Workshop: ASP.NET Core Identity
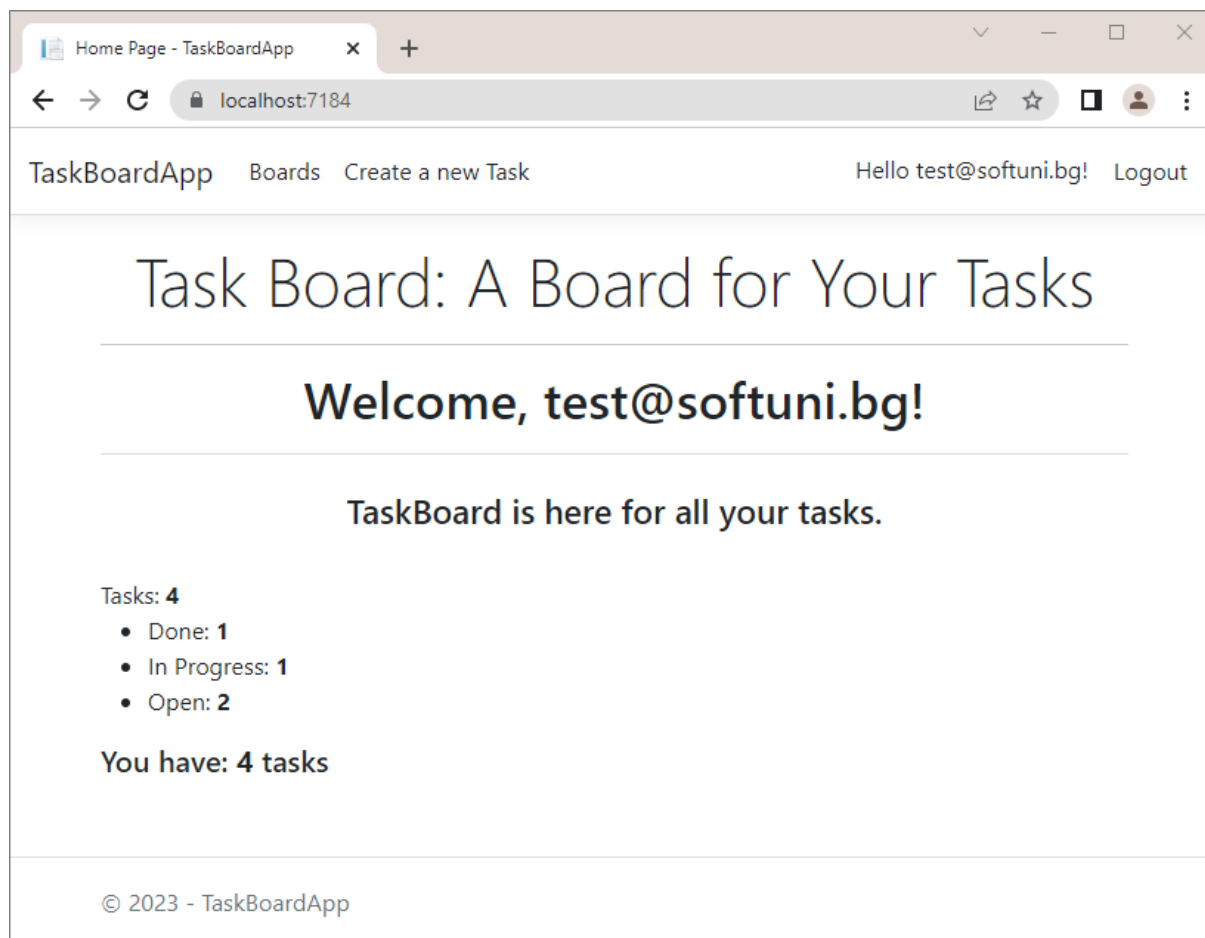
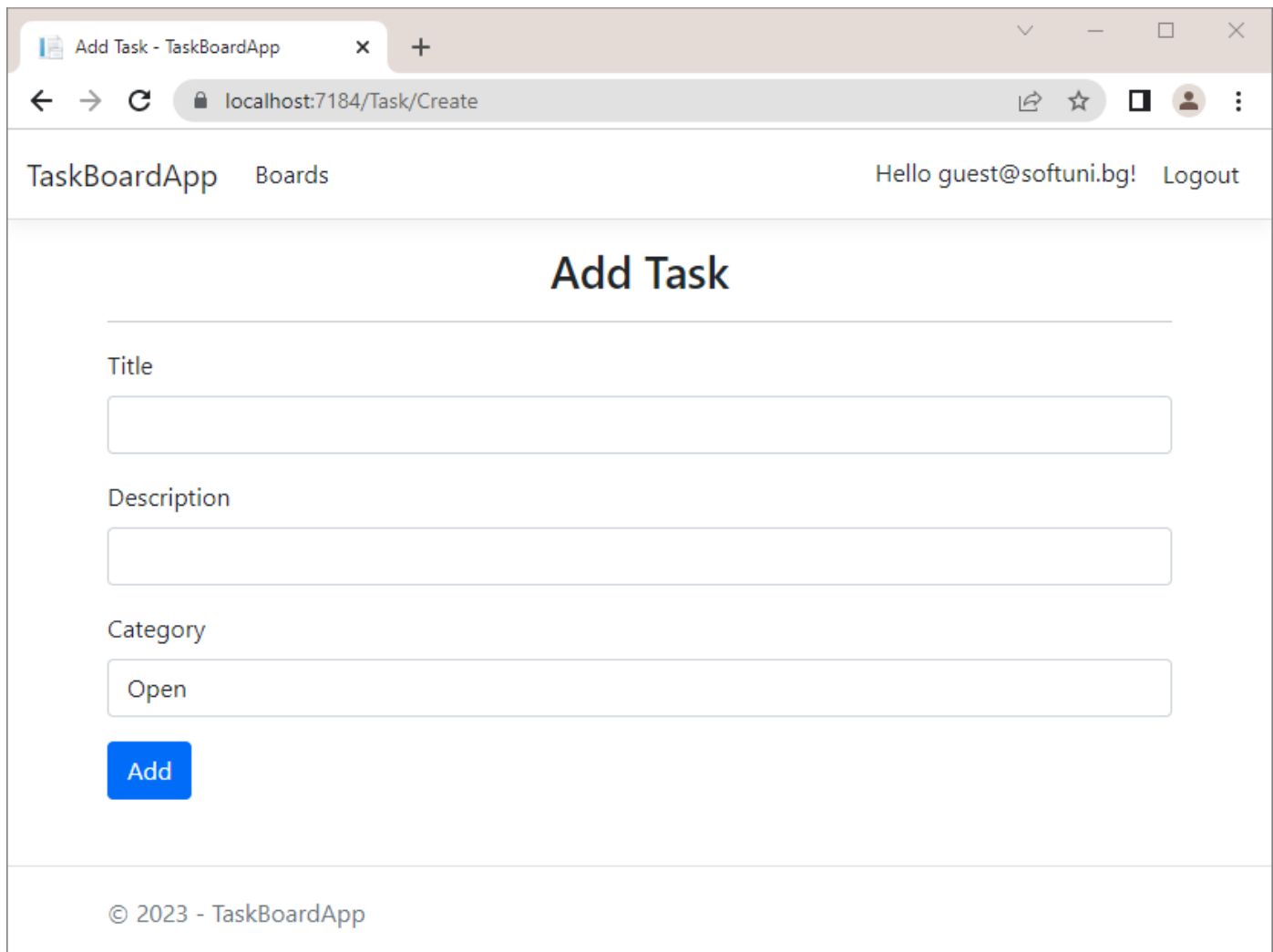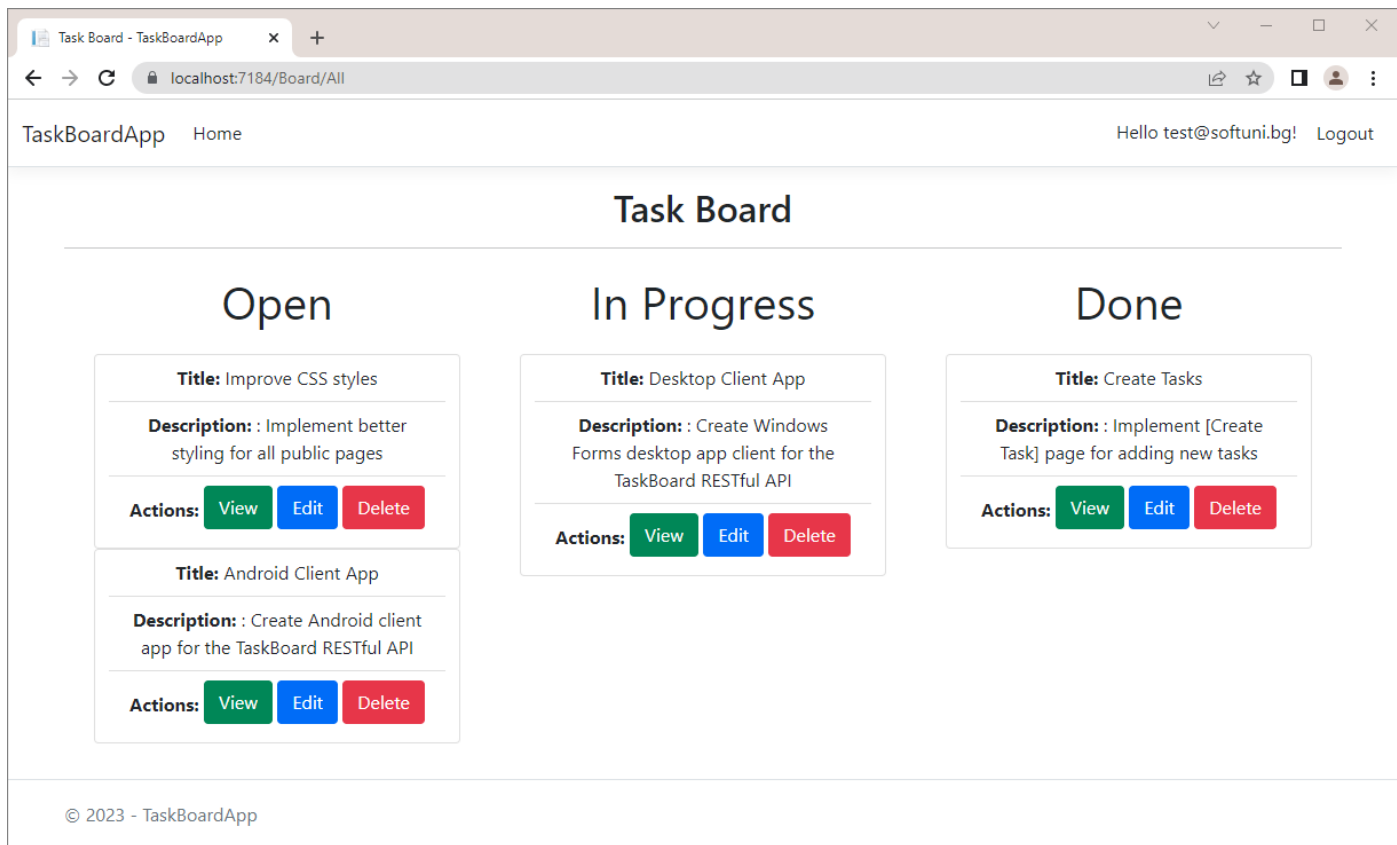Workshop for the "ASP.NET Core Fundamentals" course @ SoftUni

In this workshop we will create the **TaskBoardApp** – a Trello style app, that will hold a board of tasks. Each task will consist of a **title** and a **description**. The **tasks** will be **organized** in **boards**, which will be displayed as **columns** (sections): **Open**, **In Progress** and **Done**.
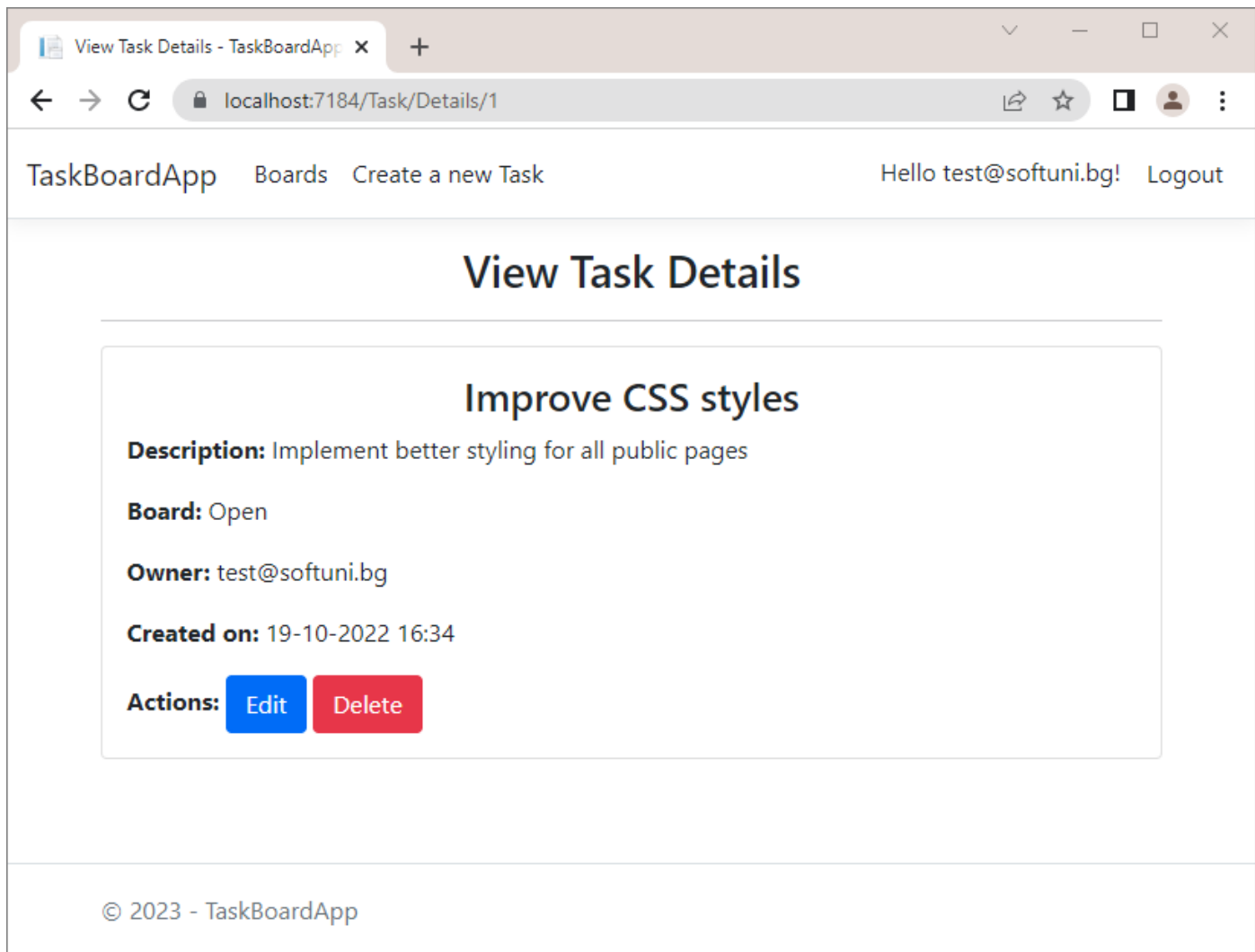
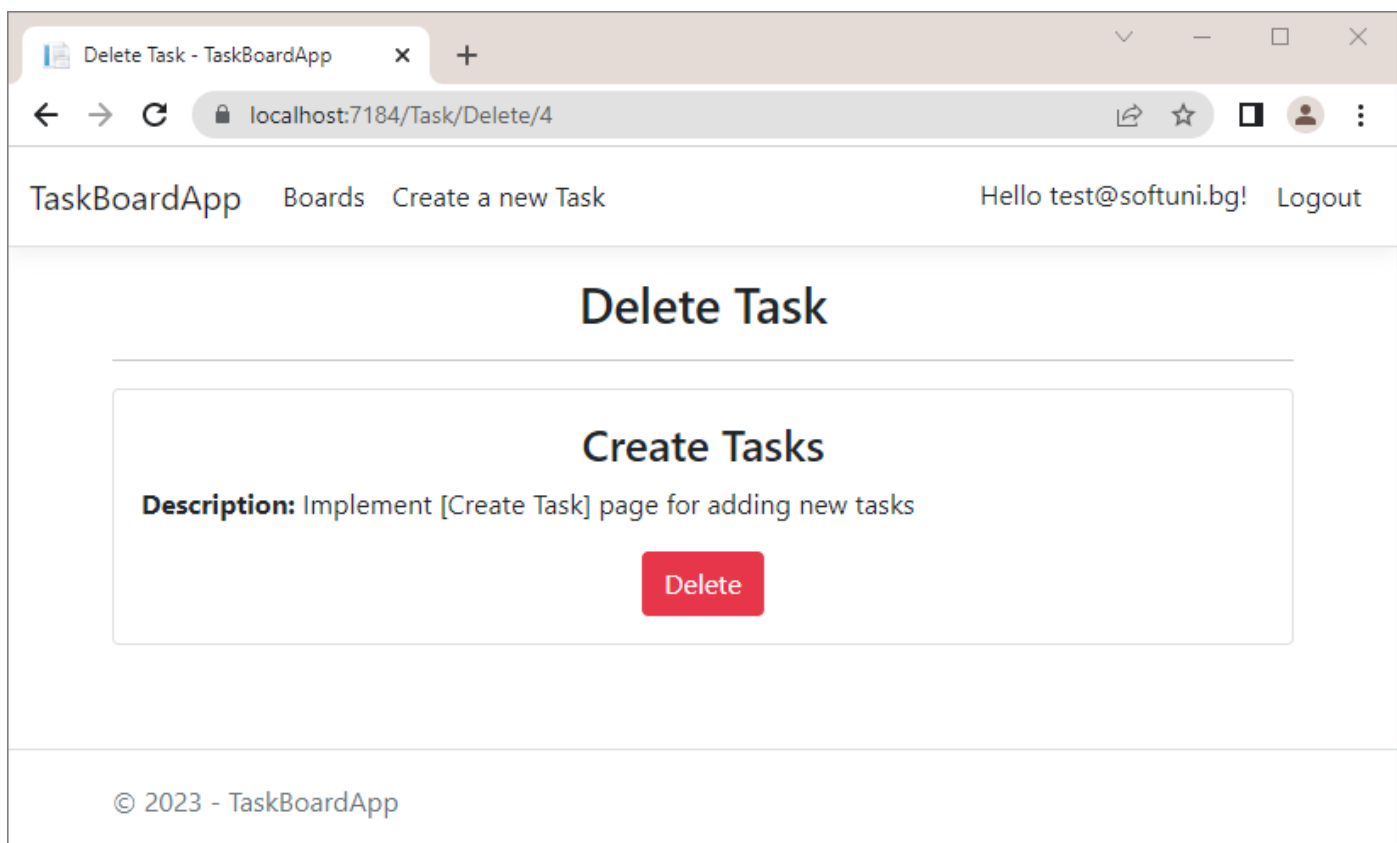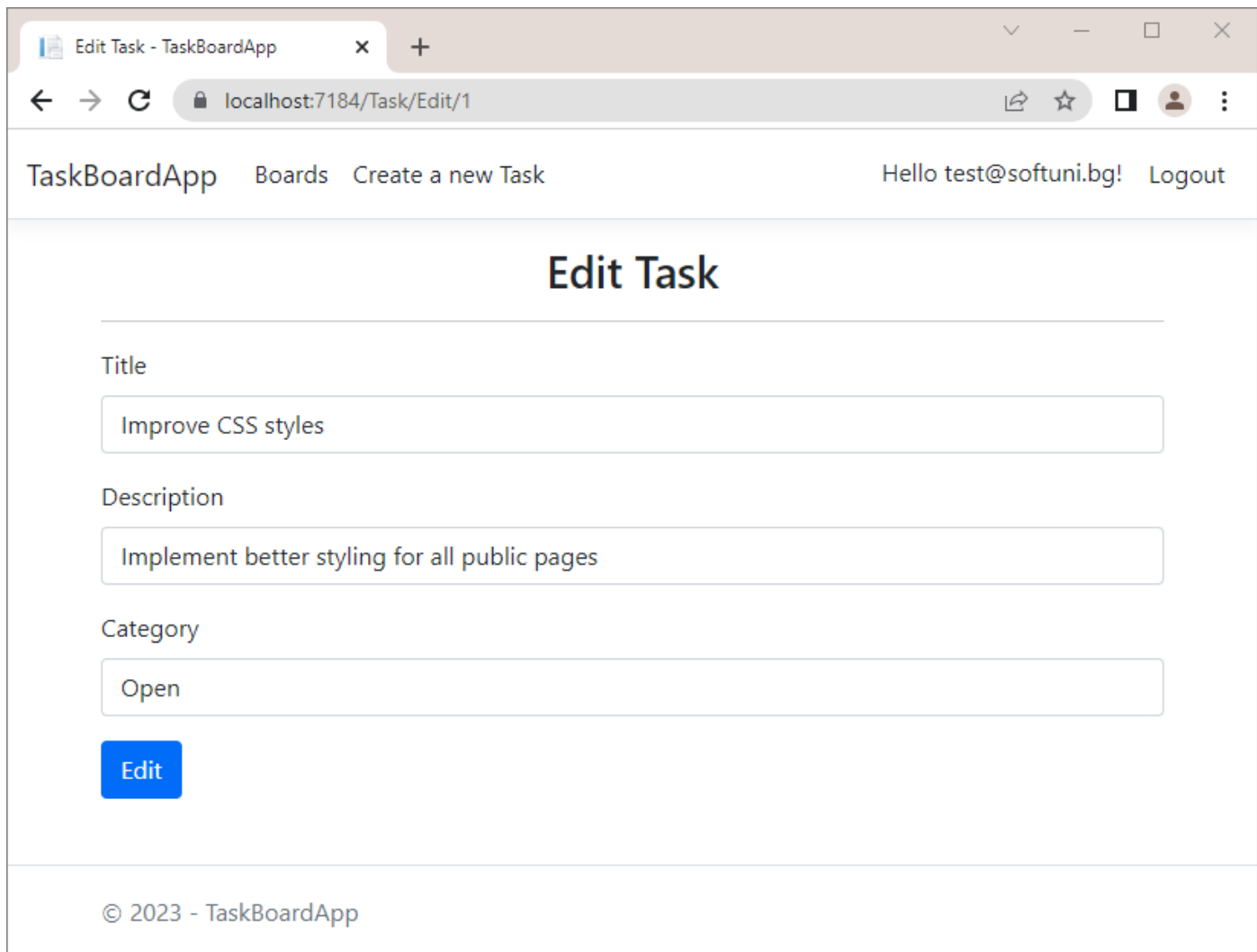The app will support the following operations:

- **Home page**: `/`
- **View** the boards with tasks: `/Boards`
- **View** task details (by id): `/Tasks/Details/:id`
- **Add** new task (title + description + board): `/Tasks/Create`
- **Edit** task / move to board: `/Tasks/Edit/:id`
- **Delete** task: `/Tasks/Delete/:id`

Here is how each page in our app will look like:

TaskBoardApp   Home   Hello test@softuni.bg!   Logout

Task Board

Open

Title: Improve CSS styles

Description: : Implement better styling for all public pages

Actions: View Edit Delete

Title: Android Client App

Description: : Create Android client app for the TaskBoard RESTful API

Actions: View Edit Delete

In Progress

Title: Desktop Client App

Description: : Create Windows Forms desktop app client for the TaskBoard RESTful API

Actions: View Edit Delete

Done

Title: Create Tasks

Description: : Implement [Create Task] page for adding new tasks

Actions: View Edit Delete

© 2023 - TaskBoardApp

TaskBoardApp   Boards   Hello guest@softuni.bg!   Logout

Add Task

Title

Description

Category

Open

Add

© 2023 - TaskBoardApp

TaskBoardApp   Boards   Create a new Task   Hello test@softuni.bg!   Logout

View Task Details

Improve CSS styles

Description: Implement better styling for all public pages

Board: Open

Owner: test@softuni.bg

Created on: 19-10-2022 16:34

Actions: Edit Delete

© 2023 - TaskBoardApp

Edit Task

Title

Improve CSS styles

Description

Implement better styling for all public pages

Category

Open

Edit

© 2023 - TaskBoardApp

Delete Task

Create Tasks

Description: Implement [Create Task] page for adding new tasks
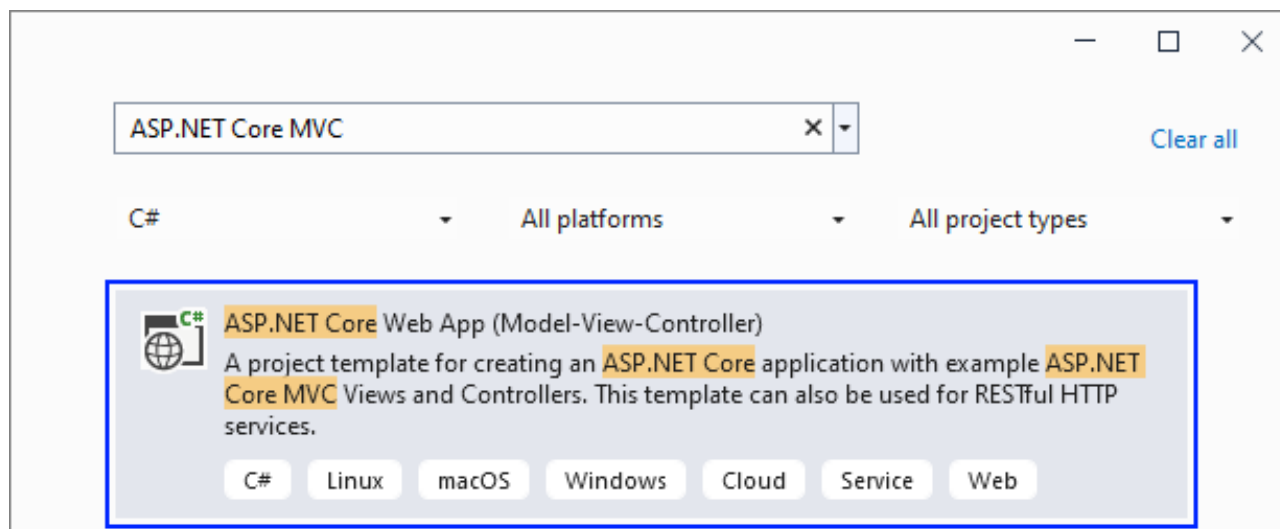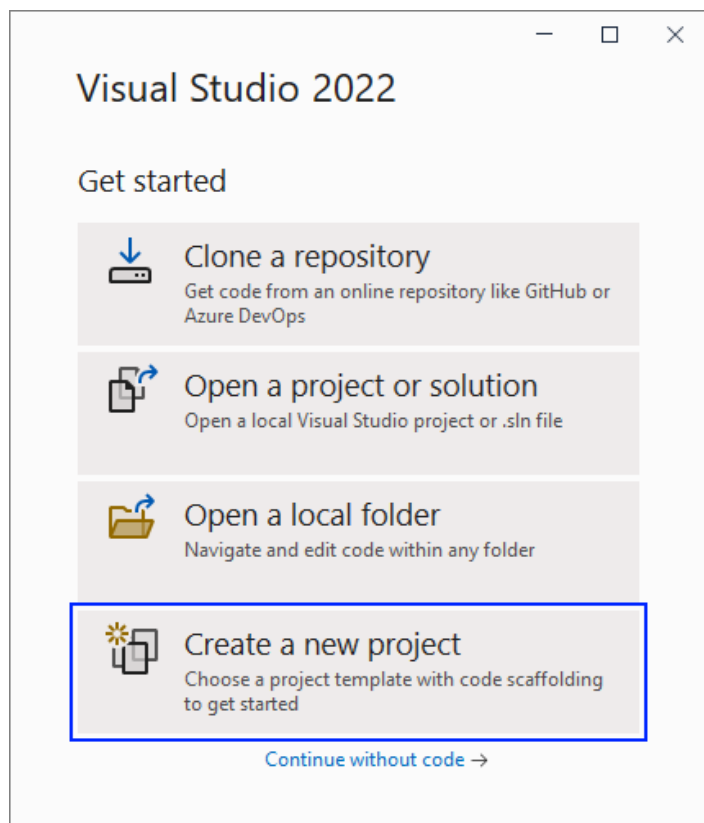
Delete

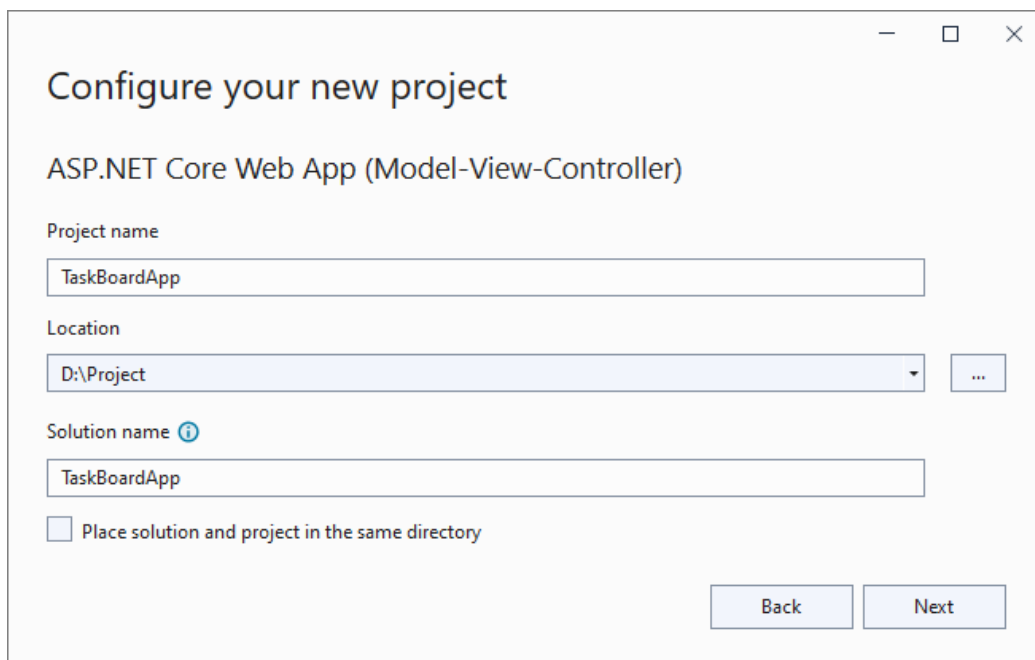© 2023 - TaskBoardApp

SoftUni

1. Create New Project

First, we need to create an **ASP.NET Core MVC application** in **Visual Studio**. Open VS and follow the steps to **create** the app. The app name should be `TaskBoardApp`.
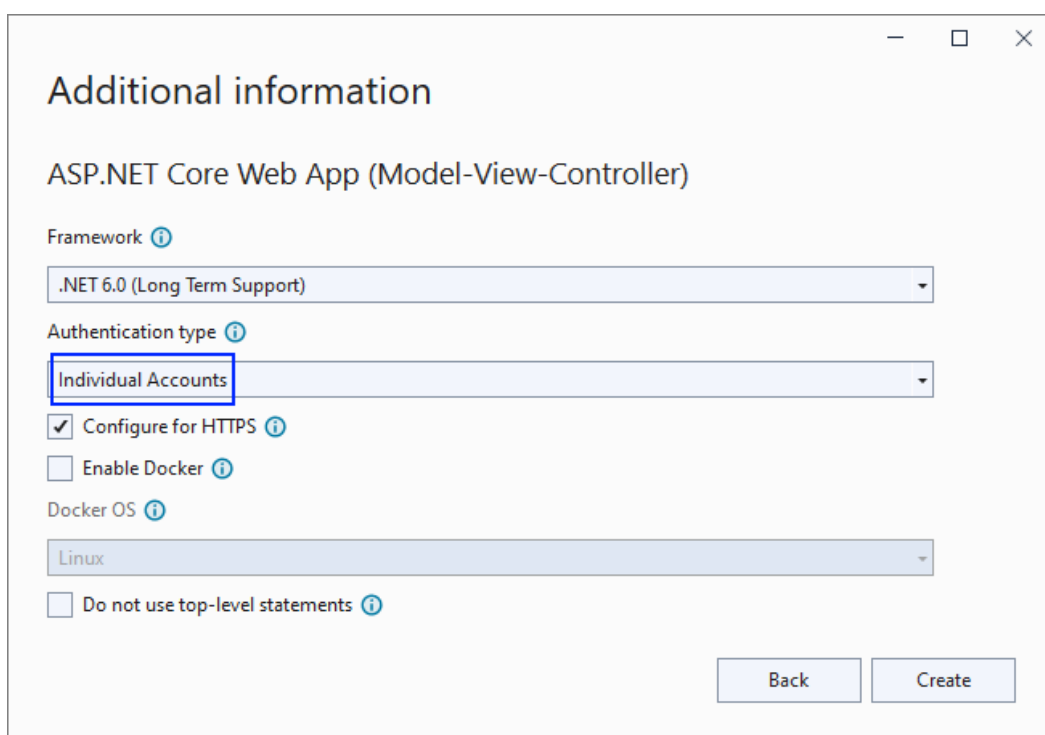
Set up the app with "**Individual Accounts**" **Authentication type**, as we want to have "**Register**" and "**Login**" functionalities.

2. Examine the App and Clean the Project

Examine the App

As we chose the **Individual Accounts** as **Authentication type** upon creating our project, we see that there are some differences in the project from the other project templates that we have been using until now:
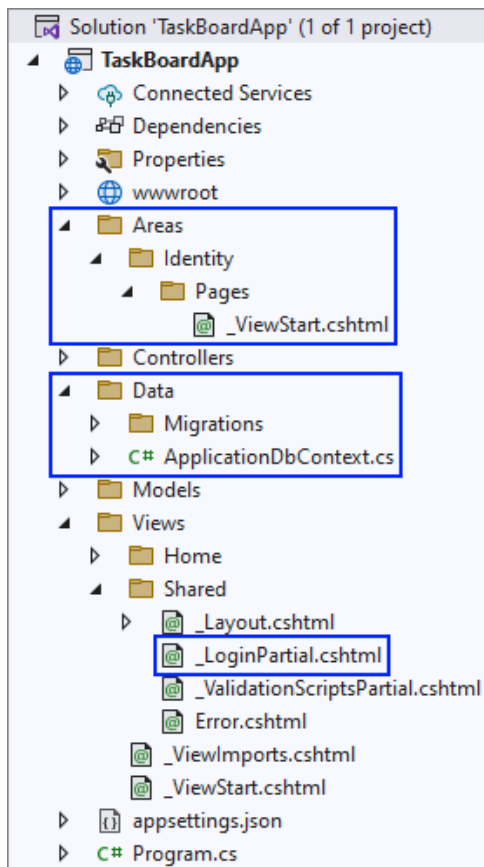
The "**Areas**" folder will hold the template **files** that will be generated after we **scaffold Identity**.

The "**Data**" folder is automatically added. It contains the **db context** for our app and a "**Migrations**" folder. It will also hold the **data models** that we will create during this workshop, as well as the **DataConstants** class

There is another **shared view** in the "**/Views/Shared**" folder, called **_LoginPartial.cshtml**, which holds the **partial view** for the **user navigation**, that **displays** the **Login**, **Logout** and **Register** buttons.

Run the created app in the **browser**. The "**Home**" **page** looks like this:
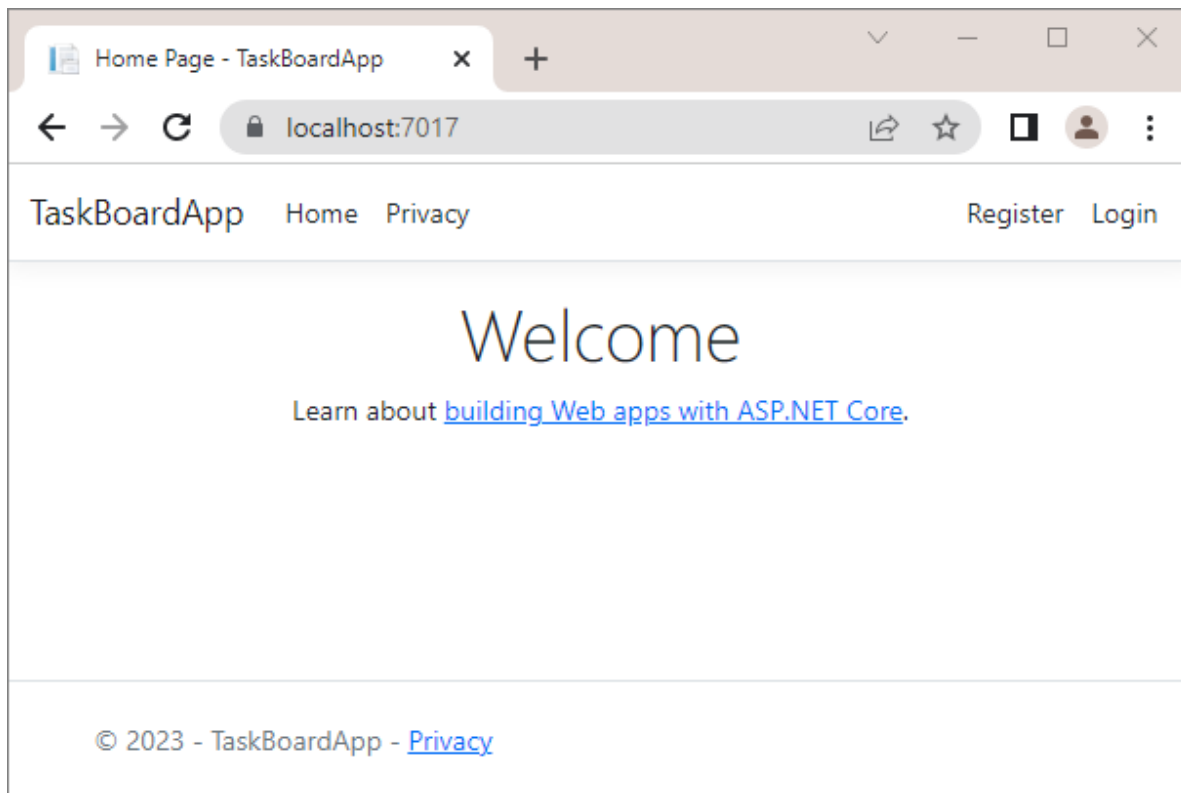
The difference between the projects from the previous sessions and this one is that our project now has "**Register**" and "**Login**" pages. They come from the "**Individual Accounts**" functionality that we added to the app.

The "**Register**" **page** looks like shown below but the **register functionality is still not working**:
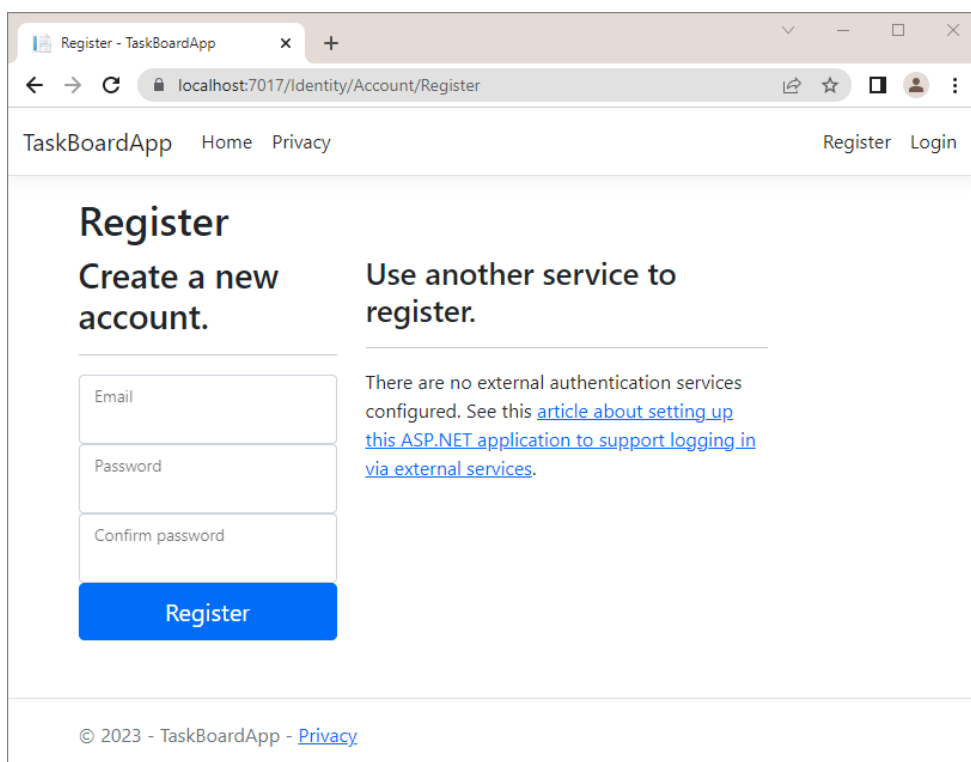
The "**Login**" **page** looks like shown below but the **register functionality is also still not working**.

Now, let's take a look at the **Program** class. In the previous workshop we had to add the following code by hand, but because of the type of the project template that we're using, now it is added automatically:

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Let's take a look at the "**appsettings.json**" file. As you can see, the file already has the connection string. This also comes from the project template that we chose.

```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-TaskBoardApp-afc5a4d9-147f-4be0-9f2d-66012864af65;Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

Leave it this way **for now**.

Clean the Project

Now, let's examine the **HomeController** class. We'll be needing only the **Index()** action, so we can delete the unnecessary code. The class should look like this:

```
public class HomeController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }
}
```

As we deleted the **Privacy()** action method from the **HomeController**, it's best to delete the view from our project. Find it in the "**/Views/Home**" **folder** and **delete it**:
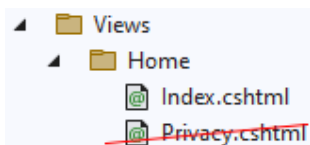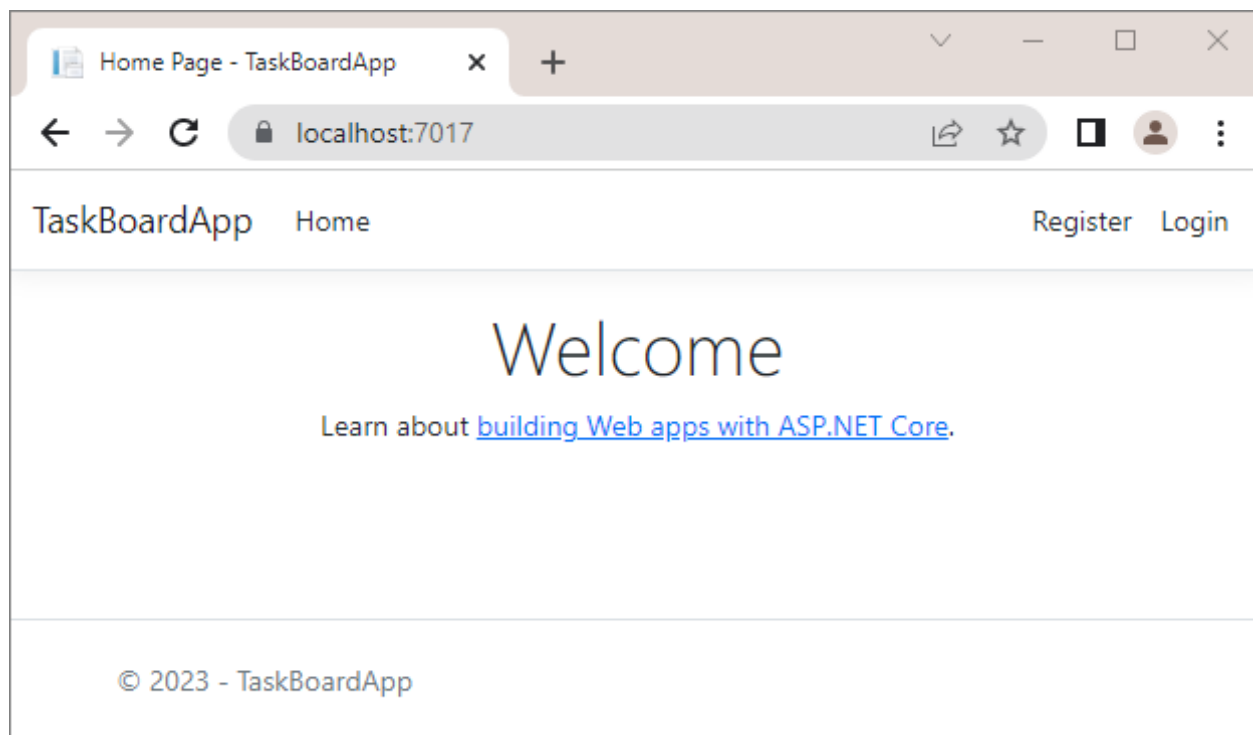

Let's modify the **_Layout.cshtml** file and remove the **Privacy** links as we already deleted the view and the action. We should also remove **2023** and type **@DateTime.Now.Year** instead, so that we see the current year in the browser, whenever we run the app.

```html
<ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
    </li>
</ul>

<footer class="border-top footer text-muted">
    <div class="container">
        &copy; @DateTime.Now.Year - TaskBoardApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
    </div>
</footer>
```

Now when we run our app, it should look like this:

3. Implement Register/Login/Logout

Scaffold Identity

To change the "**Login**" and "**Register**" **pages** and their logic, we should first **access their source code**. To do this, we should **scaffold the Identity pages**, which means to generate the pages code.

The **scaffolded pages** will be part of the "**/Areas/Identity**" **folder**. To scaffold, **right-click** on the "**Identity**" **folder** and choose **[Add]** → **[New Scaffolded Item…]**:
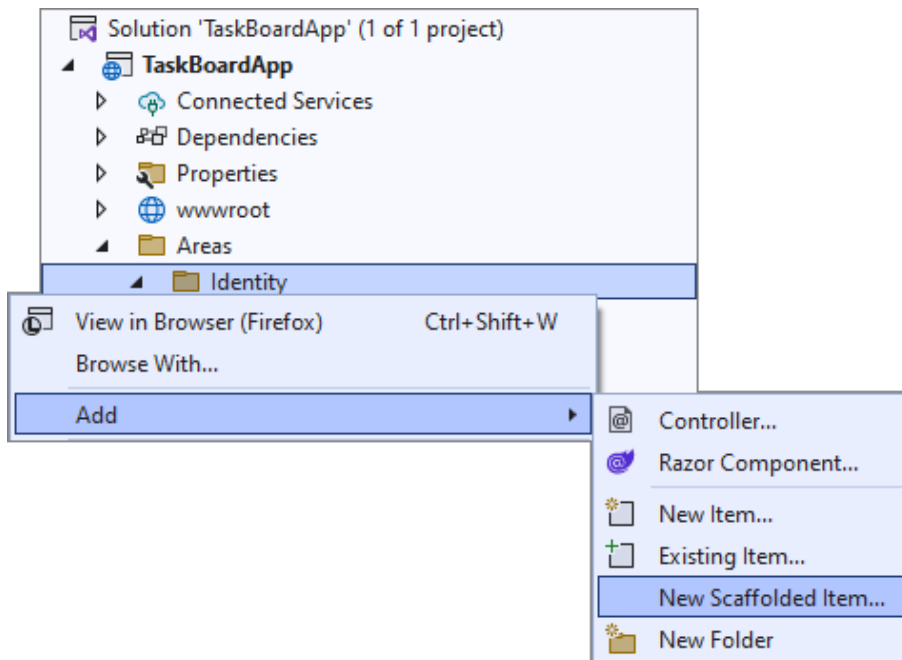
On the next step, go to the **[Identity] tab** and **choose its only option**:
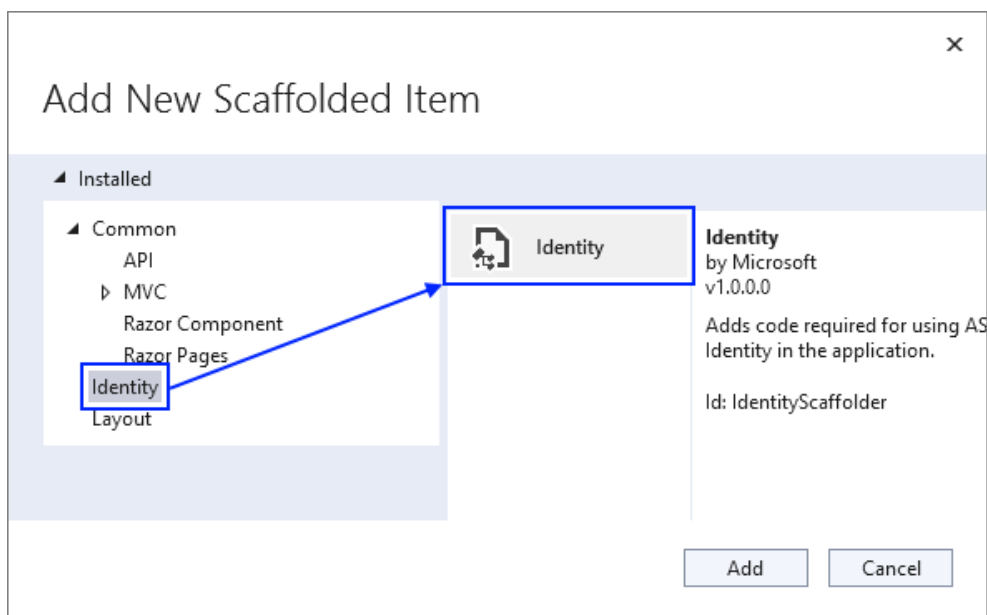
Then, on the "**Add Identity**" **window** you should set the **_Layout.cshtml** as a **layout page**, check the pages to be **scaffolded** ("**Login**", "**Register**" and "**Logout**") and **select the db context class** of our app. Do it like this:

Click the **[Add] button** and examine the **scaffolded pages** in the "**/Areas/Identity**" **folder**:
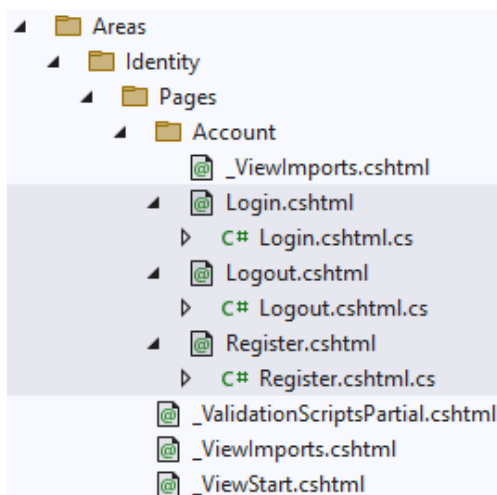
Note that the **generated pages** are **Razor pages**. They have two files – one with extension **.cshtml** and one with **.cshtml.cs**. The **Login.cshtml**, **Logout.cshtml** and **Register.cshtml files** are **Razor pages** and the logic behind them is in the **Login.cshtml.cs**, **Logout.cshtml.cs** and **Register.cshtml.cs files**. The **LoginModel, LogoutModel** and **RegisterModel** classes hold the **logic** behind the pages. They have **OnGetAsync(…)** and **OnPostAsync(…) methods**, which are responsible for **handling requests** to the page.

You can now **clear some generated files** and modify others. First, we can **move the namespaces** from the **_ViewImports.cshtml file** in the "**/Areas/Identity/Pages**" **folder** to the **_ViewImports.cshtml file** in the "**/Areas/Identity/Pages/Account**" **folder** and delete the unnecessary file.

The left classes should be the following:

You can also **delete** the `ScaffoldingReadMe.txt` **file** from the solution, as we don't need it:

Apply migrations

When we were examining the project structure, you may have noticed that in the "**Data**" folder was another one, called "**Migrations**", and there was a file called **00000000000000_CreateIdentitySchema.cs**. We will now **apply to the database the migration** for the identity schema. Before that, however, let's give the **database a good name**.

To do that, first we need to modify the connection string in the "**appsettings.json**" file. Edit your **Server** and modify the **Database** name to be **TaskBoardApp**:

```
"ConnectionStrings": {
  "DefaultConnection": "Server=        \\SQLEXPRESS;Database=TaskBoardApp Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

Now, let's open the **Package Manager Console** and type **Update-Database**.

Open **Server Management Studio** and examine the database **tables** – we can examine **all** of the **tables** that come with **Identity**:

- TaskBoardApp
  - Database Diagrams
  - Tables
    - System Tables
    - FileTables
    - External Tables
    - Graph Tables
    - dbo.__EFMigrationsHistory
    - dbo.AspNetRoleClaims
    - dbo.AspNetRoles
    - dbo.AspNetUserClaims
    - dbo.AspNetUserLogins
    - dbo.AspNetUserRoles
    - dbo.AspNetUsers
    - dbo.AspNetUserTokens
  - Views
  - External Resources
  - Synonyms
  - Programmability
  - Service Broker
  - Storage
  - Security
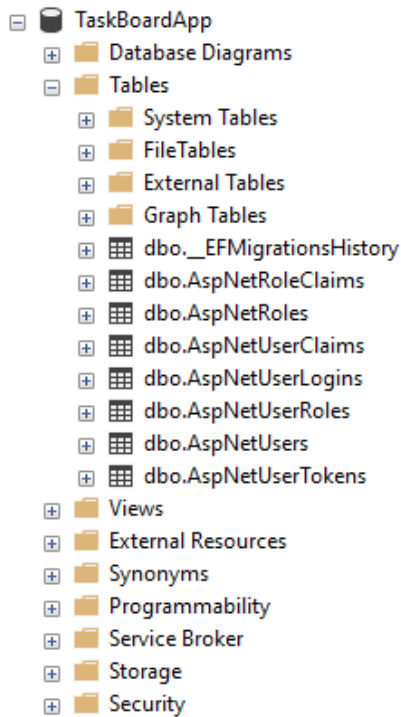
Modify the "Register" Page

Now we want to **modify our "Register" page**. It should **not have external logins**, but should have **fields for email** and **password**. Note that the generated templates treat username and email the same way. For now we will work with email only and in the next course we'll learn how to create and modify our own users.

The "**Register**" page should look like this:

Go to the **Register.cshtml file** to **clear the unnecessary view code**. We want **to remove the section for registering with an external provider**. The "**Register.cshtml**" file should look like show below. Also, if you want your form to be on the **center of the screen**, as it looks better, add the following **CSS classes**:

```
Register.cshtml
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1 class="text-center">@ViewData["Title"]</h1>

<div class="row d-flex align-items-center justify-content-center">
    <div class="col-md-4">
        <form id="registerForm" asp-route-returnUrl="@Model.ReturnUrl" method="post">
            <h2 class="text-center">Create a new account.</h2>
            <hr />
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-floating">
                <input asp-for="Input.Email" class="form-control" autocomplete="username" aria-required="true" />
                <label asp-for="Input.Email"></label>
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.Password" class="form-control" autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.Password"></label>
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.ConfirmPassword" class="form-control" autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.ConfirmPassword"></label>
                <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
            </div>
            <button id="registerSubmit" type="submit" class="w-100 btn btn-lg btn-primary">Register</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Now it is time to **clear** the **RegisterModel class** from things we won't use like **external logins**, **email sender**, etc. Your class should look like this:

```csharp
[AllowAnonymous]
// 6 references
public class RegisterModel : PageModel
{
    private readonly SignInManager<IdentityUser> _signInManager;
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IUserStore<IdentityUser> _userStore;

    // 0 references
    public RegisterModel(
        UserManager<IdentityUser> userManager,
        IUserStore<IdentityUser> userStore,
        SignInManager<IdentityUser> signInManager)
    {
        _userManager = userManager;
        _userStore = userStore;
        _signInManager = signInManager;
    }

    [BindProperty]
    // 11 references
    public InputModel Input { get; set; }

    // 2 references
    public string ReturnUrl { get; set; }

    public class InputModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        // 4 references
        public string Email { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
            MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        // 4 references
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        // 3 references
        public string ConfirmPassword { get; set; }
    }

    public async Task OnGetAsync(string returnUrl = null)
    {
        ReturnUrl = returnUrl;
    }
```

```csharp
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email, CancellationToken.None);

        var result = await _userManager.CreateAsync(user, Input.Password);

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);

        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    return Page();
}

private IdentityUser CreateUser()
{
    try
    {
        return Activator.CreateInstance<IdentityUser>();
    }
    catch
    {
        throw new InvalidOperationException($"Can't create an instance of '{nameof(IdentityUser)}'. " +
            $"Ensure that '{nameof(IdentityUser)}' is not an abstract class and has a parameterless constructor, or alternatively " +
            $"override the register page in /Areas/Identity/Pages/Account/Register.cshtml");
    }
}
```

Now **open** the "`Register`" **page** in the browser. It should look as shown on the beginning of the task. Try to **register a new user**. You should see the following error messages:
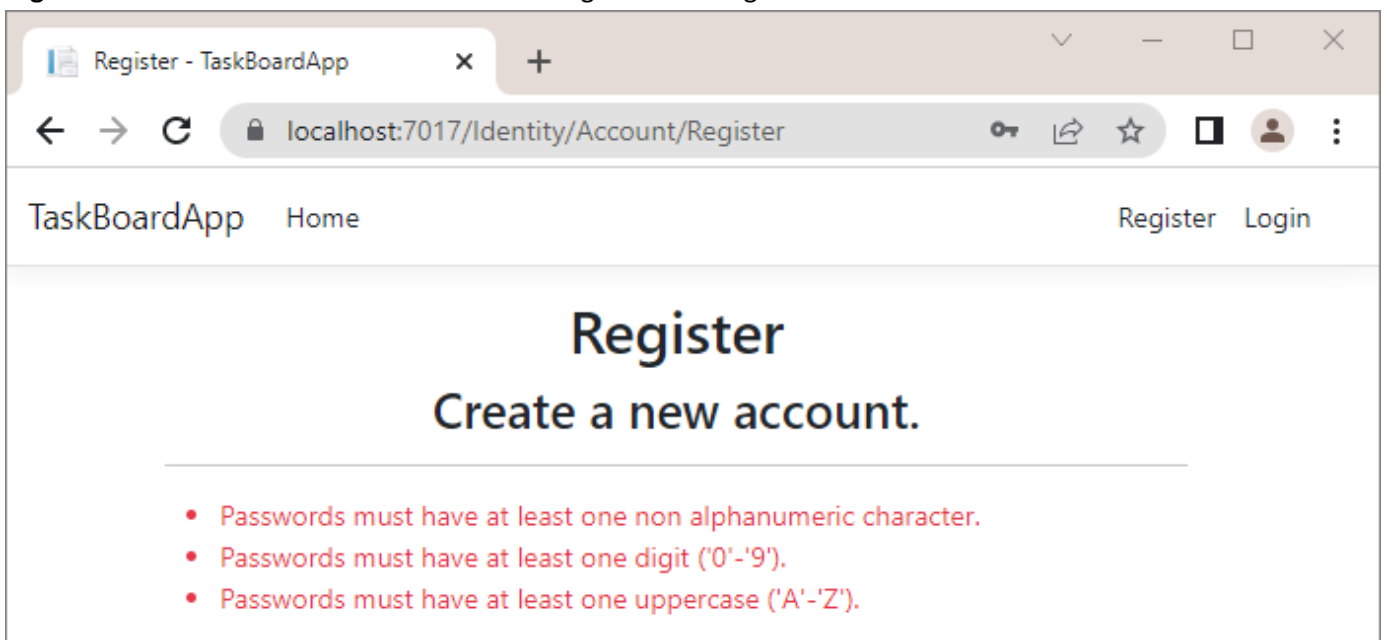
This is because our app is using the **default** password requirements, which are too strict. We can change them by adding the following lines to the **Program** class:

```
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = false;
    options.Password.RequireDigit = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = false;
    options.Password.RequireLowercase = false;
})
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Let's try again to register a new user. The registration should be **successful** and the user should **appear in the database**.

	Id	UserName	NormalizedUserName	Email	NormalizedEmail
1	69008683-4654-4fc1-bdb9-f3a30984d94f	guest@softuni.bg	GUEST@SOFTUNI.BG	NULL	NULL

Notice that our user has a **UserName** and a **NormalizedUserName**, but doesn't have an **Email** and **NormalizedEmail**, even though we have an **email input field** in the "**Register**" page. The reason for this is that as we mentioned before, **Identity** treats username and email **the same way**. Take a look at the following lines of code:

```
var user = CreateUser();

await _userStore.SetUserNameAsync(user, Input.Email, CancellationToken.None);
```

First, **Identity** creates a user with the **CreateUser** method and then with the **SetUserNameAsync** method uses the **email**, used upon **registration**, to modify the **UserName** of the user.

Modify the "Login" Page

It is time to **modify** the "**Login**" page as well to **clear it** from unnecessary code in its generated class. The page should look like this:
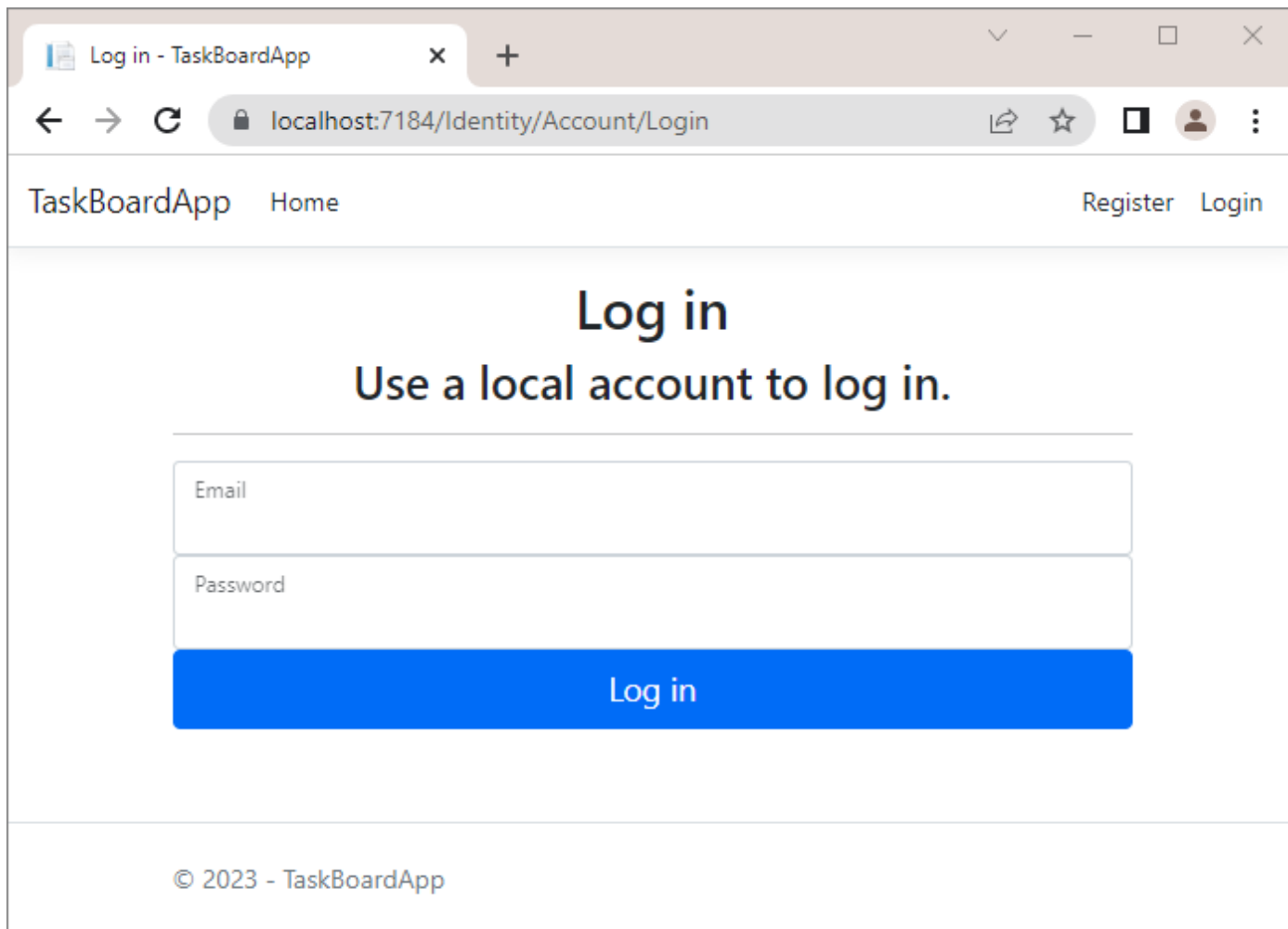
Go to the **Login.cshtml view** and **remove links for email confirmation**, **external login** and **forgotten password**. Also, add the needed classes to **center the page content**. The **Login.cshtml** view should look like this:

```cshtml
Login.cshtml
@page
@model LoginModel

@{
    ViewData["Title"] = "Log in";
}

<h1 class="text-center">@ViewData["Title"]</h1>
<div class="row d-flex align-items-center justify-content-center">
    <div class="col-md-4">
        <section>
            <form id="account" method="post">
                <h2 class="text-center">Use a local account to log in.</h2>
                <hr />
                <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                <div class="form-floating">
                    <input asp-for="Input.Email" class="form-control" autocomplete="username" aria-required="true" />
                    <label asp-for="Input.Email" class="form-label"></label>
                    <span asp-validation-for="Input.Email" class="text-danger"></span>
                </div>
                <div class="form-floating">
                    <input asp-for="Input.Password" class="form-control" autocomplete="current-password" aria-required="true" />
                    <label asp-for="Input.Password" class="form-label"></label>
                    <span asp-validation-for="Input.Password" class="text-danger"></span>
                </div>
                <div>
                    <button id="login-submit" type="submit" class="w-100 btn btn-lg btn-primary">Log in</button>
                </div>
            </form>
        </section>
    </div>
</div>
```

Follow us:

Now **modify the LoginModel class** in the `Login.cshtml.cs file` as shown below:

```csharp
[AllowAnonymous]
6 references
public class LoginModel : PageModel
{
    private readonly SignInManager<IdentityUser> _signInManager;

    0 references
    public LoginModel(SignInManager<IdentityUser> signInManager)
    {
        _signInManager = signInManager;
    }

    [BindProperty]
    8 references
    public InputModel Input { get; set; }

    1 reference
    public string ReturnUrl { get; set; }

    [TempData]
    2 references
    public string ErrorMessage { get; set; }

    public class InputModel
    {
        [Required]
        [EmailAddress]
        4 references
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        4 references
        public string Password { get; set; }
    }

    public async Task OnGetAsync(string returnUrl = null)
    {
        if (!string.IsNullOrEmpty(ErrorMessage))
        {
            ModelState.AddModelError(string.Empty, ErrorMessage);
        }

        returnUrl ??= Url.Content("~/");

        // Clear the existing external cookie to ensure a clean login process
        await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);

        ReturnUrl = returnUrl;
    }
```

```csharp
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");


    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(
            Input.Email,
            Input.Password,
            true,
            lockoutOnFailure: false);

        if (result.Succeeded)
        {
            return LocalRedirect(returnUrl);
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    return Page();
}
```

Try to log in with the **new user** we created. Login should be **successful**.

Modify the "Logout" Page

Now, if you try to logout of the app, you should logout successfully. This is because we **scaffolded** Indentity and the logout logic comes with it. However, let's clean the unnecessary code from the `Logout.cshtml.cs` file. It should look like this:

```csharp
public class LogoutModel : PageModel
{
    private readonly SignInManager<IdentityUser> _signInManager;

    0 references
    public LogoutModel(SignInManager<IdentityUser> signInManager)
    {
        _signInManager = signInManager;
    }

    0 references
    public async Task<IActionResult> OnPost(string returnUrl = null)
    {
        await _signInManager.SignOutAsync();
        if (returnUrl != null)
        {
            return LocalRedirect(returnUrl);
        }
        else
        {
            return RedirectToPage();
        }
    }
}
```

4. Define Entities

We should start with creating the data models, which we will need for our database. We will have two data model clasess – **Task** and **Board**.

Before that, let's create a **DataConstants** class in the "**Data**" folder. The **DataConstants** class will hold the constants for the required lengths of the models' properties.

```
◢  📁 Data
    ▷  📁 Migrations
    ▷  C# ApplicationDbContext.cs
    ▷  C# DataConstants.cs
```
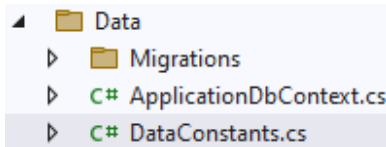
The "Task" Entity Class

The **Task** class should have the following **properties**:

- **Id** – a unique **integer**, Primary Key
- **Title** – a **string** with min length **5** and max length **70** (**required**)
- **Description** – a **string** with min length **10** and max length **1000** (**required**)
- **CreatedOn** – date and time
- **BoardId** – an **integer**
- **Board** – a **Board** object
- **OwnerId** – an **integer** (**required**)
- **Owner** – an **IdentityUser** object

First, create a new folder, called "**Models**" in the "**Data**" folder:

```
◢  📁 Data
    ▷  📁 Migrations
       📁 Models
    ▷  C# ApplicationDbContext.cs
    ▷  C# DataConstants.cs
```
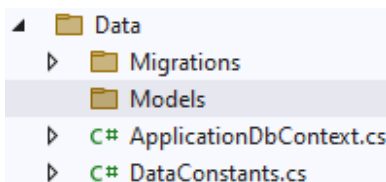
Then, create constants for the **Title** and **Description** properties length in the **DataConstants** class:

```csharp
public class DataConstants
{
    0 references
    public class Task
    {
        public const int TaskMaxTitle = 70;
        public const int TaskMinTitle = 5;

        public const int TaskMaxDescription = 1000;
        public const int TaskMinDescription = 10;
    }
}
```

Then, create a **Task** class in the "**/Data/Models**" folder and use them in the **Task.cs** file and create its **properties** as shown below:

```csharp
public class Task
{
    0 references
    public int Id { get; set; }

    [Required]
    [MaxLength(TaskMaxTitle)]
    0 references
    public string Title { get; set; } = null!;

    [Required]
    [MaxLength(TaskMaxDescription)]
    0 references
    public string Description { get; set; } = null!;

    0 references
    public DateTime CreatedOn { get; set; }

    0 references
    public int BoardId { get; set; }

    0 references
    public Board? Board { get; set; }

    [Required]
    0 references
    public string OwnerId { get; set; } = null!;

    0 references
    public IdentityUser User { get; set; } = null!;
}
```

> ⚠️ Note that the **OwnerId property** is of type **string** and has the **[Required] attribute**, as the **IdentityUser class** has a **string** for **Id**.

The "Board" Entity Class

The **Board** class should have the following properties:

- **Id** – a unique **integer**, Primary Key
- **Name** – a **string** with min length **3** and max length **30** (**required**)
- **Tasks** – a collection of **Task**

First, create constants for the **Name** property length in the **DataConstants** class:

```csharp
public class Board
{
    public const int BoardMaxName = 30;
    public const int BoardMinName = 3;
}
```

Then, use the constant in the **Board** class and create its **properties** as shown below. Don't forget that you should **initialize** the **Tasks property** either **inline** or in a **constructor** (use only one of the ways for your whole app):

```csharp
public class Board
{
    0 references
    public int Id { get; init; }

    [Required]
    [MaxLength(BoardMaxName)]
    0 references
    public string Name { get; init; } = null!;

    0 references
    public IEnumerable<Task> Tasks { get; set; }
        = new List<Task>();
}
```

Note that you don't need to **set** the **Id property** as a **primary key** in any way – this is done **by default** in **EF Core** because of the "**Id**" **property name**. Also, that's why we don't need to **set the BoardId** and **OwnerId properties** of the **Task** class as **foreign keys**, as we have **named the properties by the convention** "**{className}Id**" ("**Board**" + "**Id**" and "**Owner**" + "**Id**").

As you now have **all entity classes** your app needs, use them for the database.

5. Define the DbContext

To start with, it is a good idea to **rename** the **ApplicationDbContext class** to **TaskBoardAppDbContext**, so that it is relevant to the idea of our application:
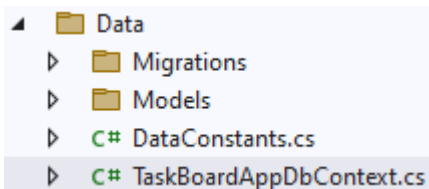
```
▲  📁 Data
   ▷  📁 Migrations
   ▷  📁 Models
   ▷  C# DataConstants.cs
   ▷  C# TaskBoardAppDbContext.cs
```

```csharp
public class TaskBoardAppDbContext : IdentityDbContext<IdentityUser>
{
    0 references
    public TaskBoardAppDbContext(DbContextOptions<TaskBoardAppDbContext> options)
        : base(options)
    {
    }
}
```

Note that our **TaskBoardAppDbContext** inherits **IdentityDbContext**.

We should use the **Migrate()** method in the constructor in order for the changes to be applied to the database directly.

```csharp
public class TaskBoardAppDbContext : IdentityDbContext<IdentityUser>
{
    0 references
    public TaskBoardAppDbContext(DbContextOptions<TaskBoardAppDbContext> options)
        : base(options)
    {
        Database.Migrate();
    }
}
```

Create the **DbSet** properties for the tables in the database:

```csharp
public DbSet<Board> Boards { get; set; }
    0 references
public DbSet<Task> Tasks { get; set; }
```

As the **Task** is an ambigious reference between our **Task** class and the **System.Threading.Tasks.Task** class, we should define which **Task** class we are using:

```csharp
using Task = TaskBoardApp.Data.Models.Task;
```

> ⚠️ **EF Core** uses a "**cascade**" **delete** by default when **removing an entity**. This means that if a record in the **parent table is deleted**, then the **corresponding records** in the child table **will automatically be deleted**. To prevent this from happening, it's a good practice to set the **delete behavior** to "**restrict**", so that an entity, which has **connections** to other entities in the database, **won't be deleted.**

To do this, we should **override** the **OnModelCreating(ModelBuilder builder) method** in the **TaskBoardAppDbContext class**:

```csharp
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
}
```

Then, we should set the foreign key relations and change the delete behaviour.

At the end, invoke the **base OnModelCreating() method**:

```csharp
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
    .Entity<Task>()
    .HasOne(t => t.Board)
    .WithMany(b => b.Tasks)
    .HasForeignKey(t => t.BoardId)
    .OnDelete(DeleteBehavior.Restrict);

    base.OnModelCreating(modelBuilder);
}
```

Now our **database structure** is ready. If you migrate it now, however, it will be **created with empty tables**. For this reason, let's see how to **seed some data to fill in the database tables**.

6. Seed the Database

Now, we want to **populate the database** with an **initial set of data**. This will include **four events, three boards (OpenBoard, InProgressBoard** and **DoneBoard)** and **one user.**

First, create properties for the above objects in the **TaskBoardAppDbContext** class:

```csharp
private IdentityUser TestUser { get; set; }
    2 references
private Board OpenBoard { get; set; }
    1 reference
private Board InProgressBoard { get; set; }
    1 reference
private Board DoneBoard { get; set; }
```

Then, we will use **separate methods** to **add data to these objects**, which will be **added** to the **corresponding database tables** in the **OnModelCreating(ModelBuilder builder) method**. Add the following lines of code to the method, before invoking the base one:

```csharp
        SeedUsers();
        modelBuilder
            .Entity<IdentityUser>()
                .HasData(TestUser);

        SeedBoards();
        modelBuilder
            .Entity<Board>()
            .HasData(OpenBoard, InProgressBoard, DoneBoard);


        modelBuilder
            .Entity<Task>()
            .HasData(new Task()...,
            new Task()...,
            new Task()...,
            new Task()...);


        base.OnModelCreating(modelBuilder);
    }
}
```

> ⚠️ When **invoking more that one seeding method**, it is important that the **seeding methods are invoked in the correct order**, as they **depend** on each other.

```
modelBuilder
    .Entity<Task>()
    .HasData(new Task()
    {
        Id = 1,
        Title = "Improve CSS styles",
        Description = "Implement better styling for all public pages",
        CreatedOn = DateTime.Now.AddDays(-200),
        OwnerId = TestUser.Id,
        BoardId = OpenBoard.Id
    },
    new Task()
    {
        Id = 2,
        Title = "Android Client App",
        Description = "Create Android client app for the TaskBoard RESTful API",
        CreatedOn = DateTime.Now.AddMonths(-5),
        OwnerId = TestUser.Id,
        BoardId = OpenBoard.Id
    },
    new Task()
    {
        Id = 3,
        Title = "Desktop Client App",
        Description = "Create Windows Forms desktop app client for the TaskBoard RESTful API",
        CreatedOn = DateTime.Now.AddMonths(-1),
        OwnerId = TestUser.Id,
        BoardId = InProgressBoard.Id
    },
    new Task()
    {
        Id = 4,
        Title = "Create Tasks",
        Description = "Implement [Create Task] page for adding new tasks",
        CreatedOn = DateTime.Now.AddYears(-1),
        OwnerId = TestUser.Id,
        BoardId = DoneBoard.Id
    });
```

Start by implementing the **SeedUsers()** method, which will create the **TestUser** for the database. Note that we are using the **IdentityUser** class, which is the **default** user provided by ASP.NET Core Identity.

First, we will need to **instantiate the PasswordHasher class**, which will **save our user's password as a hash** in the database:

```
private void SeedUsers()
{
    var hasher = new PasswordHasher<IdentityUser>();
```

Then, **create the user** with an **id** (its value does not matter), **username and normalized username,** which will be am email, a**nd** with a **hashed password**. Note that the **normalized username** and **normalized email** should be added or the user will not be able to **log in** the app. Do it like this:

```
TestUser = new IdentityUser()
{
    UserName = "test@softuni.bg",
    NormalizedUserName = "TEST@SOFTUNI.BG",
};
```

Finally, we should turn the password into hashed representation for security reasons.

```
TestUser.PasswordHash = hasher.HashPassword(TestUser, "softuni");
}
```

Follow us:

Next, seed the boards, which will be saved as **OpenBoard**, **InProgressBoard** and **DoneBoard**:

```
private void SeedBoards()
{
    OpenBoard = new Board()
    {
        Id = 1,
        Name = "Open"
    };

    InProgressBoard = new Board()
    {
        Id = 2,
        Name = "In Progress"
    };

    DoneBoard = new Board()
    {
        Id = 3,
        Name = "Done"
    };
}
```

The tasks are already created and added with this code:

```
modelBuilder
    .Entity<Task>()
    .HasData(new Task()
    {
        Id = 1,
        Title = "Improve CSS styles",
        Description = "Implement better styling for all public pages",
        CreatedOn = DateTime.Now.AddDays(-200),
        OwnerId = TestUser.Id,
        BoardId = OpenBoard.Id
    },
    new Task()
    {
        Id = 2,
        Title = "Android Client App",
        Description = "Create Android client app for the TaskBoard RESTful API",
        CreatedOn = DateTime.Now.AddMonths(-5),
        OwnerId = TestUser.Id,
        BoardId = OpenBoard.Id
    },
    new Task()
    {
        Id = 3,
        Title = "Desktop Client App",
        Description = "Create Windows Forms desktop app client for the TaskBoard RESTful API",
        CreatedOn = DateTime.Now.AddMonths(-1),
        OwnerId = TestUser.Id,
        BoardId = InProgressBoard.Id
    },
    new Task()
    {
        Id = 4,
        Title = "Create Tasks",
        Description = "Implement [Create Task] page for adding new tasks",
        CreatedOn = DateTime.Now.AddYears(-1),
        OwnerId = TestUser.Id,
        BoardId = DoneBoard.Id
    });
```

Now we have a **db context** with **seeded data** and our **database is ready to be migrated**.

7. Create a Migration

Now we will now **create a new migration** to the database.

To do that, **open** the `Package Manager Console` from `[Tools]` → `[NuGet Package Manager]` → `[Package Manager Console]` to write **commands** for **managing migrations**:
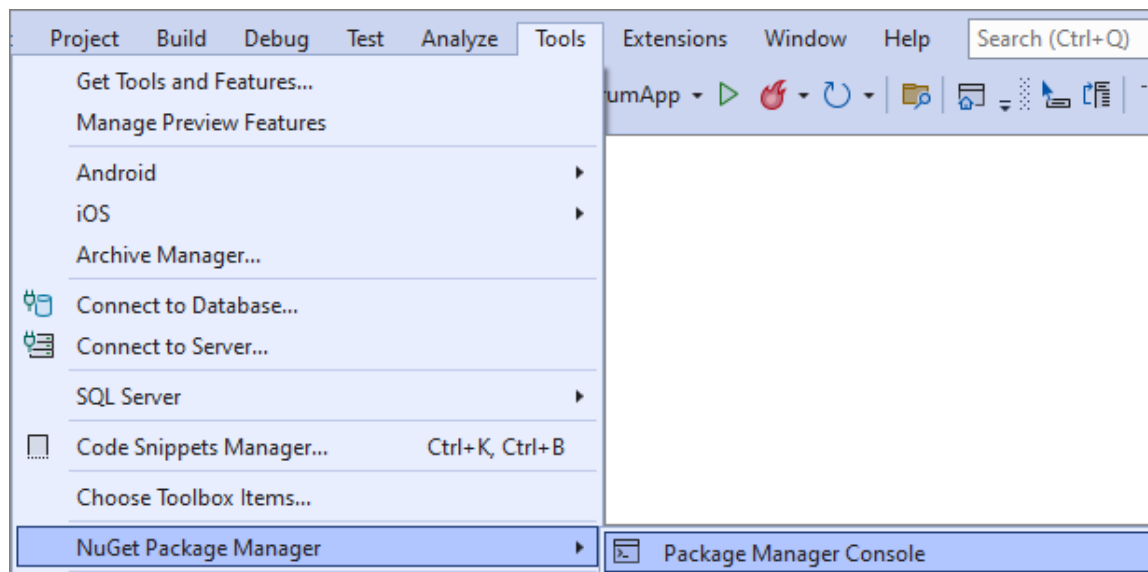
Type a command for adding a new migration called "`CreatedTables`" to the "`/Data/Migrations`" folder and press enter to execute it.

Now you should have a **new** migration in the **"Migrations"** folder**:**
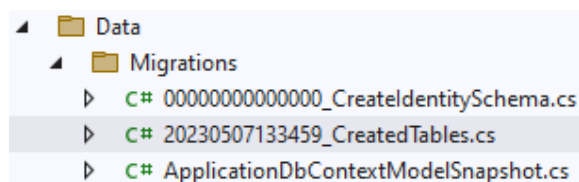
In the **console**, write a command for **updating the database** and **press** `[Enter]` to **execute it**.

You can examine the tables from the database in **Server Management Studio**. Note that now we have two users in the `AspNetUsers` table – the one that we registered in the beginning of the workshop and the one that we seeded in the previous section.

```
□ ■ TaskBoardApp
    ⊞ ■ Database Diagrams
    ⊟ ■ Tables
        ⊞ ■ System Tables
        ⊞ ■ FileTables
        ⊞ ■ External Tables
        ⊞ ■ Graph Tables
        ⊞ ⊞ dbo.__EFMigrationsHistory
        ⊞ ⊞ dbo.AspNetRoleClaims
        ⊞ ⊞ dbo.AspNetRoles
        ⊞ ⊞ dbo.AspNetUserClaims
        ⊞ ⊞ dbo.AspNetUserLogins
        ⊞ ⊞ dbo.AspNetUserRoles
        ⊞ ⊞ dbo.AspNetUsers
        ⊞ ⊞ dbo.AspNetUserTokens
        ⊞ ⊞ dbo.Boards
        ⊞ ⊞ dbo.Tasks
    ⊞ ■ Views
    ⊞ ■ External Resources
    ⊞ ■ Synonyms
    ⊞ ■ Programmability
    ⊞ ■ Service Broker
    ⊞ ■ Storage
    ⊞ ■ Security
```

Now **run the app** in the browser – there should not be **any errors**. Try logging with the new user **test@softuni.bg**. You should be able to log in **successfully**.

8. Modify View for Users

Let's change how the navigation bar looks for a user that is logged in.

Display User Names in the Navigation Bar

Let's remove the link from the "**My Profile**" on the "**Hello** " message.

Go to the **_LoginPartial.cshtml view** and remove the link. The file should look as shown below:

```
_LoginPartial.cshtml + ×
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User))
    {
        <li class="nav-item">
            <a class="nav-link text-dark">Hello @User.Identity?.Name!</a>
        </li>
        <li class="nav-item">
            <form class="form-inline" asp-area="Identity" asp-page="/Account/Logout"
                asp-route-returnUrl="@Url.Action("Index", "Home", new { area = "" })">
                <button type="submit" class="nav-link btn btn-link text-dark">Logout</button>
            </form>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Register">Register</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
        </li>
    }
</ul>
```

9. Create the MVC Structure of the App

Now that we are ready with the **Login/Logout** and **Register** part, it's time to create the MVC structure of our app. We will have three controllers – **HomeController**, **TaskController** and **BoardController**.

Home page

We already have the **HomeController** in our **project** and we already modified it, but we still have to modify the view, too.

Go to the **/View/Home/Index.cshtml** file and change the code as shown below:

```
Index.cshtml + ×
@{
    ViewData["Title"] = "Home Page";
}

<div class="text-center">
    <h1 class="display-4">Task Board: A Board for Your Tasks</h1>
</div>
```

Later the index page will display the tasks count according to the board they are attached to. For now, we will leave the view this way.

Display Boards

Let's modify our app, so it we can see the three types of boards. In order to do so, first we need to create a boards controller in the "**Controllers"** folder.

For now we will only inject the **TaskBoardAppDbContext** through the constructor and assign it to a variable to use it:

```csharp
public class BoardController : Controller
{
    private readonly TaskBoardAppDbContext _data;

    0 references
    public BoardController(TaskBoardAppDbContext context)
    {
        _data = context;
    }
```
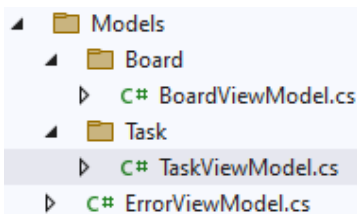
Our next step should be creating a "**Board**" folder in the "**Models**" folder and the **BoardViewModel** in the "**Models/Board**" folder. Create

```csharp
public class BoardViewModel
{
    0 references
    public int Id { get; init; }
    0 references
    public string Name { get; init; } = null!;
    0 references
    public IEnumerable<TaskViewModel> Tasks { get; set; }
        = new List<TaskViewModel>();
}
```

Each of the boards should display the tasks that are assigned to it, so the **BoardViewModel** should have a property for the collections of tasks. This means that we need another view model for the tasks. Let's add it to the "**Models**" folder. As we will have several models for the tasks, we can create a "**Task**" folder in the "**Models**" one, so that the files are more organized.

```
▲   📁 Models
    ▲   📁 Board
        ▷   C# BoardViewModel.cs
    ▲   📁 Task
        ▷   C# TaskViewModel.cs
    ▷   C# ErrorViewModel.cs
```

The **TaskViewModel** should look like this:

```csharp
public class TaskViewModel
{
    0 references
    public int Id { get; init; }
    0 references
    public string Title { get; init; } = null!;
    0 references
    public string Description { get; init; } = null!;
    0 references
    public string Owner { get;  init; } = null!;
}
```

Now we can go back the **BoardController** and add the action that will display the boards and their tasks. The **All()** action will extract the boards and their tasks from the database to model collections, which will be passed to a view.

Follow us:

```
public async Task<IActionResult> All()
{
    var boards = await _data
        .Boards
        .Select(b => new BoardViewModel()
        {
            Id = b.Id,
            Name = b.Name,
            Tasks = b
                .Tasks
                .Select(t => new TaskViewModel()
                {
                    Id = t.Id,
                    Title = t.Title,
                    Description = t.Description,
                    Owner = t.Owner.UserName
                })
        })
        .ToListAsync();

    return View(boards);
}
```

Now we should create the "**All.csthml**" view file. We will create this view in a new folder, called "**Board**", which will be located in the "**Views**" folder.

```
▲  📁 Views
    ▲  📁 Board
          📄 All.cshtml
```

We will be using the view models from the "**/View/Board**" and "**/View/Task/**" folders, so we should import them in the **_ViewImports.cshtml** file:

```
_ViewImports.cshtml  ⊣ ×
@using TaskBoardApp
@using TaskBoardApp.Models
@using TaskBoardApp.Models.Board
@using TaskBoardApp.Models.Task
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

As the **All.cshtml** file from the "**Board**" folder contains more code, you can copy it from here:

```
@model IEnumerable<BoardViewModel>

@{
    ViewData["Title"] = "Task Board";
}

<h2 class="text-center">@ViewBag.Title</h2>
<hr />

@if (!Model.Any())
{
    <h3 class="text-center">No tasks yet!</h3>
}
else
{
    <div class="container text-center">
        <div class="row">
```

```
                @foreach (var board in Model)
                {
                    <div class="col">
                        <p class="fs-1">
                            @board.Name
                        </p>


                        @if (board.Tasks.Any())
                        {
                            <div class="row  d-flex justify-content-center">
                                @await
Html.PartialAsync("~/Views/Shared/_TaskPartial.cshtml", board.Tasks)
                            </div>
                        }
                    </div>
                }
            </div>
        </div>
}
```

You can see that the view uses a partial view named "**_TaskPartial.cshtml**" from the "**Shared**" folder. We should create it as well. You can copy the code from here:

```
@model IEnumerable<TaskViewModel>

@foreach (var task in Model)
{
    <div class="card" style="width: 20rem;">
        <ul class="list-group list-group-flush">
            <li class="list-group-item"><span class="fw-bold">Title: </span>
@task.Title</li>
            <li class="list-group-item"><span class="fw-bold">Description: </span> :
@task.Description</li>
            <li class="list-group-item">
                <span class="fw-bold">Actions: </span>
                <a asp-controller="Task" asp-action="Details" asp-route-
id="@task.Id" class="btn btn-success mb-2">View</a>
                <span></span>
                @if (User.Identity.Name == task.Owner)
                {
                    <a asp-controller="Task" asp-action="Edit" asp-route-
id="@task.Id" class="btn btn-primary mb-2">Edit</a>
                    <span></span>
                    <a asp-controller="Task" asp-action="Delete" asp-route-
id="@task.Id" class="btn btn-danger mb-2">Delete</a>
                }
            </li>
        </ul>
    </div>
}
```
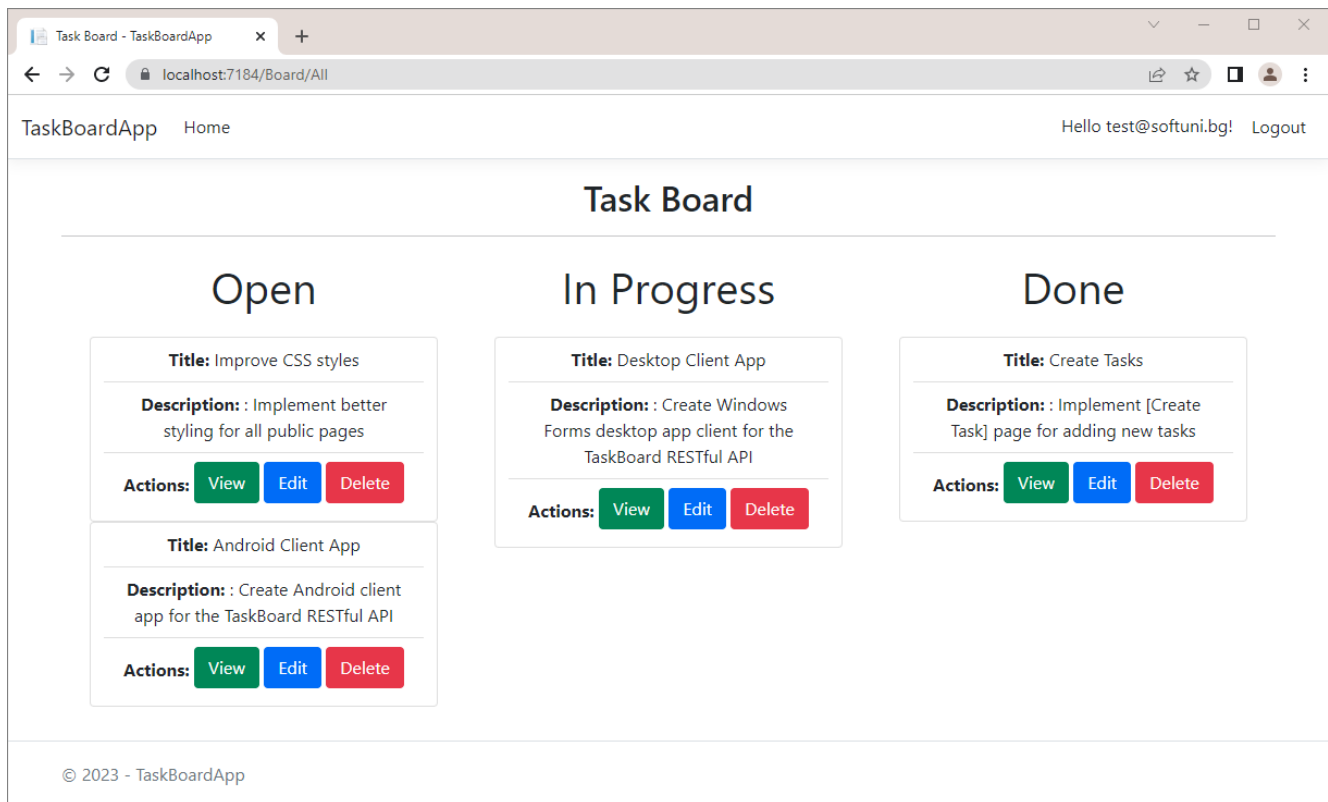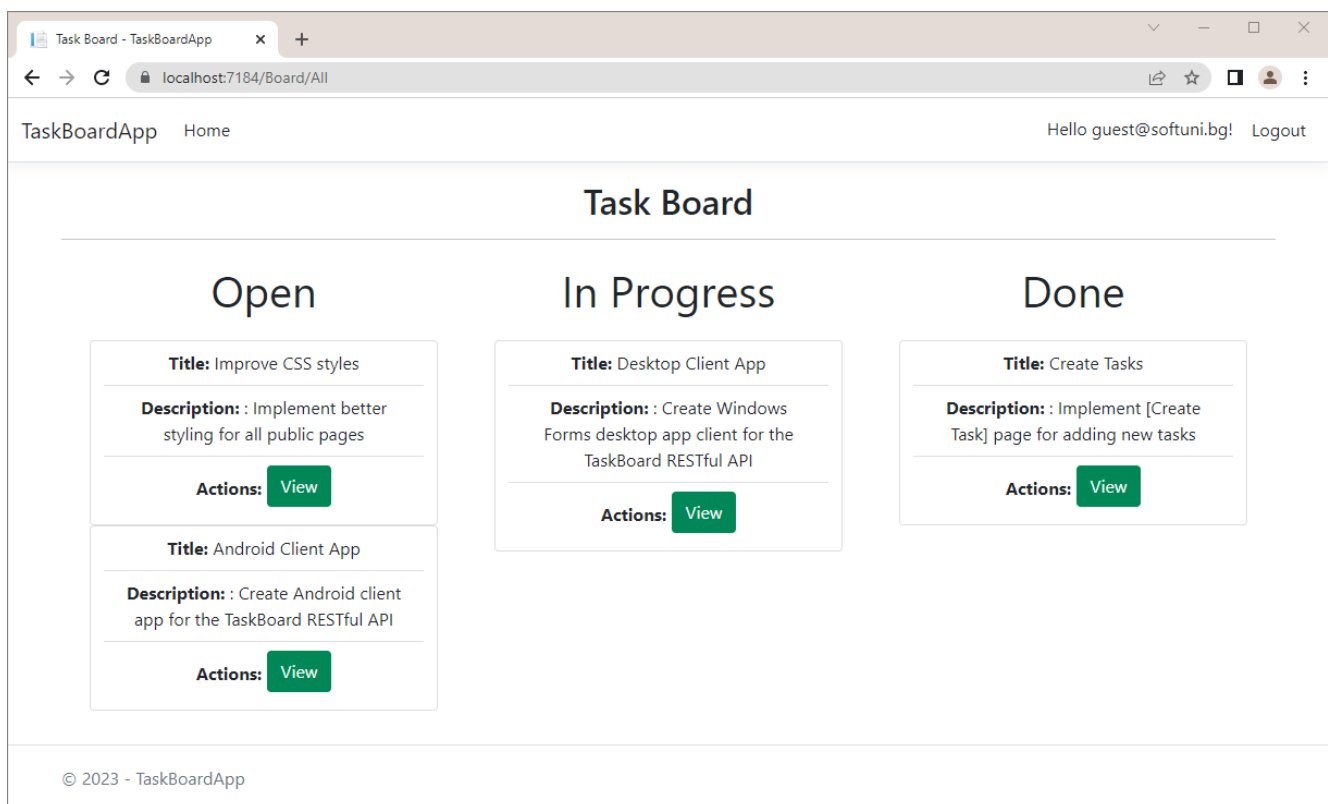
You can see that there is an **if statement**, which checks if the **logged-in user** is the **owner of the task.** In case the user is the one that **created the task,** they can see **two additional buttons – [Edit]** and **[Delete].**

Run the app in the browser and go to **/Board/All**.

When you log in with the **Guest** user, it should look like this:

SoftUni

When you log in with another user, it should look like this:

Note that the user **guest@softuni.bg** is not the owner of the tasks and they cannot see the **[Edit]** and **[Delete]** buttons.

Now we should add a link to the **/Board/All** page in the navigation. The link should be visible only to logged in users.
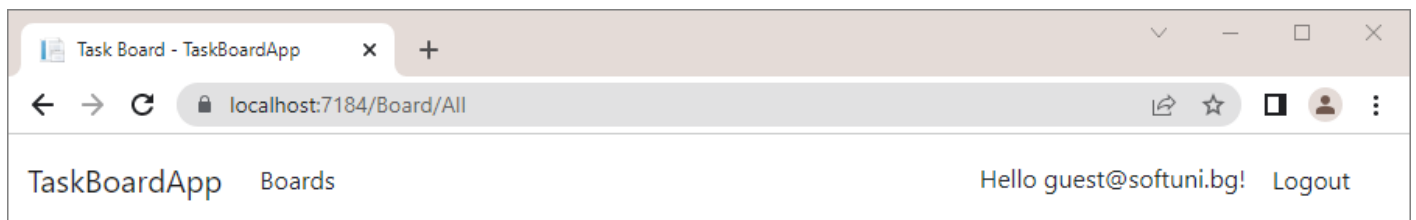
In order to do so, go to the "**_Layout.cshtml**" file and modify the code as shown below:

```
_Layout.cshtml ⊅ ✕
<!DOCTYPE html>
<html lang="en">
<head>...
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">TaskBoardApp</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse"...
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        @if (User.Identity.IsAuthenticated)
                        {
                            <li class="nav-item">
                                <a class="nav-link text-dark" asp-area="" asp-controller="Board" asp-action="All">Boards</a>
                            </li>
                        }
                        else
                        {
                            <li class="nav-item">
                                <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
                            </li>
                        }

                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
```

When you run the app, the navigation should look like this:

Create task

Now, let's extend the functionality of our app and modify it so that each user may be able to add a new task. The app should display a form for adding a task and it will look like this:

First, we need to create a **TaskFormModel** in the "/**Models/Task**" folder. We will add validation attributes to the model property. The **[Required]** attribute will check if the model property holds any value and the **[StringLength]** will check the length of the string that is held as a value.

```
public class TaskFormModel
{
    [Required]
    [StringLength(TaskMaxTitle, MinimumLength = TaskMinTitle,
        ErrorMessage = "Title should be at least {2} characters long.")]
    0 references
    public string Title { get; set; } = null!;

    [Required]
    [StringLength(TaskMaxDescription, MinimumLength = TaskMinDescription,
        ErrorMessage = "Description should be at least {2} characters long.")]
    0 references
    public string Description { get; set; } = null!;

    [Display(Name = "Board")]
    0 references
    public int BoardId { get; set; }

    0 references
    public IEnumerable<TaskBoardModel> Boards { get; set; } = null!;
}
```

As you can see, we will need another model, called **TaskBoardModel**, so you should create it in the "/**Models/Task**" folder, too.

```csharp
public class TaskBoardModel
{
    0 references
    public int Id { get; init; }
    0 references
    public string Name { get; init; } = null!;
}
```

We don't have a **TaskController** yet, so you should create it in the "**Controllers**" folder. Don't forget to inject the **TaskBoardAppDbContext** through the constructor and assign it to a variable to use it:

```csharp
public class TaskController : Controller
{
    private readonly TaskBoardAppDbContext _data;

    0 references
    public TaskController(TaskBoardAppDbContext context)
    {
        _data= context;
    }
```

Now that we have created the needed models, we should go to **TaskController** and implement the **Create()** method. This method will create a new **Task** object and then add it to the **DbSet**.

```csharp
public async Task<IActionResult> Create()
{
    TaskFormModel taskModel = new TaskFormModel()
    {
        Boards = GetBoards()
    };

    return View(taskModel);
}
```

We need an additional **GetBoards** method, so that we can display the boards in the dropdown menu later:

```csharp
private IEnumerable<TaskBoardModel> GetBoards()
    => _data
        .Boards
        .Select(b => new TaskBoardModel()
        {
            Id = b.Id,
            Name = b.Name
        });
```

Let's continue with the **Create** action method:

```csharp
[HttpPost]
0 references
public async Task<IActionResult> Create(TaskFormModel taskModel)
{
    if (!GetBoards().Any(b => b.Id == taskModel.BoardId))
    {
        ModelState.AddModelError(nameof(taskModel.BoardId), "Board does not exist.");
    }

    string currentUserId = GetUserId();

    if (!ModelState.IsValid)
    {
        taskModel.Boards = GetBoards();

        return View(taskModel);
    }

    Task task = new Task()
    {
        Title = taskModel.Title,
        Description = taskModel.Description,
        CreatedOn = DateTime.Now,
        BoardId = taskModel.BoardId,
        OwnerId = currentUserId,
    };
    await _data.Tasks.AddAsync(task);
    await _data.SaveChangesAsync();

    var boards = _data.Boards;

    return RedirectToAction("All", "Board");
}
```

We need one more additional method **GetUserId** that return the **id** of the logged in user, so that we can assing it to the **OwnerId** property of the newly created **Task**:

```csharp
private string GetUserId()
    => User.FindFirstValue(ClaimTypes.NameIdentifier);
```

The **ClaimTypes.NameIdentifier** returns the user id when needed.

Our next step is to create a new folder – "**Task**" in the "**Views**" folder, so that we can add a new "**Create.cshtml"** in it for the **"Create Task"** Page view. As there is a lot of code to write, you can copy it from here:

```html
@model TaskFormModel

@{
    ViewBag.Title = "Add Task";
}

<h2 class="text-center">@ViewBag.Title</h2>
<hr />

<div class="row">
    <div class="col-sm-12 offset-lg-2 col-lg-8 offset-xl-3 col-xl-6">
        <form asp-action="Create" method="post">
            <div class="mb-3">
```

```html
                <label asp-for="Title" class="form-label">Title</label>
                <input asp-for="Title" class="form-control" aria-required="true" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="mb-3">
                <label asp-for="Description" class="form-label">Description</label>
                <input asp-for="Description" class="form-control" aria-
required="true" />
                <span asp-validation-for="Description" class="text-danger"></span>
            </div>
            <div class="mb-3">
                <label asp-for="BoardId" class="form-label">Category</label>
                <select asp-for="BoardId" class="form-control">
                    @foreach (var board in Model.Boards)
                    {
                        <option value="@board.Id">@board.Name</option>
                    }
                </select>
                <span asp-validation-for="BoardId" class="text-danger"></span>
            </div>
            <div class="mb-3">
                <input class="btn btn-primary" type="submit" value="Add" />
            </div>
        </form>
    </div>
</div>


@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Run the app in the browser and go to **/Task/Create**. It should look like this:

All that is left to do is add a link to the "**/Task/Create**" page in the navigation. Go to the "**_Layout.cshtml**" file and modify the code as shown below:

```
@if (User.Identity.IsAuthenticated)
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Board" asp-action="All">Boards</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Task" asp-action="Create">Create a new Task</a>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
    </li>
}
```

Finally, when you run the app in the browser again, the navigation should look like this:

View task details

Now, let's extend the functionality of our app and modify it so that each user may be able to see each task details. The app should look like below:
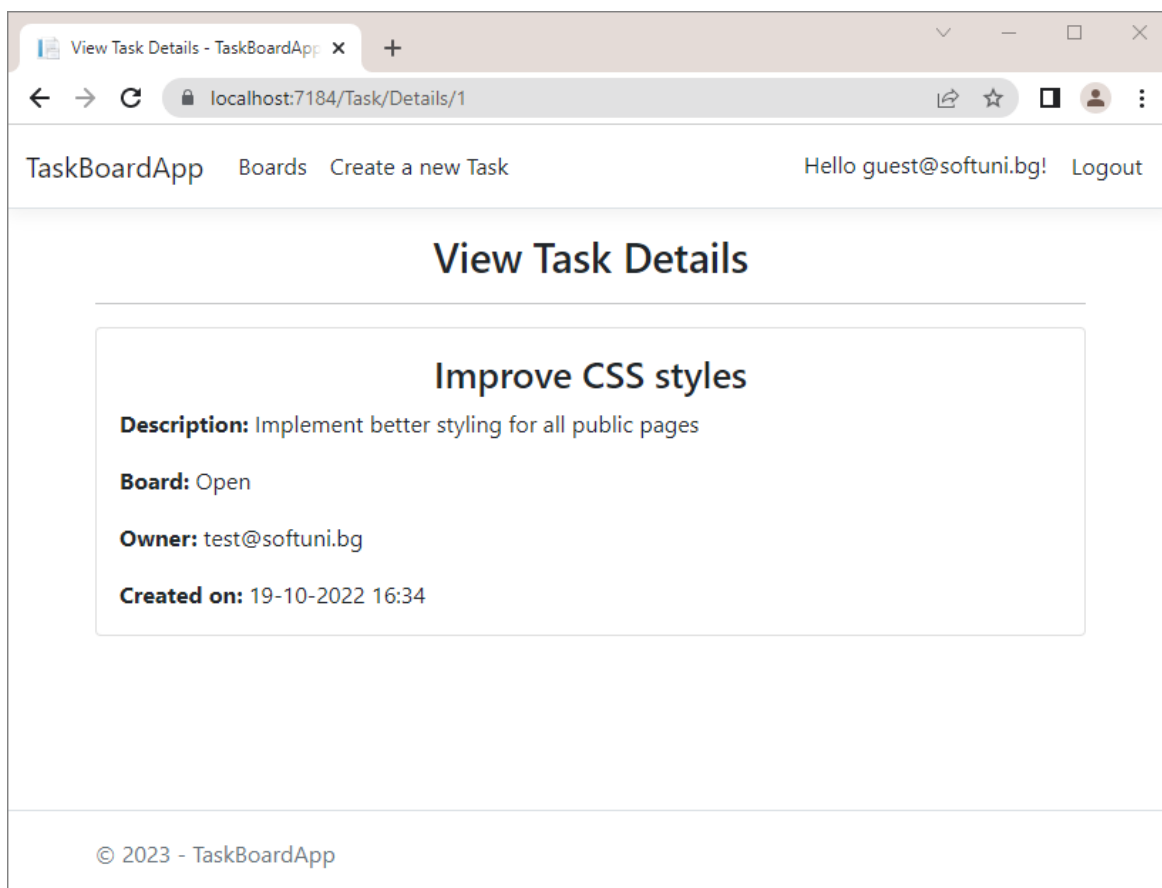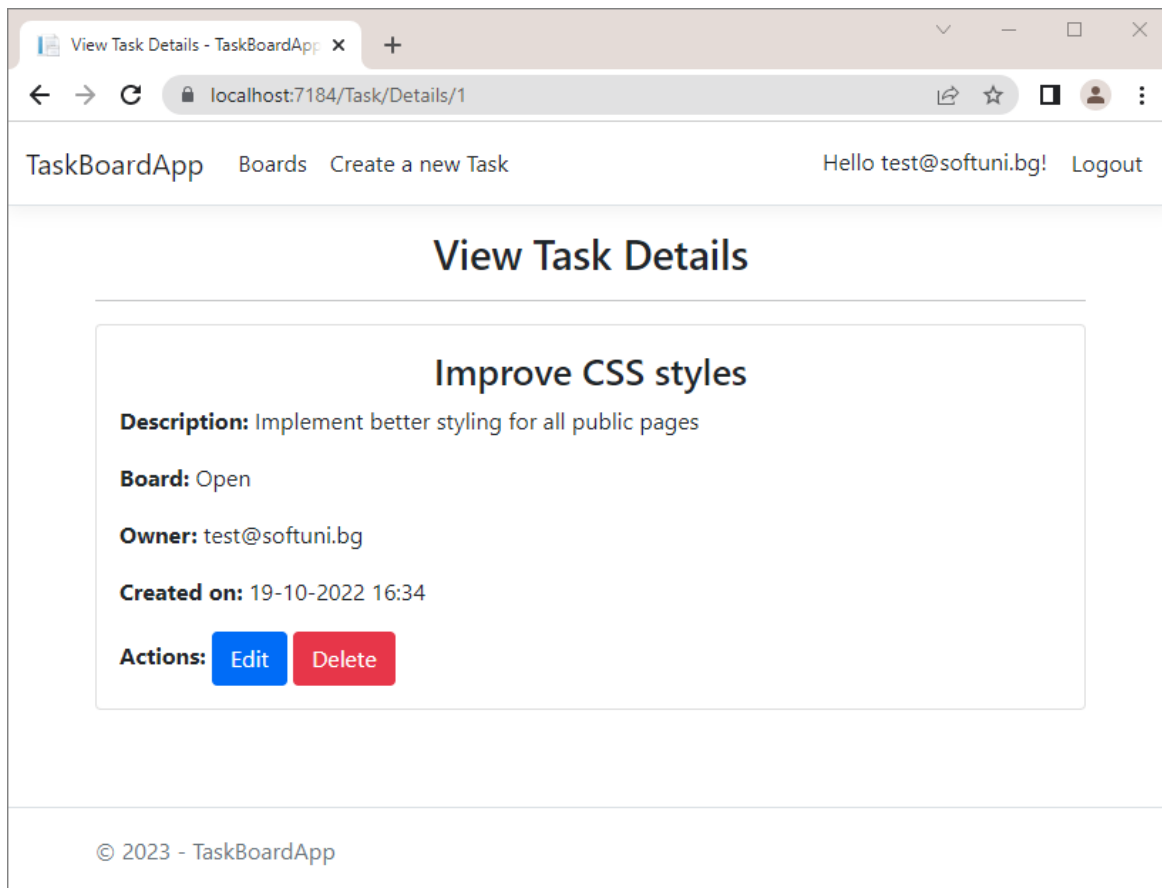
Note that the different users see different views – the reason for this is that in the first picture the logged in user is the owner of the task and in the second picture the logged in user is not the owner.

Follow us:

Let's create a **TaskDetailsViewModel** which will inherit the **TaskViewModel**. It will have two additional properties – **CreatedOn** and **Board**.

```csharp
public class TaskDetailsViewModel : TaskViewModel
{
    0 references
    public string CreatedOn { get; init; } = null!;
    0 references
    public string Board { get; init; } = null!;
}
```

Now that we have the model, we can go to the **TaskController** and add the **Details()** action.

```csharp
public async Task<IActionResult> Details(int id)
{
    var task = await _data
        .Tasks
        .Where(t => t.Id == id)
        .Select(t => new TaskDetailsViewModel()
        {
            Id = t.Id,
            Title = t.Title,
            Description = t.Description,
            CreatedOn = t.CreatedOn.ToString("dd/MM/yyyy HH:mm"),
            Board = t.Board.Name,
            Owner = t.Owner.UserName
        })
        .FirstOrDefaultAsync();

    if (task == null)
    {
        return BadRequest();
    }

    return View(task);
}
```

Now we have to create the view for the "**/Tasks/Details/{id}**" page. As the code is a lot to write, you can copy it from here:

```html
@model TaskDetailsViewModel

@{
    ViewBag.Title = "View Task Details";
}

<h2 class="text-center">@ViewBag.Title</h2>
<hr />

<div class="d-flex justify-content-center">
    <div class="card " style="width: 50rem;">
        <div class="card-body">
            <h3 class="card-title text-center">@Model.Title</h3>
            <p class="card-text"><span class="fw-bold">Description:
</span>@Model.Description</p>
            <p class="card-text"><span class="fw-bold">Board:
</span>@Model.Board</p>
            <p class="card-text"><span class="fw-bold">Owner:
</span>@Model.Owner</p>
```

```
                <p class="card-text"><span class="fw-bold">Created on:
</span>@Model.CreatedOn</p>
                @if (Model.Owner == User.Identity.Name)
                {
                    <tr class="actions">
                        <th><span class="fw-bold">Actions:</span></th>
                        <td>
                            <a asp-controller="Task" asp-action="Edit" asp-route-
id="@Model.Id" class="btn btn-primary">Edit</a>
                            <a asp-controller="Task" asp-action="Delete" asp-route-
id="@Model.Id" class="btn btn-danger">Delete</a>
                        </td>
                    </tr>
                }
        </div>
    </div>
</div>
```

Edit task

The next step in creating the **TaskBoardApp** is adding the "**Edit Task**" page. It will display a **form** for **editing a task** and it will look like this:

We already have a **TaskFormModel** and we'll use it again. First, we need to modify the **TaskController** and add an **Edit** action method, which will pass a **Task** model to the view and find and update the task in the database.

Follow us:

```csharp
public async Task<IActionResult> Edit(int id)
{
    var task = await _data.Tasks.FindAsync(id);

    if (task == null)
    {
        return BadRequest();
    }

    string currentUserId = GetUserId();
    if (currentUserId != task.OwnerId)
    {
        return Unauthorized();
    }

    TaskFormModel taskModel = new TaskFormModel()
    {
        Title = task.Title,
        Description = task.Description,
        BoardId = task.BoardId,
        Boards = GetBoards()
    };

    return View(taskModel);
}

[HttpPost]
public async Task<IActionResult> Edit(int id, TaskFormModel taskModel)
{
    var task = await _data.Tasks.FindAsync(id);

    if (task == null)
    {
        return BadRequest();
    }

    string currentUserId = GetUserId();
    if (currentUserId != task.OwnerId)
    {
        return Unauthorized();
    }

    if (!GetBoards().Any(b => b.Id == taskModel.BoardId))
    {
        ModelState.AddModelError(nameof(taskModel.BoardId), "Board does not exist.");
    }

    if (!ModelState.IsValid)
    {
        taskModel.Boards = GetBoards();

        return View(taskModel);
    }

    task.Title = taskModel.Title;
    task.Description = taskModel.Description;
    task.BoardId = taskModel.BoardId;

    await _data.SaveChangesAsync();
    return RedirectToAction("All", "Board");
}
```

Our next step is to create the view for editing a task. You can copy the code from here:

```
@model TaskFormModel

@{
    ViewBag.Title = "Edit Task";
}


<h2 class="text-center">@ViewBag.Title</h2>
<hr />

<div class="row">
    <div class="col-sm-12 offset-lg-2 col-lg-8 offset-xl-3 col-xl-6">
        <form asp-action="Edit">
            <div class="mb-3">
                <label asp-for="Title" class="form-label">Title</label>
                <input asp-for="Title" class="form-control" aria-required="true" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="mb-3">
                <label asp-for="Description" class="form-label">Description</label>
                <input asp-for="Description" class="form-control" aria-
required="true" />
                <span asp-validation-for="Description" class="text-danger"></span>
            </div>
            <div class="mb-3">
                <label asp-for="BoardId" class="form-label">Category</label>
                <select asp-for="BoardId" class="form-control">
                    @foreach (var board in Model.Boards)
                    {
                        <option value="@board.Id">@board.Name</option>
                    }
                </select>
                <span asp-validation-for="BoardId" class="text-danger"></span>
            </div>
            <div class="mb-3">
                <input class="btn btn-primary" type="submit" value="Edit" />
            </div>
        </form>
    </div>
</div>
```
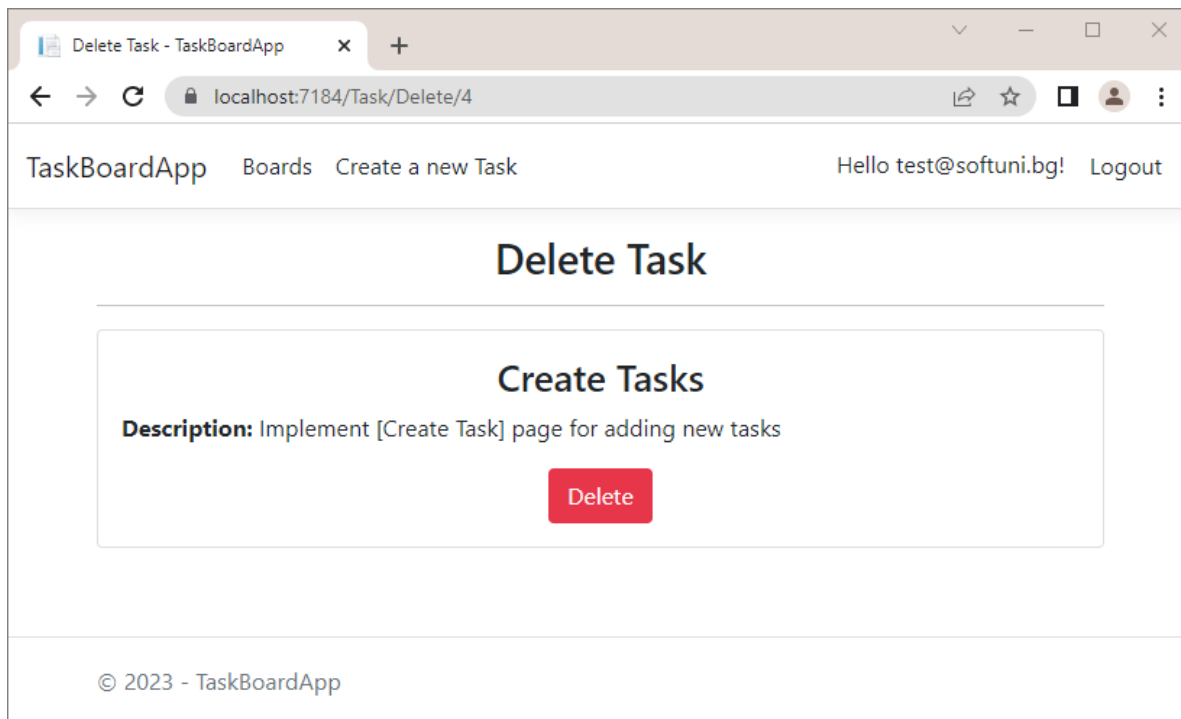
Run the app in the browser and try to edit a task.

Delete task

The view for deleting a task is similar to the view for editing a task. It should look like this:

Bur first, let's add the **Delete** action method to the **TaskController**:

```
public async Task<IActionResult> Delete(int id)
{
    var task = await _data.Tasks.FindAsync(id);
    if (task == null)
    {
        return BadRequest();
    }

    string currentUserId = GetUserId();
    if (currentUserId != task.OwnerId)
    {
        return Unauthorized();
    }

    TaskViewModel taskModel = new TaskViewModel()
    {
        Id = task.Id,
        Title = task.Title,
        Description = task.Description
    };

    return View(taskModel);
}
```

Follow us:

```
[HttpPost]
0 references
public async Task<IActionResult> Delete(TaskViewModel taskModel)
{
    var task = await _data.Tasks.FindAsync(taskModel.Id);
    if (task == null)
    {
        return BadRequest();
    }

    string currentUserId = GetUserId();
    if (currentUserId != task.OwnerId)
    {
        return Unauthorized();
    }

    _data.Tasks.Remove(task);
    await _data.SaveChangesAsync();
    return RedirectToAction("All", "Board");
}
```

After we have added the **Delete()** action, let's create the **Delete** view file.

You can copy the code from here:

```
@model TaskViewModel

@{
    ViewData["Title"] = "Delete Task";
}

<h2 class="text-center">@ViewBag.Title</h2>
<hr />



<form asp-action="Delete">
    <div class="d-flex justify-content-center">
        <div class="card " style="width: 50rem;">
            <div class="card-body">
                <h3 class="card-title text-center">@Model.Title</h3>
                <p class="card-text"><span class="fw-bold">Description:
</span>@Model.Description</p>
            </div>
            <div class="mb-3 d-flex justify-content-center">
                <input type="submit" value="Delete" class="btn btn-danger" />
            </div>
        </div>
    </div>
</form>
```

Home page

Finally, we can go back to the **Home page** so that we can modify it to show the tasks count. Let's go back to the **HomeController.** Don't forget to inject the **TaskBoardAppDbContext** through the constructor and assign it to a variable to use it:

```csharp
public class HomeController : Controller
{
    private readonly TaskBoardAppDbContext _data;

    0 references
    public HomeController(TaskBoardAppDbContext context)
    {
        _data = context;
    }
```

Now we need to create a **HomeBoardModel** and a **HomeViewModel** in the "**Models**" folder.

```csharp
public class HomeBoardModel
{
    0 references
    public string BoardName { get; set; } = null!;
    0 references
    public int TasksCount { get; set; }
}

public class HomeViewModel
{
    0 references
    public int AllTasksCount { get; init; }
    0 references
    public List<HomeBoardModel> BoardsWithTasksCount { get; init; } = null!;
    0 references
    public int UserTasksCount { get; init; }
}
```

Now that we have created the models, go back to the controller and this code:

```csharp
public async Task<IActionResult> Index()
{
    var taskBoards = _data
        .Boards
        .Select(b => b.Name)
        .Distinct();

    var tasksCounts = new List<HomeBoardModel>();
    foreach (var boardName in taskBoards)
    {
        var tasksInBoard = _data.Tasks.Where(t => t.Board.Name == boardName).Count();
        tasksCounts.Add(new HomeBoardModel()
        {
            BoardName = boardName,
            TasksCount = tasksInBoard
        });
    }

    var userTasksCount = -1;

    if (User.Identity.IsAuthenticated)
    {
        var currentUserId = User.FindFirst(ClaimTypes.NameIdentifier).Value;
        userTasksCount = _data.Tasks.Where(t => t.OwnerId == currentUserId).Count();
    }

    var homeModel = new HomeViewModel()
    {
        AllTasksCount = _data.Tasks.Count(),
        BoardsWithTasksCount = tasksCounts,
        UserTasksCount = userTasksCount
    };

    return View(homeModel);
}
```

And finally, we have to modify the view file. Go to "**/Home/Index.cshtml**" file and add the following code:

```html
@{
    ViewData["Title"] = "Home Page";
}

<div class="text-center">
    <h1 class="display-4">
        Task Board: A Board for Your Tasks</h1>
</div>

<hr />

<section class="home-page">
    @if (!User.Identity.IsAuthenticated)
    {
        <h1 class="text-center">Welcome!</h1>
    }
    else
    {
        <h1 class="text-center">Welcome, @User.Identity.Name!</h1>
    }
    <hr class="hr-2 bg-secondary" />
    <h4 class="mt-4 text-center">TaskBoard is here for all your tasks.</h4>
    <br />
```

```
    <span class="boardSpan">Tasks: <b>@Model.AllTasksCount</b></span>
    <ul>
        @foreach (var board in Model.BoardsWithTasksCount)
        {
            <li>
                <span class="boardSpan">@board.BoardName:
<b>@board.TasksCount</b></span>
            </li>
        }
    </ul>
    @if (Model.UserTasksCount > -1)
    {
        <h5>You have: <b>@Model.UserTasksCount</b> tasks</h5>
    }
</section>
```
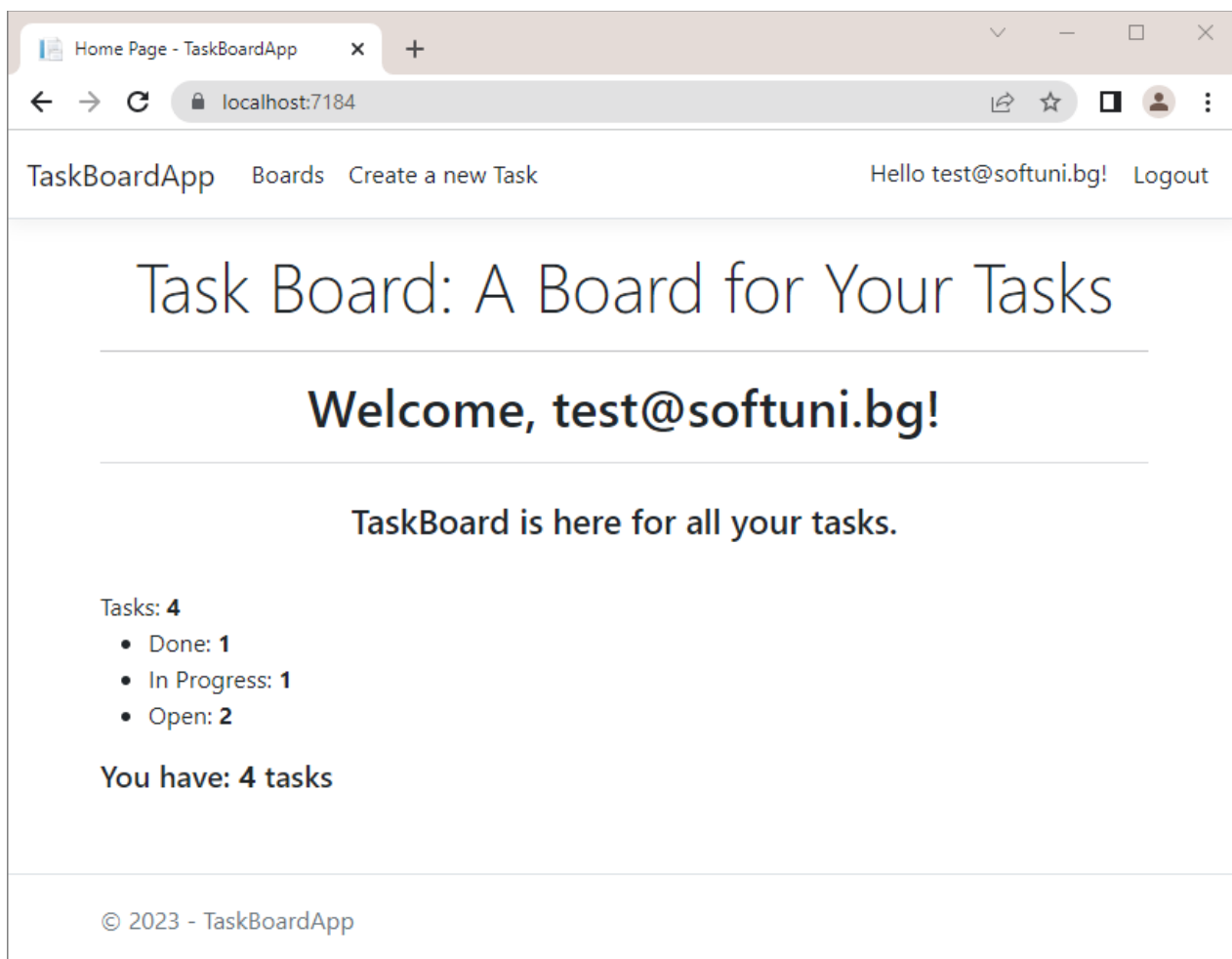
Run the app in the browser and it should look like shown below:

10. Secure the application

The not logged in users do not see the "Boards" and "Create a new Task" buttons in the navigation, but if you type for example "**Task/Create**" in the address bar, you will be able to access the page for creating a new task.

In order to avoid this, you need to add the **[Authorize]** attribute to the **TaskController** and add the following code in the **Program** class:

```
builder.Services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Identity/Account/Login";
});
```

This way, whenever an unauthorized user tries to access any page related to the **TaskController**, they'll be redirected to the **Login** page.