# Razor Pages App med Database og User

Denne opgave går primært ud på at forstå opbygningen af en Razor Pages app der implementerer et **User**-begreb, samt tilføje kode til appen.

Den givne Visual Studio solution rummer en Razor Pages app, der imple­­menterer grundlæggende funk­tioner (se alle, opret og slet) for tre domæne-klasser **Kunde**, **Bil** og **Leje**. Derudover er der defineret en **User**-klasse.

## Trin 1 (Setup)

Download og unzip (VIGTIGT) filen **RosBilRPMedUser.zip** fra Moodle.

Hvis du ikke allerede har en lokal database ved navn **RosBilDB**, skal du op­rette denne database. Derefter skal – uanset om du har databasen i for­vejen eller lige har oprettet den – køre scriptet i **DBScript.sql** på den. NB: Hvis du ikke kører på en lokal database, skal du selv tilrette connection-string i **RosBilDBContext.cs** (Linje 29).

Kør appen. Du skulle gerne kunne lave operationerne ”se alle” (via link fra start-siden), ”opret” (via link i bunden af ”se alle”-siden) og ”slet” (via knap på de enkelte elementer på ”se alle”-siden) for alle tre domæne-klasser. Hvis du stød­er på problemer, er det sandsynligvis din database, der ikke er sat korrekt op.

I *Models*-folderen finder du en **User**-klasse, som er auto-genereret ud fra **User**-tabellen i databasen. Desuden er der lavet et par ”extensions” til **User**-klassen i **Extensions.cs**. I sig selv er **User**-klassen ikke så kompleks.

## Trin 2 (Opret UserRepository)

Tilføj et nyt interface **IUserRepository** til **Services**-folderen**.** Du kan f.eks. tilføje det til filen **IRepository.cs**, hvor der i bunden af filen er defineret tilsvarende interfaces. Det skal således ud:

public interface IUserRepository : IRepository<User>

{

User? VerifyUser(string providedUserName, string providedPassword);

List<string> Roller { get; }

}

Tilføj derefter klassen **UserRepository** til **Services**-folderen:

public class UserRepository : EFCRepositoryBase<User, RosBilDBContext>, IUserRepository

{

public List<string> Roller

{

get { return new List<string> { "admin", "ansat" }; }

}

public User? VerifyUser(string providedUserName, string providedPassword)

{

User? user = All.FirstOrDefault(u => u.Navn == providedUserName);

// User findes ikke, eller har angivet forkert password

if (user == null || !VerifyPassword(user, providedPassword))

return null;

return user;

}

private bool VerifyPassword(User user, string providedPassword)

{

return user.Password == providedPassword;

}

}

Dette giver os en grundlæggende funktionalitet i.f.t. at kunne verificere om et givent bruger-navn og password – som f.eks. kunne være tastet ind på en Login-page – matcher de **Users** systemet kender. Bemærk også, at vi her definerer at enhver **User** enten har rollen *admin* eller *ansat*.

I første omgang fokuserer vi ikke så meget på, hvordan man rent faktisk skal kunne oprette en ny **User**.

## Trin 3 (Registrering af UserRepository som service)

Vi skal registrere vores **UserRepository** som en ”service” i applikationen. Dette gøres på samme vis som for de andre repositories. Tilføj denne linje i **Program.cs**:

builder.Services.AddSingleton<IUserRepository, UserRepository>();

## Trin 4 (User-tabellen)

Der er allerede en *User*-tabel i **RosBilDB**-databasen, med to **Users**. Tast evt. selv noget mere data ind i tabellen, så den rummer flere **Users**. Husk at være omhyggelig med at taste roller korrekt ind, da det jo ikke her bliver tjekket om de matcher de to roller vi selv har angivet (*admin* og *ansat*)

## Trin 5 (Login-page, PageModel)

Opret en ny folder **UserLogin** under **Pages** (NB: Vær omhyggelig med at give folderen præcis dette navn, incl. stort bogstav først). Tilføj en ny *Razor Page* med navnet **Login**. **LoginModel** skal nu implementeres således (**NB**: klassen er ikke komplet endnu):

public class LoginModel : PageModel

{

private IUserRepository \_userRepository;

public static User? CurrentUser { get; set; }

[BindProperty]

public string Navn { get; set; }

[BindProperty, DataType(DataType.Password)]

public string Password { get; set; }

public string ErrorMessage { get; set; }

public LoginModel(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

}

Ideen er, at man på **Login**-page (se nedefor) kan indtaste et **Navn** 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 **Current­User**, som skal holde styr på, hvem der p.t. er logget ind på appen.

## Trin 6 (Login-page)

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

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

<div class="container">

<div>

@Model.ErrorMessage

</div>

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

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

<**label** **asp-for**="Navn" class="control-label "></**label**>

<**input** **asp-for**="Navn" 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 ret 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 **LoginModel**, mere specifikt skal **OnPost** implementeres. Tilføj denne kode til **LoginModel**:

public async Task<IActionResult> OnPost()

{

CurrentUser = \_userRepository.VerifyUser(Navn, Password);

if (CurrentUser == null)

{

ErrorMessage = "Kunne ikke logge ind";

return Page();

}

// Log ind

await HttpContext.SignInAsync(

CookieAuthenticationDefaults.AuthenticationScheme,

BuildClaimsPrincipal(CurrentUser));

return RedirectToPage("/Index");

}

private ClaimsPrincipal BuildClaimsPrincipal(User user)

{

// Opbyg Claims-liste

List<Claim> claims = new List<Claim>();

claims.Add(new Claim(ClaimTypes.Name, user.Navn));

claims.Add(new Claim(ClaimTypes.Role, user.Rolle));

// Opret ClaimsIdentity (claims plus Authentication-strategi)

ClaimsIdentity claimsIdentity = new ClaimsIdentity(

claims, CookieAuthenticationDefaults.AuthenticationScheme);

// Opret endeligt ClaimsPrincipal-objekt

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, rolle, 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**-repository at foretage en ”verificering” af en mulig **User** (givet ved det indtastede **Navn** og **Password**). Hvis denne returnerer et **User**-objekt, er dette tegn på at verificeringen lykkedes, og selve ”Log ind”-processen kan sættes i gang.

P.S: Vi har ikke i denne opgaver specielt fokus på forstå authentication-begrebet i dybden, se evt. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-9.0> 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 til dette:

builder.Services.AddRazorPages(options =>

{

// Angiv hvilke foldere login giver adgang til

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

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

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

});

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

// Angiv og konfigurér cookie-baseret Authentication

builder.Services.AddAuthentication(

CookieAuthenticationDefaults.AuthenticationScheme).AddCookie(options =>

{

options.LoginPath = "/UserLogin/Login";

options.AccessDeniedPath = "/UserLogin/AccessDenied";

});

Til sidst skal der tilføjes denne ene linje lige før kaldet af **app.UseAuthorization**):

app.UseRouting();

app.UseAuthentication(); // Aktivér cookie-baseret 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 **/UserLogin/Login**. (**AccessDenied** kommer i spil senere).

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-9.0> 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 (LoginModel.CurrentUser == null) // Force Signout on startup

{

HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);

}

}

## Trin 10 (Afprøv)

Afprøv, om appen virker efter hensigten. 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 **UserLogin**-folderen, med navnet **Logout**. Denne Page skal ikke have nogen UI, så vi skal kun implementere **LogoutModel**:

public class LogoutModel : PageModel

{

public async Task<IActionResult> OnGet()

{

LoginModel.CurrentUser = 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 RosBilRP.Pages.UserLogin

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 (LoginModel.CurrentUser == null) // Ingen logget ind -> vis "Log ind"

{

<li class="nav-item">

<**a** class="btn btn-primary" **asp-area**="" **asp-page**="/UserLogin/Login">Log ind</**a**>

</li>

}

else // En User er logget ind -> vis navn og "Log ud"

{

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

<span>User:</span> <span><b>@LoginModel.CurrentUser.Navn</b></span>

</li>

<li class="nav-item">

<**a** class="btn btn-primary" **asp-area**="" **asp-page**="/UserLogin/Logout">Log ud</**a**>

</li>

}

}

</ul>

## Trin 13 (Afprøv)

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

Nu har vi sådan set implementeret et grundlæggende **User**-begreb, men vi bruger ikke rigtig informationen om den **User** der p.t. er logget ind til noget, og vi har heller ikke nogen mulighed for at oprette en ny **User**. Det følger i de næste trin, hvor vi også skal se på det problem at alle password p.t. blot ligger som alminde­lig tekst i databasen, hvilket ikke er hensigtsmæssigt i.f.t. sikker­hed.

## Trin 14 (Opret-page og OpretModel for User)

Opret en ny folder **Users** under **Pages**. Tilføj en ny *Razor Page* med navnet **Opret**. **OpretModel** skal nu implementeres således:

[Authorize(Roles = "admin")]

public class OpretModel : PageModel

{

private IUserRepository \_repo;

[BindProperty]

public User Element { get; set; } = new User();

public SelectList Roller { get; set; }

public OpretModel(IUserRepository repo)

{

\_repo = repo;

Roller = new SelectList(\_repo.Roller);

}

public IActionResult OnPost()

{

// Tjek om det indtastede data er validt

if (!ModelState.IsValid)

{

return Page();

}

// Send data videre til repository

\_repo.Create(Element);

// Vend tilbage til startsiden

return RedirectToPage("/Index");

}

}

Bemærk den øverste linje [Authorize(Roles = "admin")]. Denne ”annotation” medfører, at man kun kan tilgå denne Page, hvis man har rollen *admin*. Altså er det kun **Users** med denne rolle, der vil kunne oprette nye **Users**.

Derudover minder denne Page-model ret meget om de øvrige Page-models for **Opret**-pages. Der er jo heller ikke så underligt, da **User** jo reelt set bare er endnu en domæne-klasse. Det er mere den måde den bruges på andre steder i appen, der gør den til noget særligt.

## Trin 15 (Opret-page for User)

**Opret**-pagen – som ikke er specielt bemærkelsesværdig i sig selv – skal implementeres således:

<h1 class="display-5 mb-3">Opret ny User</h1>

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

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

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

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

</div>

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

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

<**select** **asp-for**="@Model.Element.Rolle" **asp-items**="@Model.Roller"

class="form-control">

</**select**>

</div>

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

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

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

</div>

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

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

</div>

</**form**>

## Trin 16 (AccessDenied-page)

Under Trin 8 nævnte vi kort en **AccessDenied**-Page, som vi nu implementerer. Opret derfor en ny Razor Page med navnet **AccessDenied** i *UserLogin*-folderen. Selve implementationen er meget simpel:

public class AccessDeniedModel : PageModel

{

public void OnGet()

{

}

}

Og i .cshtml:

<h1 class="display-3 mb-5">Adgang ikke tilladt</h1>

<h3>Du skal have admin-rolle for at se denne side</h3>

## Trin 17 (\_Layout.cshtml)

Vi skal nu tilføje navigation hen til vores **Opret**-page for **User**. 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**="/Users/Opret"><i>Opret ny User</i>

</**a**>

</li>

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

## Trin 18 (\_Layout.cshtml)

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

(**NB**: Du skal også tilføje denne linje øverst i **\_layout.cshtml**: @using System.Security.Claims)

@{

// Vis kunne dette menu-punkt, hvis en User med "admin"-rolle er logget ind

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**="/Users/Opret"><i>Opret ny User</i>

</**a**>

</li>

}

}

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

## Trin 19 (Password-kryptering, overblik)

Vi kan nu – når vi er logget på med **admin**-rollen – oprette nye **Users**. Lige nu ligger passwords dog 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.
* I **OnPost** kaldes **Create** på vores **User**-repository, som gemmer den nye **User** i databasen. Men i stedet for blot at gemme data som det er angivet på **User**-objektet, vil vi nu ændre 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 20 (Password-kryptering, implementation)

Vi kan implementere ovenstående ved at opdatere vores **User**-repository. I klassen **UserRepository** skal vi først tilføje dette instance field og denne constructor:

private PasswordHasher<string> \_passwordHasher;

public UserRepository()

{

\_passwordHasher = new PasswordHasher<string>();

}

Derefter tilføjes denne metode til **UserRepository**:

public override int Create(User user)

{

user.Password = \_passwordHasher.HashPassword(user.Navn, 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 udskiftet **VerifyPassword**-metoden, på denne måde:

private bool VerifyPassword(User user, string providedPassword)

{

PasswordVerificationResult result =\_passwordHasher.VerifyHashedPassword(

user.Navn, user.Password, providedPassword);

return result == PasswordVerificationResult.Success;

}

## Trin 21 (Password-kryptering, afprøvning)

Nu er vi i stilling til at afprøve vores forbedrede håndtering af passwords. Som det første skal du slette alt data 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 **User** med en *admin*-rolle!? Det kommer vi udenom på følgende måde:

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

if (true || Context.User.Claim…

I **/Users/Opret.cshtml.cs**, udkommentér **Authorize**-annoteringen, således:

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

Kør appen; nu burde du kunne få lov at oprette en ny **User**, f.eks. med **Navn**= ”admin”, **Rolle**=”admin” og **Password** = ”admin”. Tjek at der bliver tilføjet en ny række 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**-begrebet, f.eks. med at have flere forskellige roller end de to vi har angivet. Det kunne være en rolle til at se data, en anden til at ændre data, og så videre.