Razor Pages opgavesæt (2.semester)

Dette opgavesæt rummer et antal opgaver omhandlende brug af *Razor Pages*. Opgaverne er rettet mod SWC 2.semester, og de vil benytte ”data services” som bruger *Entity Framework Core v7* til interaktion med en relationel database.

De opgaver som omhandler håndtering af et *User*-begreb er stærkt inspireret af opgaver oprindeligt lavet af **Henrik Høltzer**. Al kredit til Henrik Høltzer for at lave disse opgaver!

Opgaverne vil generelt fokusere mere på funktionalitet end på visuel finish. Fokus er på brug af *Razor Pages* og *EFCore*, 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 (dette gælder dog ikke **WebShopRPV0**). I *Unsolved*-mappen udgør projektet i mappen *WebShopRPVx* således udgangspunktet for opgave **WebShopRP.X**, og i *Solved*-mappen udgør projektet i mappen *WebShopRPVx* ligeledes en løsning for opgave **WebShopRP.X**.

[WebShopRP.0 (Razor Pages repetition) 2](#_Toc133836161)

[WebShopRP.1 (Razor Pages – overblik og Pages for Customer) 17](#_Toc133836162)

[WebShopRP.2 (Razor Pages – Pages for Order) 22](#_Toc133836163)

[WebShopRP.3 (Razor Pages – Statistics services) 25](#_Toc133836164)

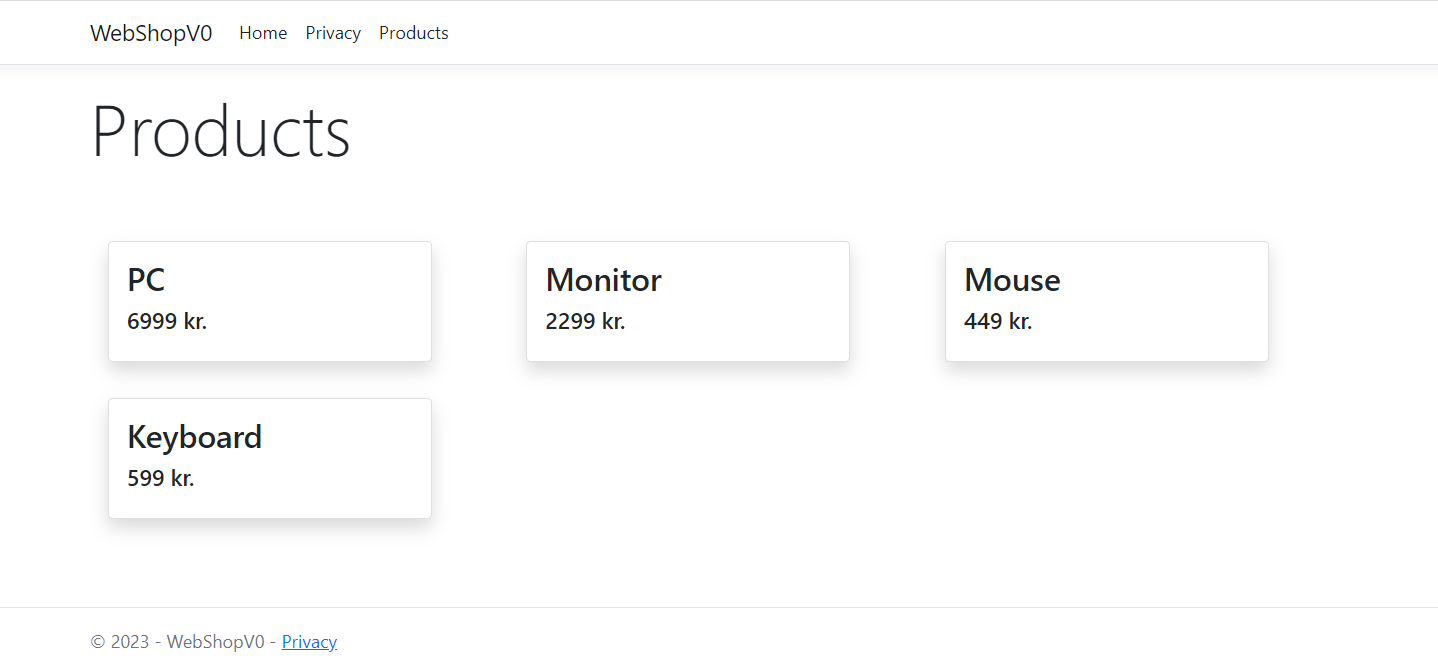
[WebShopRP.4 (Razor Pages – User authentication med Cookies) 29](#_Toc133836165)

[WebShopRP.5 (Razor Pages – rolle-baseret authorization) 36](#_Toc133836166)

# WebShopRP.0 (Razor Pages repetition)

Dette er ”nulte” opgave i opgave-serien **WebShopRP**, mest tænkt som repetition.

Her 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 (WebShopRPV0)

Opret et nyt projekt af typen: **ASP.NET Core Web App** med navnet: **WebShopRPV0**. Under *Framework* vælges **.NET 7.0 (Standard Term Support)**.

## 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**. 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 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** skal 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 i en **OnGet**-metode, som hentes via en **Service**. **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 se 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; }

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 Product-oversigen

return RedirectToPage("All");

}

}

Bemærk vores **Data**-property af typen **Product**, som har annoteringen **[BindProperty]**. Denne gør det muligt at ”binde” denne propery 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. Men vi har jo ikke selv initialiseret **Data**…? Det gør frameworket netop for os, når denne **Create**-page bliver konstrueret! Det er også derfor det er vigtigt at **Product** har en constructor uden parametre, for det er netop denne constructor frameworket bruger.

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 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 😊.

.

# WebShopRP.1 (Razor Pages – overblik og Pages for Customer)

Dette er første opgave i opgave-serien **WebShopRP**, og den tager udgangspunkt i versionen i mappen **Unsolved/WebShopRPV1**. Med ”første” menes der her den første opgave, som tager afsæt i et projekt med en hel del på forhånd givne klasser.

Her er der taget et ganske stort skridt op fra den ”nulte” opgave, idet den givne version af **WebShopRP** fra starten rummer et ret stort antal klasser og pages. Her følger en oversigt over det givne (ud over det, som følger med en standard *Razor Page* applikation):

|  |  |
| --- | --- |
| **Mappe** | **Indhold og formål** |
| **Models/Generated** | Indeholder resultatet af at køre *EF Core Power Tools* på databasen (se senere), hvilket medfører generering af fire domæne-klasser **Product**, **Customer**, **Order** og **User**. |
| **Models/Extended** | Indeholder partielle definitioner af de fire domæne-klasser. Deres formål er i store træk at definere hvilke interfaces klasserne implementerer, samt at im­ple­mentere metoden **Update**, fra interfacet **IUpdateFromOther**. |
| **Models/interfaces** | Rummer et antal interfaces, som domæne-klasserne hver især implementerer visse af (se **Models/Extended**). De vigtigste interfaces er **IHasId**, **IHasName**, **IHasOrders** samt **IUpdateFromOther**. De to sidste interfaces **IStatistics** og **ITotalStatistics** benyttes i.f.m. implementering af statistik-services i de senere opgaver. |
| **Models/Stat** | Rummer klasser til implementering af statistik-services, som kommer i spil i de senere opgaver |
| **Models/WebShop­DbContext.cs** | Den applikations-specifikke **DbContext**-klasse, som blev genereret ved at køre *EF Core Power Tools* på databasen. |
| **Pages/Base** | Rummer et antal klasser, som bruges som base-klasser for PageModel-klasser. For de tre domæne-klasser **Product**, **Customer** og **Order** kommer vi til at lave de samme Pages (**Create**, **Edit**, **Delete**, **View**, **All**), og da de i store træk kommer til at fungere på samme måde – d.v.s. at **Delete** for **Product** kommer til at virke på samme måde som **Delete** for **Customer,** o.s.v. – giver det mening at lave base-klasser for hver slags Page. Desuden skal (næsten) alle Pages have en data-bærende property samt en reference til en data service, hvorfor det giver mening også at lave et par base-klasser på øverste niveau. |
| **Pages/Products** | Rummer konkrete Pages og tilhørende PageModels for **Product**-klassen, mere specifikt de fem Pages **Create**, **Edit**, **Delete**, **View**, **All**. Vi kommer til at skulle lave tilsvarende samlinger af Pages for hhv. **Customer** og **Order**. |
| **Services/Data/**  **Interfaces** | Rummer interfacet **IDataService**, som er et generelt interface for en data ser­vice, samt interfacet **IProductDataService**, som er specifikt for **Product**. Vi kom­mer til at tilføje flere interfaces til denne folder. |
| **Services/Data/Base** | Rummer base-klasserne **EFCDataServiceBase** og **EFCDataServiceAppBase**,som implementerer **IDataService** ved brug af *Entity Framework Core*. |
| **Services/Data/EFC­ProductDataService.cs** | **Product**-specifik data-service implementation, baseret på de ovennævnte base-klasser. Vi kommer til at lave tilsvarende klasser for de andre domæne-klasser. |
| **Services/Stat** | Klasser som implementerer statistik-services. Vi vender tilbage til disse klasser i de senere opgaver. |
| **Pages/Shared/**  **\_Layout.cshtml** | I denne Page er der tilføjet et **<li>**-tag, som gør det muligt at navigere til **All**-pagen for **Product**. Vi kommer til at tilføje lignende tags for de andre domæne-klasser, samt for **User**-orienteret funktionalitet. |
| **Program.cs** | Her er der tilføjet en registrering af **IProductDataService**. Vi kommer til at til­føje lignende registreringer af data-services for andre domæne-klasser. |
| **appsettings.json** | Her kan man finde den connection string, som skal bruges til at skabe forbind­else til databasen. Herved undgås, at den ligger direkte i selve koden. Den benyttes i metoden **OnConfiguring** i **WebShopDBContext**-klassen. |

## Trin 0 (Database, appsettings.json)

Applikationen bruger en database som datakilde. Lav en ny lokal *SQL Server* database kaldet **WebShopDB** (**NB**: hvis du laver databasen et andet sted, skal du selv rette connection string til i **appsettings.json**), og kør efterfølgende det SQL-script, som ligger i **DBScript.sql**. Kontrollér, at tabellerne er blevet genereret korrekt, og at der er kommet data i tabellerne (tabellen *User* skal være tom).

## Trin 1 (Overblik, Models)

Der er mange klasser i spil, og det er helt sikkert udfordrende at få overblik over det hele. Start med at se folderen **Models/Interface** igennem, og fokusér på de fire interfaces **IHasId**, **IHasName**, **IHasOrders** samt **IUpdateFromOther**. Det er hver især nogle ganske små interfaces.

Se efterfølgende nærmere på **Product**-klassen i **Models/Generated**-folderen. Den er blevet genereret af *Entity Core Power Tools*. Hvordan er den relateret til de fire ovennævnte interfaces? Hvad mangler i.f.t. at ”opfylde” de fire interfaces? Fortsæt til **Product**-klassen i **Models/Extended**-folderen. Den er tilføjet for at supplere den auto-genererede **Product**-klasse. Hvad er det specifikt den tilføjer? Hvordan hænger den sam­lede definition af **Product**-klassen sammen med definitionen af tabellen *Product*?

Et lidt tricky spørgsmål: Hvorfor har **Product** mon en **Orders** property af typen **ICollection<Order>**? Dette kræver, at du ser lidt nærmere på databasens opbygning. Hvordan er *Order*-tabellen relateret til *Product*-tabellen?

## Trin 2 (Overblik, Pages/Base)

Denne folder rummer som nævnt en hel del klasser, som agerer som base-klasser for PageModel-klasserne hørende til forskellige kategorier af Pages (**Create**, **Edit**, **Delete**, **View**, **All**). Den øverste klasse i arve-hierar­kiet er **PageModelAppBase**, som de fleste PageModel-klasser kommer til at arve fra (dog oftest ikke direk­te, men via andre base-klasse, som selv arver fra **PageModelAppBase**).

Start med at danne dig en forståelse af elementerne i **PageModelAppBase**, og efterfølgende **PageModel­ExistingData**. Intentionen er at skubbe de elementer (instance field med reference til en data service, public properties som rummer data, constructor som initialiserer reference til data service, **OnGet**/**OnPost** implementation) så langt op i arve-hierarkiet som muligt. Hvilke elementer er ikke dækket af disse to base-klasser?

Arbejd dig nu gennem klasserne **CreatePageModel**, **DeletePageModel**, **EditPageModel**, **ViewPageModel** og **GetAllPageModel** (vi springer **StatsPageModel** over for nu). Deres navne skulle gerne indikere, hvilken kategori af Pages hver klasse hører til. Læg for hver klasser mærke til:

* Hvilken base-klasse arver klassen fra (og hvorfor)?
* Hvilke elementer bliver gjort specifikke i hver klasse?

Du skulle gerne ende med en rimelig forståelse for, hvorfor hver af disse klasser udgør en rimelig base-klasse for de specifikke PageModel-klasser, vi senere skal impementere.

## Trin 3 (Overblik, Services/Data)

Denne folder rummer som nævnt de klasser, som implementerer et generelt data-service koncept ved hjælp af *Entity Framework Core*. Start med at se på de to interfaces i folderen **Interfaces**. Hvilket interface er generelt? Hvilket interface er **Product**-specifikt? Og hvorfor fylder det sidste så lidt?

Fortsæt med de to base-klasser i folderen **Base**. Den vigtigste er **EFCDataServiceBase**, som er et forsøg på at lave en helt generel – men dog baseret på *Entity Framework Core* – implementation af **IDataService** interfacet. Giver implementation af metoderne fra **IDataService** mening for dig? Hvilket formål tjener de to **protected** metoder?

Tag til sidst et kig på **EFCProductDataService**. Hvorfor er **GetAllWithIncludes** mon implementeret på denne måde? Prøv at køre applikationen, gå til oversigten over **Products**, og klik på **View** på f.eks. *Gaming Laptop*. Bemærk teksten *”Part of 2 Orders”*. Prøv at udkommentere **GetAllWithIncludes** fra **EFCProductDataSer­vice**, og kør igen. Hvad bliver teksten nu? Og hvorfor mon? Husk at indkommentere igen 😊.

## Trin 4 (Overblik, Pages/Products)

Nu er vi endelig i stilling til at se på, hvordan nogle konkrete Pages og PageModels er implementeret. Folderen **Pages/Products** rummer alle Pages + PageModels for de fem Pages (**Create**, **Edit**, **Delete**, **View**, **All**). Start med at se på PageModel-klasserne for alle fem sider. De rummer ikke meget kode… Hvad er det, der gør dem **Product**-specifikke?

Fortsæt til **All.cshtml**. Dette er således Page for ”se alle” funktionen. Her er benyttet et Card-orienteret layout (dette er ikke i sig selv så vigtigt, man kunne også have brugt et Tabel-orienteret layout). Det vig­tigste er at forstå, hvordan denne kode kan generere et antal Cards, som svarer til antallet af **Products**. Benærk også de tre anchor-tags; hvad er deres funktion? Hvordan er det specificeret, hvilken Page de skal navigere hen til? Og hvad bruges **asp-route-id** til (Hint: hvordan er **OnGet** implementeret på f.eks. en **Edit**-page?)

Arbejd dig efterfølgende gennem de fire andre Pages (**Create**, **Edit**, **Delete**, **View**). Flere af dem minder meget om hinanden. Prøv at køre appen og brug de forskellige funktioner. Giver udseendet af de forskellige Pages mening i.f.t. hvordan de er defineret i **.cshtml**-filerne?

**NB**: Fandt du en (mulig) fejl ved din afprøvning af af appen…? Mere specifikt ved brug af **Delete**…? Hvis ikke, så prøv at slette et af de **Products**, som blev oprettet ved kørsel af SQL-scriptet i **DBScript.sql**. Det går ikke så godt 😐. Hvad er problemet mon…? Prøv at oprette et helt nyt **Product**, og prøv så efterfølgende at slette det. Gik det godt? Vi ser nærmere på problemet i næste opgave, men hvis du har mod på det, så prøv at se lidt nærmere på relationerne mellem tabellerne i databasen. Her kan der være et hint om, hvad pro­blemet er.

## Trin 5 (Implementation, Pages/Customers)

Efter at have forstået hvordan hele ”familien” af **Product**-orienterede Pages er bygget op, skal vi nu – langt om længe – selv lave noget. Du skal nu implementere en tilsvarende familie af Pages for **Customer**-klassen. Dette kan i meget høj grad gøres ved at lægge sig tæt op af implementationen for **Product**. Mere specifikt skal du altså:

1. Få overblik over, hvordan **Customer**-klassen er anderledes end **Product**-klassen (der er ikke den store forskel)
2. Implementere en **Customer**-specifik data-service (= et nyt interface og en ny klasse). Husk at overveje, om du også bør override **GetAllWithIncludes**.
3. Registrere din nye data-service i **Program.cs**
4. Lave en ny folder **Pages/Customers.**
5. Implementere de fem Pages (**Create**, **Edit**, **Delete**, **View**, **All**) for **Customer** i **Pages/Customers**. Igen: du kan for hver Page lægge dig meget tæt op af implementationen af den tilsvarende Page for **Product**.
6. Tilføje et navigation-link i **\_Layout.cs**, lige under det tilsvarende link for **Product**.

Husk – selvfølgelig – at teste, om alle fem Pages virker som de skal 😊. **NB**: Du vil kunne opleve en fejl ved **Delete**, som svarer meget til hvad vi så for sletning af **Products**. Som sagt ser vi nærmere på dette problem i den næste opgave.

## Trin 6 (Reflektion)

Med de givne base-klasser – samt de eksisterende **Product**-Pages – var det muligvis ikke den helt store op­gave at ”kopiere” det hele for **Customers**. Men føler du også, at din forståelse er fulgt med? Prøv om du kan redegøre for, hvordan f.eks. **Create**-pagen for **Customer** virker, ned i de mindste detaljer. Overvej f.eks. disse spørgsmål:

* Hvordan ved **Create**-pagen, at det indtastede data skal bruges i en **OnPost**-metode?
* Hvor er den property, som det indtastede data bindes til?
* Hvor er referencen til data-servicen?
* Hvor – og hvornår – bliver referencen til data-servicen initialiseret?
* Hvor – og hvornår – bliver referencen til data-servicen benyttet?
* Hvordan er **OnGet**/**OnPost** implementeret?

Hvis du ikke føler du kan svare på disse spørgsmål, bør du tage en runde mere med at forstå de base-klasser, som de konkrete PageModel-klasser arver fra.

# WebShopRP.2 (Razor Pages – Pages for Order)

Dette er anden opgave i opgave-serien **WebShopRP**, og den tager udgangspunkt i versionen i mappen **Unsolved/WebShopRPV2** (denne version udgør også et løsningsforslag til **WebShopRP.1**).

I denne opgave udbygger vi applikationen ved at gentage Trin 5 fra **WebShopRP.1**, men nu for **Order**-klassen. **Order**-klassen er en smule mere kompleks, idet *Order*-tabellen refererer til både *Product*-tabellen og *Customer*-tabellen med *foreign keys*. Dette afspejler sig også i den **Order**-klasse, som *EF Core Power Tools* har genereret.

## Trin 1 (Models/Generated/Order.cs)

Start med at undersøge **Order**-klassen (som er genereret af *EF Core Power Tools*). Hvordan hænger den sammen med *Order*-tabellen? Hvorfor har den både f.eks. en **ProductId** property, men også en **Product** property? Hvilken af de to properties er det, vi selv kan sætte, og hvilken property er det, som *Entity Framework Core* sætter for os? Med andre ord; hvis vi vil lave et nyt **Order**-objekt, som skal referere til eksisterende **Product**- og **Customer**-objekter, hvilke properties skal vi så sætte, før vi ”overdrager” det nye **Order**-objekt til *Entity Framework Core* (via vores data-services)? Hint: en af forcerne ved *Entity Framework Core* er, at objekt-referencer via id’er bliver automatisk ”resolvet” til rigtige objekt-referencer.

## Trin 2 (Models/Extended/Order.cs)

Undersøg **Order**-klassen i denne folder. Hvorfor implementerer **Order** ikke **IHasName** og **IHasOrders**? Og hvorfor kopierer **Update**-metoden ikke de to properties **Customer** og **Product**?

## Trin 3 (Implementation, Pages/Order)

Du skal nu implementere de fem Pages (**Create, Edit, Delete, View, All**) for **Order**-klassen. Dette kan gøres ved at tage udgangspunkt i de tilsvarende implementationer for **Product** eller **Customer**, men der er nogle vigtige forskelle.

Mere specifikt skal du altså:

1. Implementere en **Order**-specifik data-service (= et nyt interface og en ny klasse). Husk at overveje, om du også bør override **GetAllWithIncludes** (det er nok ganske vigtigt her).
2. Registrere din nye data-service i **Program.cs**
3. Lave en ny folder **Pages/Orders.**
4. Implementere **All**-page. M.h.t. PageModel-klassen er den helt som de andre, men m.h.t. Page skal du overveje, hvordan man bedst præsenterer en **Order** på et Card. Husk, at man jo nemt kan få fat på information om den **Customer**/**Product**, som en given **Order** er relateret til.
5. Implementere **Create**-page. Den største udfordring her er, hvordan man skal gøre det muligt for brugeren at vælge det **Product** og den **Customer**, som ordren skal henvise til. Dette gøres nemmest ved brug af en **SelectList** (se nedenfor).
6. Implementere **Delete**-page. Den bør ligge meget tæt på de andre **Delete**-pages.
7. Implementere **Edit**-page. Her skal man igennem de samme overvejelser m.h.t. brug af **SelectList** som ved **Create**, men faktisk vil implementationen være stort set identisk.
8. Implementere **View**-page. Her behøver man jo faktisk ikke at bruge **SelectList**, da man jo ikke kan ændre noget på denne Page. Implementationen af denne burde derfor være rimeligt ligetil.
9. Tilføje et navigation-link i **\_Layout.cs**, lige under det tilsvarende link for **Customer**.
10. Husk – selvfølgelig – at teste, om alle fem Pages virker som de skal 😊.

## Trin 4 (Et problem med Delete)

Vi har jo et udestående problem med **Delete**-funktionen for **Product** og **Customer**… Pro­blemet er følgende: Når vi har en **Order** som refererer til en **Customer** og et **Product**, så er det jo ikke så smart hvis vi sletter den **Customer**/**Product**, da referencerne på **Order** dermed ikke længere er valide. Faktisk bør det være sådan, at man ikke kan slette en **Customer**/**Product**, hvis der er en **Order** der refererer til den. Er det mon svært at afgøre…? Det er heldigvis ret nemt, for både **Customer** og **Product** har jo en **Orders**-property! Så reglen må være: En **Customer**/**Product** må kun slettes, hvis dens **Orders**-property er tom.

Dette implementer vi ved et par små ændringer: Tilføj en ny property til **IHasOrders**-interfacet

bool HasAnyOrders { get; }

Implementér denne property i både **Product-** og **Customer**-klassen (i **Models/Extended**-folderen):

public bool HasAnyOrders => Orders != null && Orders.Count > 0;

Endelig skal vi tilrette **All.cshtml** for både **Product** og **Customer**:

@if (!customer.HasAnyOrders)

{

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

type="button"

title="Delete"

**asp-page**="Delete"

**asp-route-id**="@customer.Id"><i class="fa fa-trash"></i>

</**a**>

}

På denne måde bliver *Delete*-knappen slet ikke vist, hvis man ikke må slette den pågældende **Customer** /**Product**. Prøv det!

Om **SelectList**: En **SelectList** en en kontrol-type, som egner sig godt i situationer, hvor brugeren skal vælge et enkelt objekt fra en liste af objekter. I PageModel-klassen vil man blot have properties af denne type, f.eks. således:

public SelectList CustomerList { get; set; }

public SelectList ProductList { get; set; }

Disse skal så initialiseres i PageModel-constructoren. Her specificerer man

* De elementer, man skal kunne vælge mellem.
* Den værdi, som bliver ”returneret”, når brugeren vælger et element (defineres i form af et property-navn)
* Den værdi, der skal vises for hvert element i listen (defineres i form af et property-navn)

For de to lister vi skal bruge her, vil det naturlige valg se således ud:

public CreateModel(

IOrderDataService dataService,

IProductDataService productDataService,

ICustomerDataService customerDataService) : base(dataService)

{

CustomerList = new SelectList(customerDataService.GetAll(), nameof(Customer.Id), nameof(Customer.Name));

ProductList = new SelectList(productDataService.GetAll(), nameof(Product.Id), nameof(Product.Name));

}

D.v.s. vi bruger de relevante data-services til at hente alle **Products** og **Customers**, og for begge lister er det **Id** som bruges som retur-værdi, og **Name** som vises i listen.

I Page ser det tilsvarende således ud:

<**select** **asp-for**="@Model.Data.CustomerId" **asp-items**="@Model.CustomerList" class="form-control"> </**select**>

D.v.s. **asp-for** fortæller hvilken property på **Order**-objektet som skal sættes til den valgte værdi, og **asp-items** refererer til vores property af typen **SelectList**. Der vil naturligvis være et tilsvarende **select**-tag for **Product**-listen. Men overordnet set er det faktisk ikke meget mere kompliceret end et almindeligt **input**-tag i en form-kontrol.

# WebShopRP.3 (Razor Pages – Statistics services)

Dette er tredje opgave i opgave-serien **WebShopRP**, og den tager udgangspunkt i versionen i mappen **Unsolved/WebShopRPV3** (denne version udgør også et løsningsforslag til **WebShopRP.2**).

I denne opgave ser vi et eksempel på en anden form for service, mere specifikt en **statistik-service**, som kan udregne noget simpel statistik på ordrer, set fra et produkt- eller kunde-perspektiv.

Produkt-statistikken dækker følgende:

* For hvert produkt:
  + Totalt antal ordrer, som produktet indgår i
  + Totalt styk-antal for dette produkt
  + Total pris (pris gange styk-antal)
* Samlet:
  + Totalt antal ordrer for alle produkter
  + Totalt styk-antal for alle produkt
  + Total pris for alle bestilte produkter

Kunde-statistikken dækker følgende

* For hver kunde
  + Totalt antal produkter, som kunden har bestilt
  + For hvert produkt: Total styk-antal, som kunden har bestilt
  + Total pris for kundens ordrer
* Samlet:
  + Totalt antal ordrer for alle kunder
  + For hvert produkt: Totalt styk-antal for alle kunder
  + Total pris for alle kundernes ordrer

Således er kunde-statistikken en anelse mere kompleks end produkt-statistikken, da man gerne vil have en en ”per-produkt” sum for hver kunde.

M.h.t. hvordan statistikkerne visuelt kan præsenteres, kunne det være noget i denne stil:

Et billede, der indeholder tekst, skærmbillede, skærm

Automatisk genereret beskrivelse

Et billede, der indeholder tekst, indendørs, skærmbillede

Automatisk genereret beskrivelse

## Trin 1 (Overblik, Models/Stat)

Denne folder rummer klasser til repræsentation af de omtalte statistikker. Start med at undersøge de to base-klasser **StatisticsBase** og **TotalStatisticsBase**. De er tænkt som base-klasser for at repræsentere ordre-statistik for hhv. en enkelt ”entity” (i praksis en kunde eller et produkt) samt for en samling af disse. Tag også et kig på de interfaces, klasserne implementerer. Hvordan er det sikret, at disse base-klasser kun kan blive brugt på data-typer, som har en **Orders**-property?

Tag herefter et kig på de to klasser **ProductStatistics** og **CustomerStatistics** (og de tilsvarende **Total…**-klasser). De er begge ret simple, men klasserne for **Customer** er dog en smule mere komplekse. På hvilken måde, og hvorfor?

## Trin 2 (Overblik, Services/Stat)

Denne folder rummer klasser til implementering af de omtalte statistik-services. Start med at undersøge de interfaces der ligger i **Interfaces**-folderen (de er alle meget simple). Fortsæt derefter til **StatisticsService­Base**-klassen i **Base**-folderen. Denne klasse er lidt atypisk, idet den tager hele tre type-parametre! Selve implementationen af **GetStats** er dog ikke i sig selv voldsomt kompliceret. Man kan dog godt stille spørgs­målstegn ved, om ikke alle disse type-parametre og interfaces er mere i vejen end gavnlige… Hvad – om noget – får vi overhovedet ud af at have et grundlæggende **IStatisticsService** interface? Hvad nu hvis man fjernede det, kunne man så simplificere selve klasse-definitionerne? **PS**: Dette er mere tænkt som noget man kan fundere lidt over, uden nødvendigvis at udføre det 😊.

Tag til sidst et kig på de to konkrete service-implementationer **ProductStatisticsService** og **CustomerStatis­tics­Service**. De er ret små; hvad er egentlig deres formål? Hvilket Design Pattern peger de i retning af?

## Trin 3 (Implementation, Product statistics)

Givet disse base-klasser for repræsentation samt konstruktion af statistik-objekter, udestår egentlig ”bare” at bruge dem til at lave to Pages til at vise statistikken.

I **Pages/Products**-folderen, tilføj en ny *Razor Page* med navnet **Stats**. Implementér **StatsModel**-klassen ved brug af base-klassen **StatsPageModel** (hvis du ikke allerede har gjort det, vil det også være en god ide lige at sætte sig ind i, hvordan **StatsPageModel** er skruet sammen).

Det er lidt mere udfordrende at finde ud af, præcis hvordan selve statistikken skal præsenteres. Her vil det nok være et godt valg at bruge et tabel-orienteret layout, da der jo skal vises information for hvert produkt. Hvis du ikke kan huske hvordan man laver et tabel-orienteret layout, gives der et eksempel i slutningen af opgaveteksten (se dette).

Når **Stats**-Page er lavet, skal vi have registreret produkt-statistik servicen i **Program.cs**. Indsæt denne linje efter de linjer som registrerer vores data-services:

builder.Services.AddTransient<IProductStatisticsService, ProductStatisticsService>();

Bemærk at vi her bruger **AddTransient**. Hvad er egentlig forskellen på **AddTransient** og **AddSingleton**?

Til sidst skal du tilføje et navigation-link til **\_Layout.cs**, så man kan navigere hen til produkt-statistik fra forsiden, og selvfølgelig teste at din nye Page virker 😊.

## Trin 4 (Implementation, Customer statistics) [NB: Vanskelig]

I **Pages/Customer**-folderen, tilføj en ny *Razor Page* med navnet **Stats**. Implementér **StatsModel**-klassen ved brug af base-klassen **StatsPageModel**. Dette er imidlertdi næppe nok, da vi på en eller anden måde skal kunne præsentere kunde-statistik for hvert produkt, og altså f.eks. også skal kunne vise, at en kunde har bestilt 0 stk. af et produkt. Så vi bliver nok nødt til at have en liste over alle **Products** som en property i denne PageModel-klasse. Og den skal jo også initialiseres på en eller anden måde…

Ligeledes bliver selve layoutet af kunde-statistikken mere udfordrende, da vi nok får brug for et variabelt antal rækker og søjler, til at vise per-produkt statistik for alle kunder. Man kan sagtens have løkker inden i løkker i en layout-definition, men man skal holde tungen lige i munden… eller måske er en tabel slet ikke vejen frem? Prøv at eksperimentere lidt med forskellige tilgange til dette.

Når **Stats**-Page engang er lavet, skal vi have registreret kunde-statistik servicen i **Program.cs**. Indsæt denne linje efter den tilsvarende linje for produkt-statistik servicen:

builder.Services.AddTransient< ICustomerStatisticsService, CustomerStatisticsService>();

Til sidst skal du tilføje et navigation-link til **\_Layout.cshtml**, så man kan navigere hen til kunde-statistik fra for­siden, og selvfølgelig teste at din nye Page virker 😊.

Om **table-layout**: Dette kan være et eksempel på et table-layout, til brug med produkt-statistik:

<table class="table table-bordered table-striped">

<colgroup>

<col style="width : 25%">

<col style="width : 25%">

<col style="width : 25%">

<col style="width : 25%">

</colgroup>

<thead>

<tr class="table-dark">

<th>Id (Name)</th>

<th># Orders</th>

<th># Ordered</th>

<th>Total Cost</th>

</tr>

</thead>

<tbody>

@if (Model.Statistics != null)

{

foreach (var stat in Model.Statistics.Stats.Values)

{

<tr>

<td># @stat.Id (@stat.Name)</td>

<td>@stat.TotalOrders</td>

<td>@stat.TotalOrdered</td>

<td>@stat.TotalPrice kr.</td>

</tr>

}

<tr class="table-success">

<td>TOTAL</td>

<td>@Model.Statistics.TotalOrders</td>

<td>@Model.Statistics.TotalOrdered</td>

<td>@Model.Statistics.TotalPrice kr. </td>

</tr>

}

</tbody>

</table>

# WebShopRP.4 (Razor Pages – User authentication med Cookies)

Dette er fjerde opgave i opgave-serien **WebShopRP**, og den tager udgangspunkt i versionen i mappen **Unsolved/WebShopRPV4** (denne version udgør også et løsningsforslag til **WebShopRP.3**).

Opgaven går primært ud på at implementeres en login-funktion. Login-funktionen vil benytte en simpel Cookie-baseret authentication-strategi.

## Trin 1 (Overblik, User klassen)

**Models**-folderen rummer allerede en **User**-klassen, fordelt på de to under-foldere **Generated** og **Extended**. Tag et kig på definitionen af **User** (det er en meget simpel klasse).

## Trin 2 (UserDataService)

Tilføj et nyt interface **IUserDataService** til folderen **Services/Data/Interfaces**:

public interface IUserDataService : IDataService<User>

{

/// <summary>

/// Verifies that a provided user name and password matches a known user profile.

/// </summary>

/// <returns>User matching the provided information, otherwise null.</returns>

User? VerifyUser(string providedUserName, string providedPassword);

}

Tilføj en ny klasse **EFCUserDataService** til folderen **Services/Data**:

public class EFCUserDataService : EFCDataServiceAppBase<User>, IUserDataService

{

public User? VerifyUser(string providedUserName, string providedPassword)

{

User? user = GetAll().FirstOrDefault(u => u.UserName == providedUserName);

if (user == null || user.Password != providedPassword)

return null;

return user;

}

}

Dette giver os en meget simpel **User** data-service. I første omgang fokuserer vi ikke så meget på, hvordan man rent faktisk skal kunne oprette en ny **User**.

## Trin 3 (Registrering af User-service)

Vi skal registrere vores **User**-service som en egentlig ”service” i applikationen. Dette gøres på samme vis som for de andre data-services. Tilføj denne linje i **Program.cs**:

builder.Services.AddSingleton<IUserDataService, EFCUserDataService>();

## Trin 4 (User-tabellen)

Der er allerede en *User*-tabel i **WebShopDB**-databasen, men den er tom. Tast selv noget data ind i tabellen, så den rummer et par **Users**.

## Trin 5 (LogIn-page, PageModel)

Opret en ny folder **LogIn** under **Pages**. Tilføj en ny *Razor Page* med navnet **LogInPage**. **LoginPageModel** skal nu implementeres således (**NB**: klassen er ikke komplet endnu):

public class LogInPageModel : PageModel

{

private IUserDataService \_userDataService;

public static User LoggedInUser { get; set; }

[BindProperty]

public string UserName { get; set; }

[BindProperty, DataType(DataType.Password)]

public string Password { get; set; }

public string Message { get; set; }

public LogInPageModel(IUserDataService userDataService)

{

\_userDataService = userDataService;

}

}

Ideen er, at man på **LogIn**-page (se nedefor) kan indtaste et **UserName** og et **Password**, som vi gemmer i de tilsvarende properties v.h.a. data-binding. Bemæk **DataType**-annoteringen på **Password**; dette gør, at man ikke kan se password på skærmen, når brugeren indtaster det. Bemærk også den statiske property **Logged­InUser**, som skal holde styr på, hvem der p.t. er logget ind på applikationen.

## Trin 6 (LogIn-page)

Tilføj denne kode til **LogInPage.cshtml,** under den øverste sektion med @-direktiverne:

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

<div class="container">

<div>

@Model.Message

</div>

<form method="post">

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

<label asp-for="UserName" class="control-label "></label>

<input asp-for="UserName" class="form-control" />

</div>

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

<label asp-for="Password" class="control-label"></label>

<input asp-for="Password" class="form-control" />

</div>

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

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

</div>

</form>

</div>

Denne form er i sig selv ganske simpel, da den blot rummer to tekst-bokse samt en **sumbit-button**.

## Trin 7 (LogIn-page, PageModel fortsat)

Vi skal have tilføjet noget mere kode til **LogInPageModel**, mere specifikt skal **OnPost** implementeres. Tilføj denne kode til **LogInPageModel**:

public async Task<IActionResult> OnPost()

{

LoggedInUser = \_userDataService.VerifyUser(UserName, Password);

if (LoggedInUser == null)

{

Message = "Invalid attempt";

return Page();

}

// Log in with identity

await HttpContext.SignInAsync(

CookieAuthenticationDefaults.AuthenticationScheme,

BuildClaimsPrincipal(LoggedInUser));

return RedirectToPage("/Index");

}

private ClaimsPrincipal BuildClaimsPrincipal(User user)

{

// Build Claims

List<Claim> claims = new List<Claim> {

new Claim(ClaimTypes.Name, user.UserName)

};

if (user.UserName == "admin")

claims.Add(new Claim(ClaimTypes.Role, "admin"));

// Create claims-based identity

ClaimsIdentity claimsIdentity = new ClaimsIdentity(

claims,

CookieAuthenticationDefaults.AuthenticationScheme);

// Create and return principal

return new ClaimsPrincipal(claimsIdentity);

}

Her er flere ting at bemærke: først og fremmest bruger vi nu en asynkron implementation af **OnPost**. Dette er nødvendigt for at kunne lave kaldet **HttpContext.SignInAsync**, som er asynkront. Desuden opbygger me­toden **BuildClaimsPrincipal** et **ClaimsPrincipal**-objekt, som – kort fortalt – rummer relevant information om en bruger, såsom brugernavn, password, roller, m.v.. I sidste ende bruges dette objekt til at logge på appen, med kaldet **HttpContext.SignInAsync**.

Bemærk også, at vi her lader det være op til vores **User**-service at foretage en ”verificering” af en mulig **User** (givet ved det indtastede **UserName** og **Password**). Hvis denne returnerer et **User**-objekt, er dette tegn på at verificeringen lykkedes, og selve ”Sign in”-processen kan sættes i gang.

Endelig skal det også bemærkes, at vi – i tilfælde af at **UserName** er ”admin” – tildeler brugeren en **Role** som **admin**. Dette kommer i spil i den næste opgave…

Vi har ikke i disse opgaver specielt fokus på forstå authentication-begrebet i dybden, se evt. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-7.0> for mere information, hvis du er nysgerrig.

## Trin 8 (Program.cs)

Nu skal vi sætte brugen af vores **User**-koncept op. Vi vil gerne opnå, at appen har denne opførsel i.f.t. om nogen er logget ind eller ej:

* Ingen logget ind: Alle forsøg på navigation til Pages med data vil blive omdirigeret til **LogIn**-page
* Nogen logget ind : Alt virker som før, med fuld adgang til alle Pages.

Dette kræver nogle tilføjelser til **Program.cs**. Som det første skal kaldet af **builder.Services.AddRazorPages** ændres således:

builder.Services.AddRazorPages(options =>

{

// Add authorization options

options.Conventions.AuthorizeFolder("/Products");

options.Conventions.AuthorizeFolder("/Orders");

options.Conventions.AuthorizeFolder("/Customers");

});

Herefter skal der indsættes nogle linjer lige før kaldet af **builder.Build()**:

// Add cookie-based Authentication

builder.Services.Configure<CookiePolicyOptions>(options =>

{

options.MinimumSameSitePolicy = SameSiteMode.None;

});

builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie(options =>

{

options.LoginPath = "/LogIn/LogInPage";

});

var app = builder.Build();

Til sidst skal der tilføjes denne linje:

app.UseRouting();

app.UseAuthentication(); // Enables cookie-based Authentication

app.UseAuthorization();

Den første del udtrykker, at man ikke kan navigere til Pages i de angivne foldere, med mindre man har ”authorization” til det (hvilket alle p.t. har, blot de er logget ind). Den næste del udtrykker, at vi benytter Cookie-baseret authentication, og at man i tilfælde af forsøg på navigation til en Page man ikke har autho­rization til vil blive omdirigeret til **/LogIn/LogInPage**.

Vi har ikke i disse opgaver specielt fokus på forstå Cookie-baseret authentication i dybden, se evt. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-7.0> for mere information, hvis du er nysgerrig.

## Trin 9 (Index.cshtml.cs)

Når vi bruger cookie-baseret authentication, kan der efter et succesfuldt login ligge ”login-cookies” tilbage, som gør at vi efterfølgende automatisk bliver logget ind. Dette vil vi gerne undgå, derfor skal der tilføjes denne kode til **OnGet** metoden i **Index.cshtml.cs**:

public void OnGet()

{

if (LogInPageModel.LoggedInUser == null) // Force Signout on startup

{

HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);

}

}

## Trin 10 (Afprøv)

Afprøv, om appen virker efter hensigten (har du husket at tilføje nogle **Users** til *User*-tabellen i databasen?). Prøv evt. at udkommentere et af kaldene af **AuthorizeFolder** i **Program.cs**, for at se hvilken effekt det har.

## Trin 11 (Logout-page)

Tilføj en ny *Razor Page* til **LogIn**-folderen, med navnet **LogOutPage**. Denne Page skal ikke have nogen UI, så vi skal kun implementere **LogOutPageModel**:

public class LogOutPageModel : PageModel

{

public async Task<IActionResult> OnGet()

{

LogInPageModel.LoggedInUser = null;

await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);

return RedirectToPage("/Index");

}

}

## Trin 12 (\_Layout.cshtml)

Vi skal have tilføjet nogle knapper i menu-baren, så man kan navigere direkte til login- og logout-funk­tion­erne. Som den allerførste linje i **\_Layout.cshtml**, indsæt denne linje:

@using WebShopRP.Pages.LogIn

Alle de eksisterende **nav-item** elementer er omgivet af et **<ul>**-tag. Tilføj nu nedenstående **<ul>**-tag lige under det eksisterende **<ul>**-tag:

<ul class="navbar-nav">

@{

if (LogInPageModel.LoggedInUser == null) // No one logged in -> show "Log in"

{

<li class="nav-item">

<**a** class="btn btn-primary" **asp-area**="" **asp-page**="/LogIn/LogInPage">Log in</**a**>

</li>

}

else // Someone logged in -> show user name and "Log out"

{

<li class="nav-item nav-link text-dark me-3">

<span>User:</span> <span><b>@LogInPageModel.LoggedInUser.UserName</b></span>

</li>

<li class="nav-item">

<**a** class="btn btn-primary" **asp-area**="" **asp-page**="/LogIn/LogOutPage">Log out</**a**>

</li>

}

}

</ul>

## Trin 13 (Afprøv)

Afprøv, om den nye login/logout-funktionalitet virker efter hensigten 😊.

# WebShopRP.5 (Razor Pages – rolle-baseret authorization)

Dette er femte opgave i opgave-serien **WebShopRP**, og den tager udgangspunkt i versionen i mappen **Unsolved/WebShopRPV5** (denne version udgør også et løsningsforslag til **WebShopRP.4**).

I den forrige opgave så vi, hvordan man kan begrænse adgang til en Page med brug af et **User**-koncept. Men hvordan skal vi få oprettet en **User**? Og hvem skal have lov til at oprette nye **Users**? Vi indfører derfor et **admin**-koncept, således at kun brugere som har en **admin**-”rolle” kan oprette nye **Users**. Endelig skal vi også se, hvordan man kan kryptere passwords.

## Trin 1 (CreateUser-page, CreateUserModel)

Opret en ny folder **Admin** under **Pages**. Tilføj en ny *Razor Page* med navnet **CreateUser**. **CreateUserModel** skal nu implementeres således:

// Annotation ensures that only users with "admin" role can access this page.

[Authorize(Roles = "admin")]

public class CreateUserModel : CreatePageModel<User>

{

public CreateUserModel(IUserDataService userDataService)

: base(userDataService, "/Index")

{

}

}

Som kommentaren siger, kan man kun tilgå denne Page, hvis man har en **admin**-rolle. Husker du, at vi i Trin 7 i den forrige opgave bemærkede, at vi – i tilfælde af at **UserName** er ”admin” – tildeler brugeren en **Role** som **admin**? Det er præcis det vi gør brug af her. I praksis kommer det til at betyde, at kun brugeren med *UserName* = ”admin” får adgang til denne side.

Bemærk også, at denne klasse bruger **CreatePageModel** som base-klasse, hvorved vi jo får selve **Create**-logikken ”gratis”.

## Trin 2 (CreateUser-page, CreateUser-page)

**CreateUser**-page - som ikke er specielt bemærkelsesværdig i sig selv – skal implementeres således:

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

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

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

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

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

</div>

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

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

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

</div>

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

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

</div>

</**form**>

## Trin 3 (AccessDenied-page)

Vi får også brug for en ”access denied”-page (se senere). Opret derfor en ny folder **Account** under **Pages**, og tilføj en ny *Razor Page* til **Account**-folderen, med navnet **AccessDenied**.Den er ret simpel, og implemente­res således:

public class AccessDeniedModel : PageModel

{

public void OnGet()

{

}

}

Og

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

<h3>You have to be logged in as admin</h3>

Bemærk, at det faktisk er vigtigt at denne Page ligger præcis i **Pages/Account**-folderen, da dette er ”default setting” for en *Razor Pages* applikation, m.h.t. hvor man omdirigeres til, hvis man forsøger at tilgå en Page man ikke har adgang til.

## Trin 4 (WebShopDB, tilføjelse af admin-bruger)

I **WebShopDB**-databasen skal du nu tilføje en ny row i *User*-tabellen, med *UserName* = ”admin” og *Password* = ”secret”. Det bliver denne bruger, som får adgang til at oprette nye **Users**.

## Trin 5 (\_Layout.cshtml)

Vi skal nu tilføje navigation hen til vores **CreateUser**-page. Alle de eksisterende **nav-item** elementer i **\_Layout.cshtml** er rummet i et **<ul>**-tag. Tilføj nu nedenstående **nav-item** lige efter det sidste **nav-item** i **<ul>**-tagget:

<li class="nav-item">

<**a** class="nav-link text-dark"

**asp-area**=""

**asp-page**="/Admin/CreateUser"><i>Create User</i></**a**>

</li>

Kør appen, log ind som en anden end ”admin”, og prøv at klikke på *Create User*. Nu skulle du gerne se ”access denied”-page. Prøv nu at logge ind som ”admin”; nu skulle det gerne være muligt at oprette en ny **User** 😊.

## Trin 6 (\_Layout.cshtml)

Det ville øge brugervenligheden hvis man kun kan se navigation-linket *Create User*, når man er logget ind som en bruger, der har ret til at oprette nye **Users**. Udskift derfor koden fra Trin 5 med denne kode:

@{ // Only show if user with "admin" Role is logged in.

if (Context.User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.Role)?.Value == "admin")

{

<li class="nav-item">

<**a** class="nav-link text-dark" **asp-area**="" **asp-page**="/Admin/CreateUser"><i>Create User</i></**a**>

</li>

}

}

**NB**: Du skal også tilføje denne linje øverst i **\_layout.cshtml**:

@using System.Security.Claims

Kør appen, log ind som en anden end ”admin”; nu burde du ikke kunne se *Create User* linket. Prøv nu at logge ind som ”admin”; nu skulle det gerne være muligt at oprette en ny **User** 😊.

## Trin 7 (Password-kryptering, overblik)

Vi kan nu – når vi er logget på med **admin**-rettigheder – oprette nye **Users**. Lige nu ligger *Password* som almindelig tekst i *User*-tabellen, hvilket kan være risikabelt. Derfor vil vi gerne kryptere disse passwords. Dette gøres ved brug af klassen **PasswordHasher**, som findes i **Microsoft.AspNetCore.Identity**.

Principperne er som følger:

* Når der oprettes en ny **User**, tastes der et password ind.
* Vi kalder **Create** i vores **User**-service, som gemmer den nye **User** i databasen. Men i stedet for blot at gemme data som det er givet på **User**-objektet, ændrer vi nu værdien af **Password**-property til en ”krypteret” værdi
* Denne ”krypterede” værdi fås ved at kalde metoden **HashPassword** på et **PasswordHasher**-objekt.
* Denne metode laver en såkaldt ”hashing” af password, som resulterer i en meget lang **string**, som man ikke på nem vis kan ”unhashe”. Det er så denne ”hashede” **string**, der gemmes som *Password* i data­basen.
* Når en **User** forsøger at logge på, vil vedkommende jo indtaste sit almindelige password. Derfor skal verifikationen af en **User** nu ændres, så den nu kontrollerer om det indtastede password kan hashes til den værdi, der er gemt i databasen. Dette gøres ved at kalde metoden **VerifyHashed­Password** på et **PasswordHasher**-objekt.

## Trin 8 (Password-kryptering, implementation)

Vi kan implementere ovenstående ved at opdatere vores **User**-service. I klassen **EFCUserDataService** (i **Services/Data**-folderen) skal vi først tilføje dette instance field:

private PasswordHasher<string> \_passwordHasher = new PasswordHasher<string>();

Derefter tilføjes denne metode:

public override int Create(User user)

{

user.Password = \_passwordHasher.HashPassword(user.UserName, user.Password);

return base.Create(user);

}

Bemærk, at vi således “intercepter” kaldet til base-klassens **Create**, og derved får mulighed for at opdatere **Password** på **User**-objektet til den hashede værdi.

Vi skal også have opdateret **VerifyUser**-metoden, på denne måde:

public User? VerifyUser(string providedUserName, string providedPassword)

{

User? user = GetAll().FirstOrDefault(u => u.UserName == providedUserName);

if (user == null ||

\_passwordHasher.VerifyHashedPassword(

providedUserName,

user.Password,

providedPassword)

!= PasswordVerificationResult.Success)

return null;

return user;

}

## Trin 9 (Password-kryptering, afprøvning)

Nu er vi i stilling til at afprøve vores forbedrede **User**-håndtering. Som det første skal du slette alle rows i *User*-tabellen, da de jo alle er oprettet med ikke-krypterede passwords. Dette giver imidlertid et lille pro­blem, for hvordan skal vi nu overhovedet få oprettet nye **Users**, når vi ikke har en **admin**-bruger!? Dette kommer vi udenom på følgende måde:

I **\_Layout.cshtml**, opdatér den **if**-statement, som omringer *Create User*-linket, således:

if (true || Context.User.Claim…

I **CreateUser.cshtml.cs**, udkommentér **Authorize**-annoteringen, således:

// [Authorize(Roles = "admin")]

Kør appen; nu burde du kunne få lov at oprette en **admin**-bruger, med *UserName* = ”admin” og **Password** = ”secret”. Tjek at der bliver tilføjet en row til *User*-tabellen. Du vil også kunne se hvordan et ”hashet” pass­word ser ud.

Luk appen, og ret de to ændringer fra foroven tilbage igen. Kør appen, og log på som ”admin”. Nu skulle du kunne oprette nye **Users**. Prøv det, og prøv efterfølgende at logge ind som en af disse nyoprettede **Users**.

Det var det 😊. Hvis du vil, kan du eksperimentere yderligere med **User**-konceptet, f.eks. med at have flere forskellige **Roles**. Det kunne være en **Role** til at se data, en anden til at ændre data, og en tredje til at se statistikker. Dette vil nok kræve at **User**-klassen udvides med en **Role**-property, som så også skal kunne sættes ved oprettelse af en ny **User**.