Exercise 16 - User management for passport booking

In this exercise we will build a small web application to administer and book gym sessions. The focus of the exercise is to practice user management by using the Identity framework. We will also repeat many of the points we learned earlier during the course. If you feel rusty, feel free to look back at previous exercises.

We build out our application step by step through iterations of new versions and try to focus on one problem at a time. Feel free to discuss points and solutions with your classmates, but write your code individually. It is very important that you write your own code for the exercise to be an effective learning experience.

The application becomes more complex with each version and the description places more and more responsibility on you to find solutions yourself. It is not obvious that you will finish all versions in time, but then it is the journey rather than the destination that counts. However, many of the more important elements are included in the first versions.

For the first parts, there is also a conclusion for those who are unsure that they thought correctly. But try it yourself first, if you are still unsure or don't understand, you have to ask so we can straighten out the question marks!

We will code the exercise together during the week but may not focus on exactly the same things as you practice here.

Application description

The application has two main user groups: administrators and gym members.
We will differentiate these using roles.

What is to be implemented is the following:

- As **a visitor,** I want to be able to log in.
- As **a visitor ,** I **only** want to be able to see a list of gym sessions. Only logged in visitors can interact with them.

- As **a member,** I want to be able to book a gym session.
- As **a member ,** I want to be able to cancel a gym session
- As **a member,** I want to be able to see who has booked a gym session.

- As **an administrator ,** I want to be able to do everything that members can do. • As **an administrator ,** I want to be able to add new gym sessions.
- As **an administrator,** I want to be able to edit an existing gym session.

Database:

- A gym session can have many members booked on it.
- A member can be booked for several gym sessions

Entities:

- A gym session has a name, a start time, a length (in time) and a description.
- A user has a name, a timestamp when membership was started, an email address and a password.

The database thus only contains a many-to-many relationship between member and passport (plus the classes we get "in the bargain" with Identity). You will therefore need to create a "connection table" between these entities. We did it together in https://github.com/Lexicon-NET-2023/LexiconUniversity
However, we set up both types of relationships there. Both through Enrollment and directly from Student to Course.

With the new EF Core, it is no longer necessary to create this connection table manually, but we will do it anyway. It is a bigger challenge to do it manually especially how you need to write your LINQ queries.

Build the application

Gym booking v0.1 - A scale project

1.

Create a new MVC project select *"Authentication: Individual Accounts"*

2.

Create a new class in the Models folder: GymClass.cs with the public "properties":

- int Id { get; set; }
- string Name { get; set; }
- DateTime StartTime { get; set; }
- TimeSpan Duration { get; set; }
- DateTime EndTime { get { return StartTime + Duration;} }
- String Description { get; set; }

Note how EndTime only has a "get method" and thus will not be explicitly stored in the database, but calculated based on the session's start time and time span.
We'll wait until later to extend our ApplicationUser class with its new "properties".

3.

Add a DbSet for the GymClass objects to the ApplicationDbContext class. Be careful not to end up inside the constructor.

4.

Create a new class in the Models folder: ApplicationUser. We let the class inherit from IdentityUser.
We'll leave it as it is for now.

Now we're almost ready to build our project and take it for a first round of testing, but first we need to establish our *entity relationships* through *navigation properties* and *migrate* our database.

5.

We start by creating a new class ApplicationUserGymClass for the connection table itself. It is a naming convention that the class is named via the names of the other classes. We also follow the other naming conventions when we add properties.

Class name + id for foreign keys. Id or Class Name + Primary Key Id.

6.

Create a navigation property in the GymClass class. A pass can be booked by several members, so it must contain a collection of these members. Note here that we set our connection class as the navigation property.

Conclusion see point 1.

7.

Create a corresponding *navigation property* in the ApplicationUser class. A member can book several passes, so the class needs a collection of these passes. Note that here, too, we specify our connection class as the navigation property.

Conclusion see point 2.

8. Unfortunately, the functionality to define a composite key using data annotations does not yet exist. So we get to use fluent api to accomplish that. We use an override of OnModelCreating in the ApplicationDbContext class.

Conclusion see point 3.

9.

We now need to register in the Program class which type of object we use to represent our users. As well as what type to represent roles. Here we use Microsoft's default implementation.

Conclusion see point 4.

10.

Go to the Context class and add the types with which we will create the context. Look at what the base classes need!

Conclusion see point 5.

11.

In order for the identity files to be displayed in solution explorer, we need to generate them. We do that by Scaffolding them. Add > New Scaffolded Item select Identity in the left column, then Add.

We set our layout in Viewstart so we leave the first field blank. Then we choose to overwrite the files we need.

A good start can be Register, Login and Index. If we later need to change several files, we can just run the Wizard again.

Then we select our ApplicationDbContext which is already in the Data folder. We do not choose to create a new context as we usually do.

Since we have replaced IdentityUser with our own implementation ApplicationUser, we need to replace all instances where we still use services with IdentityUser. It should be specific in a file you need to change.

Conclusion see point 6.

If you scaffolded Identity before you were finished with the configuration in startup and in the context, you will also have to change all these classes, alternatively generate them again.

Documentation if something is still unclear: https:// learn.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity? view=aspnetcore-8.0&tabs=visual-studio#scaffold-identity-into -an-empty-project

Now that our basic *POCOs* are in place with navigation properties, we can *"scaffold"* a GymClassController with views.

13.
Right-click the Controllers folder and select Add > Controller. In the Wizard, you then select "MVC Controller with views, using Entity Framework"
Model class should be your new "GymClass" and as context you use ApplicationDbContext.
The rest is filled in automatically, so just click " *Add"*

When everything is in place, we just need to migrate our entities and relationships to the database and we are done with the shell application version 0.1.

14.
Add-Migration, Update.Database

Now we are ready to build the project and take it for a test drive! When you have the web application in front of you, you try to register a user. Also visit localhost:#portnr#/GymClasses and create a gym session! Does it work? Brilliant!

Also take the opportunity to open your *server object explorer* and look at the database that has been generated. Feel free to use EF Core power tools to bring up a database schema with relationships.

v0.1 done!

Gym booking v0.2 - Passport booking

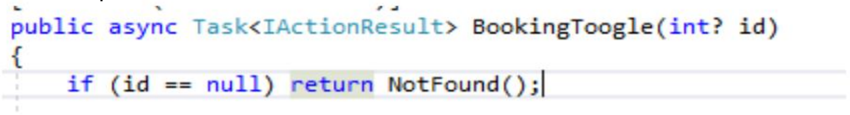
Now that we can register members and create gym passes, the natural next step in the functionality is to let members book in for passes.

In this version we will (1) write a method in our GymClassController that books on or off the logged in member to a session. We will then (2) create a link in the index view to this action and finally (3) expand the pass's details view to list the members who have booked on the pass. In all these steps there are more elegant solutions, but for now we want to keep it simple and efficient.

1.

We start by creating a *BookingToggle method* in our GymClassController:

Create a public method as follows:



```
public async Task<IActionResult> BookingToogle(int? id)
{
    if (id == null) return NotFound();
```

First of all, we need to find out which passport to book. Then we also need to find out the ID of the currently logged in member in order to probably be booked or canceled from the pass.

In the method declaration we see that the method is public and has the return type "ActionResult", we also see that it takes an input parameter Id which is an integer. This is the ID of the gym session we want to book.

With the gym pass we have already created, we can look to see if the member is among its AttendingMembers. Depending on the result, we book or unbook the member from the current pass.

Implement the rest of the method! (hint: use UserManager)

2.

To now be able to use our new action method, we need to add a link to our index view.
We open the Index view for GymClasses (/views/GymClasses/Index.cshtml).

If we look at the already existing action links to details/edit/delete, we must do exactly the same with the link text "book" and which links to the "BookingToggle" *endpoint* . Just like in the other links, we want to send the ID of the passport as an input parameter.

3.

Although it is not visible in the application, logged-in members can now book on and off the pass.
(If your logic you implemented in point 1. works :))

The last addition in this version is an opportunity to see this as well.
We open the Details view (/views/GymClasses/Details.cshtml)

At the bottom of the table, we add a printout using a foreach loop that prints the email of all members who booked the pass. In a later version we will replace the email with a name, but so far our user objects do not have a name property.

Version 0.2 is ready. Build the application and try clicking around!

Gym booking v0.3 - Logged in or not logged in, that's the question

In the task description, it says that logged in and non-logged in users should have different access to views and functionality on the page, and now that we're finally starting to get some functionality, it might be time to lock off parts of the page.

In this version:

(1) we clean up the application a bit by removing the HomeController and its views, instead we let (2) *the routing* directly take us to the list of passes. In conjunction with this, we also need to (3) clear the now dead links from our nav bar.

Then we will (4) lock off the actions that non-logged in users should not have access to. We also take care to (5) hide links that those who are not logged in do not have access to.
However, we are waiting with roles and administrators for a later version.

1.

Delete the /Controllers/HomeController.cs file and the entire /views/Home folder.

2.

Open the Program.cs file and other default controllers from “Home” to “GymClasses”

3. Open the file /Views/Shared/_Layout.cshtml
Delete the list of links in the nav bar.

Change the link text on *the page header* from "Application name" to "Gym booking" or another suitable name. Also, re-point it from the home controller to the GymClasses controller.

4

Go to GymClassesController and look at what action methods are there. Which should only a logged-in user have access to? Use the [Authorize] annotation to block these.

5.

Open Index.cshtml
From the User class, you can get answers to whether a visitor is logged in or not. Using User.Identity.IsAuthenticated returns a Boolean value that is true if the visitor is logged in, false otherwise. Use this in an *if statement* to only generate the links in views for logged in users.

Note that in code blocks you cannot write "loose strings" without enclosing them in html tags.
This applies e.g. the pipe characters (“|”) that separate certain links in the generated views. Then use <span> | for these.

Gymbokning v0.4 - The All-Mighty Admin

In this version, we will continue the work with roles. We will seed the role "Admin" and also a user who receives this. At a later stage, one can imagine a control panel where it is possible to upgrade regular users to the Admin role, but for now we'll settle for just one.

Since examples of how you can code the Seed have been given during previous code-a-longs, I will not go into details here in the exercise description, but refer to your own notes and previous projects on the class's GitHub.

Conclusion see point 7.

Seed() method:

Seed a role named “Admin”.

Seed a user with the username "admin@Gymbokning.se" and any password.

Add the user "admin@Gymbokning.se" to the role "Admin".

GymClassesController:

Block Edit-, Create- and Delete-actions from all visitors who do not belong to the role "admin"

Index-vyn:

Hide Links that only admin has access to. Use User.IsInRole("Admin") for your if statements.

Gym booking v0.5 - The user in focus?

In version 0.5, it is high time to expand our ApplicationUser class

1.

Add properties:
string FirstName { get; set; };
string LastName { get; set; };
string FullName => @”{FirstName} {LastName}”;
DateTime TimeOfRegistration { get; set; };

(For those who want to try adding TimeOfRegistration as a Shadow property)

2.

Make the same addition in the Register class (Areas/Identity/Pages/Account/Register.cshtml)

3.

Add these fields to the registration view (/Views/Account/Register.cshtml). However, TimeOfRegistration should not be filled in by the user, but set in the controller on the server side to the actual registration time.

(or automatically in an override on SaveChanges)

4.

Update GymClasses details view (/Views/GymClasses/Details.cshtml) to show FullName instead of Email.

5.

Update the seed of your user object with the new properties. This applies above all to the properties that are not nullable.

6.

Migrate the models and update the database

Gym booking v beta 1.0 - The user in focus!

We are getting ready to launch our application as version 1.0 for beta testers. For the first time the application will be used by potential end users and now it's time to polish!

1.

The "Book" link should say "Book off" if the user is already booked on the pass. And of course the opposite!

2.

Old passports whose date has passed should not be displayed in the passport list unless this is specifically requested (button, checkbox or something similar, possibly a separate view for "History")

3.

A logged in user should see two new links in the nav bar: "Booked Pass" and "History".
"Booked Passes" shows a filtered pass list with only the passes the user has signed up for.
"History" shows old passes that the user has taken and whose date has passed.

4.

A logged in user should no longer see their email or username in the upper right corner when they log in. Instead, their first and last name must be printed. Solve this by adding a new claim when a user is created in Register.cshtml.cs. Use _userManager.AddClaimAsync(). In _LoginPartial you can then extract this claim.

5.

Input Validation in all fields

6.

Update the color theme

7.

Other small fixes you discover

We are ready to release version 1.0 for beta testing! Nice work :)

ouch! (Conclusion) :)

1. public ICollection<ApplicationUserGymClass> AttendingMembers { get; set; }
2. public ICollection<ApplicationUserGymClass> AttendedClasses { get; set; }

3. protected override void OnModelCreating(ModelBuilder modelBuilder)

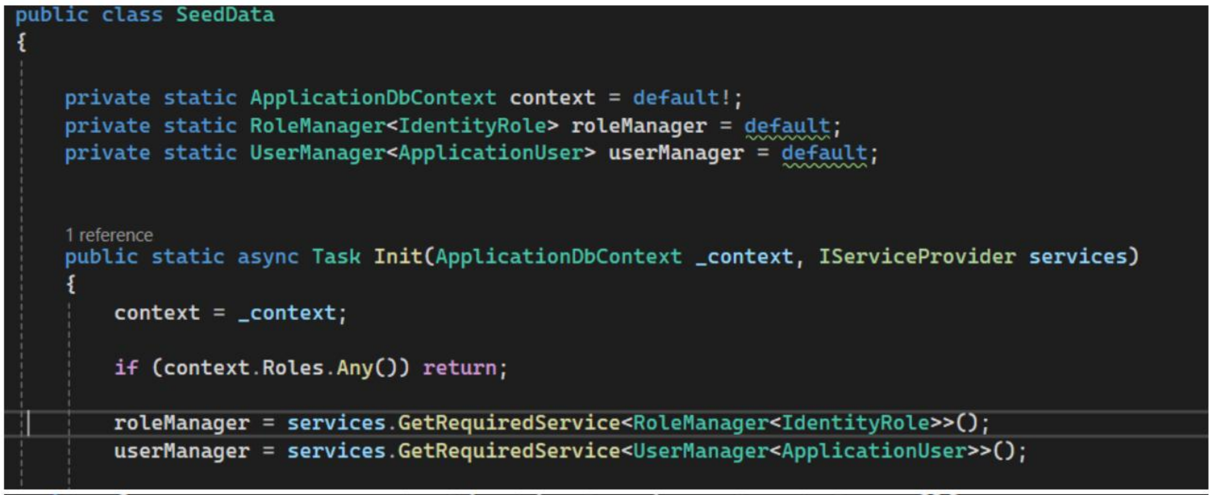
```
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<ApplicationUserGymClass>()
                .HasKey(t => new { t.ApplicationUserId, t.GymClassId });
        }
```

4. services.AddDefaultIdentity<IdentityUser>() <= Byt ut IdentityUser till ApplicationUser

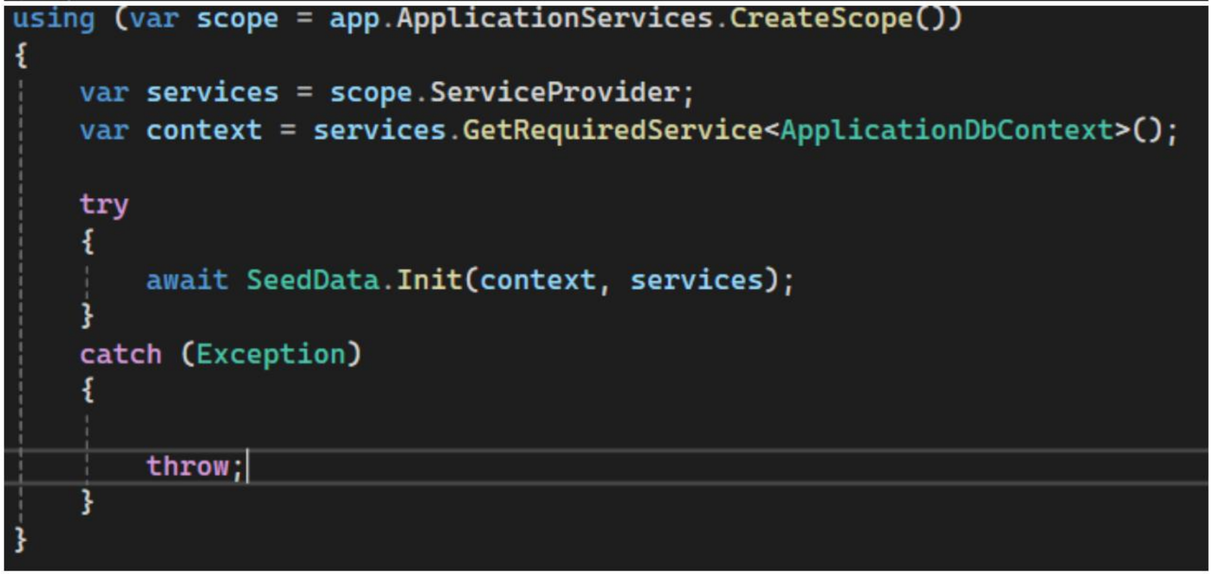
```
                .AddRoles<IdentityRole>()                    <= We need to add
```
to this line
```
        .AddEntityFrameWorkStores<ApplicationDbContext>();
```

5. ApplicationDbContext : IdentityDbContext<ApplicationUser, IdentityRole, string>
6. _LoginPartial! replace IdentityUser with ApplicationUser

7. Start code for Seeddata:



```
public class SeedData
{

    private static ApplicationDbContext context = default!;
    private static RoleManager<IdentityRole> roleManager = default;
    private static UserManager<ApplicationUser> userManager = default;


    1 reference
    public static async Task Init(ApplicationDbContext _context, IServiceProvider services)
    {
        context = _context;

        if (context.Roles.Any()) return;

        roleManager = services.GetRequiredService<RoleManager<IdentityRole>>();
        userManager = services.GetRequiredService<UserManager<ApplicationUser>>();
```



```
using (var scope = app.ApplicationServices.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ApplicationDbContext>();

    try
    {
        await SeedData.Init(context, services);
    }
    catch (Exception)
    {

        throw;
    }
}
```