Workshop: Users, Security, App Layers and Web API

Workshop for the "ASP.NET Advanced" course @ SoftUni

The "**House Renting System**" **ASP.NET Core MVC App** is a Web application for **house renting**. **Users** can look at **all houses** with their **details**, **rent a house** and look at **their rented houses**. They can also **become Agents**. **Agents** can **add houses**, see their **details** and **edit** and **delete** only **houses they added**. The **Admin** has **all privileges** of **Users** and **Agents** and can see **all registrations** in the app and **all made rents**.

1. Add Custom User

Now, we will **create our custom user**. It will **extend the default user** from ASP.NET Core by **adding first and last name properties**.

First, create a **ApplicationUser model class** in the "**/Data/Models**" **folder**, which should **inherit IdentityUser** (the default user). It should also have **properties for first and last name**. They should be **required** and have **restrictions**. Write the class like this:

```
public class ApplicationUser : IdentityUser
{
    [Required]
    [MaxLength(UserFirstNameMaxLength)]
    0 references
    public string FirstName { get; set; } = null!;


    [Required]
    [MaxLength(UserLastNameMaxLength)]
    0 references
    public string LastName { get; set; } = null!;
}
```

The **maximum and minimum lengths** of properties are part of the **DataConstants class**:

```
public class ApplicationUser
{
    // First name
    public const int UserFirstNameMaxLength = 12;
    public const int UserFirstNameMinLength = 1;

    // Last name
    public const int UserLastNameMaxLength = 15;
    public const int UserLastNameMinLength = 3;
}
```

Now we should **replace** the **IdentityUser** with our **custom user** everywhere in our code (except for the **migrations** – they will be changed later).

First, go to **Program.cs** and **modify the Identity service**:

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

Next, modify the **HouseRentingDbContext class**, which should **inherit the IdentityDbContext** with **ApplicationUser**:

```
public class HouseRentingDbContext : IdentityDbContext<ApplicationUser>
{
```

Don't forget to change **IdentityUser** to **User** on **all places**. Also, you should **add first and last names** to the **seeded user records**. The rest of the class looks like this when changed:

```
    private ApplicationUser AgentUser { get; set; }
    4 references
    private ApplicationUser GuestUser { get; set; }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        SeedUsers();
        builder.Entity<ApplicationUser>()
                .HasData(AgentUser, GuestUser);

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

        AgentUser = new ApplicationUser()
        {
            Id = "dea12856-c198-4129-b3f3-b893d8395082",
            UserName = "agent@mail.com",
            NormalizedUserName = "agent@mail.com",
            Email = "agent@mail.com",
            NormalizedEmail = "agent@mail.com",
            FirstName = "Linda",
            LastName = "Michaels"
        };

        AgentUser.PasswordHash =
            hasher.HashPassword(AgentUser, "agent123");

        GuestUser = new ApplicationUser()
        {
            Id = "6d5800ce-d726-4fc8-83d9-d6b3ac1f591e",
            UserName = "guest@mail.com",
            NormalizedUserName = "guest@mail.com",
            Email = "guest@mail.com",
            NormalizedEmail = "guest@mail.com",
            FirstName = "Teodor",
            LastName = "Lesly"
        };

        GuestUser.PasswordHash =
            hasher.HashPassword(AgentUser, "guest123");
    }
```

Then, you should **modify the type of the ApplicationUser property** in the **Agent entity class**:

```
public class Agent
{
    5 references
    public int Id { get; init; }

    [Required]
    [MaxLength(PhoneNumberMaxLength)]
    4 references
    public string PhoneNumber { get; set; } = null!;

    [Required]
    5 references
    public string UserId { get; set; } = null!;

    1 reference
    public ApplicationUser User { get; init; } = null!;
}
```

At the end, we should also modify the "**_LoginPartial.cshtml**" **file**, which has **injected services** with **IdentityUser**. Import the **namespace** of the **ApplicationUser class** directly in the view (as we don't need it anywhere else). Then, make the **SignInManager** and **UserManager use the custom user**. Modify the view like this:

_LoginPartial.cshtml

```
@using Microsoft.AspNetCore.Identity
@using HouseRentingSystem.Data.Models;
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager
@inject IAgentService agents
```

Finally, let's **create a migration**, which will create **new columns in the database**. Open the **Package Manager Console** and **add a new migration** with:

```
Add-Migration AddedUserColumns -o Data/Migrations
```

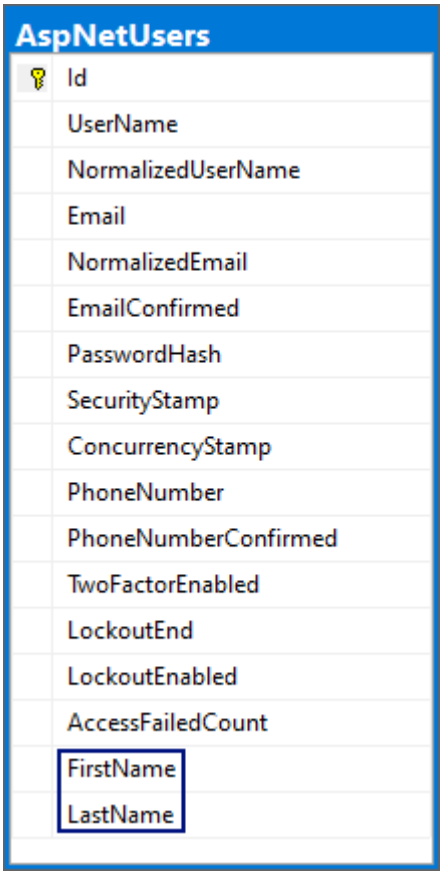
The migration should be created successfully. Look at it – it should have code for **adding** the "**FirstName**" and "**LastName**" **columns and the new seeded data**.

Apply the migrating by updating the database with:

```
Update-Database
```

Note that there may be some other **updates of the database in the migration** but if you run the app and there are **no errors** then everything is fine.

Run the app in the browser and make sure that there are **no errors** because of the new migration. Then, open **SQL Server Management Studio** and examine the "**AspNetUsers**" **table**. It should have the "**FirstName**" and "**LastName**" **columns**:



In addition, our **seeded users** should have the **names we added**:

	Id	UserName	Email	FirstName	LastName
1	6d5800ce-d726-4fc8-83d9-d6b3ac1f591e	guest@mail.com	guest@mail.com	Teodor	Lesly
2	dea12856-c198-4129-b3f3-b893d8395082	agent@mail.com	agent@mail.com	Linda	Michaels
3	e3b4a709-cdc0-459a-93f1-163d801b665f	test2@softuni.bg	test2@softuni.bg		

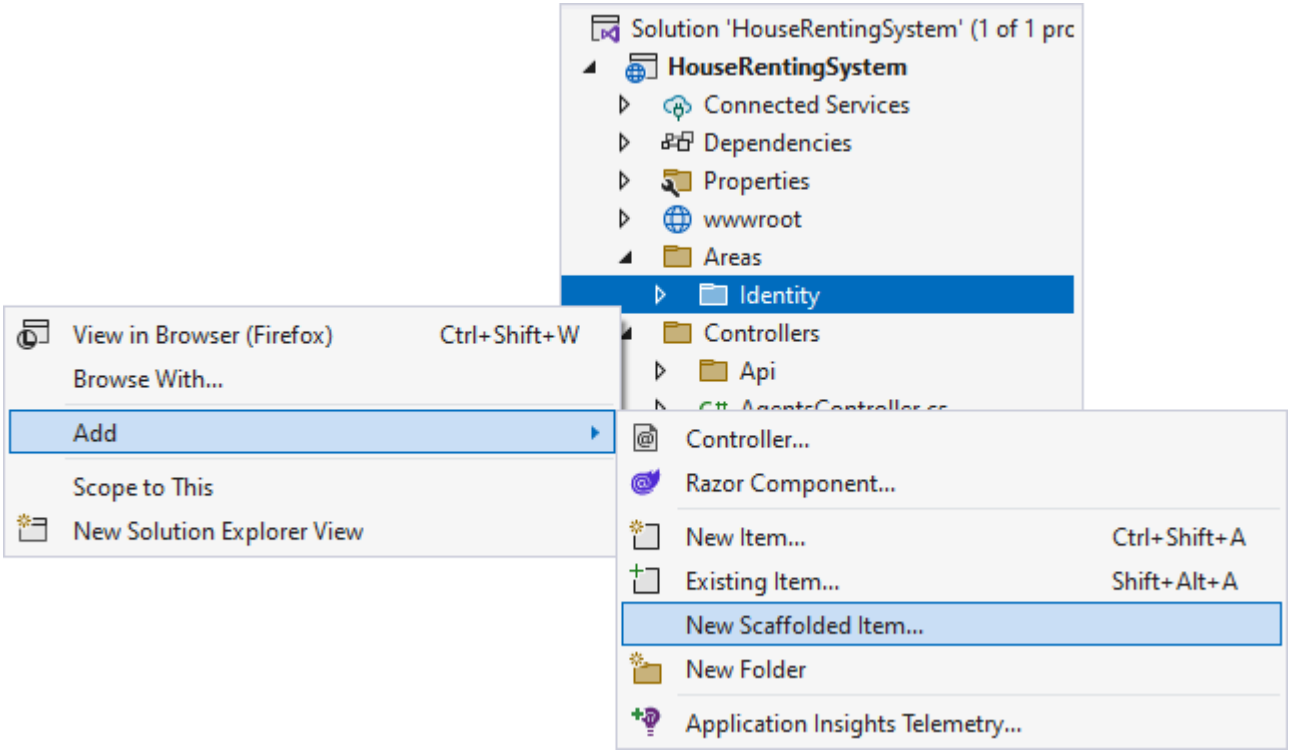
2. Modify Registration and Login

In this task we want to **clear** the "**`Login`**" and "**`Registration`**" **pages** from **functionalities that we won't use**. Also, we want to **add new fields** to the **registration form**.

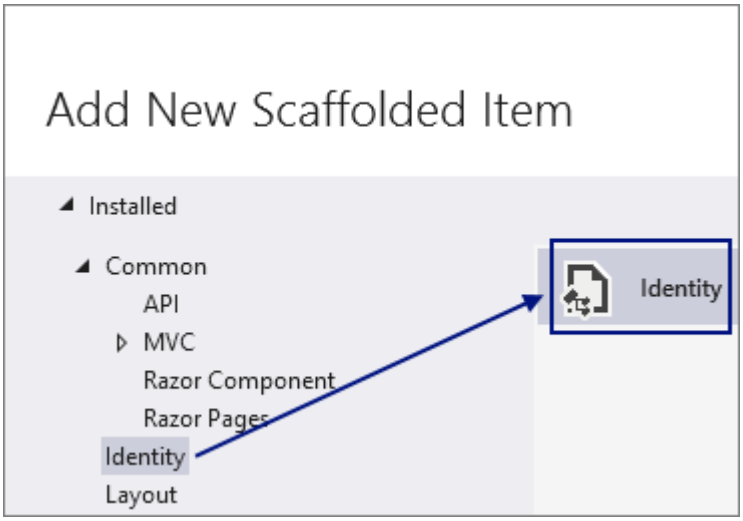
Scaffold Identity

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.

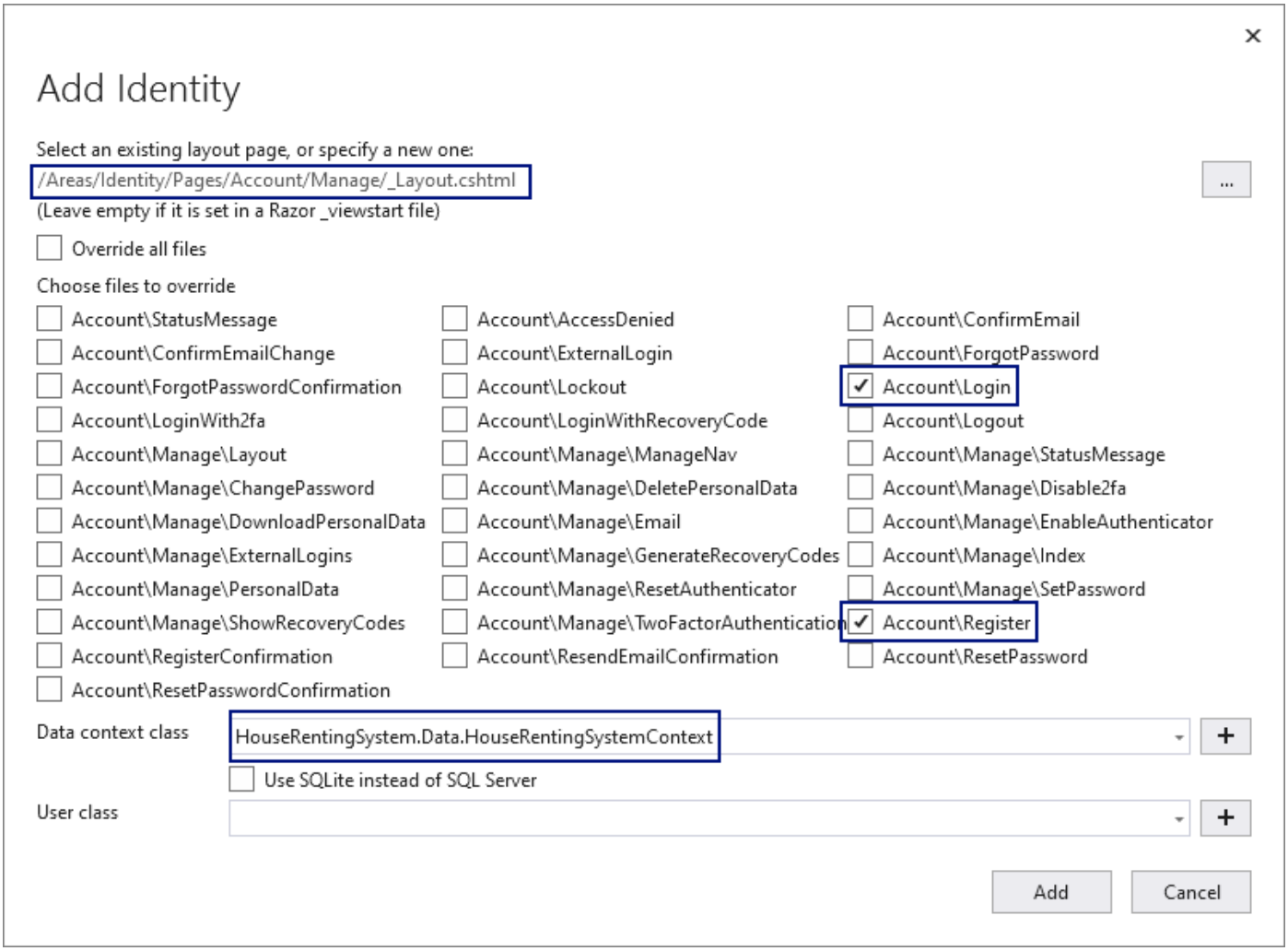
The **scaffolded pages** will be part of the "**`/Areas/Identity`**" **folder**. To scaffold, **right-click** on the "**`Identity`**" **folder** and choose **`[Add]`** → **`[New Scaffolded Item…]`**:



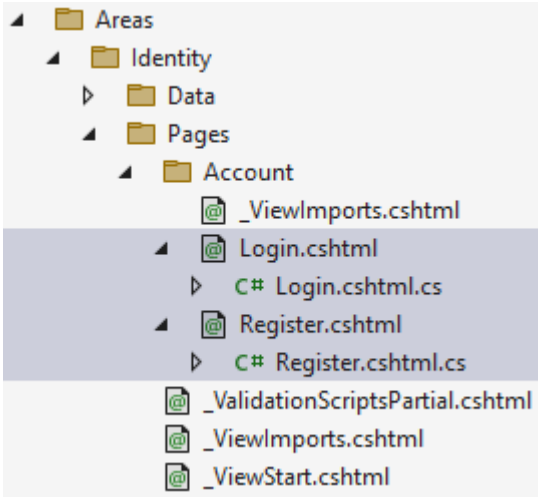
On the next step, go to the **`[Identity]` tab** and **choose its only option**:



Then, on the "**Add `Identity`" window** you should set the **`_Layout.cshtml`** as a **layout page**, check the pages to be **scaffolded** ("**`Login`**" and "**`Register`**") and **select the db `context` class** of our app. Do it like this:



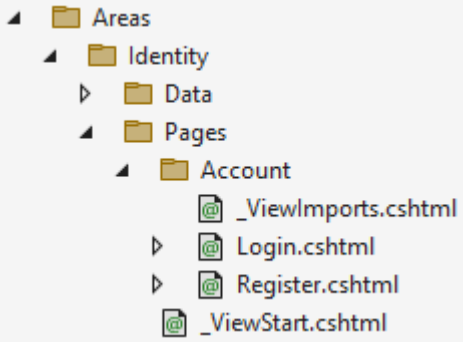
Click the **[Add] button** and examine the **scaffolded pages** in the "**/Areas/Identity**" **folder**:



Note that the **generated pages** are **Razor pages**. They have two files – one with extension **.cshtml** and one with **.cshtml.cs**. The **Login.cshtml** and **Register.cshtml files** are Razor pages and the logic behind them is in the **Login.cshtml.cs** and **Register.cshtml.cs files**. The **LoginModel** 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, you can delete the new Data folder, as you won't need it. You can also **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**:

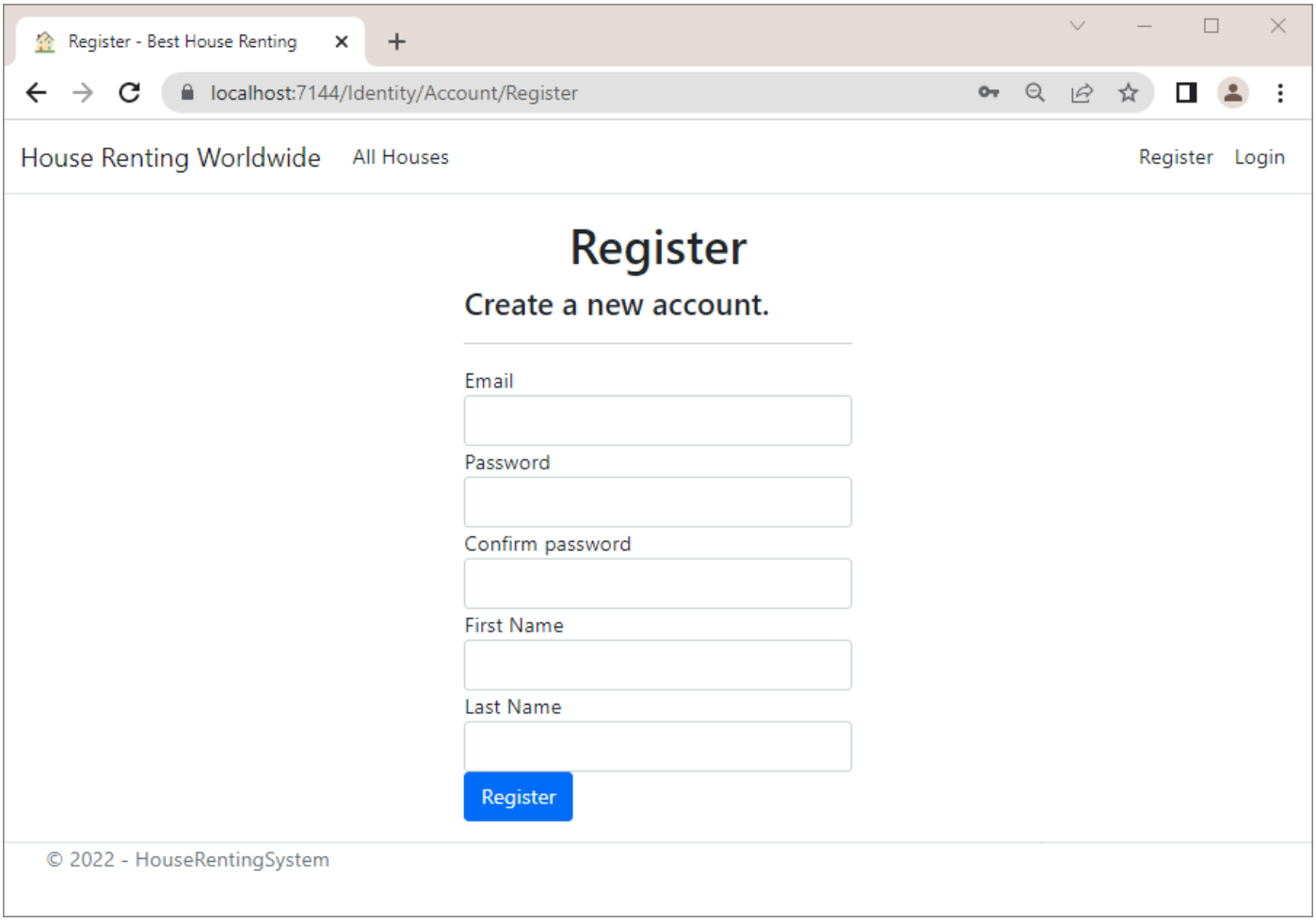
Delete the classes in the "**Areas**" **folder**, which are not in the "**Account**" **folder**. **Leave** only the **_ViewStart.cshtml file** – others are **unnecessary**. The left classes should be the following:



You can also **delete** the **ScaffoldingReadMe.txt file** from the solution.

Modify the "Register" Page

Now we want to **modify our** "**Register**" **page**. It should **not have external logins**, but should have **fields for first and last names**. It 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**. Also, we need to **add text fields** for the **first and last names** of the user. 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**:

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

<div class="row">
    <div class="col-md-4 offset-md-4">
        <form id="registerForm" asp-route-returnUrl="@Model.ReturnUrl" method="post">
            <h2>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>
            <div class="form-floating">
                <input asp-for="Input.FirstName" class="form-control" autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.FirstName"></label>
                <span asp-validation-for="Input.FirstName" class="text-danger"></span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.LastName" class="form-control" autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.LastName"></label>
                <span asp-validation-for="Input.LastName" class="text-danger"></span>
            </div>
            <button id="registerSubmit" type="submit" class="w-100 btn btn-lg btn-primary">Register</button>
        </form>
    </div>
</div>
```

Now we will add the "**FirstName**" and "**LastName**" **properties** to the **InputModel** in the **RegisterModel class**. Open the "**Register.cshtml.cs**" **file** and do it like this:

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

    [Required]
    [Display(Name = "First Name")]
    [StringLength(UserFirstNameMaxLength,
        MinimumLength = UserFirstNameMinLength)]
    3 references
    public string FirstName { get; set; }

    [Required]
    [Display(Name = "Last Name")]
    [StringLength(UserFirstNameMaxLength,
        MinimumLength = UserFirstNameMinLength)]
    3 references
    public string LastName { get; set; }
```

As we **added the properties**, it is important to use them when **creating a user to fill the database columns**. To do this, **modify** the **OnPostAsync(…) method** like this:

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

    if (ModelState.IsValid)
    {
        var user = new ApplicationUser
        {
            UserName = Input.Email,
            Email = Input.Email,
            FirstName = Input.FirstName,
            LastName = Input.LastName,
        };
```

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:

```
[AllowAnonymous]
6 references
public class RegisterModel : PageModel
{
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly UserManager<ApplicationUser> _userManager;

    0 references
    public RegisterModel(
        UserManager<ApplicationUser> userManager,
        SignInManager<ApplicationUser> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }
    [BindProperty]
    21 references
    public InputModel Input { get; set; }

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

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

    [Required]
    [Display(Name = "First Name")]
    [StringLength(UserFirstNameMaxLength,
        MinimumLength = UserFirstNameMinLength)]
    4 references
    public string FirstName { get; set; }

    [Required]
    [Display(Name = "Last Name")]
    [StringLength(UserFirstNameMaxLength,
        MinimumLength = UserFirstNameMinLength)]
    4 references
    public string LastName { 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;
}
```

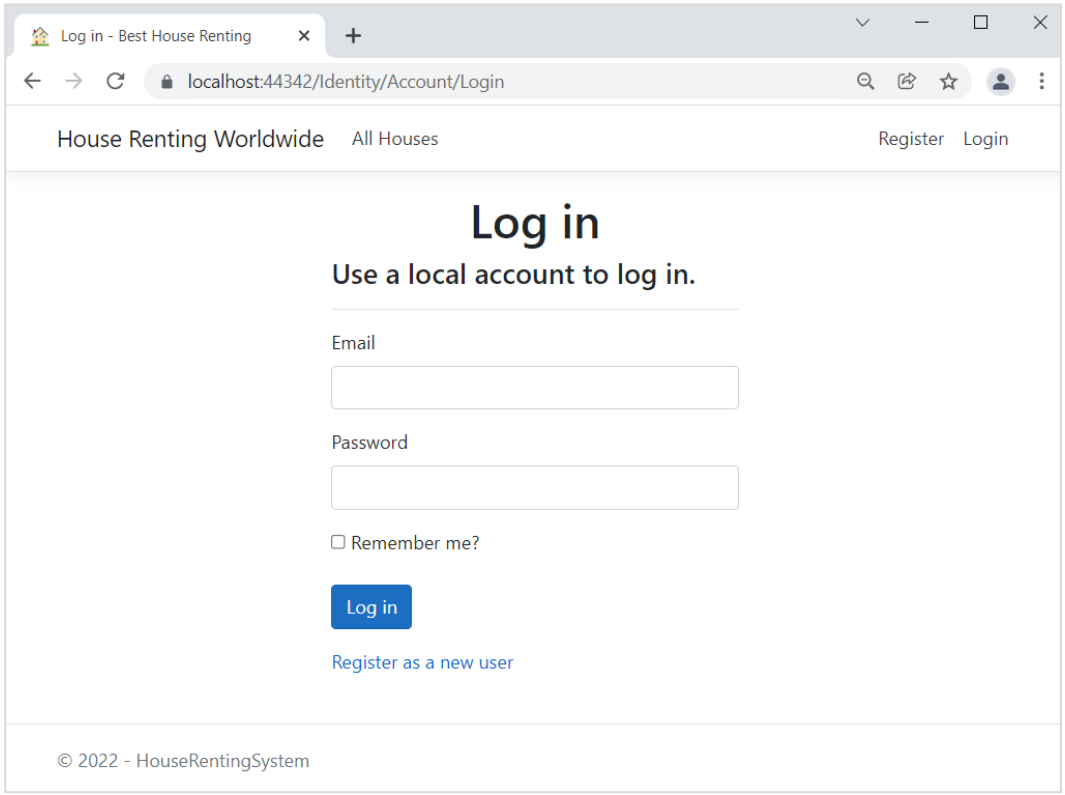
```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");

    if (ModelState.IsValid)
    {
        var user = new ApplicationUser
        {
            UserName = Input.Email,
            Email = Input.Email,
            FirstName = Input.FirstName,
            LastName = Input.LastName,
        };

        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();
}
```

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**. The registration should be **successful** and the user should **appear in the database**. They should also have a **first** and **last names**:

	Id	UserName	Email	FirstName	LastName
1	fa82d51c-c6b2-454f-aff8-2c6907a341ea	michael@mail.com	michael@mail.com	Michael	Klein

Modify the "Login" Page

It is time to **modify** the "**`Login`**" **page** as well in order to **clear it** from unnecessary code in its generated class. It 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 code should look like this:

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

<h1 class="text-center">@ViewData["Title"]</h1>
<div class="row">
    <div class="col-md-4 offset-md-4">
        <section>
            <form id="account" method="post">
                <h2>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>
                    <div class="checkbox">
                        <label asp-for="Input.RememberMe" class="form-label">
                            <input class="form-check-input" asp-for="Input.RememberMe" />
                            @Html.DisplayNameFor(m => m.Input.RememberMe)
                        </label>
                    </div>
                </div>
                <div>
                    <button id="login-submit" type="submit" class="w-100 btn btn-lg btn-primary">Log in</button>
                </div>
            </form>
        </section>
    </div>
</div>
```

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

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

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

    [BindProperty]
    13 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("~/");

        await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);

        ReturnUrl = returnUrl;
    }
```

```
    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**.

3. Display New User Data

As we now have **first and last names** of users, we can **display them** in different places to improve our app.

Step 1: Display User Names in the Navigation Bar

Let's use the **new user data** in the **welcoming** in our **navigation bar**. It should look like this:

However, **not all of our users** have **first and last names**. If a logged-in user **does not have the names**, their **username should be display**, as before:

We also want to **remove the link** for the "**My Profile**" **page** in the "**Hello, **" **message**.

If you want **all of your users to have first and last names** you can **delete and create the database again** or **edit them manually** through SSMS (but this is **bad practice** and you should **not** do it in real projects).

Let's **modify the navigation bar**. To do this, we will first **create a service** with a method to **return the user names as a string**. Create an "**ApplicationUser**" **folder** in "**Contracts**" and "**Services**" with an **IApplicationUserService interface** and a **ApplicationUserService class**:

The **IApplicationUserService** should **define** the following method:

```
public interface IApplicationUserService
{
    0 references
    Task<string> UserFullName(string userId);
}
```

The **UserService class** should **implement the above interface method**. When the **user has names**, they should be returned as a **string**. When they don't, **null should be returned**. Write the class like this:

```
public class ApplicationUserService : IApplicationUserService
{
    private readonly HouseRentingDbContext _data;

    0 references
    public ApplicationUserService(HouseRentingDbContext data)
    {
        _data = data;
    }

    1 reference
    public async Task<string> UserFullName(string userId)
    {
        var user = await _data.Users.FindAsync(userId);

        if(string.IsNullOrEmpty(user.FirstName)
            || string.IsNullOrEmpty(user.LastName))
        {
            return null;
        }

        return user.FirstName + " " + user.LastName;
    }
}
```

Add the service in **Program class** so that you will be able to use it:

```
builder.Services.AddTransient<IApplicationUserService, ApplicationUserService>();
```

As we want to **inject the service in a view**, we should go to the **_ViewImports.cshtml file** and **add the class namespace**:

_ViewImports.cshtml

```
@using HouseRentingSystem
@using HouseRentingSystem.Models
@using HouseRentingSystem.Models.Home
@using HouseRentingSystem.Models.House
@using HouseRentingSystem.Models.Agent
@using HouseRentingSystem.Services.House
@using HouseRentingSystem.Services.Agent;
@using HouseRentingSystem.Services.ApplicationUser;
@using HouseRentingSystem.Contracts.Agent;
@using HouseRentingSystem.Contracts.House;
@using HouseRentingSystem.Contracts.ApplicationUser;
@using HouseRentingSystem.Infrastructure;

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Now go to the "**_Login.cshtml" view** and let's **modify it**. **Inject IUserService** and **use the method** we created to check whether the **current user has names**. Depending on this, **display different messages**, which should **not contain** a link to the "**My Profile**" **page**. Do it like this:

_LoginPartial.cshtml

```
@using Microsoft.AspNetCore.Identity
@using HouseRentingSystem.Data.Models;
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager
@inject IAgentService agents
@inject IApplicationUserService user
```

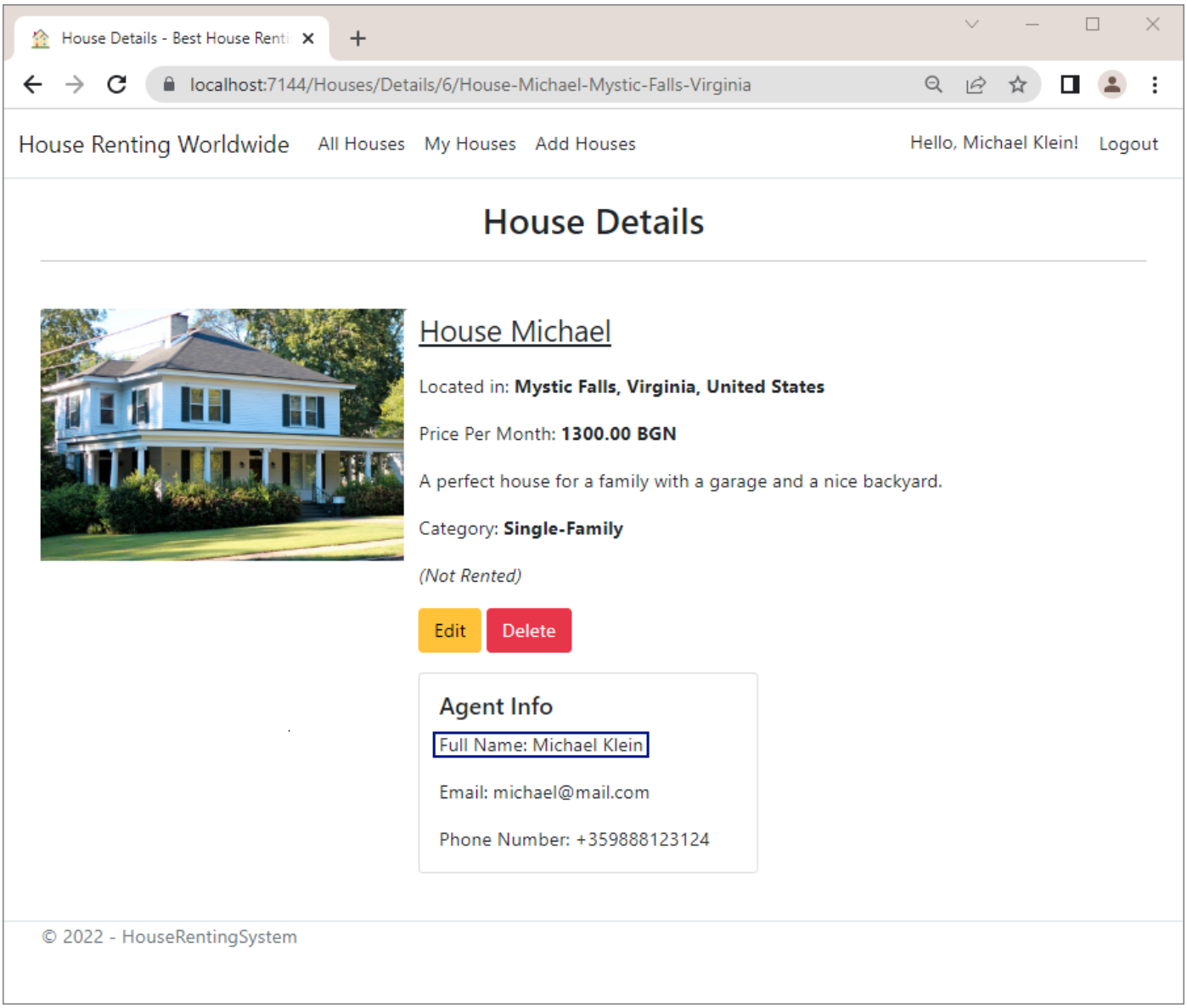
```
<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User))
    {
        @if (await agents.ExistsById(User.Id()) == false)...
        <li class="nav-item">
            @if(await user.UserFullName(User.Id()) is var fullName && fullName != null)
            {
                <a class="nav-link text-dark">Hello, @fullName!</a>
            }
            else
            {
                <a class="nav-link text-dark">Hello, @User.Identity.Name!</a>
            }
        </li>
```

Now **run the app** and make sure that the **navigation bar displays the correct message** – with **first and last names** or with **username**, depending on **the available user data**.

Step 2: Display Agent Names

As you know, on the "**`Details`**" **page** of every house we **display the house's agent data**. Until now we displayed only the **email and phone number**, but now we will also display the **agent names** (if they are **present**).

For example, if our **new user** "**Michael Klein**" becomes an **agent** and **adds a house**, the house's "**`Details`**" **page** will be the following:



First, to **display the names** we need to add a **`FullName` property** to the **`AgentServiceModel`**:

```
public class AgentServiceModel
{
    0 references
    public string FullName { get; set; } = null!;
    1 reference
    public string Email { get; set; } = null!;
    1 reference
    public string PhoneNumber { get; set; } = null!;
}
```

Next, we need to **modify the service model**, which builds the **`HouseDetailsServiceModel`** for the "**`Details`**" **page**. Use the **`IUserService`** to **get the full name of the agent**. To do this, you should first **inject the service** in the **`HouseService` class** like this:

```
public class HouseService : IHouseService
{
    private readonly HouseRentingDbContext _data;
    private readonly IApplicationUserService _user;

    0 references
    public HouseService(HouseRentingDbContext data,
        IApplicationUserService user)
    {
        _data = data;
        _user = user;
    }
```

Now modify the **`HouseDetailsById(int id)` method** to **use the `UserFullName(…)` service method** for the **`FullName` property** of the **`AgentServiceModel`**:

```
    public async Task<HouseDetailsServiceModel> HouseDetailsById(i
    {
        return await _data
            .Houses
            .Where(h => h.Id == id)
            .Select(h => new HouseDetailsServiceModel()
            {
                Id = h.Id,
                Title = h.Title,
                Address = h.Address,
                Description = h.Description,
                ImageUrl = h.ImageUrl,
                PricePerMonth = h.PricePerMonth,
                IsRented = h.RenterId != null,
                Category = h.Category.Name,
                Agent = new AgentServiceModel()
                {
                    FullName = _user.UserFullName(h.Agent.UserId).ToString(),
                    PhoneNumber = h.Agent.PhoneNumber,
                    Email = h.Agent.User.Email,
                }
            })
            .FirstOrDefaultAsync();
    }
```

At the end, **modify** the "**`Details.cshtml`**" **view** to **display the names** if the **`FullName` property is not `NULL`**:

```
<div class="card" style="width: 18rem;">
    <div class="card-body">
        <h5 class="card-title">Agent Info</h5>
        @if(Model.Agent.FullName != null)
        {
            <p class="card-text">Full Name: @Model.Agent.FullName</p>
        }
        <p class="card-text">Email:  @Model.Agent.Email</p>
        <p class="card-text">Phone Number: @Model.Agent.PhoneNumber</p>
    </div>
</div>
```

Now look at the "**Details**" **page** and make sure the **names are displayed** when the user has them.

4. Seed Administrator

In this task we will see how to **seed an administrator** in our database. Before that, however, let's **create a class with a constant** for the **administrator role name and email**, as we are going to need it. Create the **AdminUser class** and add the **constants** like this:

```
public class AdminUser
{
    public const string AdminRoleName = "Administrator";
    public const string AdminEmail = "admin@mail.com";
}
```

Now **create the Admin** as an **ApplicationUser** in the **HouseRentingDbContext** class, as it is responsible for the database. Also, the **Admin** should be explicitly an **agent**, because the **Admin** now **has no phone number**. This is a problem, as when the **Admin creates a house**, there will be **no phone number to be shown** in the "**Agent Info**" **section**. In addition, it would be **easier to give access** to the **Admin**, when they are an agent.

For the above reasons, let's **seed the Admin as a user and an agent in the database**. Note that the new **Agent record** should have a **unique id** – there should **not** be an **Agent** with the same **id** in the database already. Do it like this:

```
public class HouseRentingDbContext : IdentityDbContext<ApplicationUser>
{
    1 reference
    private ApplicationUser AdminUser { get; set; }
    1 reference
    private Agent AdminAgent { get; set; }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        SeedUsers();
        builder.Entity<ApplicationUser>()
                .HasData(AgentUser,
                GuestUser,
                AdminUser);

        SeedAgent();
        builder.Entity<Agent>()
                .HasData(Agent,
                AdminAgent);
```

You can use the following code in the **SeedUsers()** method:

```
AdminUser = new ApplicationUser()
{
      Id = "bcb4f072-ecca-43c9-ab26-c060c6f364e4",
      Email = AdminEmail,
      NormalizedEmail = AdminEmail,
      UserName = AdminEmail,
```

```
      NormalizedUserName = AdminEmail,
      FirstName = "Great",
      LastName = "Admin"
};

AdminUser.PasswordHash = hasher.HashPassword(AgentUser, "admin123");
```

Use this code in the **SeedAgent()** method:

```
AdminAgent = new Agent()
{
      Id = 5,
      PhoneNumber = "+359123456789",
      UserId = AdminUser.Id
};
```

Be careful with the value of the id – check which is the last id in your "**Agents**" table.

As we **changed the db context** we should now **migrate the changes to the database**. Open the **Package Manager Console** in **Visual Studio** and **add a migration** to the "**Data/Migrations**" **folder** like this:

```
Add-Migration AddedAdmin -o Data/Migrations
```

The **migration should be created** and you should apply the changes to the database using:

```
Update-Database
```

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.UpdateData(
        table: "AspNetUsers",
        keyColumn: "Id",
        keyValue: "6d5800ce-d726-4fc8-83d9-d6b3ac1f591e",
        columns: new[] { "ConcurrencyStamp", "PasswordHash", "SecurityStamp" },
        values: new object[] { "bee63644-c0fb-458b-a9c0-df6c040e04b3", "AQAAAAE

    migrationBuilder.UpdateData(
        table: "AspNetUsers",
        keyColumn: "Id",
        keyValue: "dea12856-c198-4129-b3f3-b893d8395082",
        columns: new[] { "ConcurrencyStamp", "PasswordHash", "SecurityStamp" },
        values: new object[] { "8551d5e6-dedf-4c04-8e4d-f18a48892a20", "AQAAAAE

    migrationBuilder.InsertData(
        table: "AspNetUsers",
        columns: new[] { "Id", "AccessFailedCount", "ConcurrencyStamp", "Email"
        values: new object[] { "bcb4f072-ecca-43c9-ab26-c060c6f364e4", 0, "b253

    migrationBuilder.InsertData(
        table: "Agents",
        columns: new[] { "Id", "PhoneNumber", "UserId" },
        values: new object[] { 6, "+359123456789", "bcb4f072-ecca-43c9-ab26-c06
}
```

Note that some **user data is changed** as it is **not hardcoded** when the seeding is done, but this is **not a problem**, as we do not have dependencies on the changed columns.

Now **run the app** and **log in** with the **Admin**. You should see that they **can now add houses**:

Open **SSMS** and look at the "**AspNetUsers**" table for our new **Admin user**:

	Id	UserName
1	bcb4f072-ecca-43c9-ab26-c060c6f364e4	admin@mail.com

Now look at the "**Agents**" **table**. There should be the **new agent**, who is also our **Admin**:

	Id	PhoneNumber	UserId
1	6	+359123456789	bcb4f072-ecca-43c9-ab26-c060c6f364e4

5. Add Role for Admin

In this task we will see how to **create a role for our Admin**, as roles will help us **restrict and give them special access** easier. To **add a user in a role**, we will need **RoleManager**. To access it, we should go **Program class** and add **role-related services** to our app like this:

```
builder.Services.AddDefaultIdentity<ApplicationUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = false;
    options.Password.RequireDigit = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = false;
})
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<HouseRentingDbContext>();
```

Our **roles** will be of type **IdentityRole**, which is the **default for ASP.NET Core**.

To **create a role with a user of that role**, we will need to use the **UserManager** and **RoleManager services** (which come from ASP.NET). For this reason, we will **not seed the user** in the **HouseRentingDbContext class** (as it does not use services), but we will **create a separate class**.

Create the **ApplicationBuilderExtensions class** in the "**Infrastructure**" **folder**. This class should **extend** the **IApplicationBuilder**, as it has **access to the app services** we need. The class should look like this:

```
public class ApplicationBuilderExtensions
{
}
```

The **ApplicationBuilderExtensions class** will have a single method for **seeding an administrator**. To **extend the IApplicationBuilder**, the method should **accept it as a parameter**, **modify it** and then **return it**:

```
public static class ApplicationBuilderExtensions
{
    0 references
    public static IApplicationBuilder SeedAdmin(this IApplicationBuilder app)
    {
```

The **app services** we need are **scoped services**. For this reason, we **cannot just use them directly** but we need to **manually create a scope** for them. The **scope is a code block**, in which a **service exists**. **Create a scope** like this:

```
    using var scopedServices = app.ApplicationServices.CreateScope();
```

Then, **get your services** through the **ServiceProvider** like this:

```
    var services = scopedServices.ServiceProvider;

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

Now we can **use the services to create an admin role** and then a **user in that role**. The **`RoleManager`** and **`UserManager`** have **only asynchronous methods**, so we need to **create a `Task`**, which will be **awaited** to finish its execution:

```
    Task
        .Run(async () =>
        {
```

Now check if the **administrator role exists**. If it does, it means that the **`Admin` is added to the new role** and we **should not add it again**:

```
            if (await roleManager.RoleExistsAsync(AdminRoleName))
            {
                return;
            }
```

It the **role does not exist**, create it as an **`IdentityRole`**:

```
            var role = new IdentityRole { Name = AdminRoleName };

            await roleManager.CreateAsync(role);
```

Then, **get the Admin user** like this:

```
            var admin = await userManager.FindByNameAsync(AdminEmail);
```

Finally, **add the user** to the **new role** and **return the `IApplicationBuilder`** like this:

```
            await userManager.AddToRoleAsync(admin, role.Name);
        })
        .GetAwaiter()
        .GetResult();

    return app;
}
```

At the end, we should **invoke the above extension method** in **`Program` class** like this:

```
app.SeedAdmin();
```

Now **run the app** and try to **log in with the `Admin` credentials**. The login should be sucessful.

Look at the "**`AspNetRoles`**" **table**, which should have the **new `IdentityRole`** we created:

	Id	Name	NormalizedName
1	8c083475-590b-46f3-b2fa-5439e0199db7	Administrator	ADMINISTRATOR

6. Modify Admin Access

For now, our **`Admin` has the access of an ordinary user**. Now we will **modify our app**, so that we give the **`Admin` more privileges**. However, let's first summarize what the **`Admin` can** and **cannot do**:

- **`Admin`** can **add houses** and **edit** and **delete** not only their houses, but these of other users, too
- **`Admin`** can **rent houses**, which are **not rented already**
- **`Admin`** can **leave a house**, which is **rented only by them**. They cannot leave houses, which are rented by another user

- **Admin** is already an agent, so they **cannot become agents** again

In addition, the "**My Houses**" **page** will **show the created houses** of the **Admin** for now. Later, we will create a **separate page** for the **Admin** to see the **houses they added** and those that **they rented**.

To begin with, we need to **create a method**, which **returns a bool whether the current user is the Admin or not**, so that we can make **validations**. To do this, we need to **check the role of the current user**. Go to the **ClaimsPrincipalExtensions class** and **add the following method**, which **extends ClaimsPrincipal**:

```
public static class ClaimsPrincipalExtentions
{
    0 references
    public static bool IsAdmin(this ClaimsPrincipal user)
    {
        return user.IsInRole(AdminRoleName);
    }
```

Now use it in the **HouseController** to **give special access** to pages when the **current user is the Admin**. As the **Admin is an agent**, we do not need to make modifications to the functionality for adding a house (it is already working). However, we should **modify the** "**edit**" and "**delete**" **functionalities** to allow the **Admin** to **edit and delete all houses**. Do it like this:

```
[Authorize]
2 references
public class HouseController : Controller
{

    public async Task<IActionResult> Edit(int id)
    {
        if (await _houses.Exists(id) == false)
        {
            return BadRequest();
        }

        if (await _houses.HasAgentWithId(id, this.User.Id()) == false
            && User.IsAdmin() == false)
        {
            return Unauthorized();
        }

    [HttpPost]
    0 references
    public async Task<IActionResult> Edit(int id, HouseFormModel house)
    {
        if (await _houses.Exists(id) == false)
        {
            return View();
        }

        if (await _houses.HasAgentWithId(id, User.Id()) == false
            && User.IsAdmin() == false)
        {
            return Unauthorized();
        }
```

```
public async Task<IActionResult> Delete(int id)
{
    if(await _houses.Exists(id) == false)
    {
        return BadRequest();
    }

    if(await _houses.HasAgentWithId(id, User.Id())
        && User.IsAdmin() == false)
    {
        return Unauthorized();
    }
```

```
[HttpPost]
0 references
public async Task<IActionResult> Delete(HouseDetailsView
{

    if(await _houses.Exists(house.Id))
    {
        return BadRequest();
    }

    if(await _houses.HasAgentWithId(house.Id, User.Id())
        && User.IsAdmin() == false)
    {
        return Unauthorized();
    }
```

Modify the **`Rent()` method**, too. The **`Admin`** should be able to **rent free houses**. Do it like this:

```
[HttpPost]
0 references
public async Task<IActionResult> Rent(int id)
{
    if(await _houses.Exists(id))
    {
        return BadRequest();
    }

    if(await _agents.ExistsById(User.Id())
        && User.IsAdmin() == false)
    {
        return Unauthorized();
    }
```

Now we should only **modify the views** to **display the buttons correctly**. The pages we should **modify** are the "**`All Houses`**", "**`My Houses`**" and "**`Details`**" **pages**. First, go to the "**`_HousePartial.cshtml`" view file**, as it is responsible for the first two pages. It should look like this when **changed**:

_HousePartial.cshtml

```
@model HouseServiceModel
@inject IHouseService houses
@inject IAgentService agents

<div class="col-md-4">
    <div class="card mb-3">
        <img class="card-img-top" src="@Model.ImageUrl" alt="House Image">
        <div class="card-body text-center">
            <h4>@Model.Title</h4>
            <h6>Address: <b>@Model.Address</b></h6>
            <h6>...
            <h6>(@(Model.IsRented ? "Rented" : "Not Rented"))</h6>
            <br />
            <a asp-controller="House" asp-action="Details" asp-route-id="@Model.Id"
               asp-asp-route-information="@Model.GetInformation()" class="btn btn-success">Details</a>
            @if (User.Identity.IsAuthenticated)
            {
                @if (await houses.HasAgentWithId(Model.Id, User.Id())
                    || User.IsAdmin())...
                <p></p>
                @if (!Model.IsRented && await agents.ExistsById(User.Id()) == false
                    || User.IsAdmin())...
                else if (await houses.IsRentedByUserWithId(Model.Id, User.Id()))...
            }
        </div>
    </div>
</div>
```

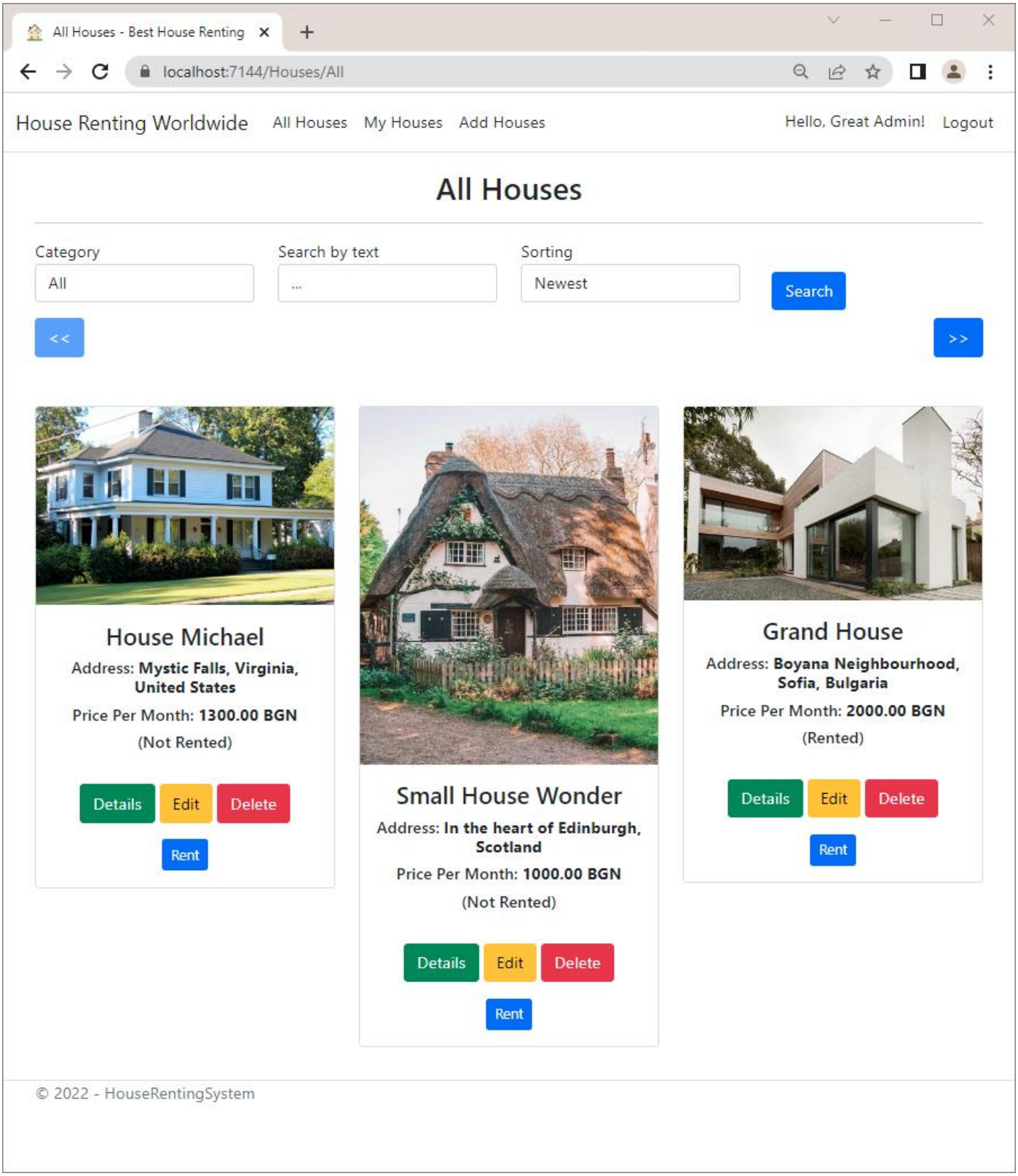
Now **modify the `Details.cshtml` view** in the same way:

Details.cshtml

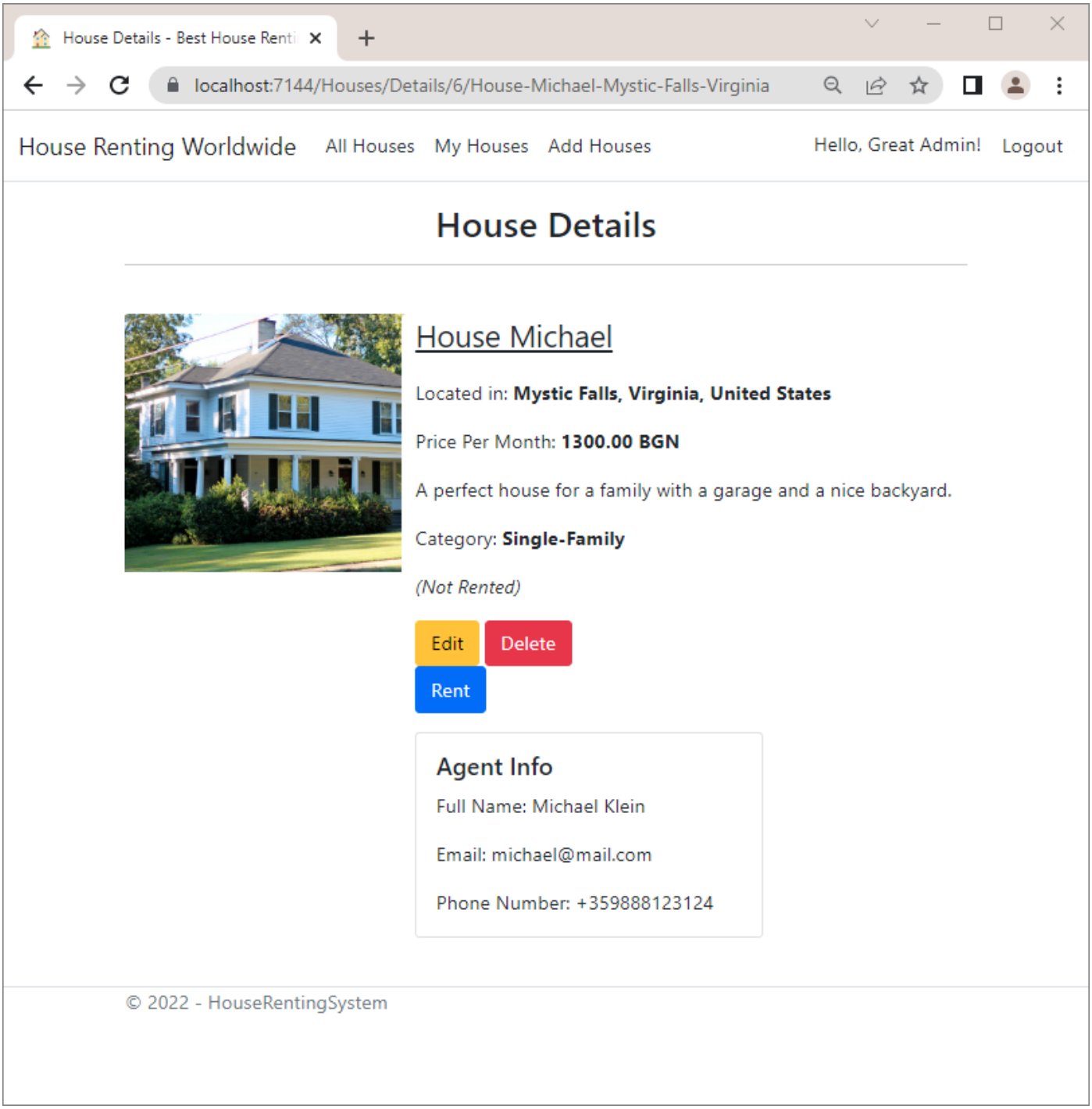
```
@model HouseDetailsServiceModel
@inject IHouseService houses
@inject IAgentService agents

@{
    ViewBag.Title = "House Details";
}

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

<div class="container" style="display:inline">
    <div class="row">
        <div class="col-4">
            <img class="card-img-top" style="width: 20rem;"
                 src="@Model.ImageUrl" alt="House Image">
        </div>
        <div class="card col-8 border-0">
            <p style="font-size:25px;"><u>@Model.Title</u></p>
            <p>Located in: <b>@Model.Address</b></p>
            <p>...
            <p>@Model.Description</p>
            <p>Category: <b>@Model.Category</b></p>
            <p><i>(@(Model.IsRented ? "Rented" : "Not Rented"))</i></p>
            <div class="form-inline">
                @if (User.Identity.IsAuthenticated)
                {
                    @if (await houses.HasAgentWithId(Model.Id, User.Id())
                        || User.IsAdmin())...
                    @if (!Model.IsRented && await agents.ExistsById(User.Id()) == false
                        || User.IsAdmin())...
                    else if (await houses.IsRentedByUserWithId(Model.Id, User.Id()))...
                }
            </div>
            <p></p>
            <div class="card" style="width: 18rem;">...
        </div>
    </div>
</div>
```

Be **careful with the use of brackets** on the above screenshots!

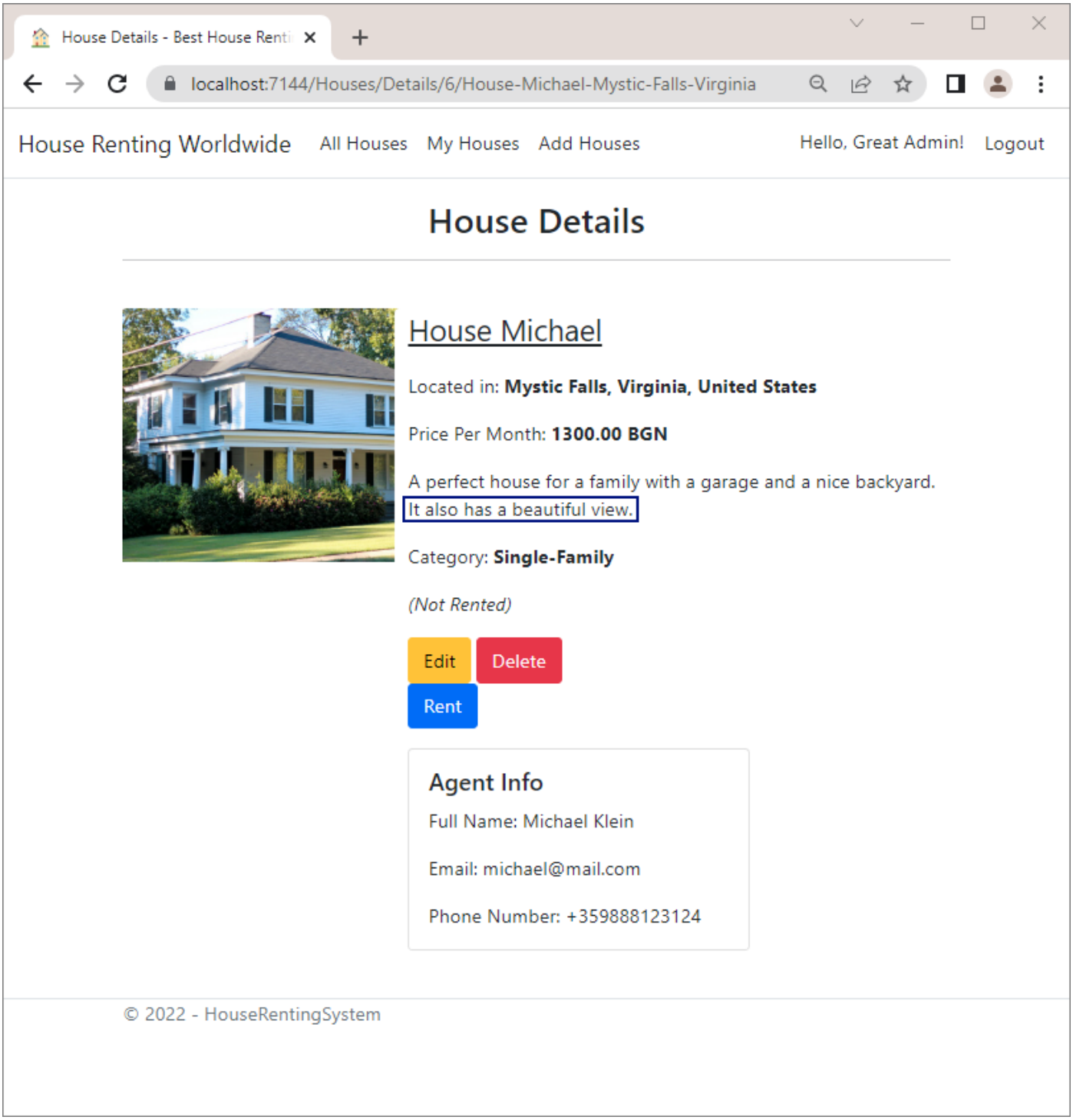
Run the app and **log in with the Admin credentials**. Go to the "**All Houses**" and you should see the **[Details]**, **[Edit]** and **[Delete] buttons** on **each house**. You should also see the **[Rent] button** on houses, which are **not rented**:



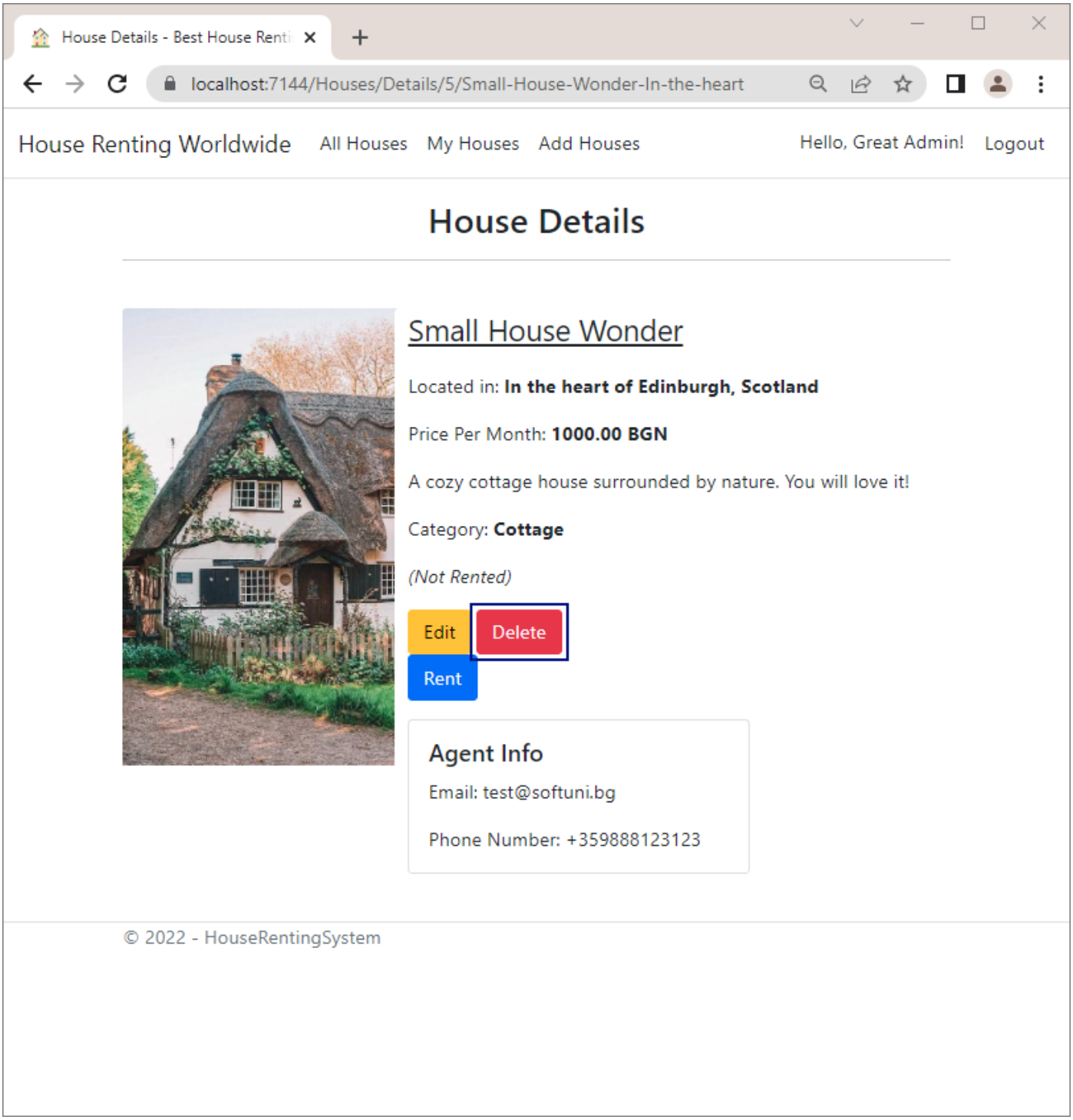
Look at the "**Details**" **page** of **any house** and make sure the **buttons are present again**:

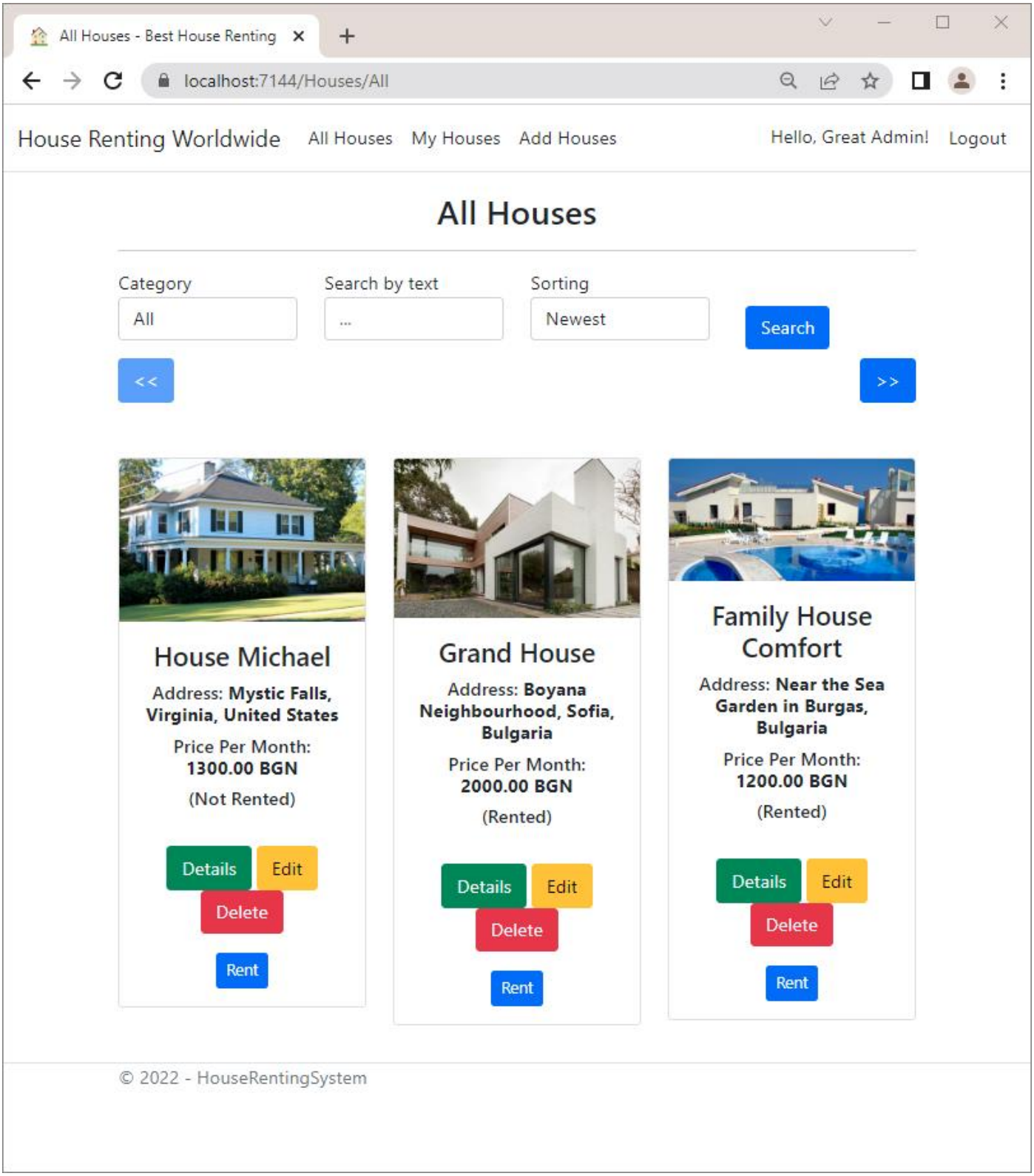


Try to edit a house of **another user**. It should be successful:

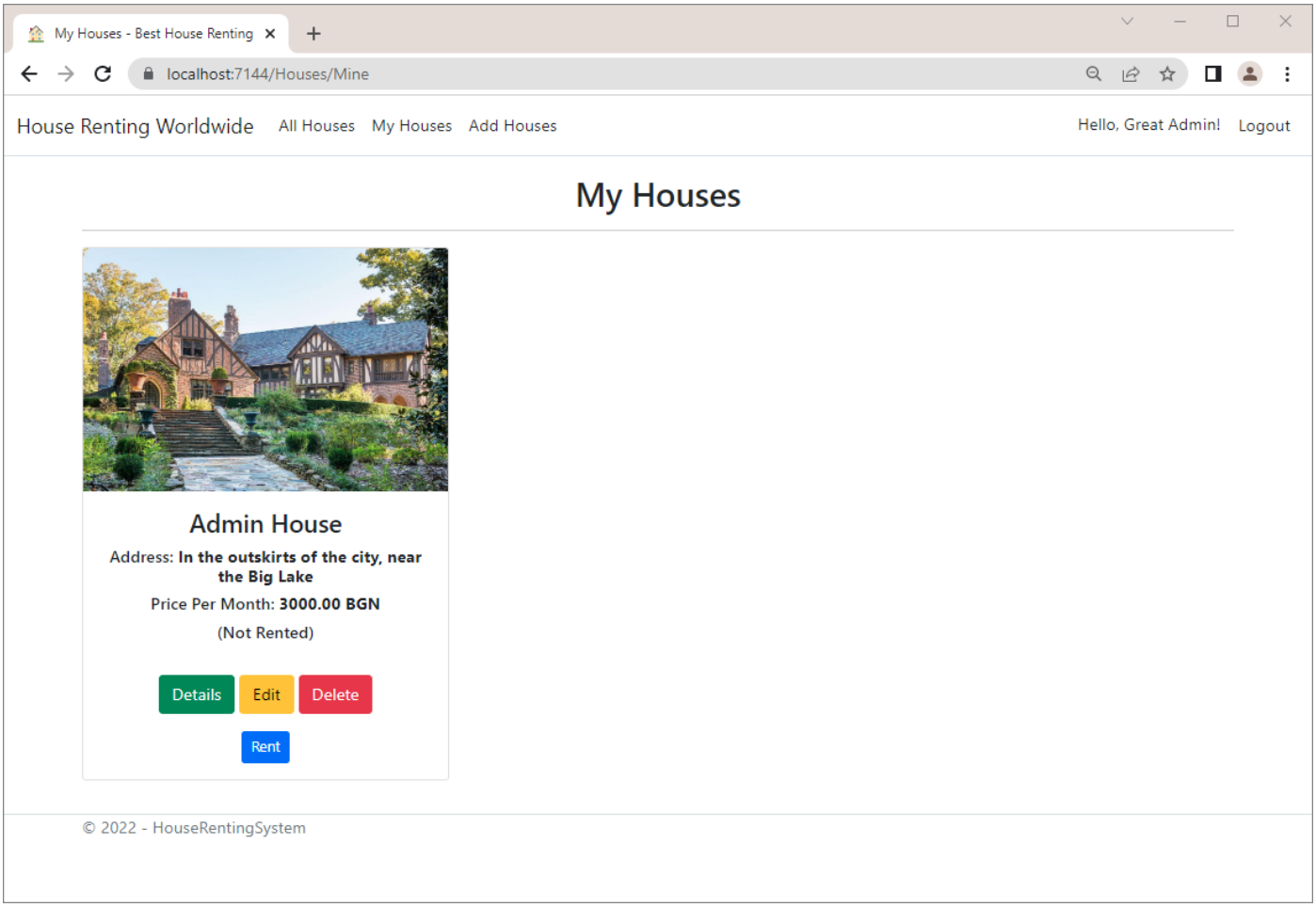


Now try to **delete a house of another user**. The deletion should be successful, too:





You can also **try to rent and then leave a house**, which is **not already rented**. Then, **create a new house** from the **`Admin` profile** and make sure that it is **created successfully** and **displayed on the "`My Houses`" page**:



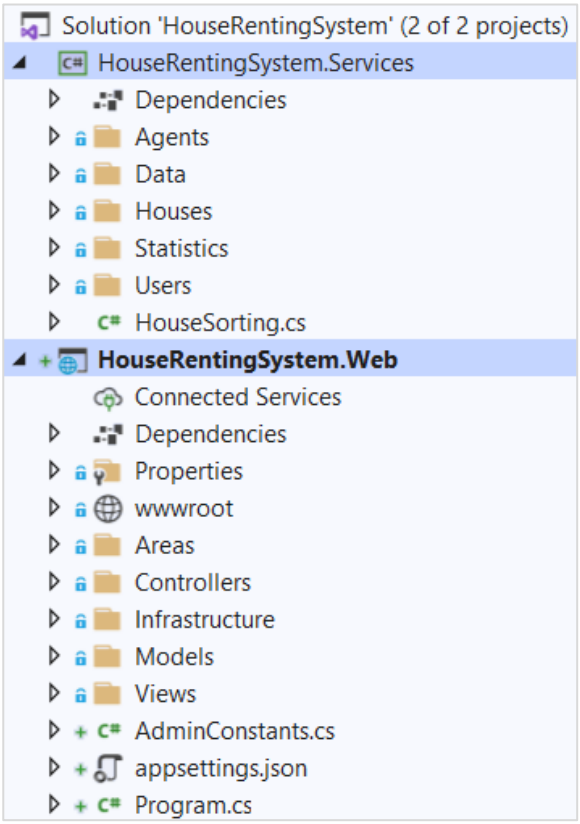
7. Separate to Multiple Projects

In this task, we will **move the service and data layer classes** to **separate projects** of type **class library**. A **class library** defines types and methods that are called by an application. In this way, we will **separate the different layers** of our app and **improve the architecture**.

In this case, we will have the following projects:

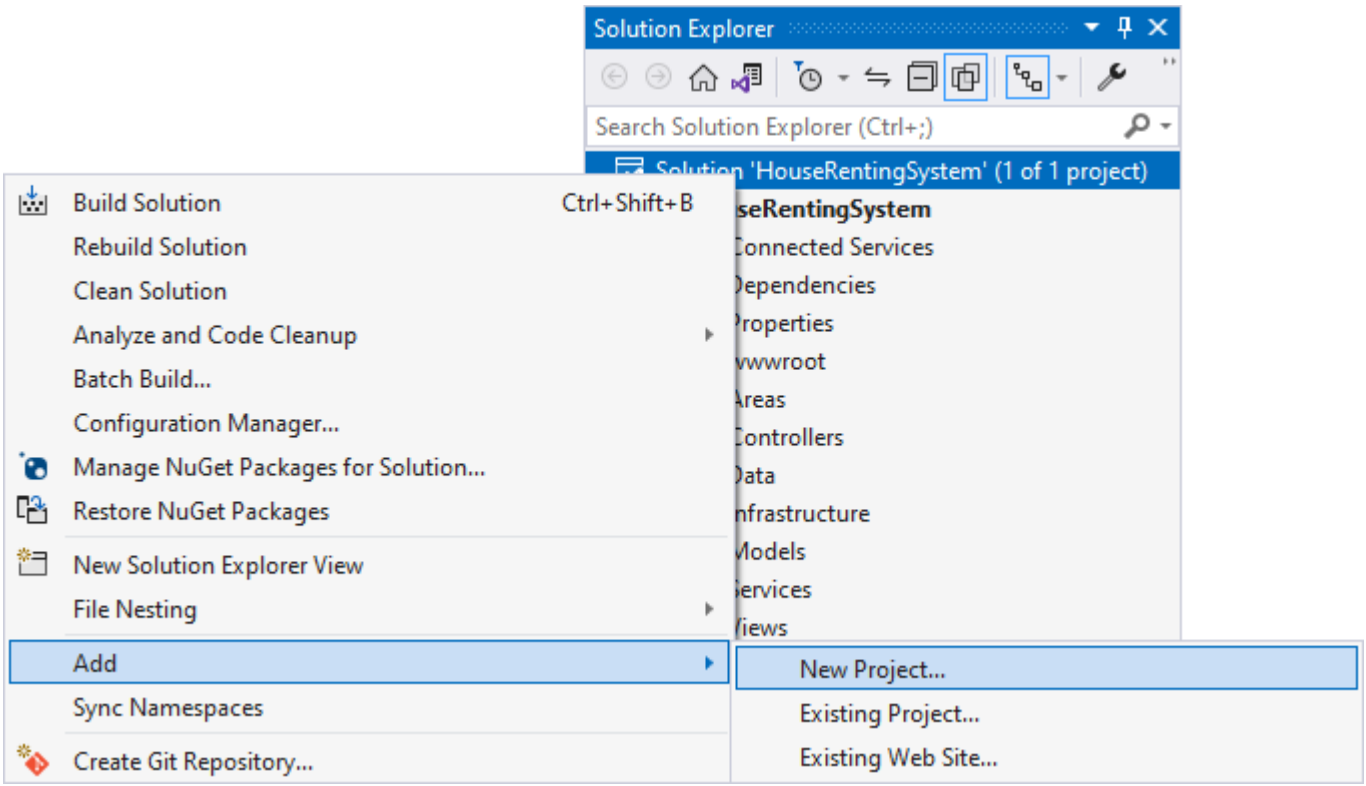
- "**`HouseRentingSystem.Core`**" project, which includes and **service-related classes**
- "**`HouseRentingSystem.Data`**" project, which includes and **data-related classes**
- "**`HouseRentingSystem.Web`**" project, which contains the **web part** with the **controllers** and **references the** "**`HouseRentingSystem.Services`**" **project**

When done, the solution will look like this:

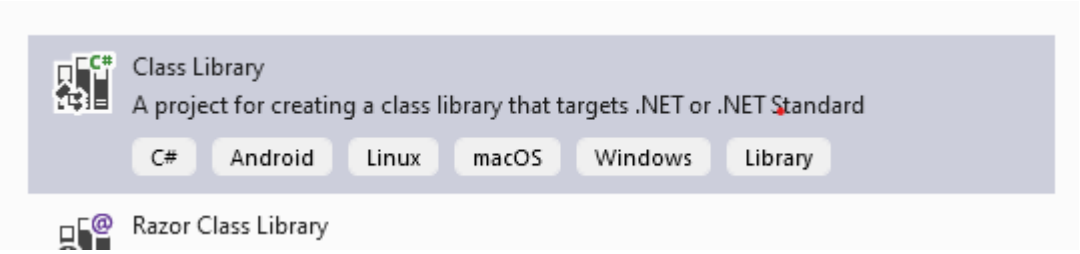


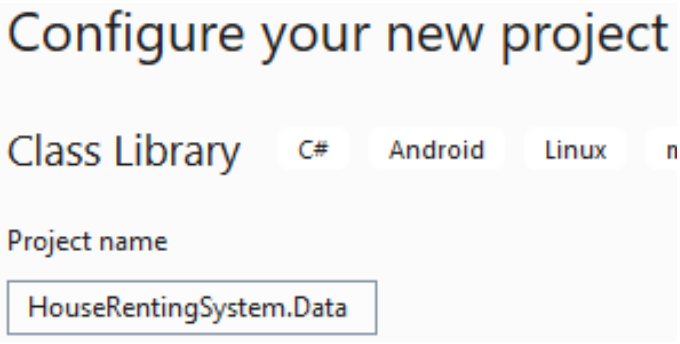
Step 1: Create a Services + Data Projects

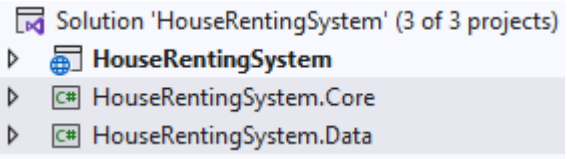
First, we will separate our **services and data layers** from the main project, as they do **not have dependencies** on other classes, but depend only on each other. We will move them to **class library** projects. To **create a class library**, **right-click** on the "**`HouseRentingSystem`**" **solution** and **choose `[Add]` → `[New Project]`**:



Next, **choose** the "**`Class library`**" **project template** and **name the project** "**`HouseRentingSystem.Services`**":



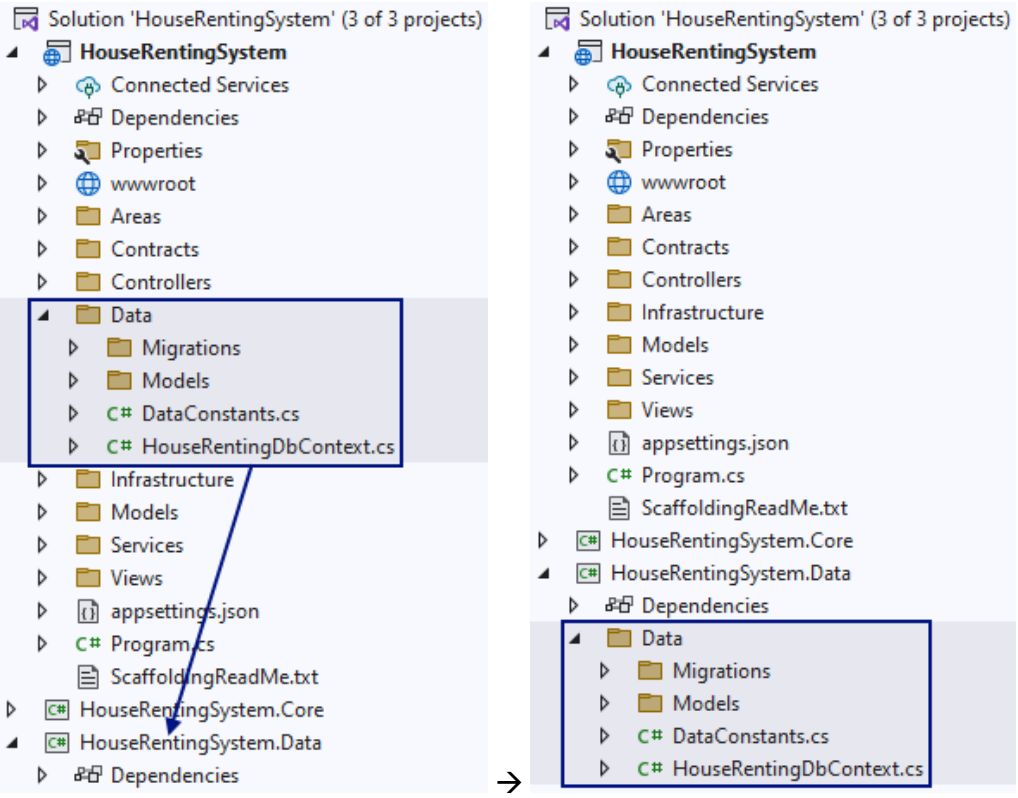




The **projects are created** and you can **delete the `Class1.cs` files**, as we won't need them.

Move the Data Layer Classes

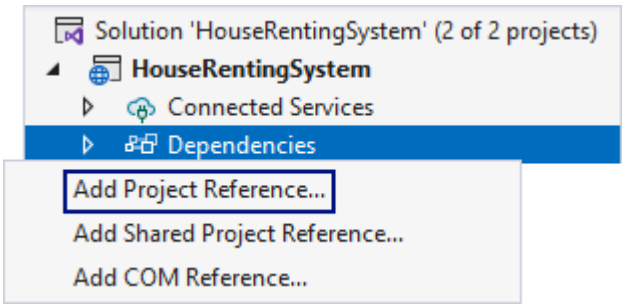
Now let's **move the "`Data`" folder** with its classes from the "**`HouseRentingSystem`**" **project** to the "**`HouseRentingSystem.Data`**" **project**:



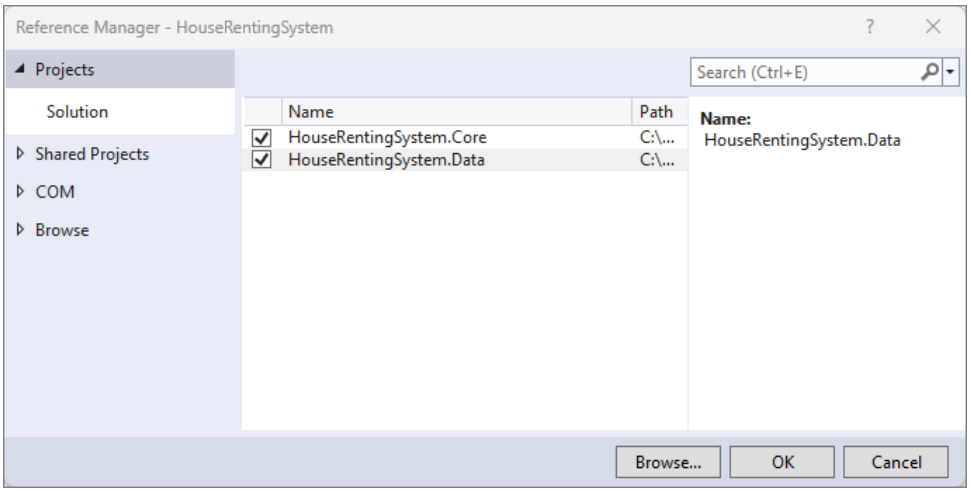
Build the project and note that there are **many errors** that we need to fix.

First, let's make the "**`HouseRentingSystem`**" **project depend** on the new "**`HouseRentingSystem.Core`**" and "**`HouseRentingSystem.Data`**"**project**, as it **needs the data classes**. To do this, **right-click** on **`[Dependencies]`** →

[Add Project Reference] and **select the** "**HouseRentingSystem.Core**" and "**HouseRentingSystem.Data**" **projects** like this:



→



Build the project again. The **errors** should be less.

However, note that all our **classes** from the "**HouseRentingSystem.Data**" **project** are from the "**HouseRentingSystem.Data.Data**" **namespace**. We should **change the namespaces** to "**HouseRentingSystem.Data.Data**" to **match the current project architecture**.

Go to **all classes** and **modify their namespace** to be the following:

```
namespace HouseRentingSystem.Data.Data
```

Note that you should be careful with the **namespaces of classes**, which are in a **subfolder**. For example, the **classes in the** "**Entities**" **folder** should have the **following namespace**:

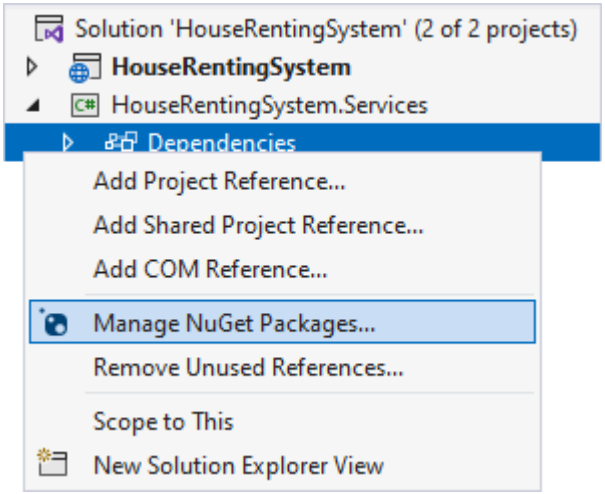
```
namespace HouseRentingSystem.Data.Data.Models
```

The "**Migrations**" **folder classes** should be part of this **namespace**:

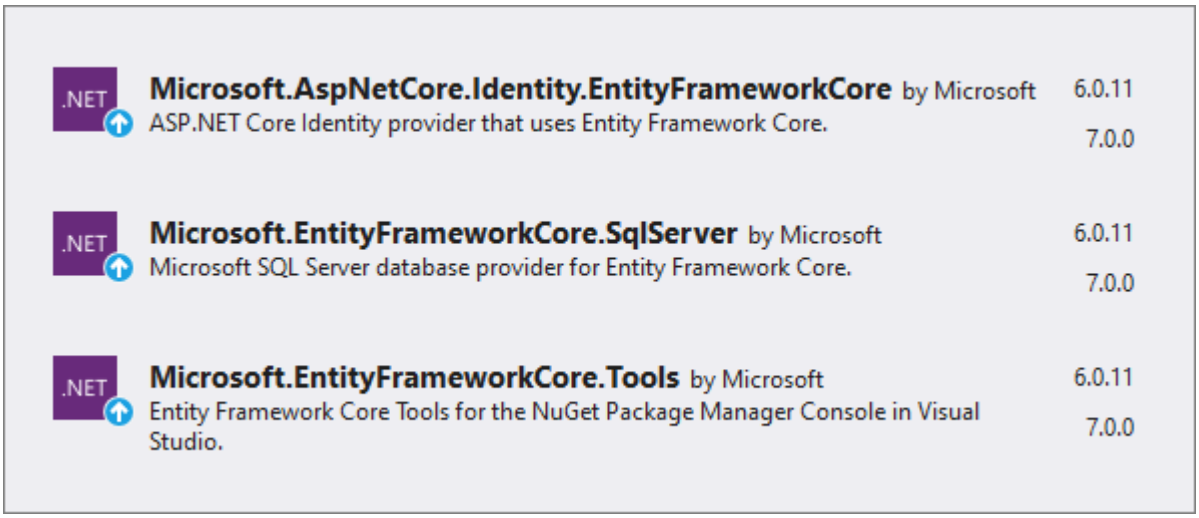
```
namespace HouseRentingSystem.Data.Data.Migrations
```

You should also **change the references to the classes** with the modified namespace. When you **build the project**, you should have **no errors**, **related to the namespaces**.

Now let's go to **NuGet Package Manager** and **let's add the packages that** the "**HouseRentingSystem.Services**" **project** needs:



Download the **packages** below but be careful with the **package versions**.



You can **delete the above packages** from the "**`HouseRentingSystem`**" **project**, as we do not need them anymore.

Build the project again. Make sure that there are **no more errors** in any of the projects.

Move the Service Layer Classes

Now we should also **move the classes and interfaces from the** "**`Services`**" and "**`Contracts`**" **folders** of the "**`HouseRentingSystem`**" to the "**`HouseRentingSystem.Core`**" **project**:

We should modify the namespaces from **`HouseRentingSystem`** to **`HouseRentingSystem.Core`**.

Don't forget to also **change the references to the classes** with the modified namespace.

We should also move the service models to the **`HouseRentingSystem.Core`** project as services depend on them. In order to do that, create a new folder named "**`Models`**" in the services layer project and inside it, create folders "**`Agent`**", "**`House`**" and "**`Statistic`**". Move the service model classes to the relevant folders and change their namespaces and modify the references to them from the other classes and projects.

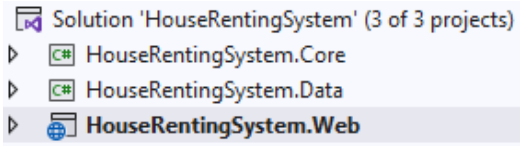
Note that all **errors** that have left are because of the **`HouseSorting` class**, because it is used in both projects, but the **new project doesn't depend on the old one**, as then we will have a **circular dependency**, which is **not allowed**. For this reason, let's **move the `HouseSorting` class** to the "**`HouseRentingSystem.Core`**" **project** and **change its namespace**.

Rename the used namespace everywhere where necessary to **clear all errors**. If you have **any other problems**, **delete all** "**`bin`**" **and** "**`obj`**" **folders** in the projects and **rebuild the solution**.

Step 2: Rename the Web Project

The "**HouseRentingSystem**" is our **web project**. For this reason, it is a good idea to **rename it**, so that it is clearer what it contains.

To do this, **right-click** on the "**HouseRentingSystem**" **project → [Rename]** and **set the name** to be "**HouseRentingSystem.Web**":



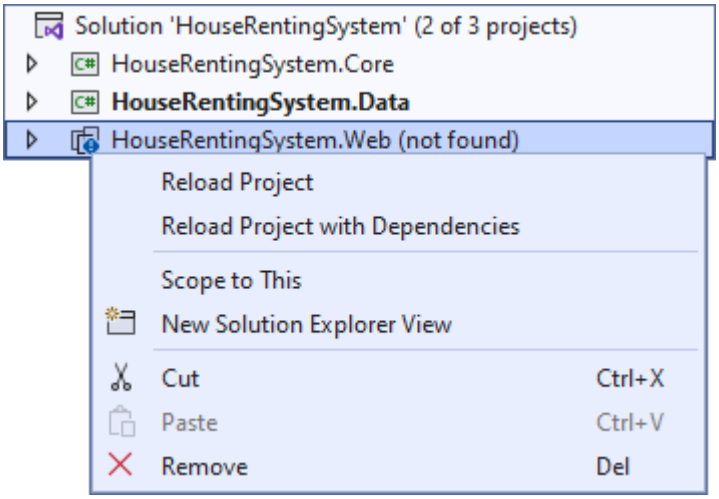
Then, **change the namespace of all classes i**n the project to be relevant to the new project name.

Do it for **all classes**. Also, **change the namespace references** to be correct. **Build the solution** and make sure that there are **no errors**.

Next, close **Visual Studio** and **navigate to the solution folder** in **File Explorer**.

Then, **rename the** "**HouseRentingSystem**" **folder** to "**HouseRentingSystem.Web**".

Open the solution in **Visual Studio** again. Now the **solution file does not recognize the** "**HouseRentingSystem.Web**" **project**, so let's **delete it** and **add it** again:



Right-click on the HouseRentinSystem solution and choose **[Add] → [Existing Project]**.

Now **run the project** and make sure that **everything works correctly** as before the separation to projects:

8. Add AutoMapper

AutoMapper is a **simple little library**, which will help us get rid of code, which **maps one object to another**. With that mapper tool, we won't do the mappings manually, which will **improve our code** and **prevent mistakes**.

Now, we should register the **AutoMapper service** in the "**HouseRentingSystem.Web**" **project**. We should **install the AutoMapper.Extensions.Microsoft.DependencyInjection NuGet package**.

Go to the **Program class** and **add the service** with the **assemblies of the service and controller classes** like this:

```
builder.Services.AddAutoMapper(
    typeof(IHouseService).Assembly,
    typeof(HomeController).Assembly);
```

Now **create classes**, in which we will **create the mappings**. They should **inherit the `Profile` class**. Create one for the **controller** and one for the **service methods mappings** in the corresponding projects.

The **`ServiceMappingProfile` class** should be in a **folder** "**`Infrastructure`**" in the "**`HouseRentingSystem.Core`** **project**:

```
public class ServiceMappingProfile : Profile
{
}
```

The **`ControllerMappingProfile` class** should be in a **folder** "**`Infrastructure`**" in the "**`HouseRentingSystem.Web`** **project**:

```
public class ControllerMappingProfile : Profile
{
}
```

Step 1: Mapping in Controller Methods

Go to the "**`HouseRentingSystem.Web`** **project** and **examine the controller methods** for manual mapping between objects. In our case, we have such **mappings in the `Edit(int id)` and `Delete(int id)` methods** of the **`HouseController`**:

```
public class HouseController : Controller
{
    public async Task<IActionResult> Edit(int id)
    {
        if (await _houses.Exists(id) == false)...

        if (await _houses.HasAgentWithId(id, this.User.Id()) == false
            && User.IsAdmin() == false)...

        var house = await _houses.HouseDetailsById(id);

        var houseCategoryId = await _houses.GetHouseCategoryId(house.Id);

        var houseModel = new HouseFormModel()
        {
            Title = house.Title,
            Address = house.Address,
            Description = house.Description,
            ImageUrl = house.ImageUrl,
            PricePerMonth = house.PricePerMonth,
            CategoryId = houseCategoryId,
            Categories = await _houses.AllCategories()
        };

        return View(houseModel);
    }
```

```
public async Task<IActionResult> Delete(int id)
{
    if(await _houses.Exists(id) == false)...

    if(await _houses.HasAgentWithId(id, User.Id())
        && User.IsAdmin() == false)...

    var house = await _houses.HouseDetailsById(id);

    var model = new HouseDetailsViewModel()
    {
        Title = house.Title,
        Address = house.Address,
        ImageUrl = house.ImageUrl,
    };

    return View(model);
}
```

In the above methods, we need to **map the HouseDetailsService model to HouseFormModel** and **HouseDetailsViewModel**. Go to the **ControllerMappingProfile class** and **create the mappings** in the **constructor** like this:

```
public class ControllerMappingProfile : Profile
{
    0 references
    public ControllerMappingProfile()
    {
        CreateMap<HouseDetailsServiceModel, HouseFormModel>();
        CreateMap<HouseDetailsServiceModel, HouseDetailsViewModel>();
    }
}
```

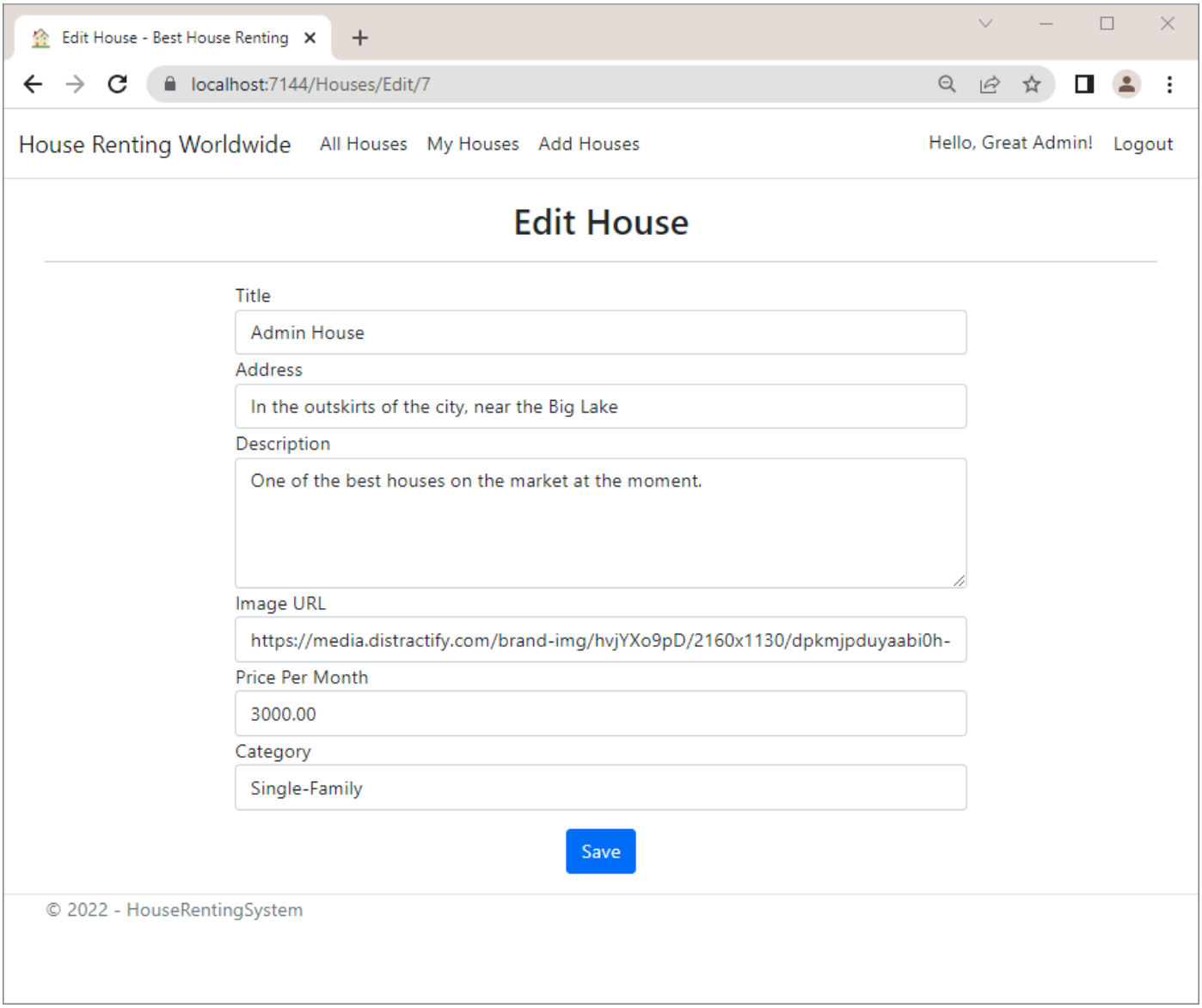
Now go back to the **HousesController class** and **inject the mapper** through the **constructor** and **assign it to a field**:

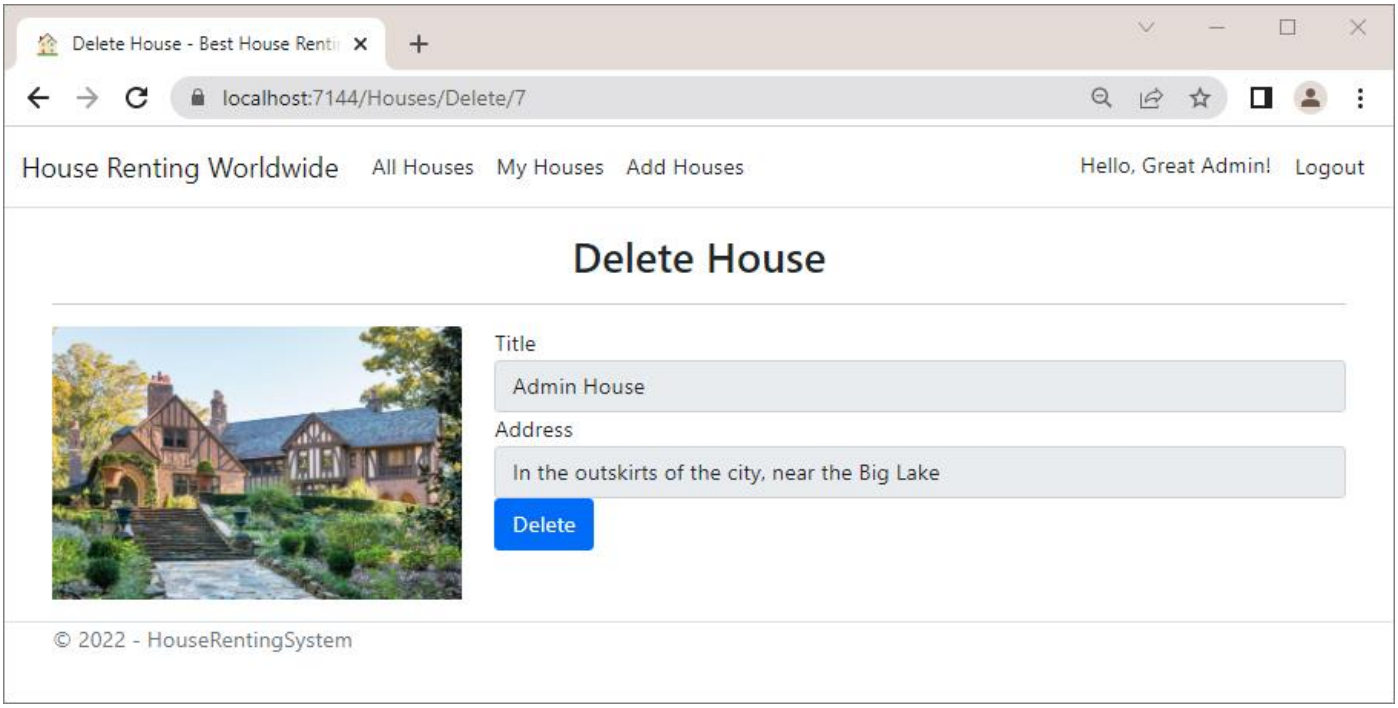
```
public class HouseController : Controller
{
    private readonly IHouseService _houses;
    private readonly IAgentService _agents;
    private readonly IMapper _mapper;

    0 references
    public HouseController(IHouseService houses,
        IAgentService agents,
        IMapper mapper)
    {
        _houses = houses;
        _agents = agents;
        _mapper = mapper;
    }
```

Modify the methods to use the mapper. Use the **Map() method** with the **result model type** and the **object to be mapped**. Note that some **properties cannot be mapped directly** because their **names in objects are not the same**, so we should take care of them **manually**. Do it like this:

```
public async Task<IActionResult> Edit(int id)
{
    if (await _houses.Exists(id) == false)...

    if (await _houses.HasAgentWithId(id, this.User.Id()) == false
        && User.IsAdmin() == false)...

    var house = await _houses.HouseDetailsById(id);

    var houseCategoryId = await _houses.GetHouseCategoryId(house.Id);

    var houseModel = _mapper.Map<HouseFormModel>(house);
    houseModel.CategoryId = houseCategoryId;
    houseModel.Categories = await _houses.AllCategories();

    return View(houseModel);
}

public async Task<IActionResult> Delete(int id)
{
    if(await _houses.Exists(id) == false)...

    if(await _houses.HasAgentWithId(id, User.Id())
        && User.IsAdmin() == false)...

    var house = await _houses.HouseDetailsById(id);

    var model = _mapper.Map<HouseDetailsViewModel>(house);

    return View(model);
}
```

Run the app and try out the mapping. Go to the "**`Edit`**" and "**`Delete`**" **pages** of any house and make sure that the **house data is displayed correctly**:





Now let's **create mapping** in the **service methods**, as well.

Step 2: Mapping in Service Methods

Service methods, which have **mappings**, are all in the **`HouseService` class**. These methods map objects as part of **LINQ queries** and use the **`ProjectTo()` method**.

To use this method, we should **install the `AutoMapper.Extensions.Microsoft.DependencyInjection` NuGet package** also in the "**`HouseRentingSystem.Services`**" **project**.

Next, you should **inject the mapper** in the **`HouseService` class**:

```
public class HouseService : IHouseService
{
    private readonly HouseRentingDbContext _data;
    private readonly IApplicationUserService _user;
    private readonly IMapper _mapper;

    0 references
    public HouseService(HouseRentingDbContext data,
        IApplicationUserService user,
        IMapper mapper)
    {
        _data = data;
        _user = user;
        _mapper = mapper;
    }
```

Now let's **modify the mapping** in each method, where needed, and **try them out**.

All(...) Method

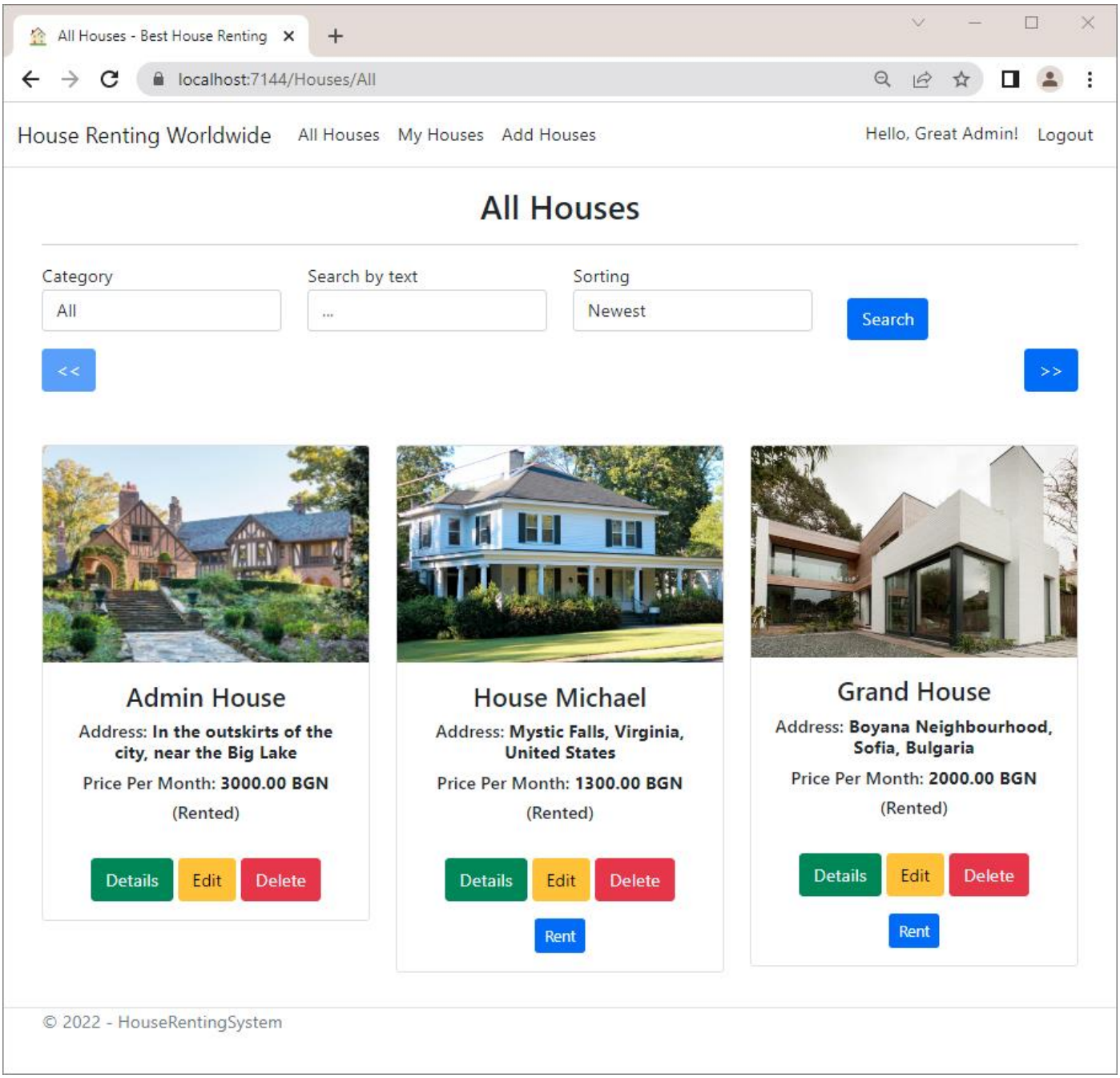
In the **`All(...)` method** we have a **mapping from `House` to `HouseServiceModel`**. However, the **`House` entity** does not have the **`IsRented` property**, which we need in the **`HouseServiceModel`**. For this reason, we should go to the **`ServiceMappingProfile` class** and **configure the mapping** like this:

```
public class ServiceMappingProfile : Profile
{
    0 references
    public ServiceMappingProfile()
    {
        CreateMap<House, HouseServiceModel>()
            .ForMember(h => h.IsRented, cfg => cfg
                .MapFrom(h => h.RenterId != null));
    }
}
```

Go back to the **`All(...)` method** of the **`HouseService` class** and **use the `ProjectTo()` method** to **map objects** in a **LINQ query**. Note that you should **provide the method with the mapper configurations**, not the mapper itself. Do it as shown below:

```
public class HouseService : IHouseService
{

    public HouseQueryServiceModel All(
        var houses = housesQuery
           .Skip((currentPage - 1) * housesPerPage)
           .Take(housesPerPage)
           .ProjectTo<HouseServiceModel>(_mapper.ConfigurationProvider)
           .ToList();
```

Now **run the app** and **navigate to the** "**`All Houses`**" **page**. It should **show the houses data**, especially if they are **rented or not**, correctly. Note that now the "**`Admin House`**" is **rented by the** "**`guest`**" **user**:



AllHousesByAgentId(...) and AllHousesByUserId(...) Methods

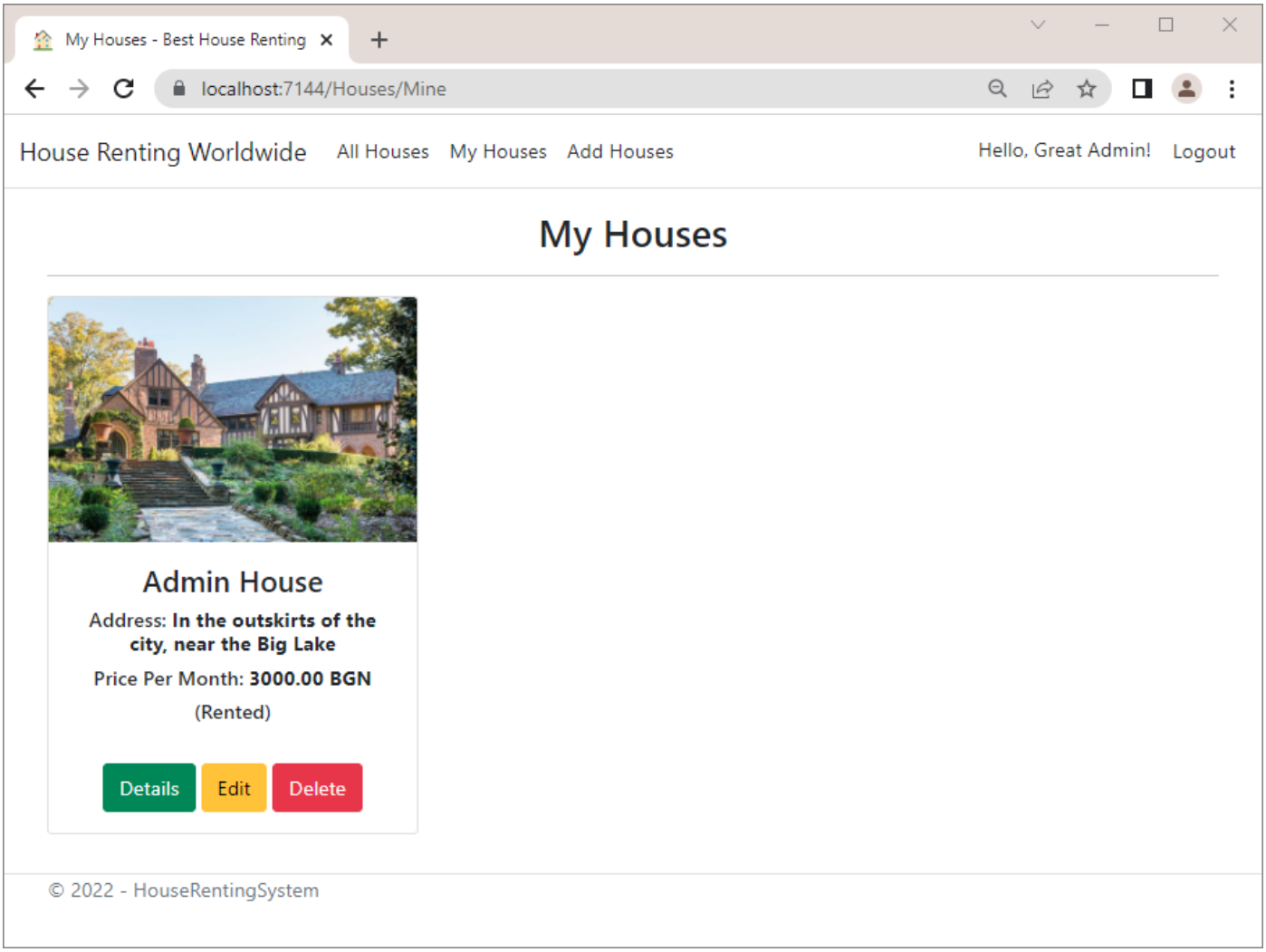
The **`AllHousesByAgentId(…)`** and **`AllHousesByUserId(…)`** methods use the private **`ProjectToModel(…)`** **method** to **map models of type `House` to `HouseServiceModel`**. However, we won't need that method anymore, as our mapping will be improved with **`AutoMapper`**.

Modify the `AllHousesByAgentId(…)` and **`AllHousesByUserId()`** **methods** and **delete the `ProjectToModel()` method** like this:

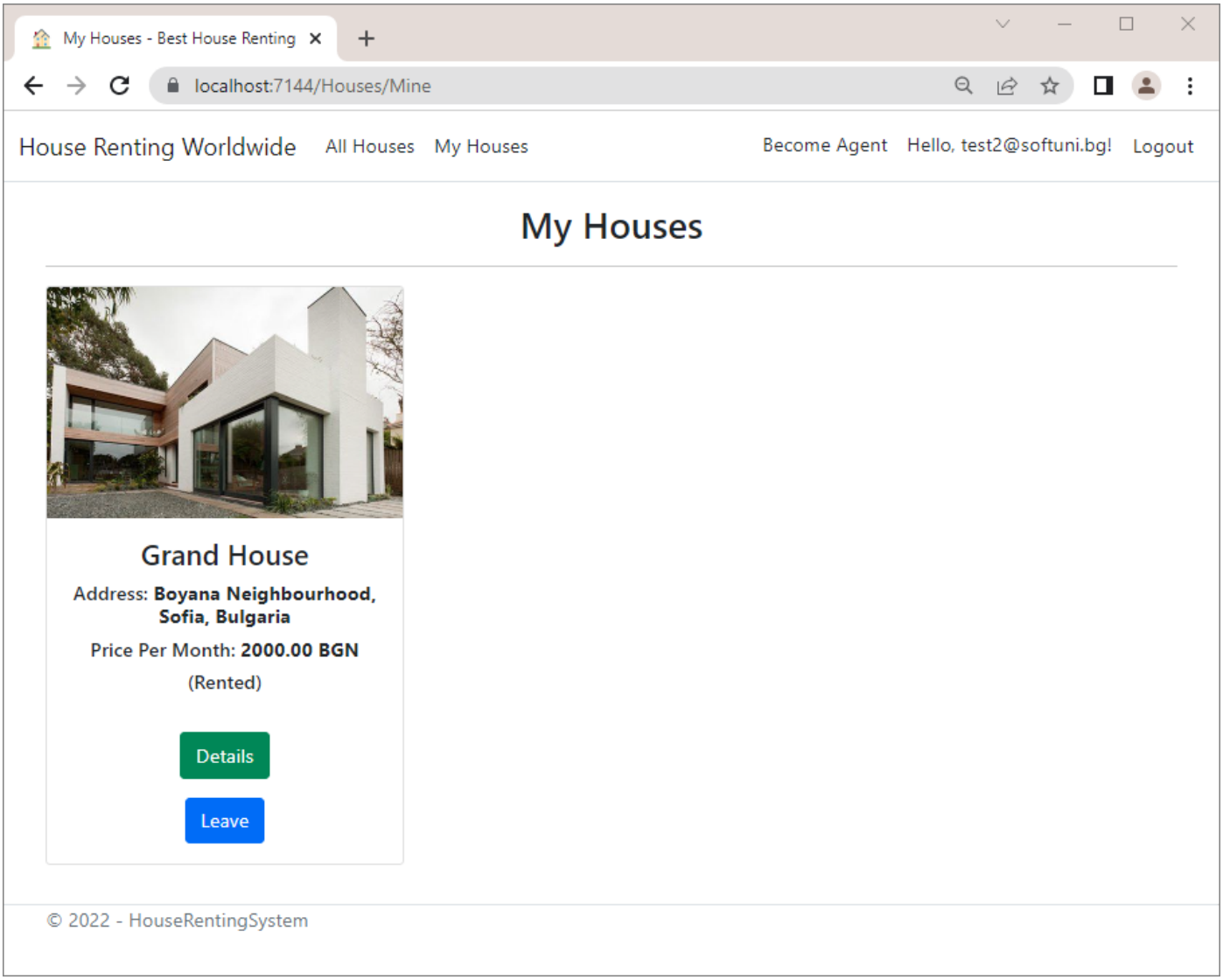
```
public class HouseService : IHouseService
{
```

```
public async Task<IEnumerable<HouseServiceModel>> AllHousesByAgentId(int agentId)
{
    var houses = await _data
          .Houses
          .Where(h => h.AgentId == agentId)
          .ProjectTo<HouseServiceModel>(_mapper.ConfigurationProvider)
          .ToListAsync();

    return houses;
}
public async Task<IEnumerable<HouseServiceModel>> AllHousesByUserId(string userId)
{
    var houses = await _data
          .Houses
          .ProjectTo<HouseServiceModel>(_mapper.ConfigurationProvider)
          .ToListAsync();

    return houses;
}
```

Run the app and navigate to the "My Houses" page with the **Admin**. The page should display the **admin's created houses** correctly:



Then, **log in with another user**, who should see the **houses they rented**:



HouseDetailsById(…) Method

The **HouseDetailsById(…) method** has **two mappings** in one – **from House to HouseDetailsServiceModel** and **from Agent to AgentServiceModel**. In addition, some **properties mapping needs to be configured**. For example, the **Category property** of the **HouseDetailsServiceModel** should have the **category name of the house** and the **Email property** of the **AgentServiceModel** should have the **user email of the agent**.

Create the mappings with the **configurations** in the **ServiceMappingProfile class** like this:

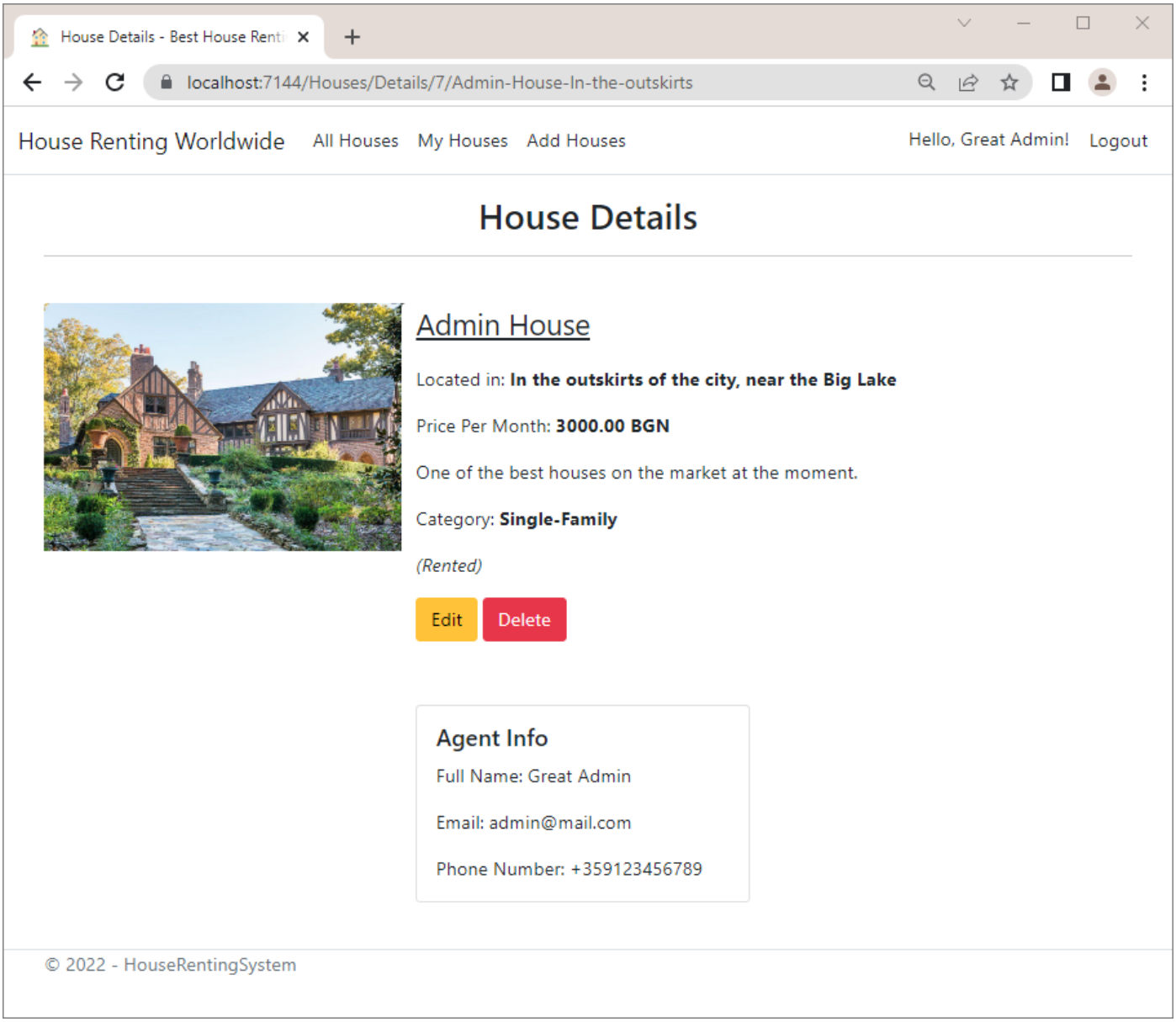
```
CreateMap<House, HouseDetailsServiceModel>()
    .ForMember(h => h.IsRented, cfg => cfg
        .MapFrom(h => h.RenterId != null))
    .ForMember(h => h.Category, cfg => cfg
        .MapFrom(h => h.Category.Name));

CreateMap<Agent, AgentServiceModel>()
    .ForMember(ag => ag.Email, cfg => cfg
        .MapFrom(ag => ag.User.Email));
```

Then, **modify the HouseDetailsById(…) method** to **get the house** from the database by **including the necessary properties** for mapping and **map the objects**. Note that the **agent's full name should be set separately**, as it is **produced by a service method**:

```
public class HouseService : IHouseService
{
```

```
public async Task<HouseDetailsServiceModel> HouseDetailsById(int id)
{
    var dbHouse = await _data
        .Houses
        .Include(h => h.Category)
        .Include(h => h.Agent.User)
        .Where(h => h.Id == id)
        .FirstOrDefaultAsync();

    var house = _mapper.Map<HouseDetailsServiceModel>(dbHouse);

    var agent = _mapper.Map<AgentServiceModel>(dbHouse.Agent);
    agent.FullName = await _user.UserFullName(dbHouse.Agent.UserId);

    house.Agent = agent;

    return house;
}
```

Run the app and **navigate to the** "**`Details`**" **page** of any house and make sure there are **no errors** and the **page contains all the data**:



LastThreeHouses() Method

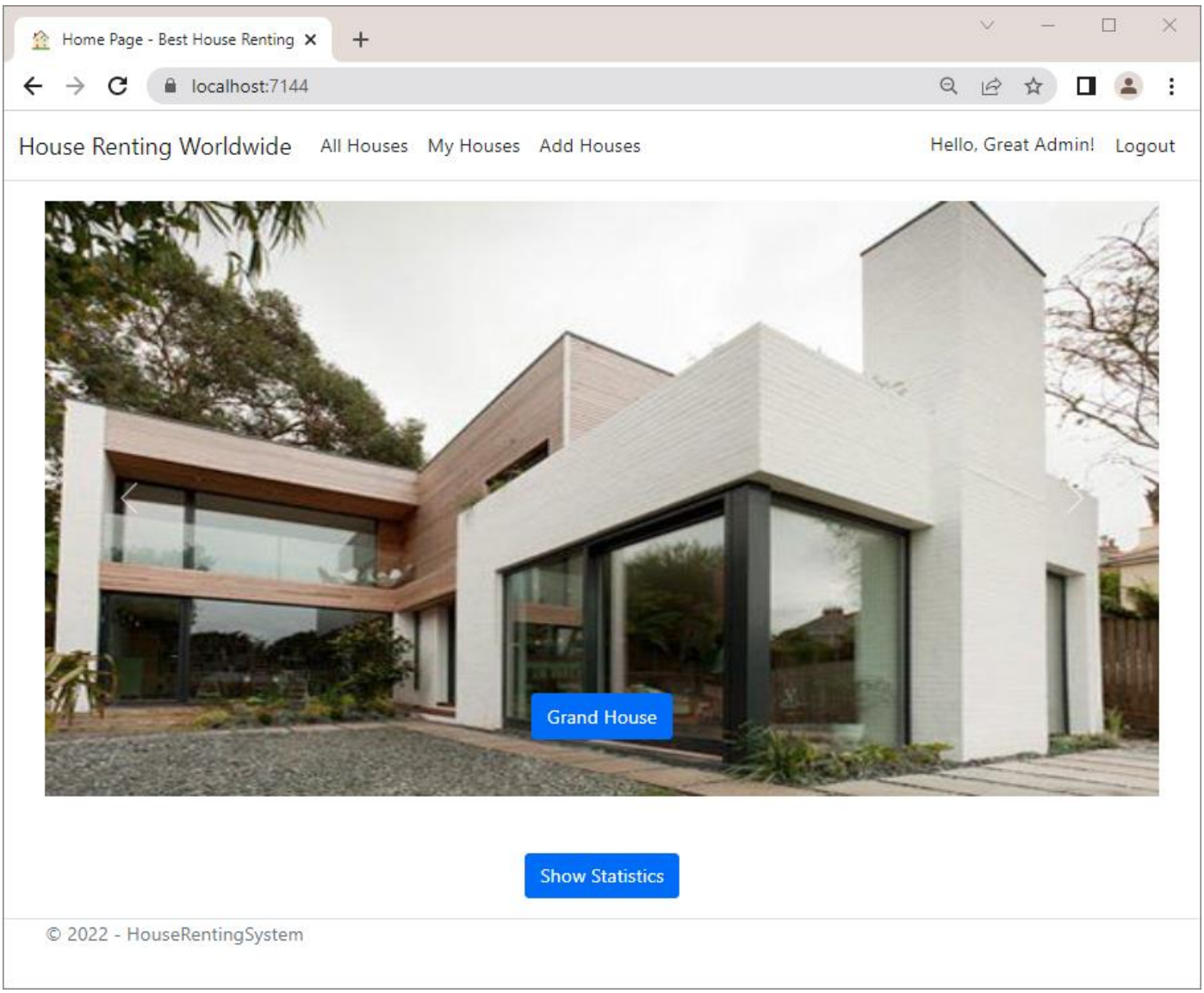
In this method we should **map `House` to `HouseIndexServiceModel`**. **Create the mapping** as shown below. It does **not need configurations**:

```
CreateMap<House, HouseIndexServiceModel>();
```

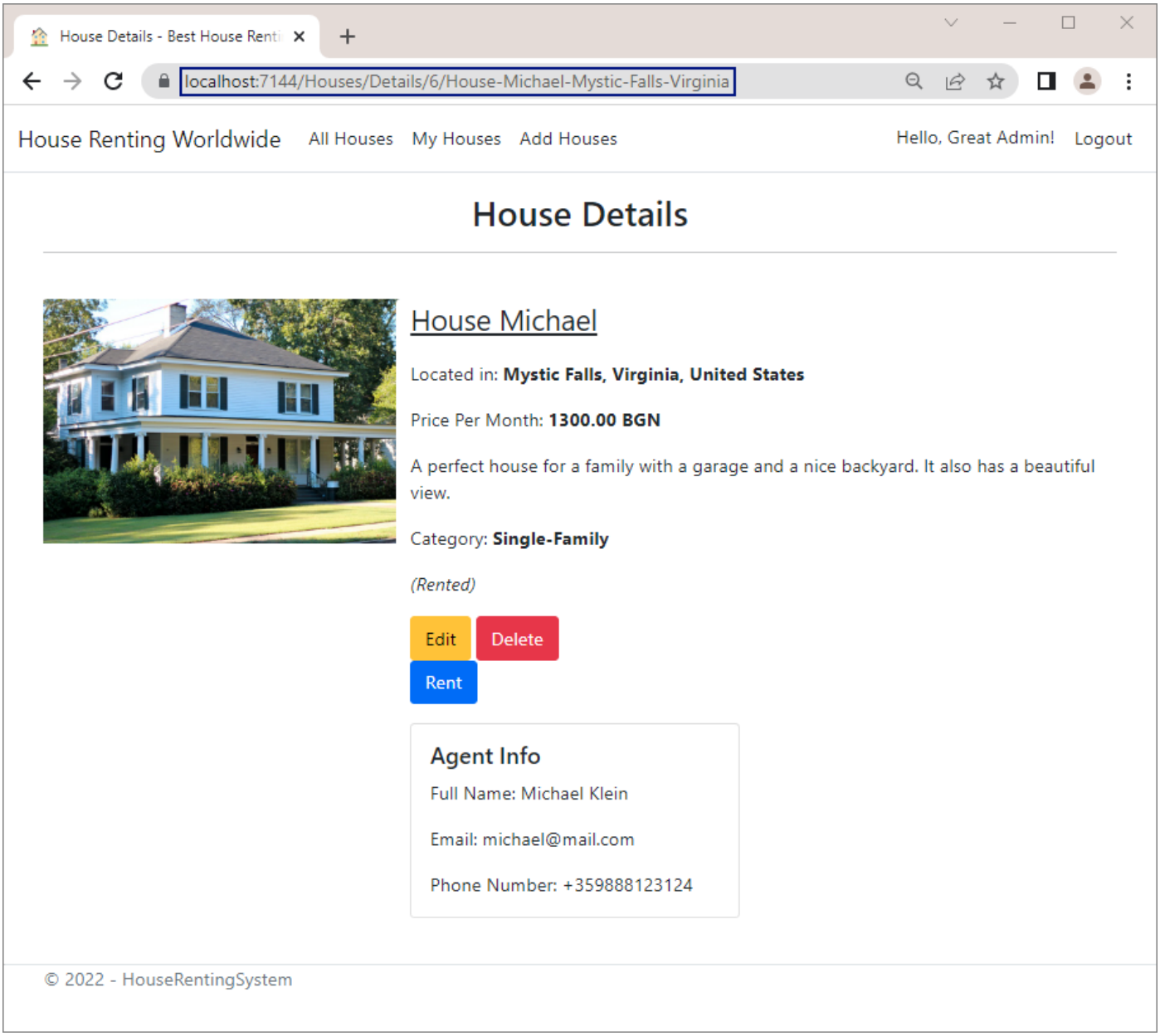
Write the **`LastThreeHouses()` method** like this:

```
public class HouseService : IHouseService
{
    public async Task<IEnumerable<HouseIndexServiceModel>> LastThreeHouses()
    {
        return _data
                .Houses
                .OrderByDescending(c => c.Id)
                .ProjectTo<HouseIndexServiceModel>(_mapper.ConfigurationProvider)
                .Take(3);
    }
```

Run the app and **examine the "`Home`" page** with the houses. It should display them as before:



The **button for the "`Details`" page** of the house should also be **working** and should **build the URL** with the house name and address correctly:



AllCategories() Method

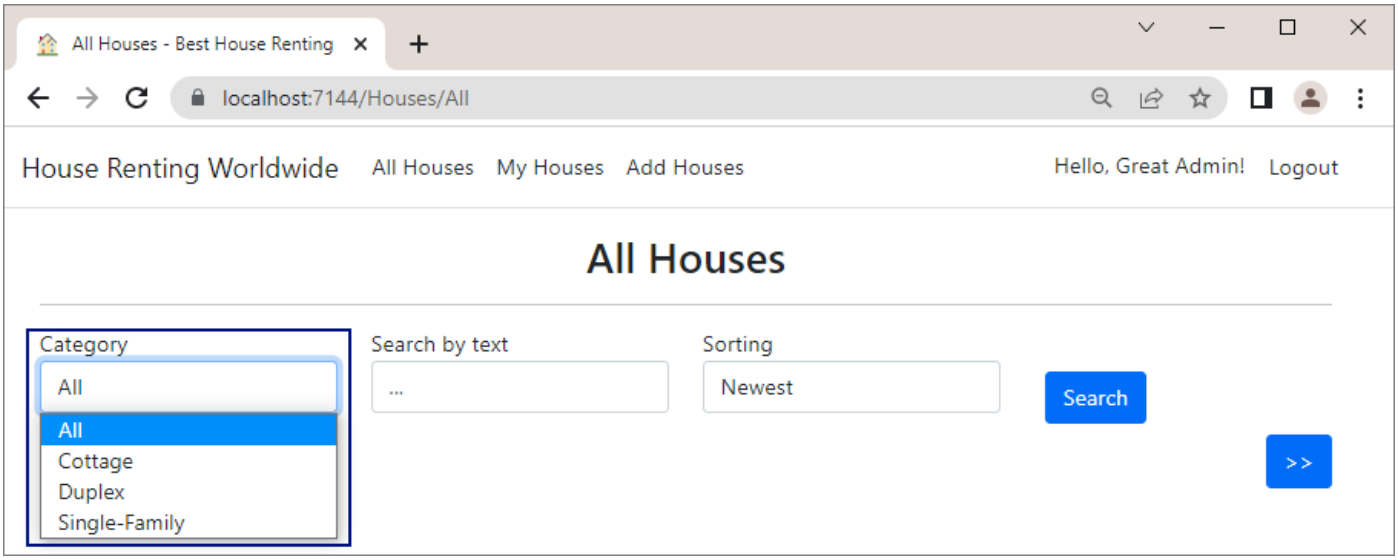
The last method we have with **mapping** is the **`AllCategories()` method**, which maps **`Category`** to **`HouseCategoryServiceModel`**. Create it like this:

```
CreateMap<Category, HouseCategoryServiceModel>();
```

The **`AllCategories()` method** should be modified like this:

```
public class HouseService : IHouseService
{

    public async Task<IEnumerable<HouseCategoryServiceModel>> AllCategories()
    {
        return await _data
                .Categories
                .ProjectTo<HouseCategoryServiceModel>(_mapper.ConfigurationProvider)
                .ToListAsync();
    }
```

Run the app and **open any page** that shows the **house categories**, for example the "**`All Houses`**" **page**. They should all be displayed:



We have **used `AutoMapper`** everywhere we can in our app and it is now improved a lot.