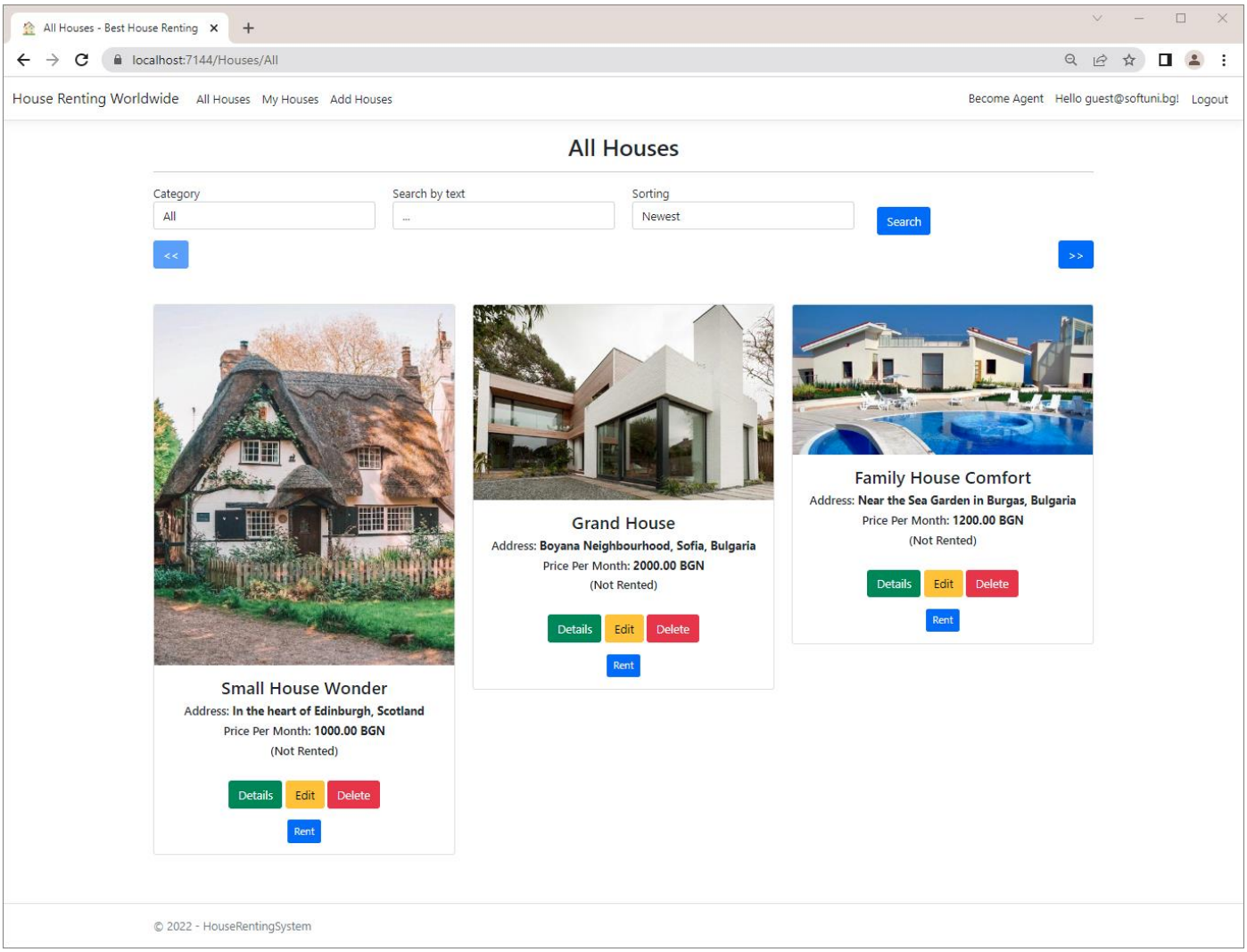
Workshop: Project Fundamentals

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**.

We will implement the app during the workshops in the course.

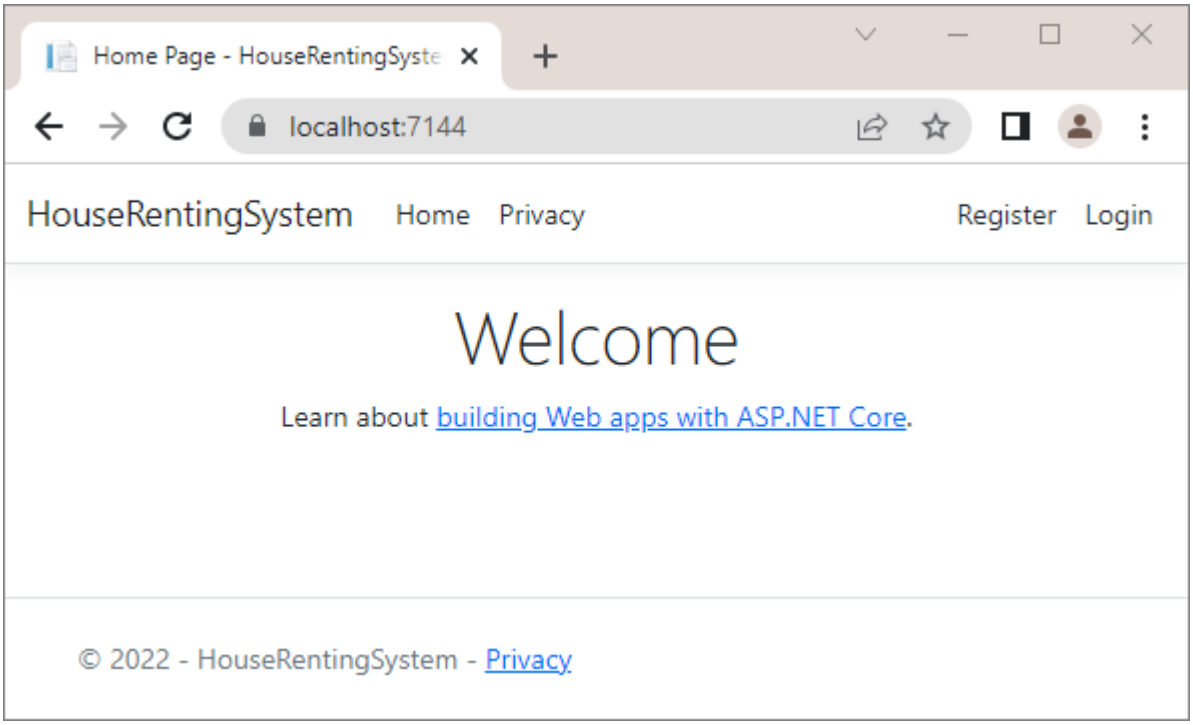


1. Create the Project

Our first task is to create an **ASP.NET Core MVC application** in **Visual Studio**. Open Visual Studio and create a new **ASP.NET Core Web App (Model-View-Controller)** with and **Individual Accounts Authentication type**, as we want to have "**Register**" and "**Login**" functionalities. The app name should be "**HouseRentingSystem**".

2. Examine the App in the Browser

Run the created app in the **browser**. It should have **four pages** for now – "**Home**", "**Privacy**", "**Register**" and "**Login**" **pages**. The "**Home**" **page** looks like this:



3. Clean Project

As you know, **ASP.NET Core** gives us a pretty good **MVC template** to work on. However, we should now define our **own style of writing and formatting code**, which will be used for the **whole app**. We will do this as it is important for our code to **look good** and be **cleaner** and **more readable**. In addition, in this way we will also **examine the project files and classes** better.

Step 1: Modify the Program class

First, go to **`Program.cs`** and look at how it is written. We should remove code comments. Now improve the configuration of the services. It is good for all commands to have an empty line between them. It should look like this:

```
var builder = WebApplication.CreateBuilder(args);

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

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = false;
})
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddControllersWithViews();
```

You can see that the **controller route mapping** is the **default** one.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.MapRazorPages();
```

For this reason, you can replace the **`MapControllerRoute(…)`** method with the **`MapDefaultControllerRoute()`** one:

```
app.MapDefaultControllerRoute();
app.MapRazorPages();
```

Step 2: Modify the HomeController Class

Now let's clean the **`HomeController` class**. To start with, we won't need a **logging functionality** in our app, so remove the **`logger` property** and the **whole class constructor**, which initializes it. We can also make the **actions** with **arrow functions** and remove the "**`Privacy`**" page from our app, as we won't be needing it.

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

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
    0 references
    public IActionResult Error()
    {
        return View(new ErrorViewModel
            { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

We should **remove** the "**`Privacy.cshtml`" view** from our project. Find it in the "**`/Views/Home`**" **folder** and **delete it**:

Step 3: Modify the _Layout.cshtml File

The "**`_Layout.cshtml`**" **view** in our app defines how our **header** and **footer look**.

Go to the **view file** in the "**`/Views/Shared`**" **folder** and let's make some changes. Start by **removing** the "**`Privacy`**" **page links** from the **header** and the **footer**, as we deleted the page. Remove the following lines from the **header**:

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <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>
    <partial name="_LoginPartial" />
</div>
```

Then, **remove** the **`<a>` tag** with the link from the **footer**:

```
<footer class="border-top footer text-muted">
    <div class="container">
        &copy; 2022 - HouseRentingSystem -
        <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
    </div>
</footer>
```

You can also replace the year with **`@DateTime.Now.Year`**:

```
<div class="container">
    &copy; @DateTime.Now.Year - HouseRentingSystem
</div>
```

Modify the **title** of the app in the **<head> tag** and **write a meaningful one**:

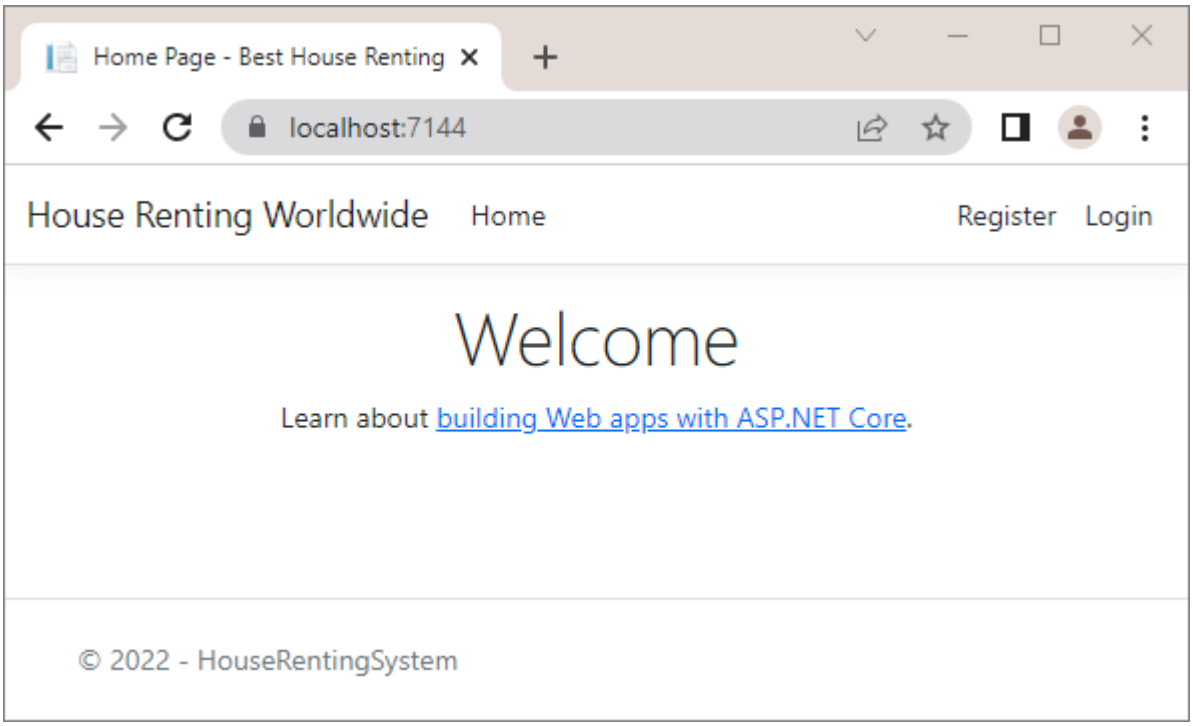
```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Best House Renting</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/HouseRentingSystem.styles.css" asp-append-version="true" />
</head>
```

Then, **change the name** of the app in the **header**, as well:

```
<div class="container-fluid">
    <a class="navbar-brand" asp-area=""
        asp-controller="Home" asp-action="Index">House Renting Worldwide</a>
```

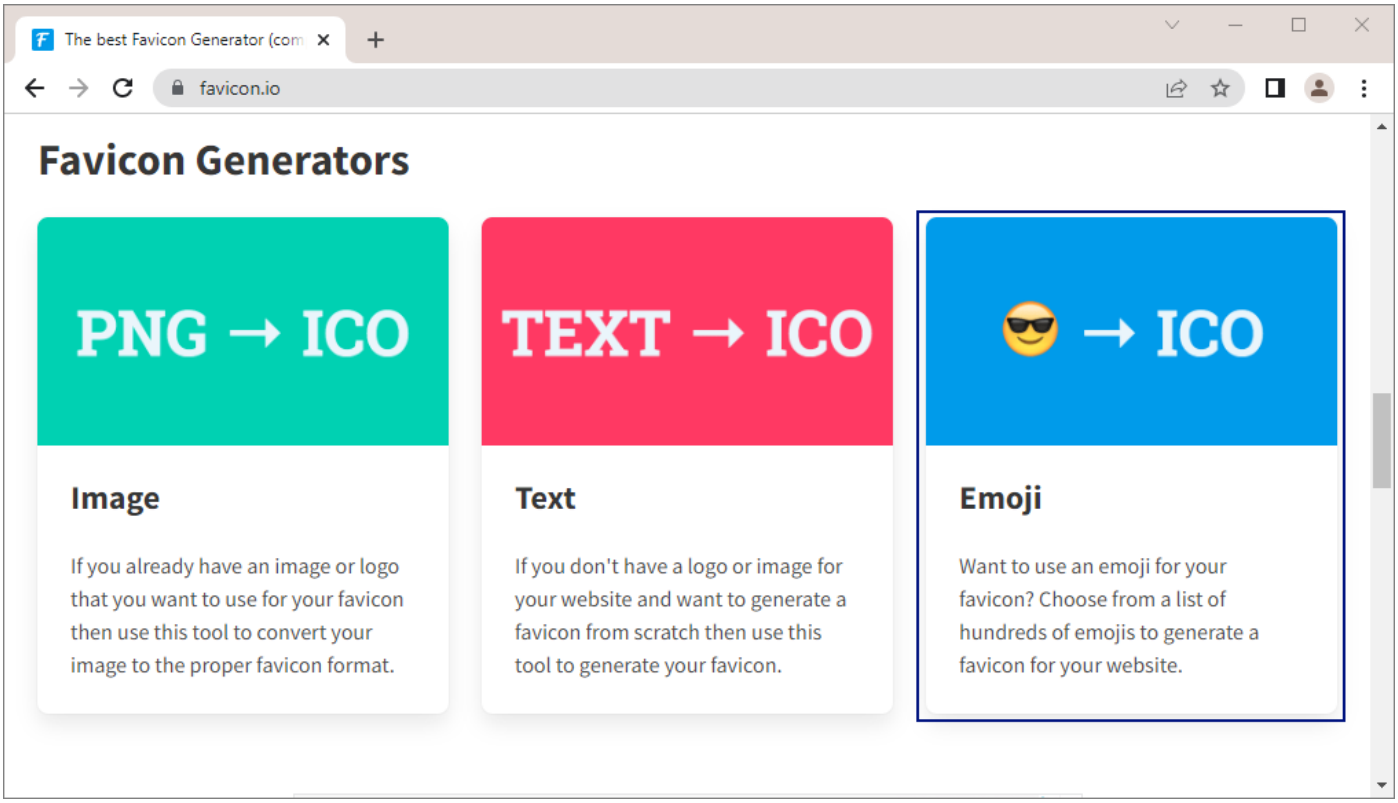
This **name** is in an **<a> tag**, as it is a **hyperlink** to the **main page** of the app.

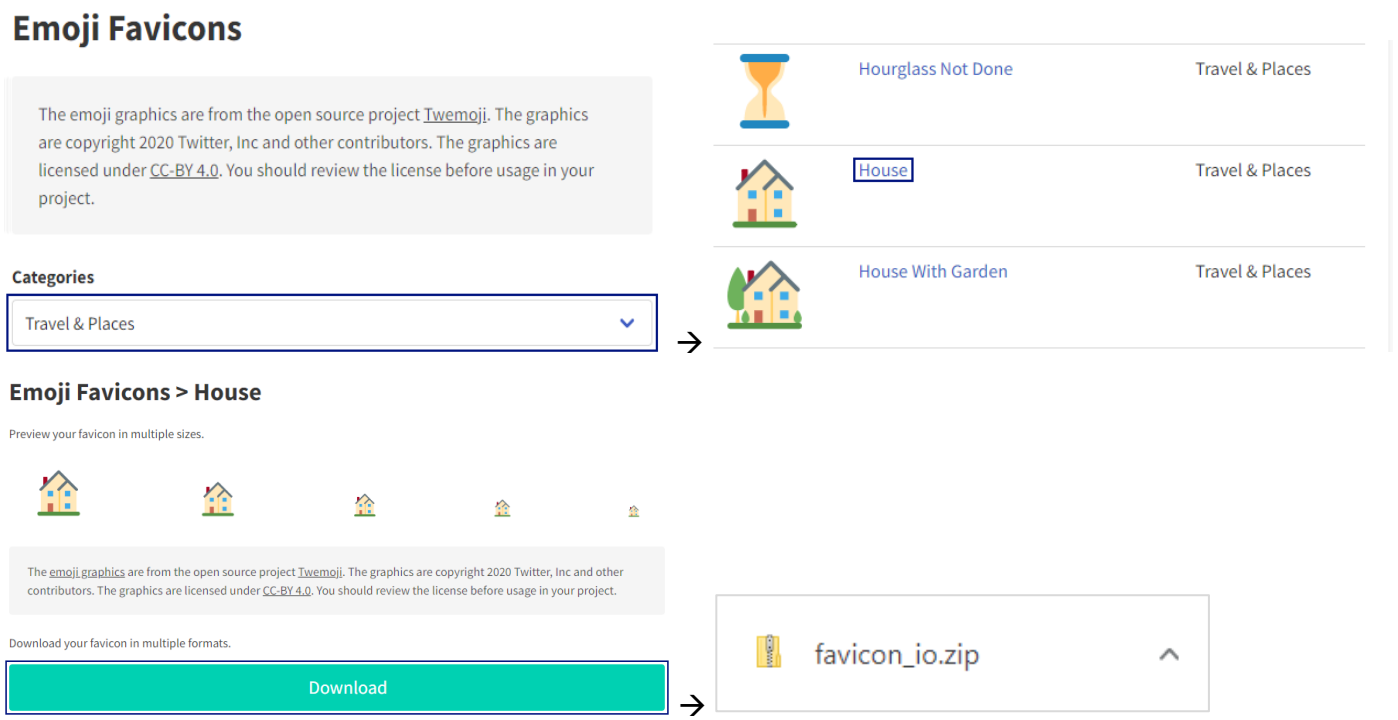
Now the **result** in the **browser** is the following:



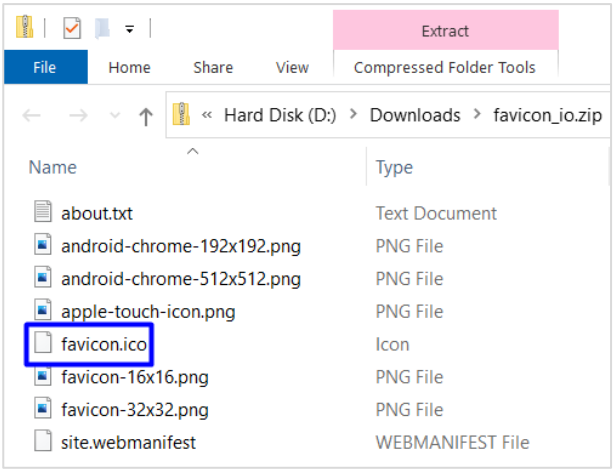
4. Add Favicon

In this task, we will see how to **add a favicon** in our **site's tab** in the **browser**. To do this, we will first go to https://favicon.io/ and **generate a favicon**. **Create** one by yourself and **download** it like this:

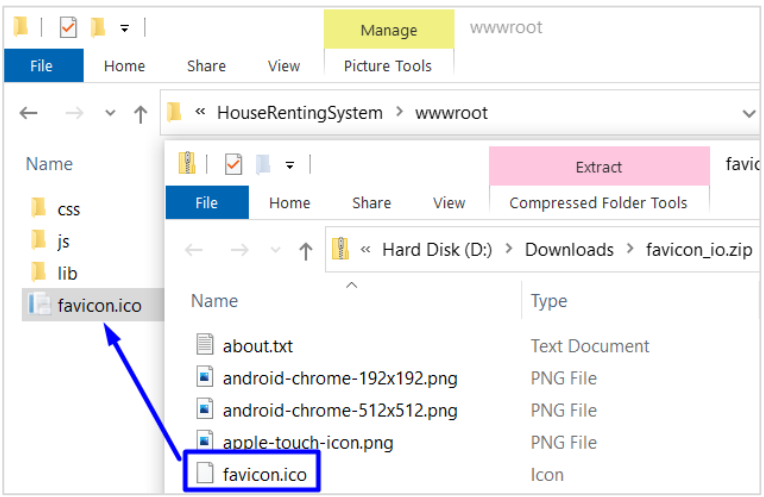




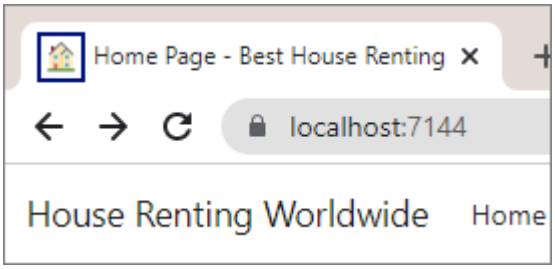
Open the **`.zip` file** and **search for** the **`favicon.ico` file**:



You should now **copy** and **paste this file** in the **project**. **Open the project** in **`File Explorer`** and look for the **`favicon.ico` file** in the "**`wwwroot`**" **directory**. **Replace the old favicon** with the **new** one:



When you are ready, **reload the project** in **VS** and **run it in the browser**. The **new favicon** should be visible in the **browser tab**:



5. Create Entity Model Classes

Now, we will create the **data models**, which we will need for our **database**. We will have **three data model classes** – **`House`**, **`Category`** and **`Agent`**.

Create the above classes in the "**`Data`**" **folder** of the project. They should be in a **separate folder** from the **`ApplicationDbContext`** class – the "**`Models`**" **folder**.

First, let's create a separate class with constants for the max and min length values. In the "**`Data`**" folder, create the **`DataConstants`** class.

Step 1: Add Category Entity Class

The **`Category` class** should have the following **properties**:

- **Id** – a unique **integer**, Primary Key
- **Name** – a **string** with max length **50** (**required**)
- **Houses** – a **collection** of **House**

Our first **entity class** should look like this:

```
public class Category
{
    8 references
    public int Id { get; init; }

    [Required]
    [MaxLength(NameMaxLength)]
    7 references
    public string Name { get; set; } = null!;

    1 reference
    public IEnumerable<House> Houses { get; init; } = new List<House>();
}
```

Step 2: Add House Entity Class

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

- **Id** – a unique **integer**, Primary Key
- **Title** – a **string** with min length **10** and max length **50** (**required**)
- **Address** – a **string** with min length **30** and max length **150** (**required**)
- **Description** – a **string** with min length **50** and max length **500** (**required**)
- **ImageUrl** – a **string** (**required**)
- **PricePerMonth** – a **decimal** with min value **0** and max value **2000** (**required**)
- **CategoryId** – an **integer** (**required**)
- **Category** – a **Category** object
- **AgentId** – an **integer** (**required**)
- **Agent** – an **Agent** object
- **RenterId** – a **string**

Step 3: Add Agent Entity Class

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

- **Id** – a unique **integer**, Primary Key
- **PhoneNumber** – a **string** with min length **7** and max length **15** (**required**)
- **UserId** – a **string** (**required**)
- **User** – an **IdentityUser** object

6. Modify DbContext Class

As we now have **all entity classes** your app needs, use them for the database. To start with, it is a good idea to **rename** the **ApplicationDbContext class** to "**HouseRentingDbContext**", so that it is connected to the idea of our application.

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

```
public DbSet<House> Houses { get; set; }
3 references
public DbSet<Category> Categories { get; set; }
5 references
public DbSet<Agent> Agents { get; set; }
```

Next, we should **override** the **`OnModelCreating(ModelBuilder builder)` method** in the **`HouseRentingDbContext` class**:

```
builder
    .Entity<House>()
    .HasOne(h => h.Category)
    .WithMany(c => c.Houses)
    .HasForeignKey(h => h.CategoryId)
    .OnDelete(DeleteBehavior.Restrict);

builder
   .Entity<House>()
   .HasOne(h => h.Agent)
   .WithMany()
   .HasForeignKey(h => h.AgentId)
   .OnDelete(DeleteBehavior.Restrict);
```

Don't forget to invoke the base **`OnModelCreating()`** method at the end.

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

7. Seed Database

Now we need to **populate the database** with an **initial set of data**. This will include **two users**, an **agent**, **three categories** and **three houses**.

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

```
private IdentityUser AgentUser { get; set; }
4 references
private IdentityUser GuestUser { get; set; }
5 references
private Agent Agent { get; set; }
2 references
private Category CottageCategory { get; set; }
4 references
private Category SingleCategory { get; set; }
3 references
private Category DuplexCategory { get; set; }
2 references
private House FirstHouse { get; set; }
2 references
private House SecondHouse { get; set; }
2 references
private House ThirdHouse { 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(…)` method**. Add the following lines to the method, before invoking the base one:

```
SeedUsers();
builder.Entity<IdentityUser>()
        .HasData(AgentUser, GuestUser);

SeedAgent();
builder.Entity<Agent>()
        .HasData(Agent);

SeedCategories();
builder.Entity<Category>()
    .HasData(CottageCategory,
             SingleCategory,
             DuplexCategory);

SeedHouses();
builder.Entity<House>()
    .HasData(FirstHouse,
             SecondHouse,
             ThirdHouse);
```

As you remember from the previous workshops, it is important that the above **seeding methods are invoked in the correct order**, as they **depend** on each other.

As the methods have a lot of data, you can copy the code from here:

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

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

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

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

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

private void SeedAgent()
{
     Agent = new Agent()
     {
```

```
            Id = 1,
            PhoneNumber = "+359888888888",
            UserId = AgentUser.Id
        };
}

private void SeedCategories()
{
        CottageCategory = new Category()
        {
            Id = 1,
            Name = "Cottage"
        };

        SingleCategory = new Category()
        {
            Id = 2,
            Name = "Single-Family"
        };

        DuplexCategory = new Category()
        {
            Id = 3,
            Name = "Duplex"
        };
}

private void SeedHouses()
{
        FirstHouse = new House()
        {
            Id = 1,
            Title = "Big House Marina",
            Address = "North London, UK (near the border)",
            Description = "A big house for your whole family. Don't miss to buy a
house with three bedrooms.",
            ImageUrl = "https://www.luxury-architecture.net/wp-
content/uploads/2017/12/1513217889-7597-FAIRWAYS-010.jpg",
            PricePerMonth = 2100.00M,
            CategoryId = DuplexCategory.Id,
            AgentId = Agent.Id,
            RenterId = GuestUser.Id
        };

        SecondHouse = new House()
        {
            Id = 2,
            Title = "Family House Comfort",
            Address = "Near the Sea Garden in Burgas, Bulgaria",
            Description = "It has the best comfort you will ever ask for. With two
bedrooms, it is great for your family.",
            ImageUrl =
"https://cf.bstatic.com/xdata/images/hotel/max1024x768/179489660.jpg?k=2029f6d9589b4
9c95dcc9503a265e292c2cdfcb5277487a0050397c3f8dd545a&o=&hp=1",
            PricePerMonth = 1200.00M,
            CategoryId = SingleCategory.Id,
            AgentId = Agent.Id
```

```
        };

        ThirdHouse = new House()
        {
            Id = 3,
            Title = "Grand House",
            Address = "Boyana Neighbourhood, Sofia, Bulgaria",
            Description = "This luxurious house is everything you will need. It is
just excellent.",
            ImageUrl =
"https://i.pinimg.com/originals/a6/f5/85/a6f5850a77633c56e4e4ac4f867e3c00.jpg",
            PricePerMonth = 2000.00M,
            CategoryId = SingleCategory.Id,
            AgentId = Agent.Id
        };
}
```

Note that only the **first house** has a **`RenterId`**, which means that it is **rented by the `GuestUser`**.

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

8. Create a Migration

We will now **create a migration** to the database. Before that, however, let's give the **database a good name**.

To do this, **edit the connection string** in the "**`appsettings.json`**" **file**. Set "**`Database`**" **name** to be "**`HouseRentingSystem`**":

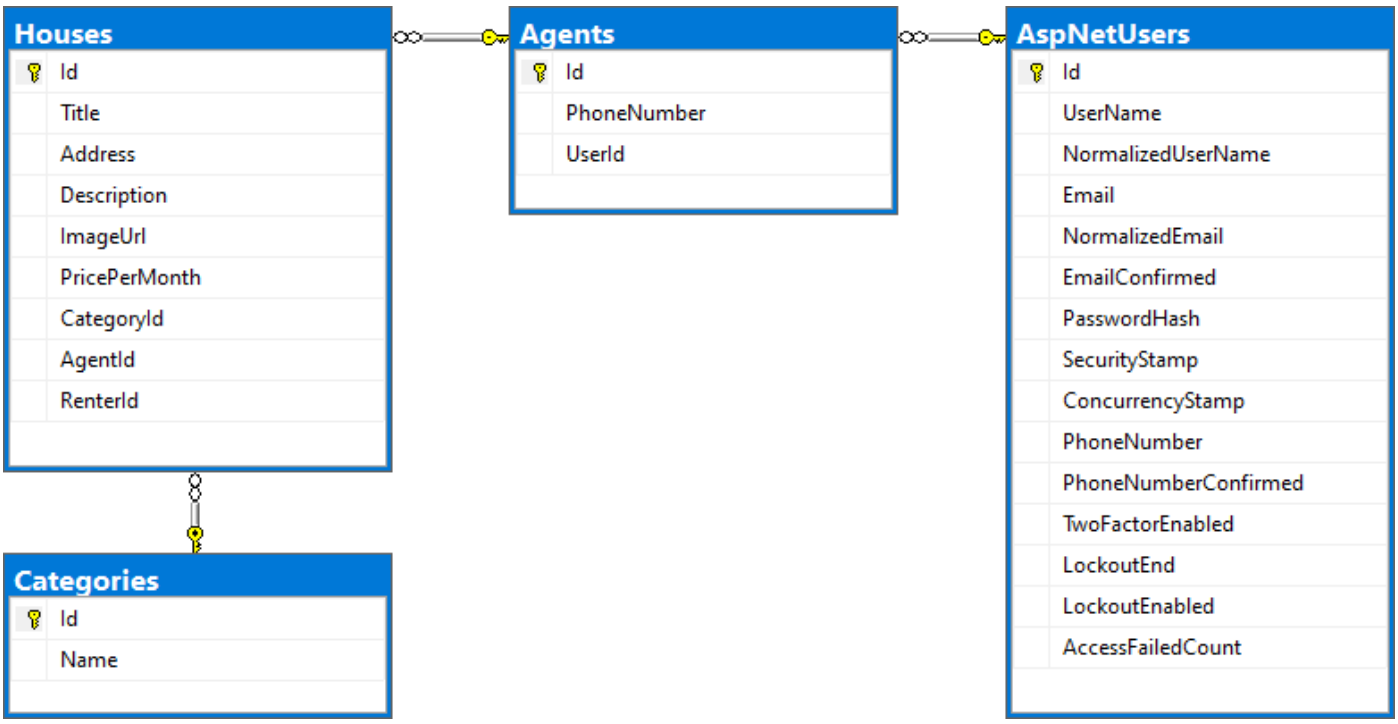
```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=HouseRentingSystem;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

Next, **open** the **`Package Manager Console`** to write **commands** for **managing migrations**. In the **console**, write a command for **adding a migration** to the "**`Data/Migrations`**" **folder** with a given **name** and **press `[Enter]`** to **execute it**. Now you should have a **new migration** in the "**`Migrations`**" **folder**.

Examine the tables and its **restrictions** in the **new migration** – if something is wrong, **delete the migration** with the "**`Remove-Migration`**" **command** or **delete the migration file**. Don't forget that you should also **delete the database** from SQL Server Management Studio, or errors will appear.

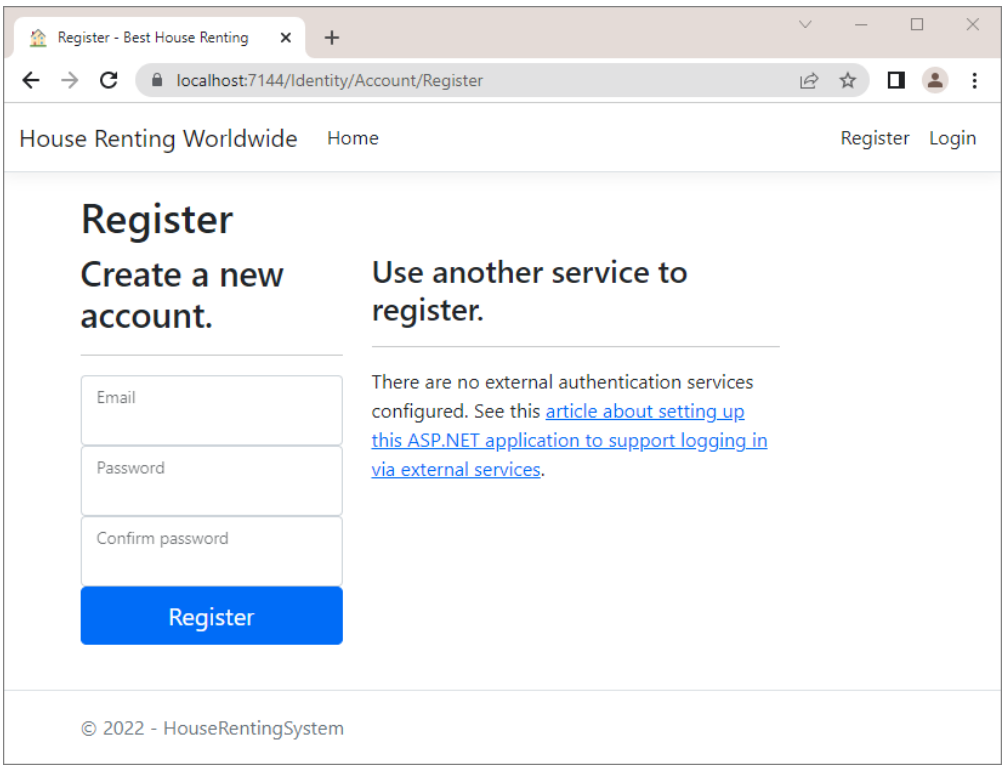
Now **run the app** in the browser – there should not be **any errors**. Then, look at the **newly-created database** in SQL Server Management Studio and **examine its tables** – all tables we created should be **present** and have the **right restrictions** and **relationships**.

Examine the **diagram of the database**, as well. It should look like this:



9. Register and Log in the App

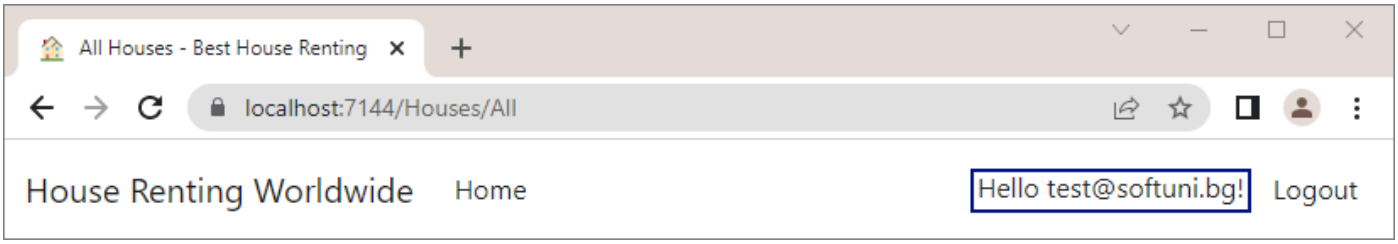
Go to the "**`Register`**" **page** in the "**`HouseRentingSystem`**" **app** and you should see the **registration form**:



If you try to **fill in the form**, you will see that **password requirements are too strict** – change this by adding the following lines in **`Program.cs`**:

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

After this, **fill in** the "**`Register`**" **form** with **valid data** and you should be **logged**:



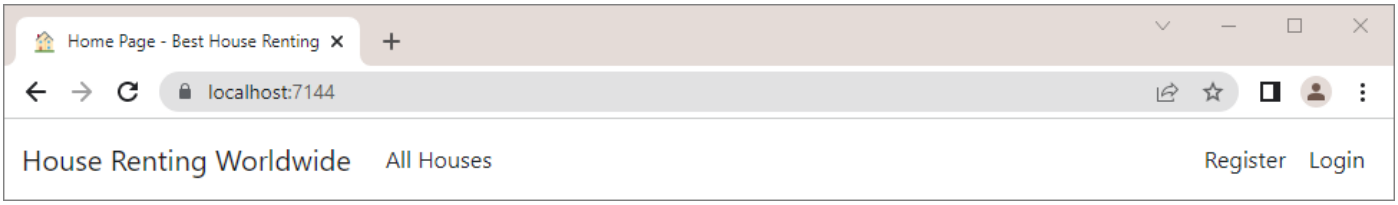
Also, the **new user** should be part of the **database** – check in the "**`AspNetUsers`**" **table**:

	Id	UserName	NormalizedUserName	Email	NormalizedEmail
1	68d4176d-087b-4284-b100-ce0f8face68b	test@softuni.bg	TEST@SOFTUNI.BG	test@softuni.bg	TEST@SOFTUNI.BG
2	6d5800ce-d726-4fc8-83d9-d6b3ac1f591e	guest@mail.com	guest@mail.com	guest@mail.com	guest@mail.com
3	dea12856-c198-4129-b3f3-b893d8395082	agent@mail.com	agent@mail.com	agent@mail.com	agent@mail.com

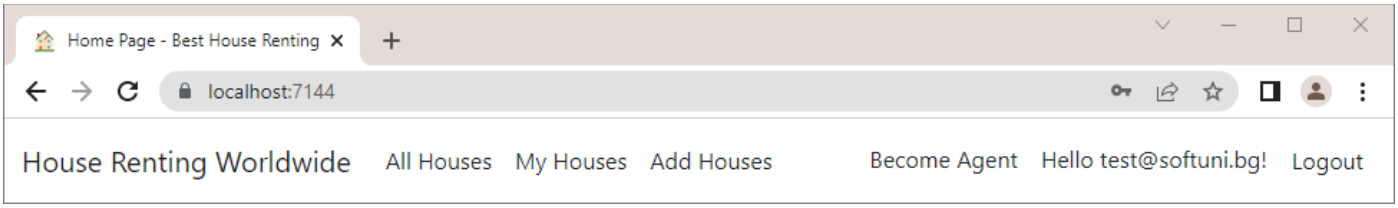
10. Modify Navigation

Now, we should **change the navigation menu** to have **links** to the "**`All Houses`**", "**`My Houses`**", "**`Add House`**" and "**`Become Agent`**" **pages** when the **user is logged-in**.

When the **user is not logged-in**, the **navigation** should look like this:



When the **user is logged-in**, it should be the following:



To do this, we need to **modify** the "**`_Layout.cshtml`**" and "**`_LoginPartial.cshtml`**" **views** in the "**`/Views/Shared`**" **folder**, as they are **responsible for the navigation menu**.

First, to **change links on the left side** of the page, go to the "**`_Layout.cshtml`**" file. In it, we want to **add links to the pages** depending on whether the **user is authenticated**. Use an **`if` statement** in the **Razor view** and the **`Identity.IsAuthenticated` property** to check whether the **current user is authenticated**:

_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">House Renting Worldwide</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
                    data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        @if (this.User.Identity.IsAuthenticated)
                        {

                        }
                        <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>
```

Then, outside of the **`if` statement**, add the "**`All Houses`**" **page link**, as the page should be **accessible to anyone**. In the **statement**, add **links** to the "**`My Houses`**" and "**`Add House`**" **pages**, which are **only for authenticated users**. Use the **`asp-controller`** and **`asp-action` tag helpers**, so that **links point to the correct controller action**. Don't forget to remove the link to the "**`Home`**" page.

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="House" asp-action="All">All Houses</a>
        </li>
        @if (User.Identity.IsAuthenticated)
        {
            <li class="nav-item">
                <a class="nav-link text-dark" asp-area="" asp-controller="House" asp-action="Mine">My Houses</a>
            </li>
            <li class="nav-item">
                <a class="nav-link text-dark" asp-area="" asp-controller="House" asp-action="Add">Add Houses</a>
            </li>
        }
    </ul>
    <partial name="_LoginPartial" />
</div>
```

Now go to the "**`_LoginPartial.cshtml`**" **view** to add the "**`Become Agent`**" **page link**. It should be **only for logged-in users** (later we will see how only non-agents to see it), so find the right place in the view code and **add a link** like this:

```
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area=""
            asp-controller="Agents" asp-action="Become">Become Agent</a>
    </li>
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello @User.Identity?.Name!</a>
    </li>
```

Try out the **navigation bar links** in the browser – they should **redirect to the correct pages**.

11. Create the MVC Structure of the App

In our "**HouseRentingSystem**" **app** we will have **three controller classes – HomeController**, **HouseController** and **AgentController**. Now we will just **create the controller actions** with the **attributes** and **models** they need, but we **won't implement them** yet, as we will do this later in the workshop.

Step 1: Modify the HomeController Class

We already have the **HomeController** in our **project** but we will modify it a bit, as we want it to **return a model**.

Start by creating a "**Home**" **folder** in the "**Models**" **folder** of our project – this is where the **models** for our **HomeController** should be created. Then, create the **IndexViewModel class** in the "**/Models/Home**" **folder**.

Use the **model** and **pass it to the view** in the **Index() method** of the **HomeController class**:

```
public class HomeController : Controller
{
    0 references
    public async Task<IActionResult> Index()
    {
        return View(new IndexViewModel());
    }
}
```

Now we should go to the **"Index.cshtml" view** in the "**/Views/Home**" **folder** and modify it to **accept an IndexViewModel**.

Before we write in the **view**, we need to go to the "**_ViewImports.cshtml" file** and **add the** "**HouseRentingSystem.Models.Home**" **namespace**, so that the **views can use the models** in our "**/Models/Home**" **folder** like this:

_ViewImports.cshtml

```
@using HouseRentingSystem
@using HouseRentingSystem.Models
@using HouseRentingSystem.Models.Home
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Add the **IndexViewModel** to the **"Index.cshtml" view** like this:

Index.cshtml

```
@model IndexViewModel

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

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about
        <a href="https://docs.microsoft.com/aspnet/core">
            building Web apps with ASP.NET Core</a>.</p>
</div>
```

Step 2: Create the HouseController Class

The **HouseController** will have all **actions** for **CRUD operations** on **houses** and **their renting**.

First, create the **HouseController class** in the "**Controllers**" **folder** of the "**HouseRentingSystem**" **project**.

The class should have some **methods,** which (for now) we will create **without implementing them fully**.

Start with writing the **All() method**, which should **return a view** with **all houses**. Create the **AllHousesQueryModel model**, which will later get information from the **request query** and use it to generate a **view**, in the "**Models**" **folder**. Create a "**Houses**" **folder** for **house models** and **add the above model class**.

We will leave the **model class** like this for now and **add properties later**. Go back to the **HouseController** and write the **All() method** to **return a view** with the newly-created **model**.

```
[Authorize]
0 references
public class HouseController : Controller
{
    [AllowAnonymous]
    0 references
    public async Task<IActionResult> All()
    {
        return View(new AllHousesQueryModel());
    }
}
```

As the actions returns a view, let's **create that view**. To do this, create a "**Houses**" (the name of the **controller**) **folder** in the "**Views**" **folder** of the project and then **create an empty Razor view** named "**All.cshtml**" (the name of the **action**).

Before we write in the **view**, we need to go to the "**_ViewImports.cshtml**" **file** and **add the** "**HouseRentingSystem.Models.Houses**" **namespace**, so that the **views can use the models** in our "**/Models/Houses**" **folder**.

Now, in the **All view**, use the **@model directive**, which specifies what **type** the view should accept from the **controller action**. In this case, we accept a **AllHousesQueryModel model**:

All.cshtml

```
@model AllHousesQueryModel
```

Then, we will **set a page title** with **ViewBag** and then use it in a **page heading**. For the **ViewBag**, we will use the **@ symbol** to **open a code block** and **write C#** in it like this:

```
@{
    ViewBag.Title = "All Houses";
}
```

The rest of the "**All.cshtml**" **file** looks like this:

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

We will improve it later in the workshop.

Go back to the **HouseController class** and continue with writing its **methods**. The next **action** is **Mine()**, which should also **return a view with houses**. Use the **[Authorize] attribute** to make the method accessible only for **authorized users**:

```
    public async Task<IActionResult> Mine()
    {
        return View(new AllHousesQueryModel());
    }
```

The **method** uses the **`AllHousesQueryModel` model**, which we have created already. **Create a view**, similar to the one we just created, which will be returned:

Mine.cshtml

```
@model AllHousesQueryModel

@{
    ViewBag.Title = "My Houses";
}

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

Next, add the **`Details(int id)` action** to the **controller class**, which should **return a view** with a **details model**:

```
public async Task<IActionResult> Details(int id)
{
    return View(new HouseDetailsViewModel());
}
```

Create the **`HouseDetailsViewModel` class** in the "**`/Models/Houses`**" **folder** and **leave it empty** for now. Then, **write the method** to return a **view** and **create the view**, as well. The **view** should be the following:

Details.cshtml

```
@model HouseDetailsViewModel

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

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

In the **`HouseController`**, add the **`Add()` method**, which should just **return a view**. Remember that only **authorized users** should be able to access the "**Add House**" **page**:

```
0 references
public async Task<IActionResult> Add()
{
    return View();
}
```

The "**`Add.cshtml`**" **view** should **contain a form for adding a new house**. However, we will **create the form later –** for now, the **view** should be the following:

Add.cshtml

```
@{
    ViewBag.Title = "Add House";
}

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

Next, write the **`Add(HouseFormModel house)` method**, which should **accept a model** from the **view** when we **create its form**. Create the **`HouseFormModel` class** in the "**`/Models/Houses`**" **folder** and **leave it empty** for now.

The **Add(HouseFormModel house) method** should be invoked on a "**POST**" **request** and be for **authorized users** only:

```
[HttpPost]
0 references
public async Task<IActionResult> Add(HouseFormModel model)
{
    return RedirectToAction(nameof(Details), new { id = "1" });
}
```

The method should **add a new house** to the **database** (which we will do later) and **redirect the user** to the "**House Details**" **page** of the **newly-created house**. The **Details(int id) controller action** accepts an **id**, but we will **hardcode** it for now.

Next **methods** to be implemented are for the "**Edit House**" **page**. The method on a "**GET**" **request** should accept a house **id**, so that we know which **house to edit** and **return a view** with the **current house information**:

The **method** for the "**POST**" **request** to the "**Edit House**" **page** should accept a house id and a **model** and **redirect** the user to the "**House Details**" **page** after the house is **modified in the database**:

```
public async Task<IActionResult> Edit(int id)
{
    return View(new HouseFormModel());
}

[HttpPost]
0 references
public async Task<IActionResult> Edit(int id, HouseFormModel house)
{

    return RedirectToAction(nameof(Details), new { id = "1" });
}
```

The **methods for deleting a house** are similar to the above methods, but they **accept** and **pass** a **HouseDetailsViewModel model**. After the **house is deleted**, the user should be **redirected** to the "**All Houses**" **page**. Write them like this:

```
public async Task<IActionResult> Delete(int id)
{
    return View(new HouseFormModel());
}

[HttpPost]
0 references
public async Task<IActionResult> Delete(HouseDetailsViewModel house)
{

    return RedirectToAction(nameof(All));
}
```

Create the "**Edit.cshtml**" and "**Delete.cshtml**" **views**, as we did with the previous ones:

Edit.cshtml

```
@{
    ViewBag.Title = "Edit House";
}

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

Delete.cshtml

```
@{
    ViewBag.Title = "Delete House";
}

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

Finally, write the **methods** for the **house renting and leaving functionality**. They should be invoked on a "**POST**" **request** and **redirect** the user to the "**My Houses**" page:

```
    [HttpPost]
    0 references
    public async Task<IActionResult> Rent(int id)
    {
        return RedirectToAction(nameof(Mine));
    }

    [HttpPost]
    0 references
    public async Task<IActionResult> Leave(int id)
    {
        return RedirectToAction(nameof(Mine));
    }
}
```

Step 3: Create the AgentController Class

The **AgentController** is taking care of the **agents functionality**.

Create the **AgentController** in the "**Controller**" **folder** and write the **Become() method** like this:

```
[Authorize]
0 references
public class AgentController : Controller
{
    0 references
    public async Task<IActionResult> Become()
    {
        return View();
    }
}
```

Create the "**Become.cshtml**" **view** in "**/Views/Agents**":

Become.cshtml

```
@{
    ViewBag.Title = "Become Agent";
}

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

For the **`Become(BecomeAgentFormModel agent)` method** you should create the **`BecomeAgentFormModel`** in the "**`/Models/Agents`**" **folder**.

Write the **method** to be **invoked** on a "**POST**" **request** and **return a redirect response**:

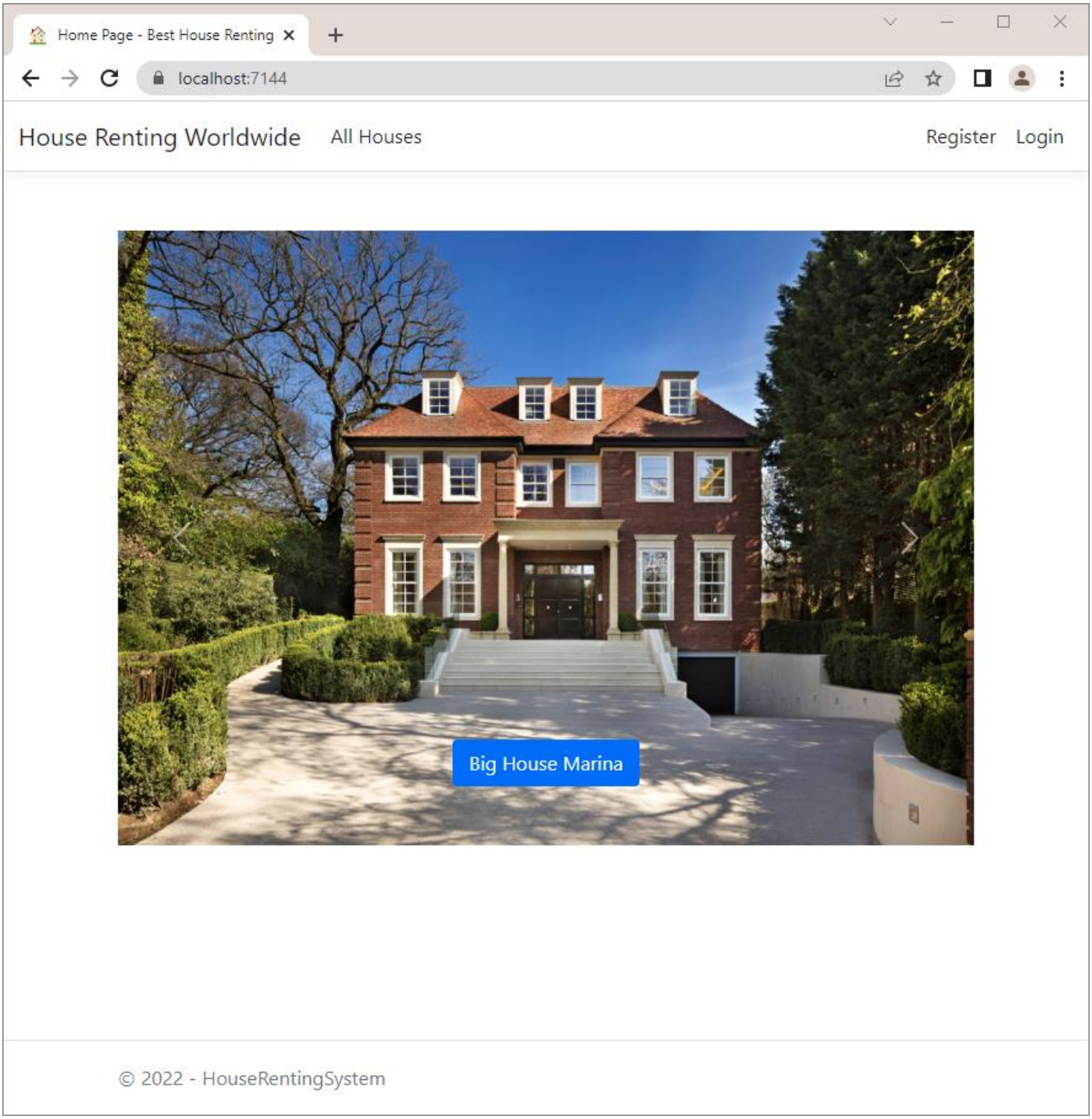
```
public class AgentController : Controller
{
    [Authorize]
    [HttpPost]
    0 references
    public async Task<IActionResult> Become(BecomeAgentFormModel agent)
    {
        return RedirectToAction(nameof(HouseController.All), "Houses");
    }
}
```

Now we have written the **structure of our controllers** (**`HomeController`**, **`HouseController`** and **`AgentController`**) and their **actions**, as well as the **models** and **views** for our **MVC app**.

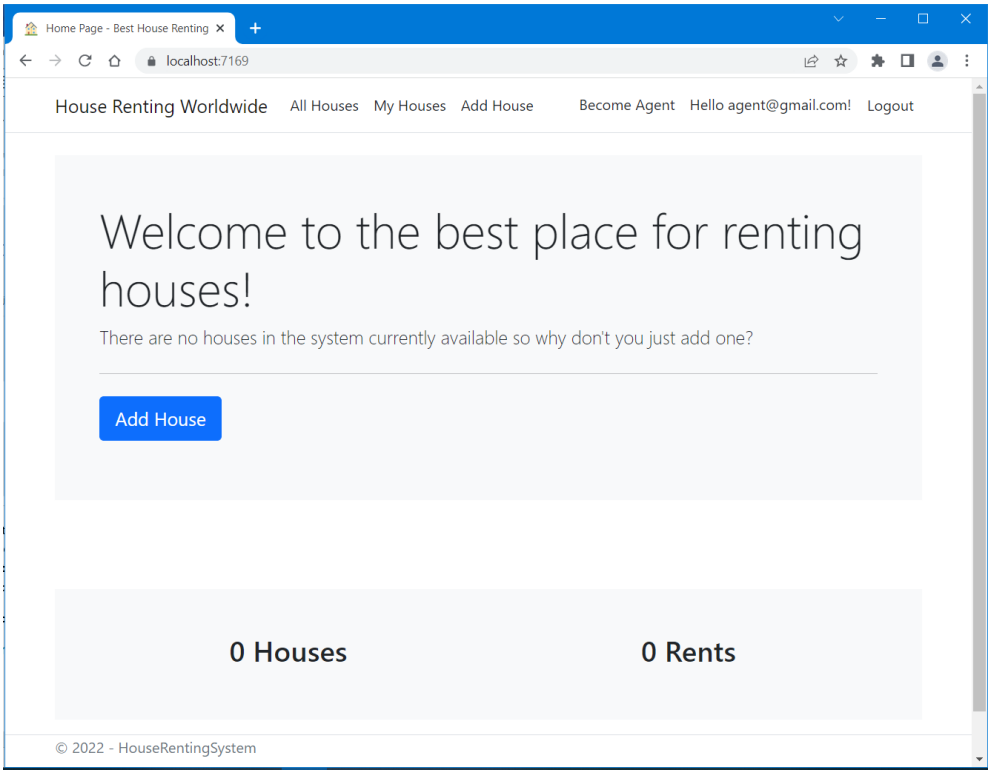
In the next task, we will start creating **services**, which will contain the **app's business logic** and **interact with the database**. Our **controllers** will use the **service methods** and depend only on them – they will **not have access to the database**. We will do this by creating **services with service models** and implementing the **logic in service methods**. Let's see how this is going to happen.

12. Implement the "Home" Page

The "**Home**" **page** of the app should always **show the last three added houses** as a **slideshow**:



If there **aren't any houses**, the "**Home**" **page** should look like this:



First, we should create two additional folders in our project – "**Contacts**" and "**Services**". They will hold the **interfaces** and the **services** classes.

The "**Home**" **page** is accessed on "**/**" and **invokes** the **Index() method** of the **HomeController class**. We should **modify** the **HomeController** class with its **Index()** method to use services. The **houses collection** for the **view** (for the houses slideshow) will be implemented by using a **service method** in the **HouseService class**. The **total houses and total rents counts** will be obtained in a **separate API controller** with a **separate service class** (you will see how to do this on the next task).

For now, create an **IHouseService interface** in the "**Contracts/House**" and a **HouseService class** in the "**Services/House**" folders.

Then, we should define the following **method** in the **IHouseService interface**, which should only **return a collection** with the **newest three houses** from the **database**:

```
public interface IHouseService
{
    2 references
    Task<IEnumerable<HouseIndexServiceModel>> LastThreeHouses();
}
```

Create a new model **HouseIndexServiceModel** to be returned by the **service method**:

```
public class HouseIndexServiceModel
{
    2 references
    public int Id { get; set; }
    3 references
    public string Title { get; set; } = null!;
    2 references
    public string ImageUrl { get; set; } = null!;
}
```

Implement the method from the **IHouseService interface** in the **HouseService class**.

Don't forget to inject the **`HouseRentingDbContext`** through the constructor and assign it to a variable to use it. After that, we will **chain several methods** to **extract what we need from the database** and use it. First, we should **get all houses** from the database and then **get the newest houses** first by **sorting them by `id` in descending order** (newer houses have a higher id). Using the the **`Select()` LINQ method** and **project the `House` entities** to **`HouseIndexServiceModel`**, as we should **pass view models to the view**. Make sure you fill in all the **`HouseIndexServiceModel` properties** with **correct data** from the **database**. Finally, we take the first three **house models.**

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

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

    2 references
    public async Task<IEnumerable<HouseIndexServiceModel>> LastThreeHouses()
    {
        return _data
               .Houses
               .OrderByDescending(c => c.Id)
               .Select(c => new HouseIndexServiceModel
               {
                   Id = c.Id,
                   Title = c.Title,
                   ImageUrl = c.ImageUrl
               })
               .Take(3);
    }
}
```

It's very important to remember to go to the "**`Program.cs`**" file and add the service like this:

```
builder.Services.AddTransient<IHouseService, HouseService>();
```

We set the **service** as **`transient`**, as we a **new instance to be created every time**.

At the end, **modify** the **`HomeController`** to use only the **`IHouseService` methods** and modify the **`Index()`** method. Don't forget to **inject the service through the constructor**, so that we can **use the service methods** when we create them. **Create a property** for the service and **set it**:

```
public class HomeController : Controller
{
    private readonly IHouseService _houses;
    0 references
    public HomeController(IHouseService houses)
    {
        _houses = houses;
    }

    0 references
    public async Task<IActionResult> Index()
    {
        var houses = await _houses.LastThreeHouses();
        return View(houses);
    }
```

Now, we have to modify the "**Index.cshtml**" file and it should accept an **IEnumerable<HouseIndexServiceModel>**. As the code is a lot, you can copy it from here:

```
@model IEnumerable<HouseIndexServiceModel>

@{
    ViewData["Title"] = "Home Page";
    var houses = Model.ToList();
}

@if (!houses.Any())
{
    <div class="mt-4 p-5 bg-light">
        <h1 class="display-4">Welcome to the best place for renting houses!</h1>
        <p class="lead">
            There are no houses in the system currently available
            so why don't you just add one?
        </p>
        <hr class="my-4">
        <p class="lead">
            @if (User.Identity.IsAuthenticated)
            {
                <a asp-controller="House" asp-action="Add" class="btn btn-primary
btn-lg"
           role="button">Add House</a>
            }
        </p>
    </div>
}

<div class="mb-5"></div>

<div id="carouselExampleControls" class="carousel slide" data-bs-ride="carousel">
    <div class="carousel-inner">
        @for (int i = 0; i < houses.Count(); i++)
        {
            var house = houses[i];
            <div class="carousel-item @(i == 0 ? "active" : string.Empty)">
                <img class="d-block w-100" style="height:500px"
                 src="@house.ImageUrl" alt="@house.Title">
```

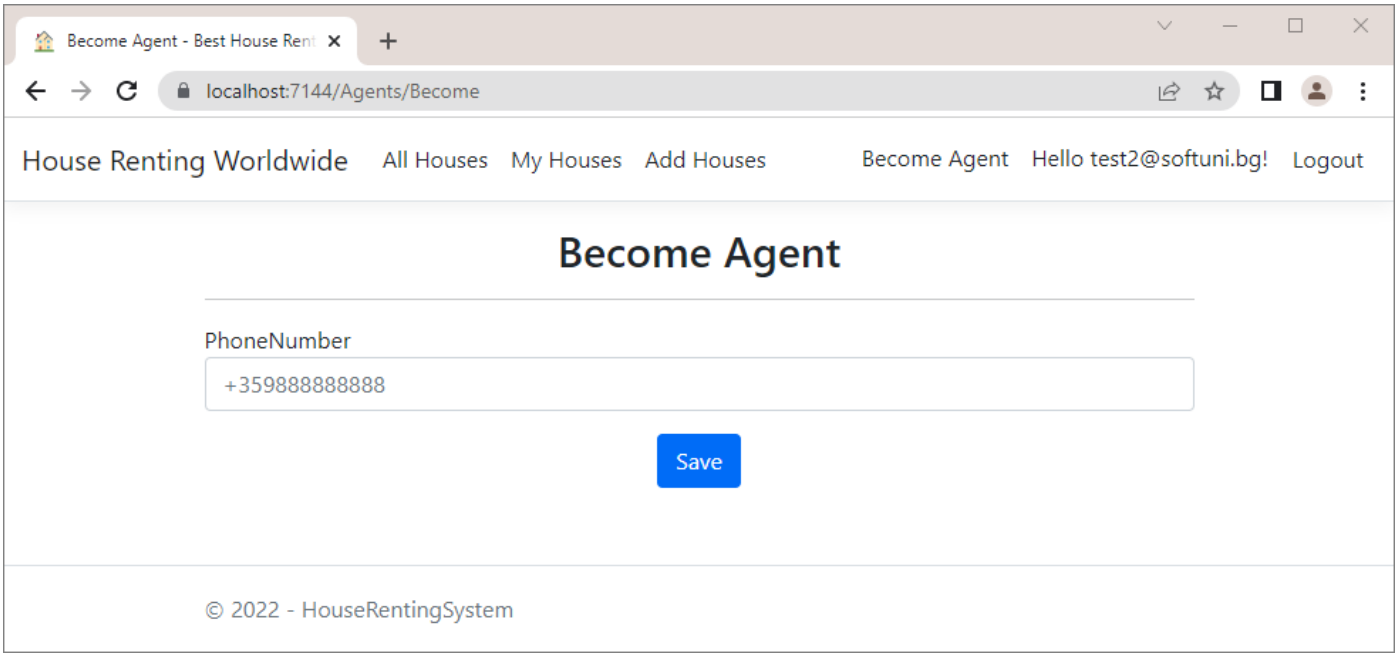
```
                <div class="carousel-caption d-none d-md-block">
                    <h5>
                        <a class="btn btn-primary" asp-controller="House" asp-
action="Details"
                       asp-route-id="@house.Id"> @house.Title</a>
                    </h5>
                </div>
            </div>
        }
    </div>
    <button class="carousel-control-prev" type="button" data-bs-
target="#carouselExampleControls" data-bs-slide="prev">
        <span class="carousel-control-prev-icon" aria-hidden="true"></span>
        <span class="visually-hidden">Previous</span>
    </button>
    <button class="carousel-control-next" type="button" data-bs-
target="#carouselExampleControls" data-bs-slide="next">
        <span class="carousel-control-next-icon" aria-hidden="true"></span>
        <span class="visually-hidden">Next</span>
    </button>
</div>
```

Try the "**Home**" **page** in the browser. You should see the **last three houses in a slideshow**.

13. Implement Agents

Our next step is to implement the Agents pages, services and functionalities.

The user should **access** the "**Become Agent**" **page** on "**/Agents/Become**" (only for **authorized users**, who are **not agents**). It should look like this:



Let's start by creating an "**Agent**" **folders** with an **IAgentService interface** and a **AgentService class** in it. Note that the **AgentService class** should **inherit IAgentService**.

```
public interface IAgentService
{
}
```

```
public class AgentService : IAgentService
{
}
```

Now go to **Program.cs** and **add the service** like this:

```
builder.Services.AddTransient<IAgentService, AgentService>();
```

Now go to the **AgentController** and **inject the service through the constructor**, so that we can **use the service methods** when we create them. **Create a property** for the service and **set it** like this:

```
[Authorize]
1 reference
public class AgentController : Controller
{
    private readonly IAgentService _agents;

    0 references
    public AgentController(IAgentService agents)
    {
        _agents = agents;
    }
```

We want to **write the business logic in a service method**. Start with the **Become() controller method** – we want it to **invoke a service method** to **check whether the currently logged-in user already exists as an agent**. The **service method**, on the other hand, will **accept parameters** from the controller action and use them to **return a result**.

Go to the **IAgentService class** and **declare a method**, which accepts an **user id** and **returns a bool**, depending on whether an **agent with the user id exists**:

```
public interface IAgentService
{
    1 reference
    Task<bool> ExistsById(string userId);
}
```

Now **implement the method** in the **AgentService class**. Before that, however, we need to **add** the **db context through the constructor** and **assign it to a property**, so that we can **obtain data from the database**. Do it as shown below:

```
public class AgentService : IAgentService
{
    private readonly HouseRentingDbContext _data;

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

Write the **logic** for the **method** like this:

```
    public async Task<bool> ExistsById(string userId)
    {
        return await _data.Agents.AnyAsync(a => a.UserId == userId);
    }
```

Go to **AgentController** and **modify** the **Become() method** to **use the service**:

```
[HttpPost]
0 references
public async Task<IActionResult> Become()
{
    if (_agents.ExistsById(User.Id()))
    {
        return BadRequest();
    }
    return View();
}
```

As you can see, the **ClaimsPrincipal** does not contain a definition for **Id**, so we'll have to add it. To **get the current user id**, let's create a **separate class** called **ClaimsPrincipalExtensions** in a **new folder** "**Infrastructure**". The reason for this is that we will need the **id** in **several classes** and want our code to be reusable. We named the **ClaimsPrincipalExtensions class** like this because it will **extend** the **ClaimsPrincipal class**, which supports **mu**ltiple **claims-based identities** (e.g. user information).

Write a method for **getting the current user id** in the class like this:

```
public static class ClaimsPrincipalExtentions
{
    0 references
    public static string Id(this ClaimsPrincipal user)
    {
        return user.FindFirst(ClaimTypes.NameIdentifier).Value;
    }
}
```

Now let's use **service methods** in the **Become(BecomeAgentFormModel model) controller method**, as well. In it, we **check whether the current user exists as an agent**, **whether an agent with a phone number exists** and whether the **user has any rents in the system**. Then, we create a **new Agent entity** and **save it to the database**.

Go to the **IAgentService class** and **define methods** for the above functionalities:

```
public interface IAgentService
{
    2 references
    Task<bool> ExistsById(string userId);

    1 reference
    Task<bool> UserWithPhoneNumberExists(string phoneNumber);

    1 reference
    Task<bool> UserHasRents (string userId);

    1 reference
    Task Create(string userId, string phoneNumber);
}
```

Implement the methods in the **AgentService class** like shown below. First, we need to check if the **current user is not already an agent**. This is done again for more security. Then we check if the user has any rents and if another agent with that phone number exists.

Next, after we are sure that we have **valid data from the model**, we will create the **`Agent`**. To do this, we should first **get the current user `id`**, as our **`Agent` class** has a **`UserId` property**. Then we **add the new `Agent` record** to the **database** through the **db `context`** class. Don't forget to **save the changes**. Do it like this:

```
    public async Task<bool> UserWithPhoneNumberExists(string phoneNumber)
    {
        return await _data.Agents.AnyAsync(a => a.PhoneNumber == phoneNumber);
    }

    1 reference
    public async Task<bool> UserHasRents(string userId)
    {
        return await _data.Houses.AnyAsync(h => h.RenterId == userId);
    }


    1 reference
    public async Task Create(string userId, string phoneNumber)
    {
        var agent = new HouseRentingSystem.Data.Models.Agent()
        {
            UserId = userId,
            PhoneNumber = phoneNumber
        };

        await _data.Agents.AddAsync(agent);
        await _data.SaveChangesAsync();
    }
}
```

Before using these methods in the `Become(BecomeAgentFormModel model)` method in the **`AgentController` class**, we should write the properties for the **`BecomeAgentFormModel`** that we have already created.

```
public class BecomeAgentFormModel
{
    [Required]
    [StringLength(PhoneNumberMaxLength, MinimumLength = PhoneNumberMinLength)]
    [Display(Name = "Phone Number")]
    [Phone]
    0 references
    public string PhoneNumber { get; set; } = null!;
}
```

The returned **view** will be "**`Become.cshtml`**". In it, our model properties names will be displayed as they are in the model class. We don't want to have a "**`PhoneNumber`**" **label** on the "**`Become Agent`**" page. It should be "**`Phone Number`**".

For this reason, we will **add the `[Display(Name = "")]` attribute** over the **`BecomeAgentFormModel` class property `PhoneNumber`** to **set its display name in views**.

Now that we have our model, we can use **the methods** in the **`Become(BecomeAgentFormModel model)`** method in the **`AgentController` class**. First, we need to check again if there is a **user with the `id`** of the **current user** as a **`UserId`**, this means that the **current user is already an agent** and a **`BadRequest`** should be returned and if the current user is **not an agent**, a **view** should be returned.

We have to **add some other validations** for **becoming an agent**, using the **`ModelState` class**. If a user tries to **become an agent** with a **phone number of another agent**, we should display an **error**. Also, a **user should have no current rents** to become an agent. Then, the **`ModelState.IsValid`** check will make sure that the **error is displayed in the browser**.

```
[HttpPost]
2 references
public async Task<IActionResult> Become(BecomeAgentFormModel model)
{
    var userId = User.Id();

    if (await _agents.ExistsById(userId))
    {
        return BadRequest();
    }

    if (await _agents.UserWithPhoneNumberExists(model.PhoneNumber))
    {
        ModelState.AddModelError(nameof(model.PhoneNumber),
            "Phone number already exists. Enter another one.");
    }

    if (await _agents.UserHasRents(userId))
    {
        ModelState.AddModelError("Error",
            "You should have no rents to become an agent!");
    }

    if (!ModelState.IsValid)
    {
        return View(model);
    }

    await _agents.Create(userId, model.PhoneNumber);

    return RedirectToAction(nameof(HouseController.All), "House");
}
```

Now let's create the "**`Become Agent`**" **form** in the view, so that the **user can send data** to the app.

In MVC, **controllers pass a model to the views** if views need its data. Then, **views display model data** or (if there is a form in the view) **send a model** filled with submitted data to the **controller**. In our case, the **`Become` view** should have a **form for submitting a phone number**, which will be used to populate the **`BecomeAgentFormModel`**. The **`BecomeAgentFormModel`** will be then **returned to the controller**.

To use the **`BecomeAgentFormModel`** in the "**`Become.cshtml`" view**, add it like this:

Become.cshtml

```
@model BecomeAgentFormModel

@{
    ViewBag.Title = "Become Agent";
}
```

Then, **add a form** with an **input field** for the **PhoneNumber property** of the **model class**. Use the **asp-for input tag helper** to bind an **HTML <input> element** to a **model property** in your **Razor view**:

You can get the code from here as it is a lot to write:

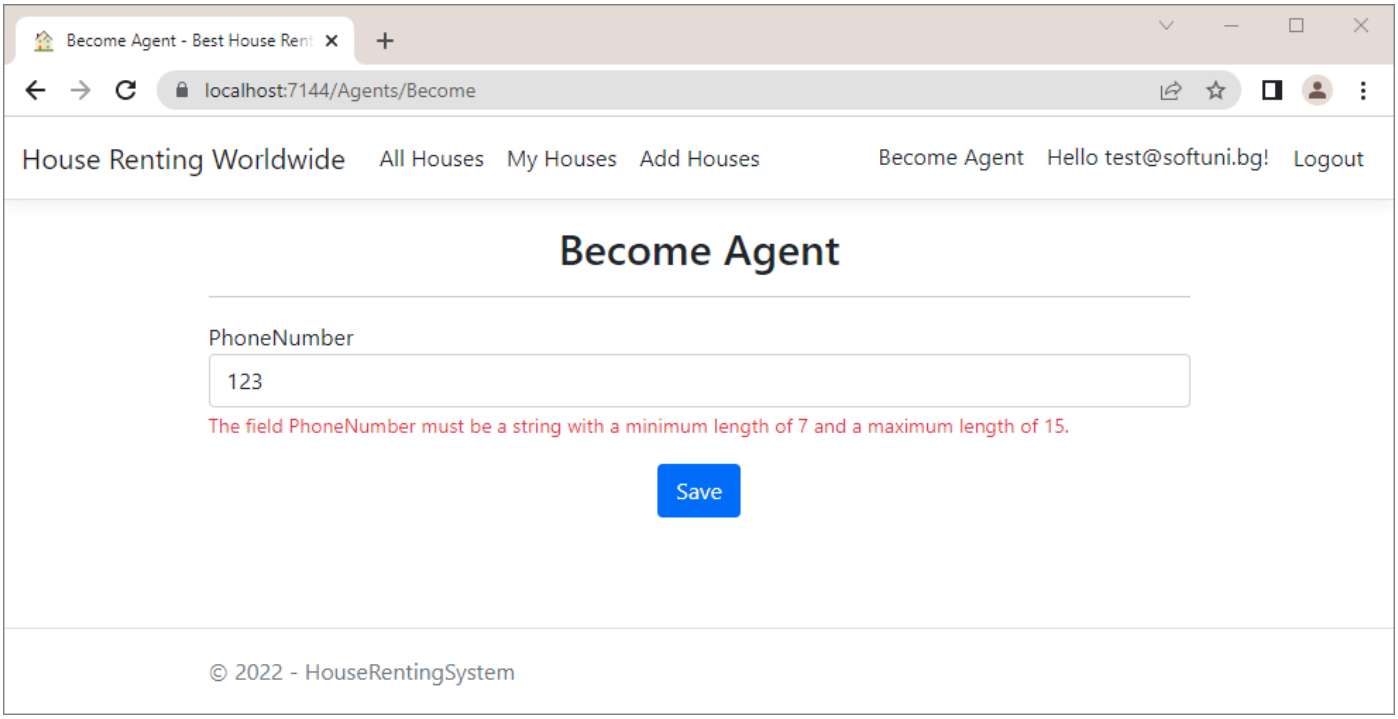
```
@model BecomeAgentFormModel

@{
    ViewBag.Title = "Become Agent";
}

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

@if (!ViewData.ModelState.IsValid && ViewData.ModelState["Error"] != null)
{
    <div class="alert alert-danger" role="alert">
        @ViewData.ModelState["Error"].Errors.First().ErrorMessage
    </div>
}

<div class="row">
    <div class="col-sm-12 offset-lg-2 col-lg-8 offset-xl-3 col-xl-6">
        <form method="post">
            <div class="form-group">
                <label asp-for="PhoneNumber"></label>
                <input asp-for="PhoneNumber" class="form-control"
placeholder="+359888888888">
                <span asp-validation-for="PhoneNumber" class="small text-
danger"></span>
            </div>
            <div class="text-center">
                <input class="btn btn-primary mt-3 " type="submit" value="Save" />
            </div>
        </form>
    </div>
</div>

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

Note that we use the **asp-validation-for tag helper** to **apply the model property restrictions** directly to the **form field**. For example, the **[PhoneNumber] attribute** over the **PhoneNumber property** in the **BecomeAgentFormModel** will validate that the form field has a **valid phone number**.

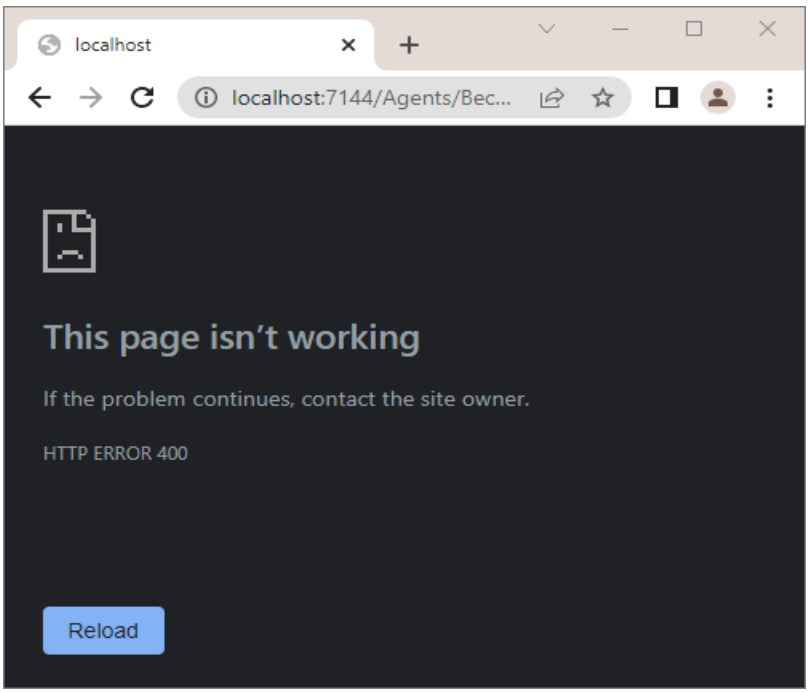
At the end, we will **define a section** in our Become **view**, which will **invoke** the "**_ValidationScriptsPartial.cshtml**" **partial view**. It checks whether the **entered form data is valid before the form is submitted**. We will see how this is done later.

Also, note that we give a **custom name** to the **model error**, as it is not connected to a property. For this reason, we should also go to the **"`Become.cshtml`" view** and **modify it to display the error**, as it won't do it by itself.

We have the first functionality in our app – the one for **becoming an agent**. Test it by **registering** or **logging** in the app and accessing "**`/Agents/Become`**". Examine the "**`Become Agent`**" **page functionality** in the browser. The page should **work correctly**. Try to **become an agent with user**, who is **already an agent**, and with one, **who is not**. Try to **fill an invalid phone number** – an **error** should appear:



Then, **fill in a valid phone number**. You should be **redirected** to the "**`All Houses`**" **page** after **pressing** the **`[Save]` button**. When you are **already an agent** and you try to **access** the "**`Become Agent`**" **page** again, a **`BadRequest`** will be returned:

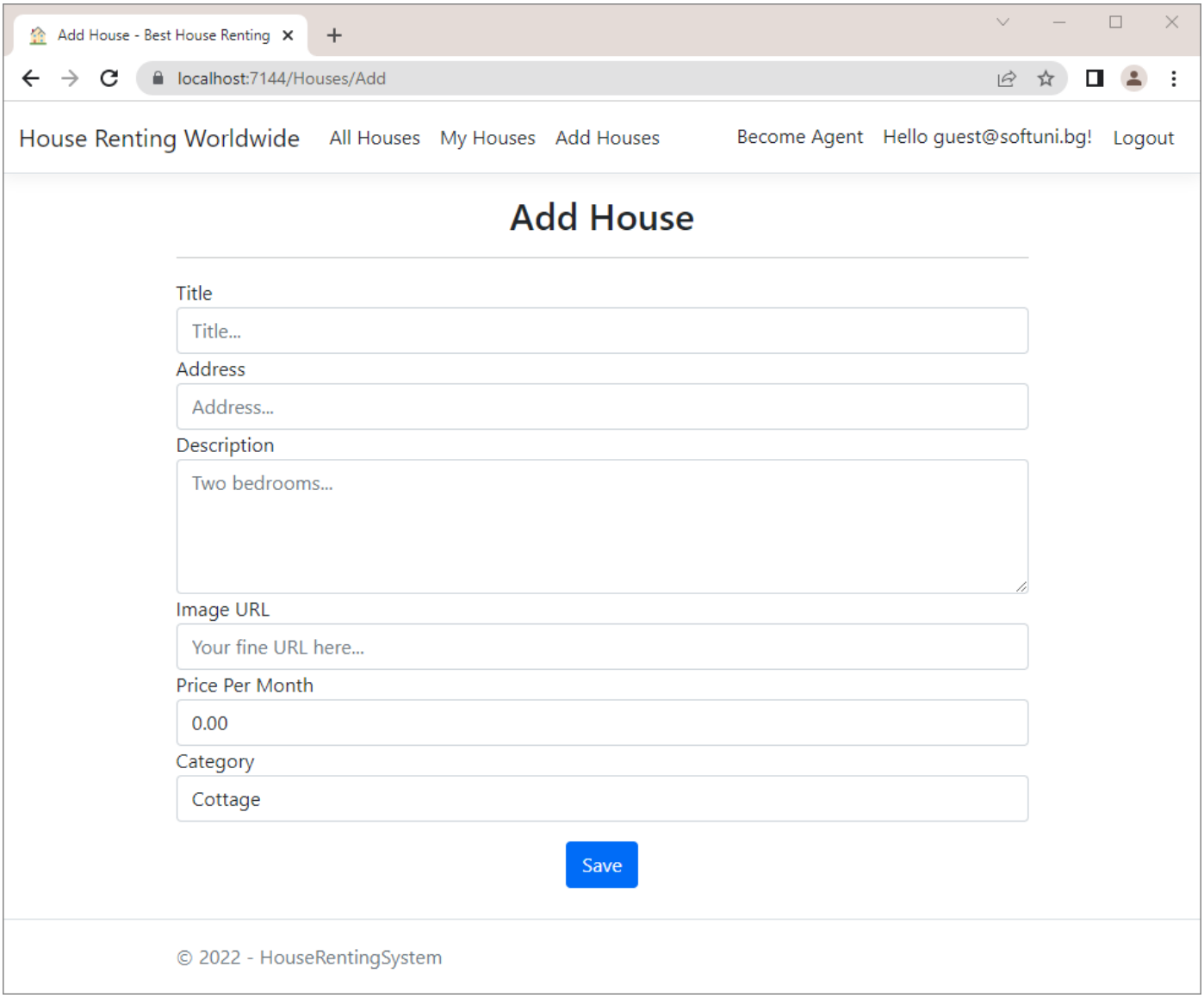


Look at the "**`Agents`**" **table** in the **database** in **`SQL Server Management Studio`**. It should have the new **`Agent` record**:

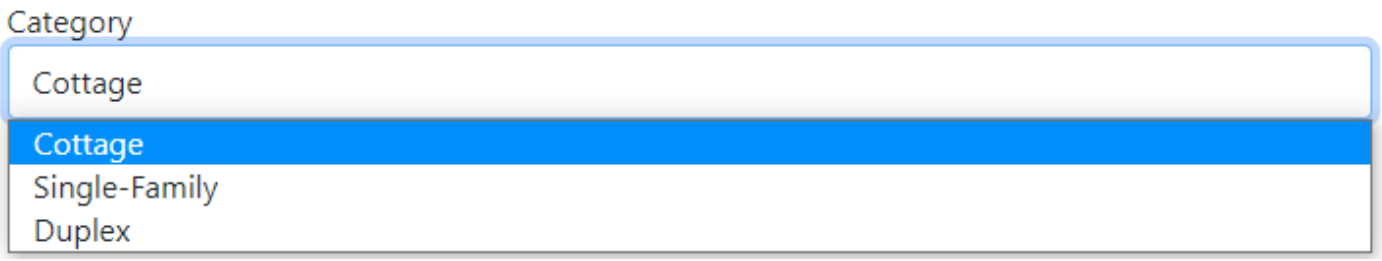
	Id	PhoneNumber	UserId
1	1	+359888888888	dea12856-c198-4129-b3f3-b893d8395082
2	2	+359888123123	68d4176d-087b-4284-b100-ce0f8face68b

14. Implement "Add House" page

Now it's time to implement the "**Add House**" **page** on "**/House/Add**" (only for **authorized users**, who are **agents**).



The user should **fill in the** "**Add House**" **form** with **valid data**, **choose a category** and press **[Submit]**. Note that available **categories should be displayed in the dropdown menu** like this:



The **form data** should be used for creating a new **House record** in the **database** and the user should be **redirected** to the "**Details**" **page** of the **newly-created house**.

The methods, which implement the above functionalities, are the **`Add()`** and **`Add(HouseFormModel model)`** **methods** in the **`HouseController` class**. Before we modify the **controller actions**, we have to **inject the service**:

```
[Authorize]
2 references
public class HouseController : Controller
{
    private readonly IHouseService _houses;

    0 references
    public HouseController(IHouseService houses)
    {
        _houses = houses;
    }
```

For this controller action, we need to **add a single service method** to **get all categories from the database** and **return them as a collection**.

First, let's create the **`HouseCategoryServiceModel`** in the "**`/Services/Houses/Models`**" **folder** for the service, because we'll need it in a few minutes. It should look like this:

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

    0 references
    public string Name { get; set; } = null!;
}
```

Now we should modify the HouseFormModel, as well. This class should have **properties** for **title**, **address**, **description**, **image URL**, **price per month**, **category id** and a **collection for categories**. As this is a model for **submitting a form**, we should **add restrictions** for **min** and **max length**, **range**, etc.

Use the **`[StringLength(…, MinimumLength = …)]` attribute** to **set max and min length** of the **`Title`**, **`Address`** and **`Description` properties**. The **constants** for **min lengths** are part of the **`DataConstants` class**, that we have already created. Don't forget that you should add the **`[Required]` attribute** over **obligatory properties**. The **`ImageUrl` property** should also have that **attribute**. Next, **restrict** the **`PricePerMonth` property** to have **values between 0 and 20000**. To do this, use the **`[Range]` attribute** and **add the min and max values**. Also, add a **custom error message** so that it is more user-friendly. Finally, add the **`CategoryId` property**, which holds the **selected category id**, and a **collection of category models**, as the user should have the **categories to choose from**.

```
public class HouseFormModel
{

    [Required]
    [StringLength(TitleMaxLength, MinimumLength = TitleMinLength)]
    0 references
    public string Title { get; set; } = null!;

    [Required]
    [StringLength(AddressMaxLength, MinimumLength = AddressMinLength)]
    0 references
    public string Address { get; set; } = null!;

    [Required]
    [StringLength(DescriptionMaxLength, MinimumLength = DescriptionMinLength)]
    0 references
    public string Description { get; set; } = null!;

    [Required]
    [Display(Name = "Image URL")]
    0 references
    public string ImageUrl { get; set; } = null!;

    [Required]
    [Range(0.00, MaxPricePerMonth,
        ErrorMessage = "Price Per Month must be a positive number and less than {2} leva.")]
    [Display(Name = "Price Per Month")]
    0 references
    public decimal PricePerMonth { get; set; }

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

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

Now, implement the **service method** for **getting all categories** like this:

```
public interface IHouseService
{
    2 references
    Task<IEnumerable<HouseIndexServiceModel>> LastThreeHouses();

    0 references
    Task<IEnumerable<HouseCategoryServiceModel>> AllCategories();
}
```

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

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

    1 reference
    public async Task<IEnumerable<HouseCategoryServiceModel>> AllCategories()
    {
        return await _data
                .Categories
                .Select(c => new HouseCategoryServiceModel
                {
                    Id = c.Id,
                    Name = c.Name,
                })
                .ToListAsync();
    }
```

Now it's time to modify the **`Add()`** method in the **`HouseController`** in order for it to use the above methods. It should look like this. In our app, **only agents are allowed to add houses**, so check if the **current user is an agent**. If they are not, **redirect** them to the "**`Become Agent`**" **page**:

```
    public async Task<IActionResult> Add()
    {
        if (await _agents.ExistsById(User.Id()) == false)
        {
            return RedirectToAction(nameof(AgentController.Become), "Agent");
        }

        return View(new HouseFormModel
        {
            Categories = await _houses.AllCategories()
        });
    }
```

Don't forget to **accept the `IAgentService` through the constructor**:

```
[Authorize]
2 references
public class HouseController : Controller
{
    private readonly IHouseService _houses;
    private readonly IAgentService _agents;

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

Now we should **modify** the **`Add(HouseFormModel model)` method**. For it, we need to **create service methods** to **check whether a category with a given id exists** and to **create a new `House` entity in the database**.

Implement the following methods in **`IHouseService`** and **`HouseService` classes**:

```
    Task<bool> CategoryExists(int categoryId);

    1 reference
    Task<int> Create(string title, string address,
        string description, string imageUrl, decimal price,
        int categoryId, int agentId);
```

```
    public async Task<bool> CategoryExists(int categoryId)
    {
        return await _data.Categories.AnyAsync(c => c.Id == categoryId);
    }

    2 references
    public async Task<int> Create(string title, string address, string description,
        string imageUrl, decimal price, int categoryId, int agentId)
    {
        var house = new HouseRentingSystem.Data.Models.House()
        {
            Title = title,
            Address = address,
            Description = description,
            ImageUrl = imageUrl,
            PricePerMonth = price,
            CategoryId = categoryId,
            AgentId = agentId
        };

        await _data.Houses.AddAsync(house);
        await _data.SaveChangesAsync();

        return house.Id;
    }
```

As you can see, the **`Create(…)` service methods** returns the **new house id** because our **controller** needs it. The **`Add(HouseFormModel model)` controller methods** should **use service methods** as shown below:

```
[HttpPost]
0 references
public async Task<IActionResult> Add(HouseFormModel model)
{
    if (await _agents.ExistsById(User.Id()) == false)
    {
        return RedirectToAction(nameof(AgentController.Become), "Agent");
    }

    if (await _houses.CategoryExists(model.CategoryId) == false)
    {
        this.ModelState.AddModelError(nameof(model.CategoryId),
            "Category does not exist.");
    }

    if (!ModelState.IsValid)
    {
        model.Categories = await _houses.AllCategories();

        return View(model);
    }

    var agentId = await _agents.GetAgentId(User.Id());

    var newHouseId = await _houses.Create(model.Title, model.Address,
        model.Description, model.ImageUrl, model.PricePerMonth,
        model.CategoryId, agentId);

    return RedirectToAction(nameof(Details), new { id = newHouseId });
}
```

You can see that our **IAgentService** interface doesn't have a **GetAgentId method,** so our next task is to **implement** it**.**

Go to the **IAgentService** and write it:

```
public interface IAgentService
{
    1 reference
    Task<int> GetAgentId(string userId);
```

Next, go to the **AgentService** class and write the **service method**:

```
public async Task<int> GetAgentId(string userId)
{
    return _data.Agents.FirstOrDefaultAsync(a => a.UserId == userId).Id;
}
```

The last for completing this functionality is implementing the view. We have already created the **"Add.cshtml"** file in the **/Views/Houses** folder. We have to **add** the **HouseFormModel** to the view.

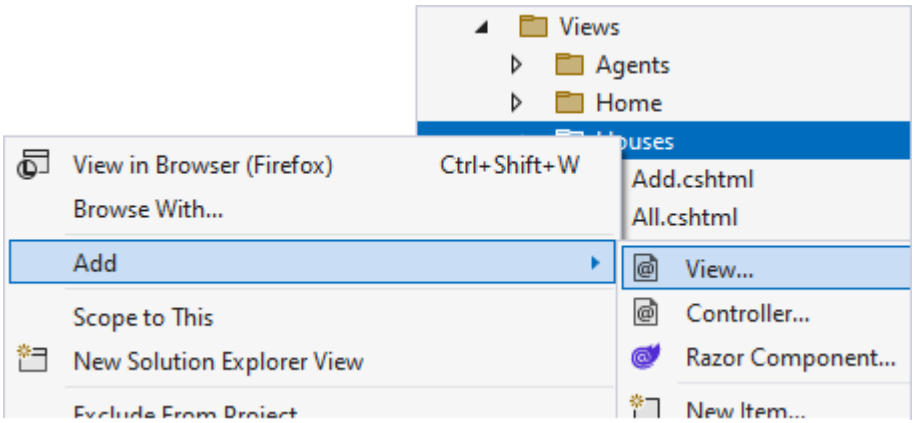
Then, we should **add the form**. However, we will **reuse this form** for the "**`Edit`**" **functionality** later. For this reason, we will create a **partial view**, called "**`_HouseFormPartial.cshtml`**", where we will **keep the form**.

In the **Add view** you should just **render the partial view** with the **`HouseFormModel`** and at the end, don't forget to **render** the **`_ValidationScriptsPartial`**, as well, because we have a **form to be checked**:

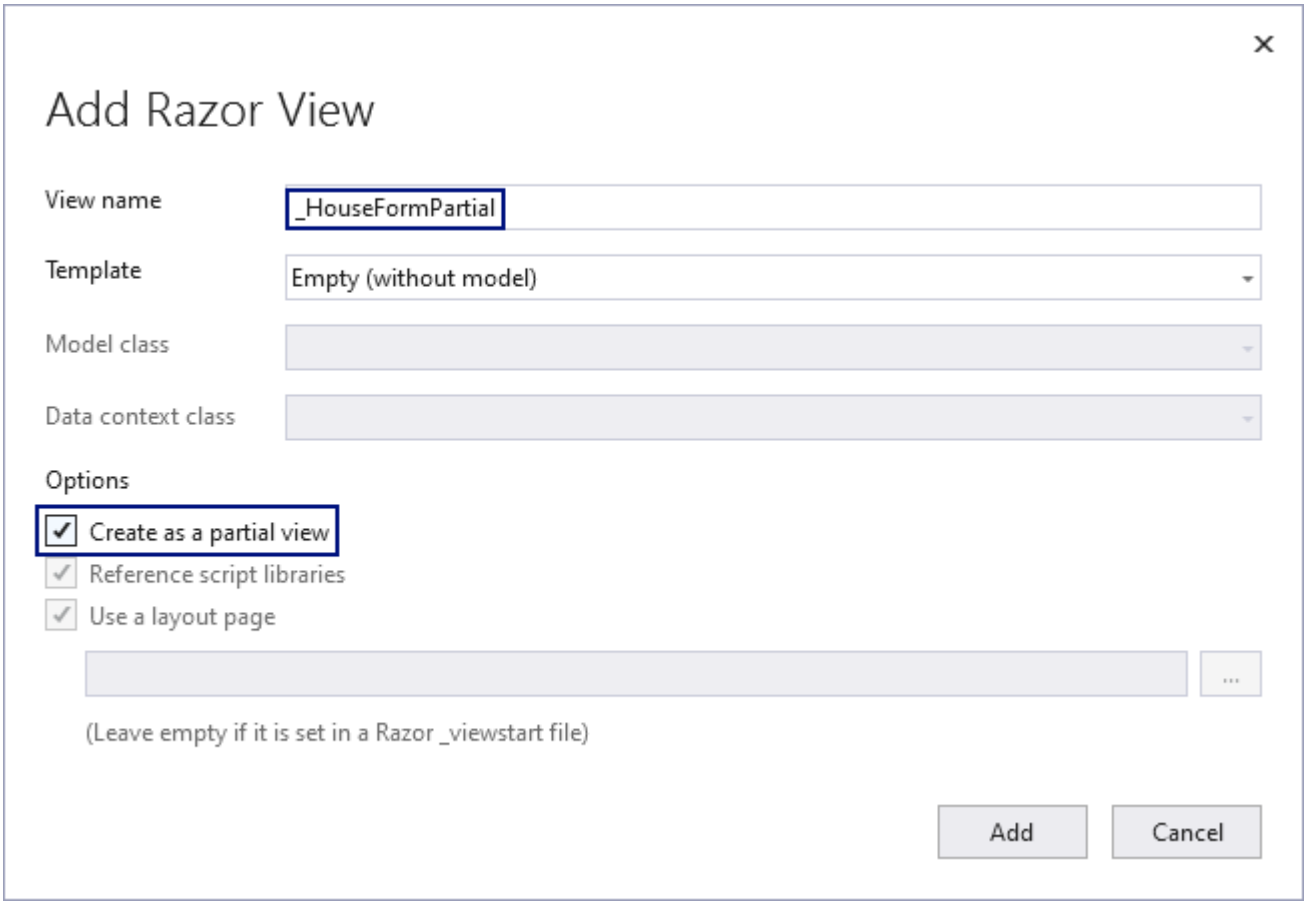
Add.cshtml

```
@{
    ViewBag.Title = "Add House";
}

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

<partial name="_HouseFormPartial" model="@Model" />

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

Create the "**`_HouseFormPartial.cshtml`**" **partial view**. To do this, **right-click** on the "**`/Views/House`**" **folder** and on **`[Add]`** → **`[View]`** and **choose** **`[Razor View]`**:



On the next window, **fill in the view name** and **check** "**`Create as a partial view`**":



The **new partial view** should first accept a **HouseFormModel** from the **view** that **renders** it. As the code is a lot to write, you can copy it from here:

```
@model HouseFormModel

<div class="row">
    <div class="col-sm-12 offset-lg-2 col-lg-8 offset-xl-3 col-xl-6">
        <form method="post">
            <div class="form-group">
                <label asp-for="Title"></label>
                <input asp-for="Title" class="form-control" placeholder="Title...">
                <span asp-validation-for="Title" class="small text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Address"></label>
                <input asp-for="Address" class="form-control"
placeholder="Address...">
                <span asp-validation-for="Address" class="small text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Description"></label>
                <textarea asp-for="Description" rows="4" class="form-control"
placeholder="Two bedrooms..."></textarea>
                <span asp-validation-for="Description" class="small text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ImageUrl"></label>
                <input asp-for="ImageUrl" class="form-control" placeholder="Your
fine URL here...">
```

```
                <span asp-validation-for="ImageUrl" class="small text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="PricePerMonth"></label>
                <input asp-for="PricePerMonth" class="form-control">
                <span asp-validation-for="PricePerMonth" class="small text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="CategoryId"></label>
                <select asp-for="CategoryId" class="form-control">
                    @foreach (var category in Model.Categories)
                    {
                        <option value="@category.Id">@category.Name</option>
                    }
                </select>
                <span asp-validation-for="CategoryId" class="small text-
danger"></span>
            </div>

            <div class="text-center">
                <input class="btn btn-primary mt-3" type="submit" value="Save" />
            </div>
        </form>
    </div>
</div>
```

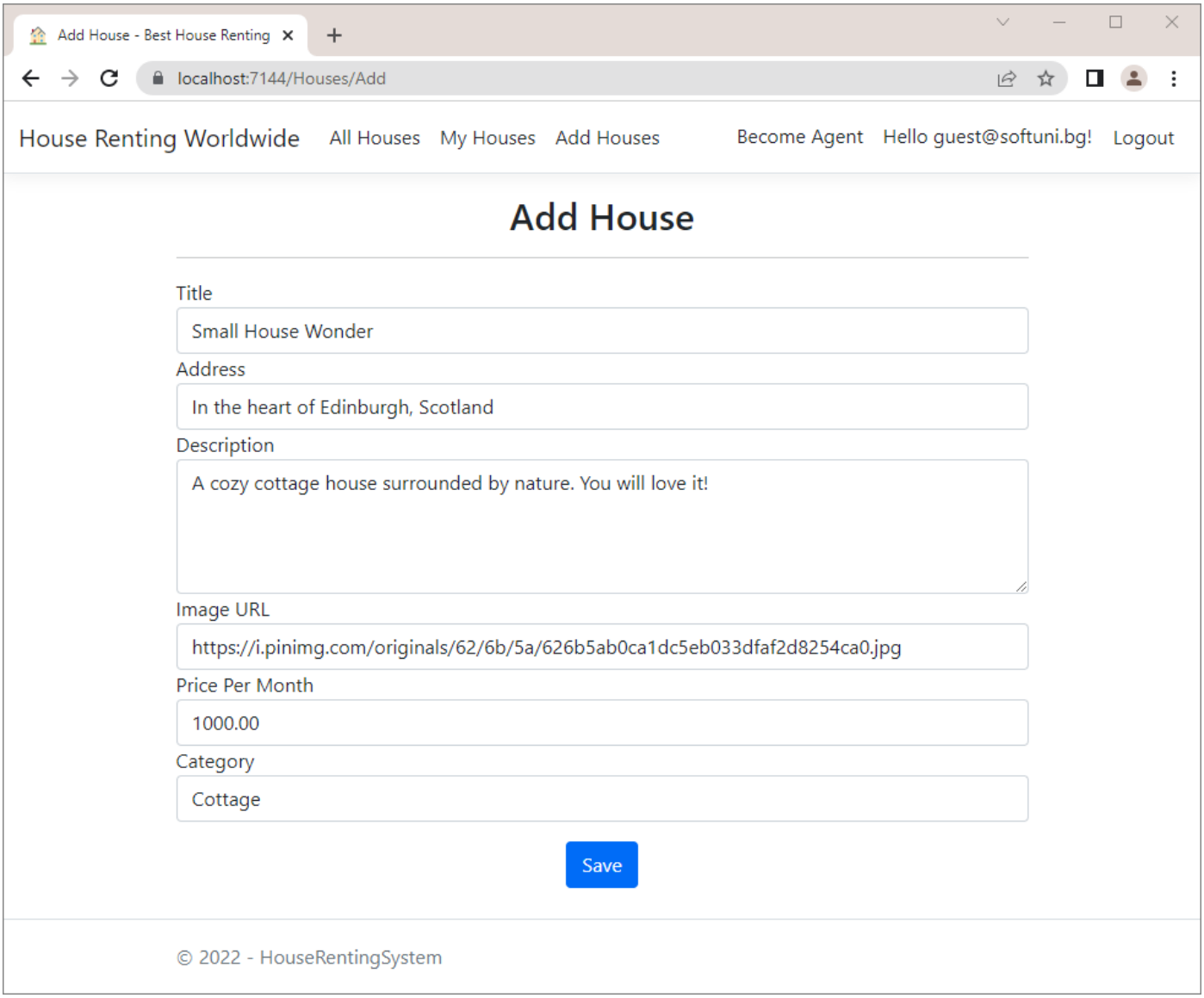
Note that we used a **`foreach` loop** in a **Razor view** to iterate through the **`Categories` collection** in the **model**. Also, look at the **`<label>` tags** of the **form** – they use the **class property name**. However, our properties have names like "**`ImageURL`**", "**`CategoryId`**", etc., which should **not be displayed** like that. That is why we used the **`[Display(Name = "")]` attribute** over some of the **`HouseFormModel` class properties**.

Now it's time to test the "**`Add Houses`**" functionality. Register and log in in the app with a user that is not yet an **`Agent`**. Go to the "**`Add Houses`**" page and you should be redirected to the "**`Become Agent`**" page as adding houses can be performed only by agents. Modify your account to become an agent and try again the "**`Add Houses`**" – you should see the "**`Add Houses`**" page.

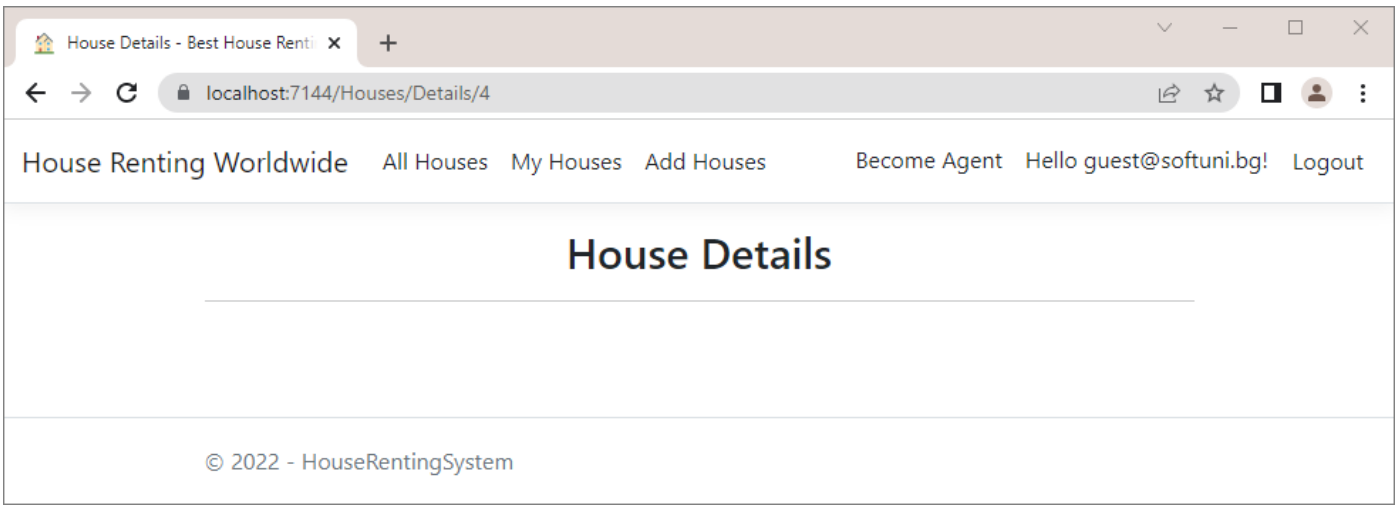
Try filling in some **invalid data** – you should see that **errors appear**. Then, **fill in valid data** and you should be **redirected** to the "**`Details`**" **page of the new house**:

You can get the above image URL from here:

https://i.pinimg.com/originals/62/6b/5a/626b5ab0ca1dc5eb033dfaf2d8254ca0.jpg



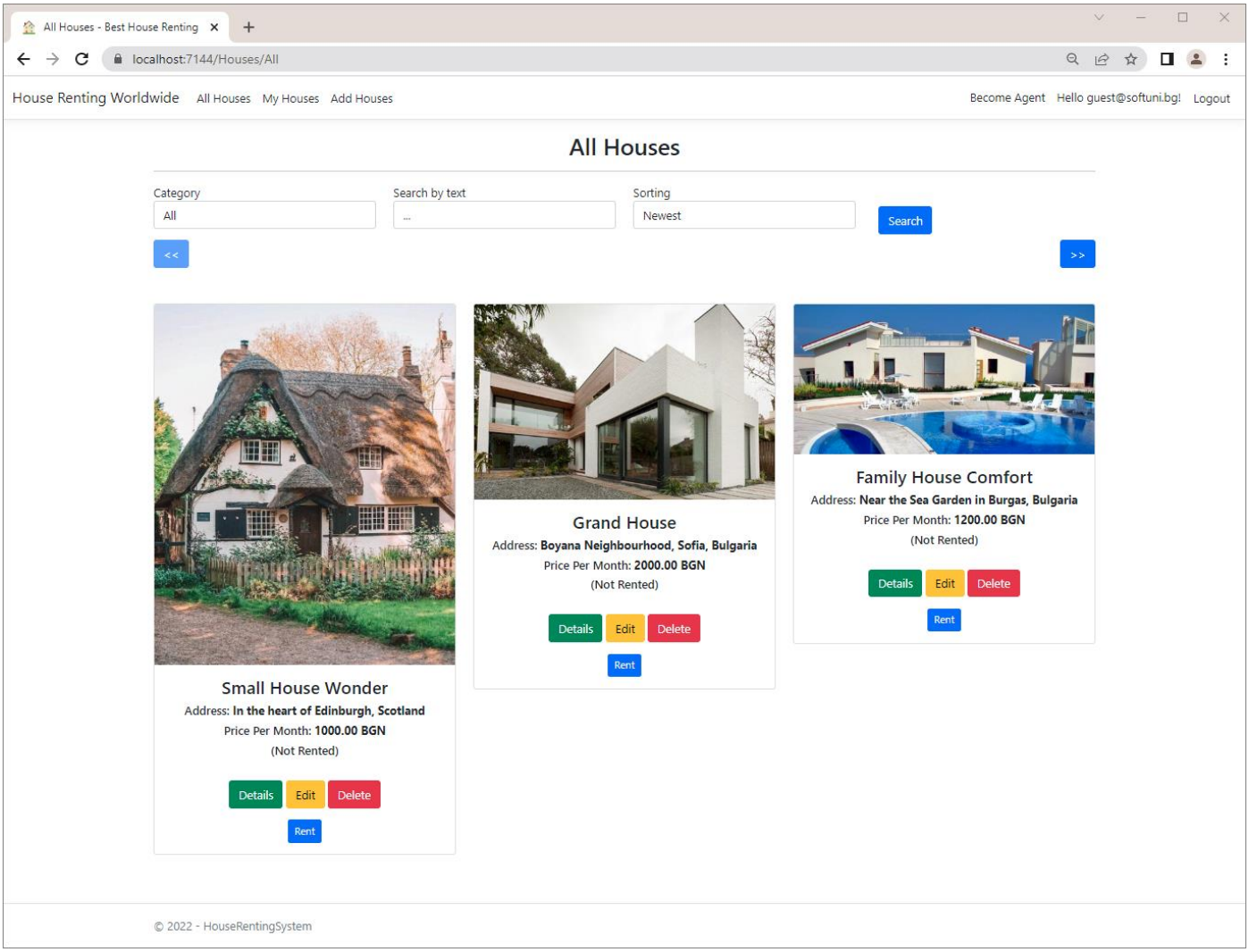
Upon pressing the [**Save**] button, you are redirected to the "**House Details**" page, but we have not yet implemented this functionality. We will do this later in the workshop, because first we must implement the "**All Houses**" page, which will give us access to the "**Edit House**" and "**Delete House**" functionalities.



Before that, look at the **new house** in the "**Houses**" **table** of the **database** and make sure it is there.

15. Implement "All Houses" page

Let's start implementing the "**`All Houses`**" page. It should **display the houses from the database** and **implement searching**, **sorting** and **paging functionalities**. It should look like this:



Start with the **`All([FromQuery] AllHousesQueryModel query)` controller method** – we want it to **invoke a service method** and **pass the result model to a view**. The service method, on the other hand, will accept **parameters from the controller action** and use them to **return a model**.

We want to create the following **workflow** in our app:

- The **controller accepts a model** from the **query string**
- It invokes a **service method** to **get the houses and their count** as a **model**
- Invokes a **service method** to **get the house categories**
- Updates the **query model's** houses, houses count and categories
- Passes the model to the **view**

First, let's add an **enum class** for the **house sorting**. Create the **`HouseSorting` class** in the "**`Infrastructure`**" **folder**. The class should have the **options for sorting** – we can **sort houses by newest**, by **price** or by **not-rented first**. The **enum class** looks like this:

```
public enum HouseSorting
{
    Newest = 0,
    Price = 1,
    NotRentedFirst = 2
}
```

Now, to implement the above-described **workflow** in the controller method, let's first **create the service methods**. Go to the **`IHouseService` class** and **declare a method for getting all houses and their count** and one for **getting the house categories**.

The **`All(…)` method** in the **`IHouseService` class** needs a **category name**, a **search term**, a **`HouseSorting` value** and the **current page** and **houses per page values** to filter and return the correct houses. It should return a **model with the houses and their count**, called **`HouseQueryServiceModel`**.

Create the **`HouseQueryServiceModel`** in a separate **folder** called "**`Models`**" in "**`/Services/Houses`**" like this:

```
public class HouseQueryServiceModel
{
    0 references
    public int TotalHousesCount { get; set; }

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

As you can see, we are going to need a **`HouseServiceModel`**, so go to the "**`/Services/Houses/Models`**" and **create** the **model.** It should look like this:

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

    0 references
    public string Title { get; set; } = null!;

    0 references
    public string Address { get; set; } = null!;

    [DisplayName("Image URL")]
    0 references
    public string ImageUrl { get; set; } = null!;

    [DisplayName("Price Per Month")]
    0 references
    public decimal PricePerMonth { get; set; }

    [DisplayName("Is Rented")]
    0 references
    public bool IsRented { get; set; }
}
```

Define the **`All()` method** like this:

```
HouseQueryServiceModel All(string category = null,
    string searchTerm = null,
    HouseSorting sorting = HouseSorting.Newest,
    int currentPage = 1,
    int housesPerPage = 1);
```

The **`AllCategoriesNames()`** method should **return all category names** as a **collection of `string`**:

```
Task<IEnumerable<string>> AllCategoriesNames();
```

Implement the **methods** in the **`HouseService` class**:

```
public HouseQueryServiceModel All(string category = null,
   string searchTerm = null,
   HouseSorting sorting = HouseSorting.Newest,
   int currentPage = 1,
   int housesPerPage = 1)
{
    throw new NotImplementedException();
}

public async Task<IEnumerable<string>> AllCategoriesNames()
{
    throw new NotImplementedException();
}
```

Now we should implement the **`All(…)` method**. Implement the logic for getting and filtering the houses to the **`All(…)` service method** like shown below.

We get the **houses from the model collection** and **convert it** to **`IQueryable`**, so that we can **perform LINQ operations** on it.

Then, we check whether we need to **filter houses by a category** and **get only the houses from the given category**. If we have a **search term**, we should get only the **`House` entities**, which **contain** it in their **title**, **address** or **description**. **Searching** should be **case-insensitive**.

It's time to do the **sorting**, too. If the user has chosen to **sort by price**, you should **order the collection in descending order by price**. If they have chosen to **sort by not-rented first**, you should **get the not-rented first**, then the **rented ones** and **sort them descending by `id`** (so that newer are first). If the user has chosen to **sort by newest** or **haven't chosen a sorting criterion**, **sort the houses by `id`** in **descending order**.

Get the houses, which are supposed to be on the **current user page**, get the **needed number of houses** and **project** them to a **`List<HouseServiceModel>`**. **Assign these houses** to the **`Houses` property** of the **current `HouseQueryService` model**. Then, **fill in the `Categories` collection** of the **model** with the **categories' names**, **without repeating** them.

Get the **total houses count** and **assign it**, as well.

```
public HouseQueryServiceModel All(string category = null,
    string searchTerm = null,
    HouseSorting sorting = HouseSorting.Newest,
    int currentPage = 1,
    int housesPerPage = 1)
{
    var housesQuery = _data.Houses.AsQueryable();

    if (!string.IsNullOrWhiteSpace(category))
    {
        housesQuery = _data
            .Houses
            .Where(h => h.Category.Name == category);
    }

    if (!string.IsNullOrWhiteSpace(searchTerm))
    {
        housesQuery = housesQuery
            .Where(h =>
                h.Title.ToLower().Contains(searchTerm.ToLower()) ||
                h.Address.ToLower().Contains(searchTerm.ToLower()) ||
                h.Description.ToLower().Contains(searchTerm.ToLower()));
    }

    housesQuery = sorting switch
    {
        HouseSorting.Price => housesQuery
            .OrderBy(h => h.PricePerMonth),
        HouseSorting.NotRentedFirst => housesQuery
        .OrderBy(h => h.RenterId != null)
        .ThenByDescending(h => h.Id),
        _ => housesQuery.OrderByDescending(h => h.Id)
    };

    var houses = housesQuery
       .Skip((currentPage - 1) * housesPerPage)
       .Take(housesPerPage)
       .Select(h => new HouseServiceModel
       {
           Id = h.Id,
           Title = h.Title,
           Address = h.Address,
           ImageUrl = h.ImageUrl,
           IsRented = h.RenterId != null,
           PricePerMonth = h.PricePerMonth
       })
       .ToList();

    var totalHouses = housesQuery.Count();
```

At the end, **return** a **`HouseQueryServiceModel`** with the **needed properties**:

```
        return new HouseQueryServiceModel()
        {
            TotalHousesCount = totalHouses,
            Houses = houses
        };
    }
```

Implement the **`AllCategoriesNames()` method**, which should only **return the categories names**:

```
public async Task<IEnumerable<string>> AllCategoriesNames()
{
    return await _data
                    .Categories
                    .Select(c => c.Name)
                    .Distinct()
                    .ToListAsync();
}
```

Now go to the **`All([FromQuery] AllHousesQueryModel query)` method** in the **`HouseController` class** and implement it to **use the service methods** we created and **return a view with the model:**

```
[AllowAnonymous]
2 references
public async Task<IActionResult> All([FromQuery] AllHousesQueryModel query)
{
    var queryResult = _houses.All(
        query.Category,
        query.SearchTerm,
        query.Sorting,
        query.CurrentPage,
        AllHousesQueryModel.HousesPerPage);

    query.TotalHousesCount = queryResult.TotalHousesCount;
    query.Houses = queryResult.Houses;

    var houseCategories = _houses.AllCategoriesNames();
    query.Categories = (IEnumerable<string>)houseCategories;

    return View(query);
}
```

As you can see, we are using the **`AllHousesQueryModel`**. Go to the **model class** and **modify** it to look like this:

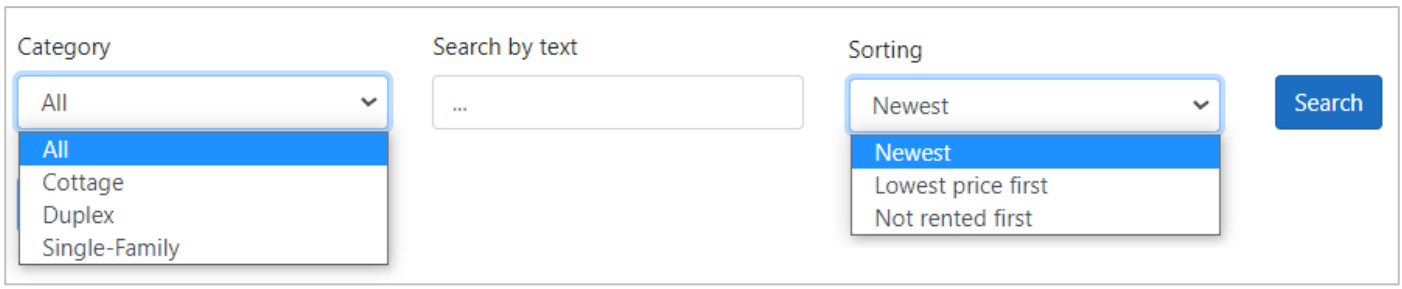
```
public class AllHousesQueryModel
{
    public const int HousesPerPage = 3;

    1 reference
    public string Category { get; init; } = null!;

    [Display(Name = "Search by text")]
    1 reference
    public string SearchTerm { get; init; } = null!;

    1 reference
    public HouseSorting Sorting { get; init; }

    1 reference
    public int CurrentPage { get; init; } = 1;

    1 reference
    public int TotalHousesCount { get; set; }

    1 reference
    public IEnumerable<string> Categories { get; set; } = null!;

    1 reference
    public IEnumerable<HouseServiceModel> Houses { get; set; }
        = new List<HouseServiceModel>();
}
```

Now you should also go and **modify** the **"`All.cshtml`" view** to use the **`HouseServiceModel`**. **Import the service model class namespace** in the **"`_ViewImports.cshtml`" file** like this:

_ViewImports.cshtml

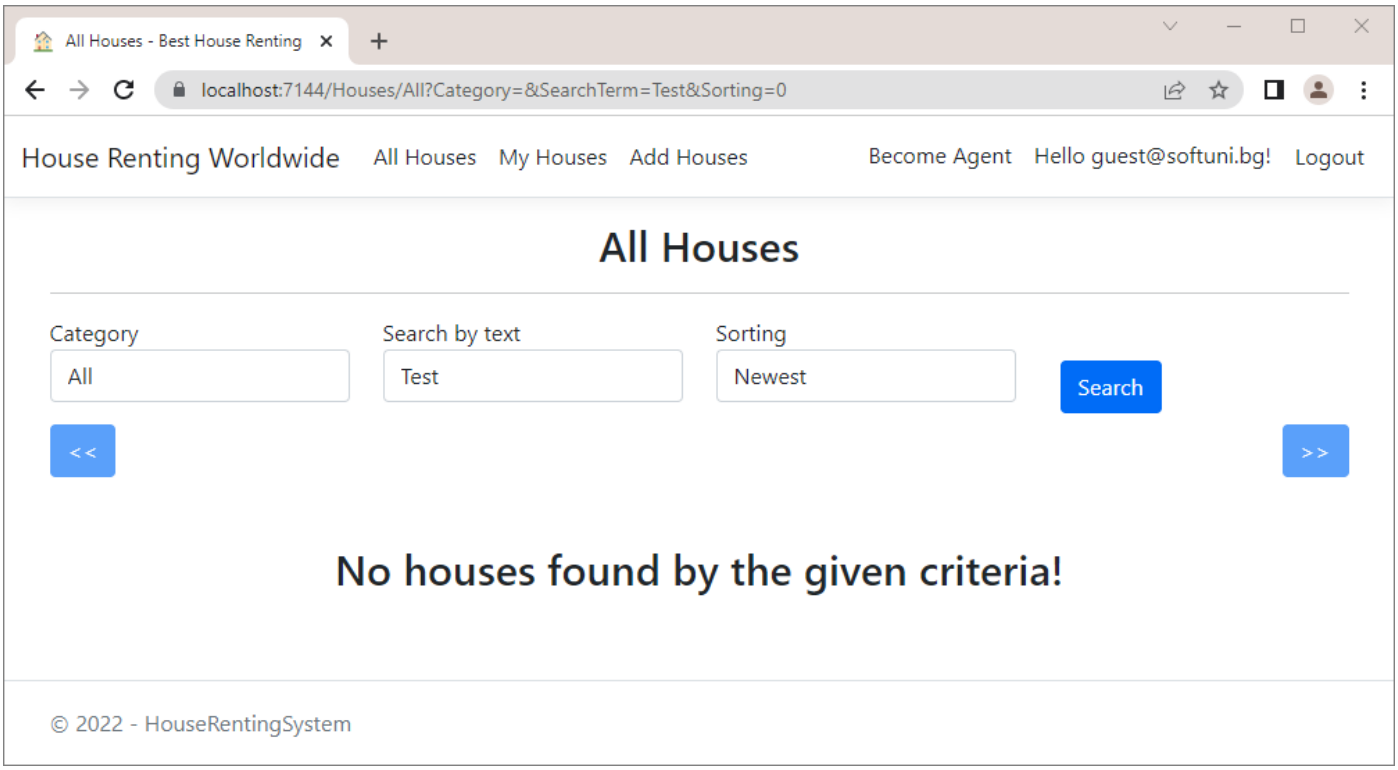
```
@using HouseRentingSystem
@using HouseRentingSystem.Models
@using HouseRentingSystem.Models.Home
@using HouseRentingSystem.Models.House
@using HouseRentingSystem.Models.Agent
@using HouseRentingSystem.Services.House
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Now write the rest of the code for the "**`All.cshtml`**" view. We need to **create a form**, which will send a "**`GET`**" **request** when **submitted**. In this way, **model properties** will be **added as query parameters** to the **URL**. The **form** will look like this in the browser:



As you can see, we should **first add a dropdown menu** with the **categories** from the **model**. We should also **add the search term field** and the **sorting dropdown**. We should also add a [**Search**] button. We should also add to the view the **paging functionality**, which should have **buttons** for the **previous and next pages** (they should be **enabled/disabled**, depending on whether there are houses on each page). It looks like this:

First, **open a code block** and **calculate the previous page**, depending on the **current one**, and the **maximal page with houses** (there should not be an empty page). We will need these variables later. Then, we should **create the button for the previous page**. If we are on the **first page**, we **disable the button**. Also, we should **add tag helpers** to the **button**. In this way, when the **button is clicked**, it will **send data for the current page**, **category**, **search term** and **sorting criteria**. Use the **`asp-route-{value}` tag helper** to **add potential route parameters**. Then, **open another code block** to **calculate if the button for the next page should be disabled** and **add** the button. At the end, if there are **no houses**, you should **display a message**:



If **there are houses**, **each of them should be displayed** by the **"`_HousePartial.cshtml`" partial view**, which we will implement next. As the code is a lot to write, you can copy it from here:

```
@model AllHousesQueryModel
```

```
@{
    ViewBag.Title = "All Houses";
}

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

<form method="get">
    <div class="row">
        <div class="form-group col-md-3 d-flex justify-content-between">
            <div class="form-group">
                <label asp-for="Category"></label>
                <select asp-for="Category" class="form-control">
                    <option value="">All</option>
                    @foreach (var category in Model.Categories)
                    {
                        <option value="@category">@category</option>
                    }
                </select>
            </div>
        </div>

        <div class="form-group col-md-3">
            <label asp-for="SearchTerm"></label>
            <input asp-for="SearchTerm" class="form-control" placeholder="...">
        </div>

        <div class="form-group col-md-3">
            <div class="form-group">
                <label asp-for="Sorting"></label>
                <select asp-for="Sorting" class="form-control">
                    <option value="0">Newest</option>
                    <option value="1">Lowest price first</option>
                    <option value="2">Not rented first</option>
                </select>
            </div>
        </div>

        <div class="col-md-3">
            <div class="form-group mt-4 p-2">
                <input type="submit" value="Search" class="btn btn-primary" />
            </div>
        </div>
    </div>
</form>


@{
    var previousPage = Model.CurrentPage - 1;
    if (previousPage < 1)
    {
        previousPage = 1;
    }

    var maxPage = Math.Ceiling((double)Model.TotalHousesCount /
        AllHousesQueryModel.HousesPerPage);
```

```
}

<div class="row mb-5">
    <div class="col-md-6 d-grid gap-2 d-md-flex justify-content-md-start">
        <a class="btn btn-primary @(Model.CurrentPage == 1 ? "disabled" :
string.Empty)"
           asp-controller="House"
           asp-action="All"
           asp-route-currentPage="@previousPage"
           asp-route-category="@Model.Category"
           asp-route-searchTerm="@Model.SearchTerm"
           asp-route-sorting="@((int)Model.Sorting)"><<<</a>
    </div>

    @{
        var shouldButtonBeDisabled = Model.CurrentPage == maxPage ||
        !Model.Houses.Any();
    }

    <div class="col-md-6 d-grid gap-2 d-md-flex justify-content-md-end">
        <a class="btn btn-primary
           @(shouldButtonBeDisabled ? "disabled" : string.Empty)"
           asp-controller="House"
           asp-action="All"
           asp-route-currentPage="@(Model.CurrentPage + 1)"
           asp-route-category="@Model.Category"
           asp-route-searchTerm="@Model.SearchTerm"
           asp-route-sorting="@((int)Model.Sorting)">>></a>
    </div>
</div>

@if (!Model.Houses.Any())
{
    <h2 class="text-center">No houses found by the given criteria!</h2>
}

<div class="row">
    @foreach (var house in Model.Houses)
    {
        <partial name="_HousePartial" model="@house" />
    }
</div>
```

Create the **"_HousePartial.cshtml" partial view** in the "**/Views/Houses**" **folder**. It should **accept** a **HouseViewModel** and **display the house card**. If the **house is rented**, display a **[Leave] button**. If not, show a **[Rent] button**. If the **current user is not authenticated** (as the "**All Houses**" page is accessible to anyone), do not show **any buttons**, except for the **[Details]** one.

Copy the code from here:

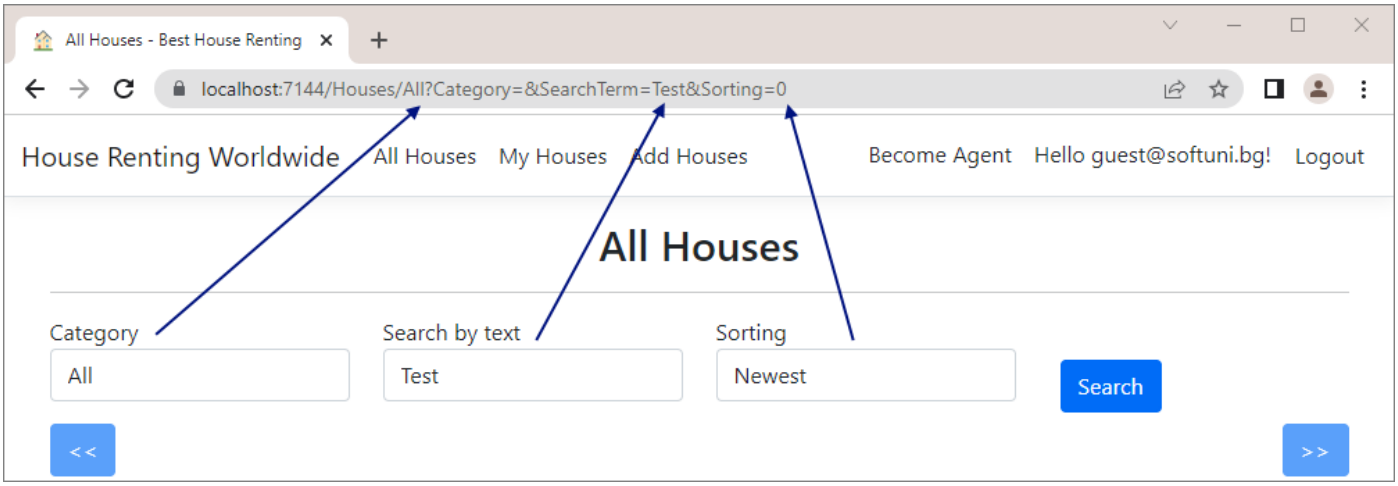
```
@model HouseServiceModel

<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>
                Price Per Month:
                <b>@String.Format("{0:f2}", Model.PricePerMonth) BGN</b>
            </h6>
            <h6>(@(Model.IsRented ? "Rented" : "Not Rented"))</h6>
            <br />
            <a asp-controller="House" asp-action="Details" asp-route-id="@Model.Id"
               class="btn btn-success">Details</a>
            @if (this.User.Identity.IsAuthenticated)
            {
                <a asp-controller="House" asp-action="Edit" asp-route-id="@Model.Id"
               class="btn btn-warning">Edit</a>
                <a asp-controller="House" asp-action="Delete" asp-route-
id="@Model.Id"
               class="btn btn-danger">Delete</a>
                <p></p>
                @if (!Model.IsRented)
                {
                    <form class="input-group-sm" asp-controller="House"
                  asp-action="Rent" asp-route-id="@Model.Id" method="post">
                        <input class="btn btn-primary" type="submit" value="Rent" />
                    </form>
                }
                else
                {
                    <form asp-controller="House" asp-action="Leave"
                  asp-route-id="@Model.Id" method="post">
                        <input class="btn btn-primary" type="submit" value="Leave"
/>
                    </form>
                }
            }
        </div>
    </div>
</div>
```

Try out if the **controller method is working correctly with the services** in the browser. To do this, comment out the **`HouseController` class code**, which **gives errors** and **run the app**.

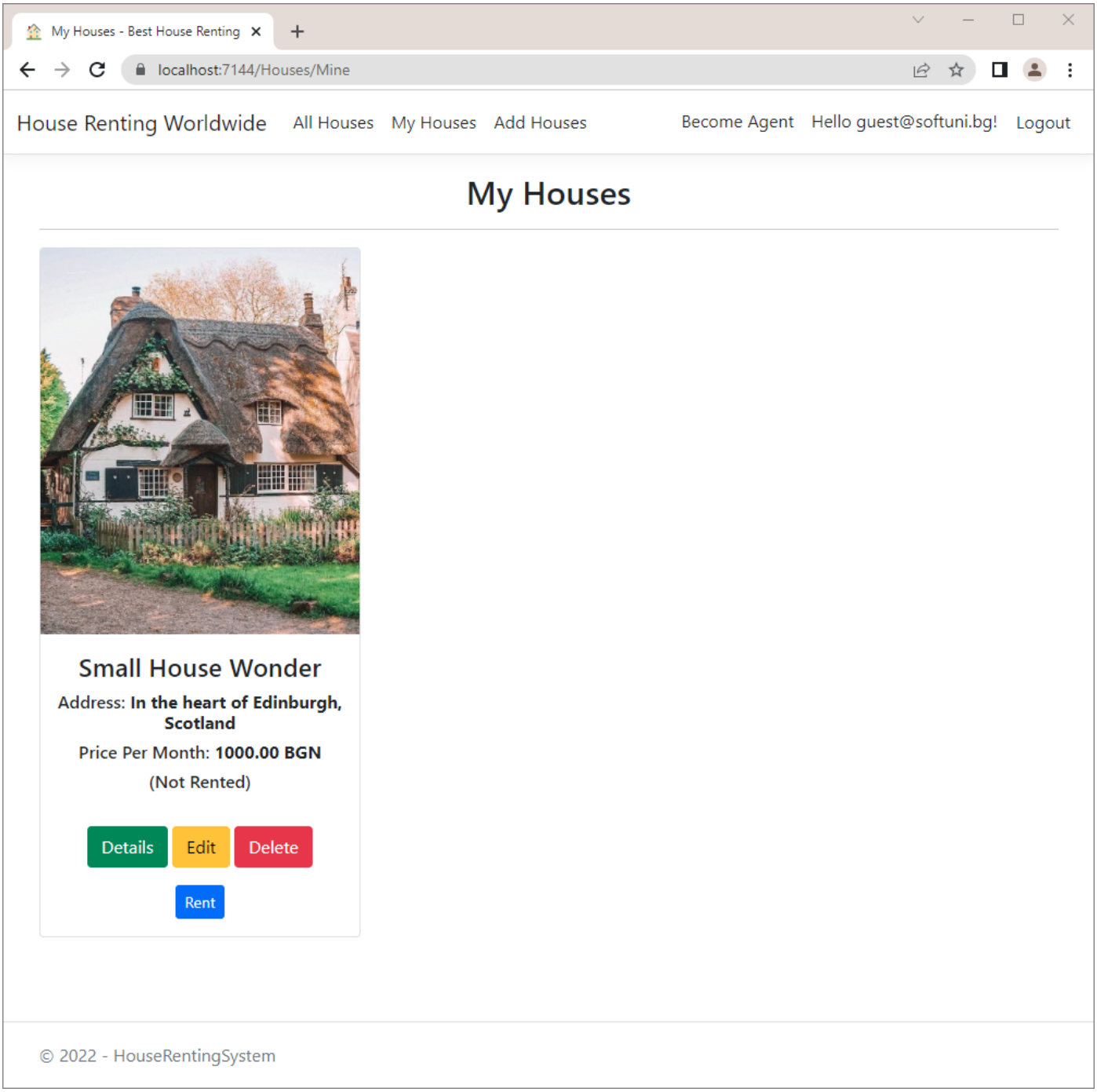
The "**`All Houses`**" page should look like before and **work as expected**. Try out the **searching**, **paging** and **sorting functionalities**.

Note that when you **add searching and sorting criteria** on the page, they will **be added as part of the URL** – that's what the **tag helpers** we added to the **view** do:



16. Implement "My Houses" page

Now it's time to implement the "**My Houses**" page. It will have **different implementations** for **users** and for **agents**. When the **user is not an agent**, they will **see the houses they rented**. When the **user is an agent**, they will see the **houses they created**. The page should look like this:



Modify the **`Mine(…)` method** in the **`HouseController` class** to **satisfy the above requirements** and to use **service methods**.

This method needs **service methods for getting the houses by a given agent id** (when the **user is an agent**) and **by a user id** (when the **user is not an agent**).

Define these two methods in the **`IHouseService`**:

```
Task<IEnumerable<HouseServiceModel>> AllHousesByAgentId(int agentId);

1 reference
Task<IEnumerable<HouseServiceModel>> AllHousesByUserId(string userId);
```

Now let's write **these methods** in the **`HouseService`**. We will add a **separate method for projecting a filtered House collection** to **`List<HouseServiceModel>`**. Note that we should **return a collection of type `HouseServiceModel`**, as the **`AllHousesQueryModel`** requires it.

Write the following **methods** in the **`HouseService` class** like this:

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

    return ProjectToModel(houses);
}

1 reference
public async Task<IEnumerable<HouseServiceModel>> AllHousesByUserId(string userId
{
    var houses = await _data
          .Houses
          .Where(h => h.RenterId == userId)
          .ToListAsync();

    return ProjectToModel(houses);
}
```

As you can see, we are going to need an additional method in the **`HouseService`** class, that should look like this:

```
private List<HouseServiceModel> ProjectToModel(List<House> houses)
{
    var resultHouses = houses
        .Select(h => new HouseServiceModel()
        {
            Id = h.Id,
            Title = h.Title,
            Address = h.Address,
            ImageUrl = h.ImageUrl,
            PricePerMonth = h.PricePerMonth,
            IsRented = h.RenterId != null
        })
        .ToList();

    return resultHouses;
}
```

Modify the **`Mine()` method** to **use the service methods** to **get the filtered houses collection** and **pass it to a view**:

```
public async Task<IActionResult> Mine()
{
    IEnumerable<HouseServiceModel> myHouses = null;

    var userId = User.Id();

    if (await _agents.ExistsById(userId))
    {
        var currentAgentId = _agents.GetAgentId(userId);

        myHouses = await _houses.AllHousesByAgentId(currentAgentId);
    }
    else
    {
        myHouses = await _houses.AllHousesByUserId(userId);
    }

    return View(myHouses);
}
```

Finally, write the "**Mine.cshtml**" **view**, which should only **accept a model** and **use** the "**_HousePartial.cshtml**" **partial view** to **display each house**.

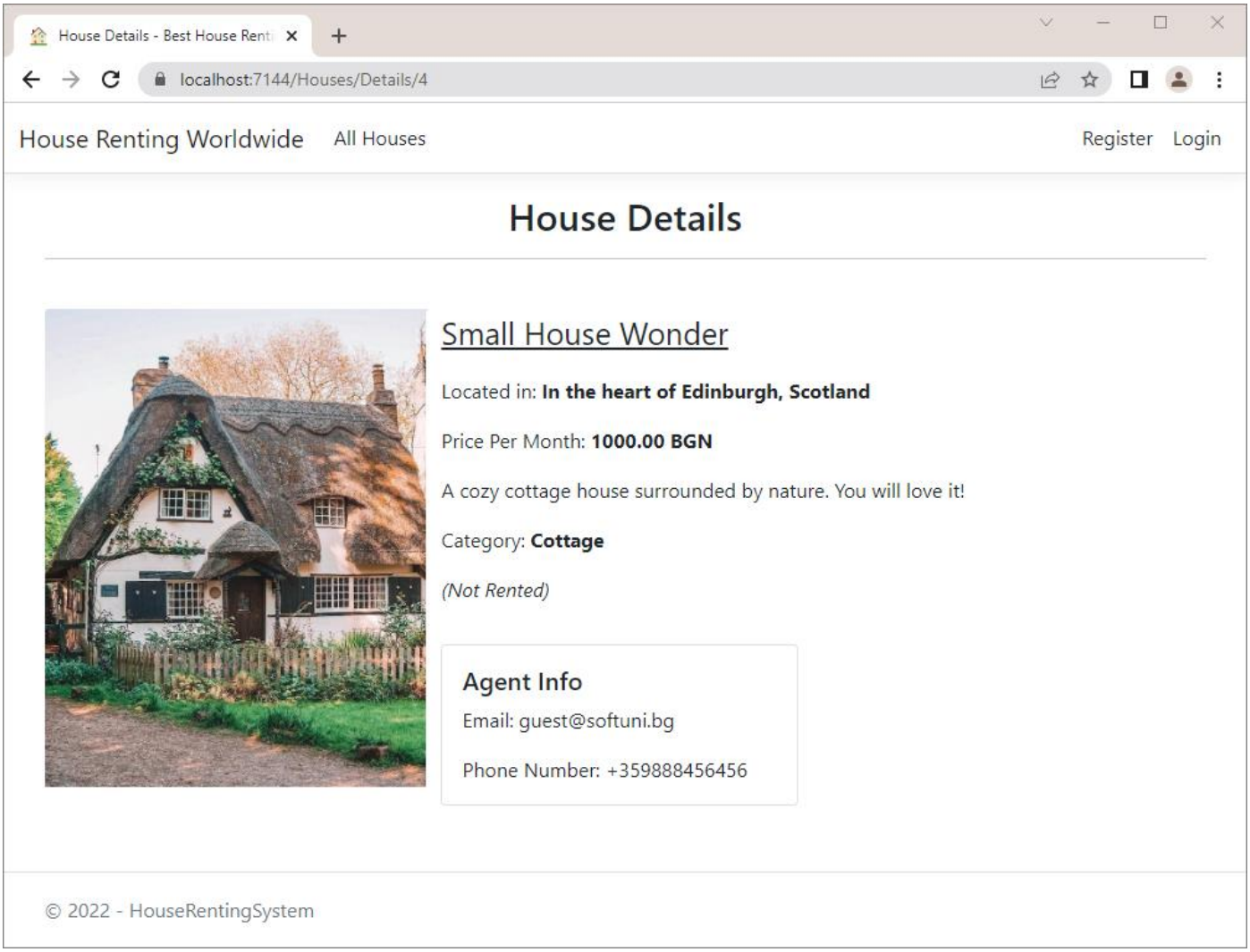
Mine.cshtml

```
@model IEnumerable<HouseServiceModel>

@{
    ViewBag.Title = "My Houses";
}

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

@if (!Model.Any())
{
    <h2 class="text-center">You have no houses yet!</h2>
}

<div class="row">
    @foreach (var house in Model)
    {
        <partial name="_HousePartial" model="@house" />
    }
</div>
```

Test the "**My Houses**" **page functionality** in the browser. When you **create a house as an agent** you should see your **created houses**. When you **rent a house as a non-agent user**, you should see your **rented houses**.

17. Implement "Details" page

The "**`Details`" page** should **show the house details** of a **house by a given `id`**. It looks like this:



The next method we should **modify** is **`Details(int id)`** in the **`HouseController`**. It will have logic for **checking whether a house with a given id exists** and for **projecting the `House`** to a **`HouseDetailsViewModel`**.

As we will implement this logic in a **service**, we need **service models**. Create the **`HouseDetailsServiceModel`** in "**`/Services/Houses/Models`**". It should **inherit `HouseServiceModel`** and have the **properties** of the **`HouseDetailsViewModel` class**:

```
public class HouseDetailsServiceModel : HouseServiceModel
{
    1 reference
    public string Description { get; set; } = null!;

    1 reference
    public string Category { get; set; } = null!;

    2 references
    public AgentServiceModel Agent {get; set;} = null!;
}
```

Now let's write the logic of **service methods**. Create the following **service methods** in the **`IHouseService` interface** and the **`HouseService` class**. **Get the house from the database** and **project it to a `HouseDetailsViewModel`**.

```
Task<bool> Exists(int id);

0 references
Task<HouseDetailsServiceModel> HouseDetailsById(int id);
```

```
public async Task<bool> Exists(int id)
{
    return await _data
        .Houses
        .AnyAsync(h => h.Id == id);
}

1 reference
public async Task<HouseDetailsServiceModel> HouseDetailsById(int id)
{
    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()
            {
                PhoneNumber = h.Agent.PhoneNumber,
                Email = h.Agent.User.Email
            }
        })
        .FirstOrDefaultAsync();
}
```

Modify the **`Details(int id)` method** in the **`HouseController`** to use the **service methods**:

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

    var houseModel = await _houses.HouseDetailsById(id);

    return View(houseModel);
}
```

Implement the **"Details.cshtml" view**, which should **accept a model** and **use its properties to display the house data**. Note that the **[Edit]**, **[Delete]**, **[Rent]** and **[Leave] buttons** should be **displayed only to authorized users**, as this page is accessible to everyone. As it is a lot of code to write, you can copy it from here:

```
@model HouseDetailsServiceModel

@{
    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>
                Price Per Month:
                <b>@String.Format("{0:f2}", Model.PricePerMonth) BGN</b>
            </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 (this.User.Identity.IsAuthenticated)
                {
                    <a class="btn btn-warning" asp-controller="House" asp-
action="Edit"
                       asp-route-id="@Model.Id">Edit</a>
                    <a class="ml-2 btn btn-danger" asp-controller="House" asp-
action="Delete"
                       asp-route-id="@Model.Id">Delete</a>
                    @if (!Model.IsRented)
                    {
                        <form class="ml-2" asp-controller="House"
                              asp-action="Rent" asp-route-id="@Model.Id"
method="post">
                            <input class="btn btn-primary" type="submit"
value="Rent" />
                        </form>
                    }
                    else
                    {
                        <form class="ml-2" asp-controller="House" asp-action="Leave"
                              asp-route-id="@Model.Id" method="post">
                            <input class="btn btn-primary" type="submit"
value="Leave" />
                        </form>
                    }
                }
            </div>
```

```
            <p></p>
            <div class="card" style="width: 18rem;">
                <div class="card-body">
                    <h5 class="card-title">Agent Info</h5>
                    <p class="card-text">Email:  @Model.Agent.Email</p>
                    <p class="card-text">Phone Number: @Model.Agent.PhoneNumber</p>
                </div>
            </div>
        </div>
    </div>
</div>
```

Try out the "**Details**" **page** in the browser and make sure that it **looks like shown above**.

18. Implement "Edit House" page

Let's modify the the **Edit(int id)** and **Edit(int id, HouseFormModel model) methods** in the **HouseController** for **editing an existing house**. We should check whether a **house with the given id exists** in the database and if the **current logged-in user is the agent**, who **created the house** (only the creator of a house can edit it). In both cases, the method should return **an error if something is wrong**.

First, go to the **IHouse** interface and create the following service method:

```
Task Edit(int houseId, string title, string address,
  string description, string imageUrl, decimal price, int categoryId);
```

Do that for the **HouseService** class, too. The method should **find by id** the house that we want to **edit**, **change** the needed info and **save** the **changes** to the database.

```
public async Task Edit(int houseId, string title, string address,
  string description, string imageUrl, decimal price, int categoryId)
{
    var house = _data.Houses.Find(houseId);

    house.Title = title;
    house.Address = address;
    house.Description = description;
    house.ImageUrl = imageUrl;
    house.PricePerMonth = price;
    house.CategoryId = categoryId;

    await _data.SaveChangesAsync();
}
```

Next, we need to modify the **Edit(int id)** and … controller methods from the **HouseController**. Do it like this:

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

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

    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);
}
```

```
[HttpPost]
0 references
public async Task<IActionResult> Edit(int id, HouseFormModel house)
{


    if (await _houses.Exists(id) == false)
    {
        return this.View();
    }

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

    if (await _houses.CategoryExists(house.CategoryId) == false)
    {
        this.ModelState.AddModelError(nameof(house.CategoryId),
            "Category does not exist.");
    }

    if (!ModelState.IsValid)
    {
        house.Categories = await _houses.AllCategories();

        return View(house);
    }

    _houses.Edit(id, house.Title, house.Address, house.Description,
        house.ImageUrl, house.PricePerMonth, house.CategoryId);

    return RedirectToAction(nameof(Details), new { id = id });
}
```

As you can see, we need some additional methods that checks if the agent that created the house, is the currently signed in agent and another one that returns the category of a house.

Write them in the **IHouseService** interface and in the **HouseService** class like shown below:

```
Task<bool> HasAgentWithId(int houseId, string currentUserId);

1 reference
Task<int> GetHouseCategoryId(int houseId);
```

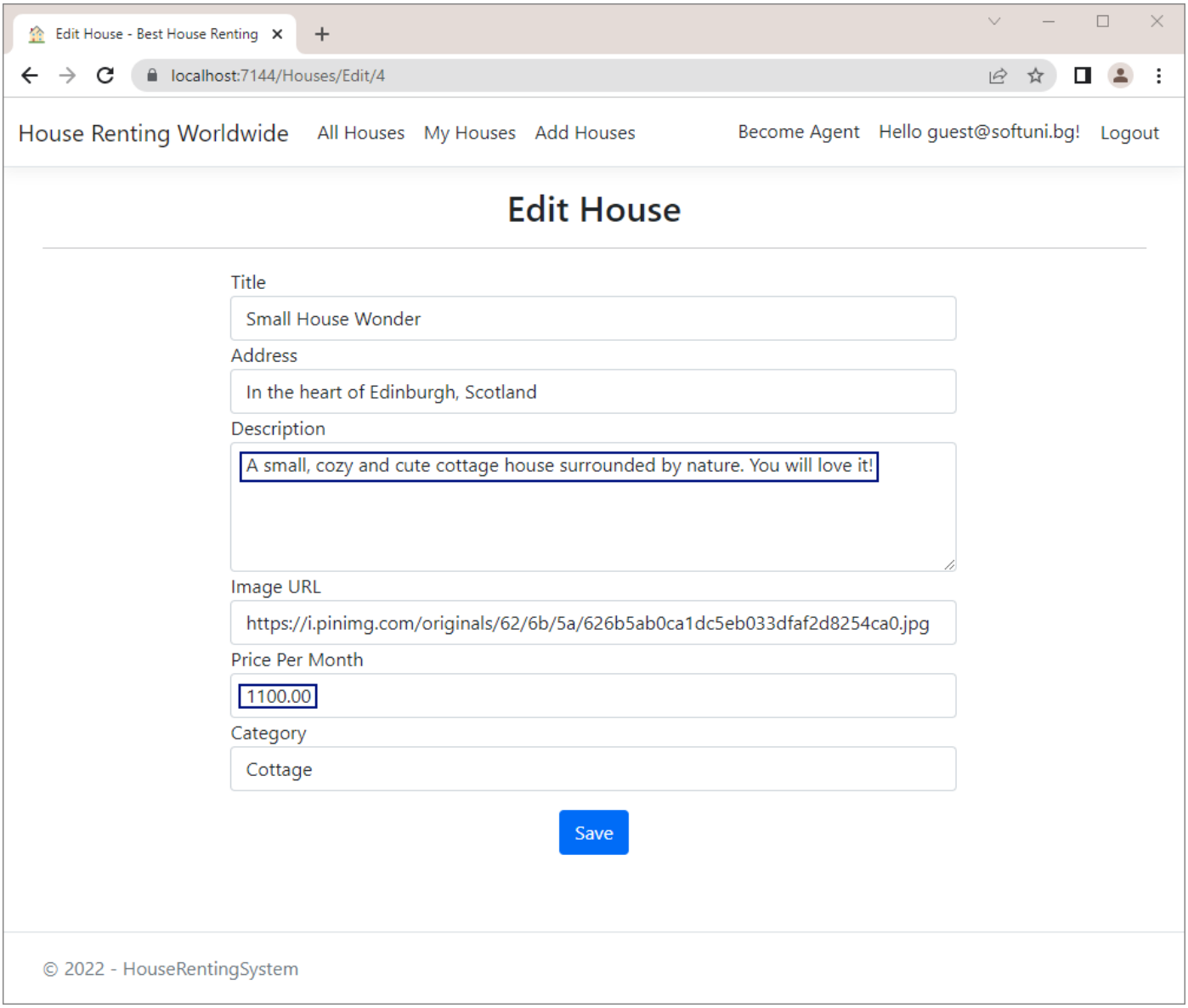
```
public async Task<bool> HasAgentWithId(int houseId, string currentUserId)
{
    var house = await _data.Houses.FindAsync(houseId);
    var agent = await _data.Agents.FirstOrDefaultAsync(a => a.Id == house.AgentId);

    if (agent == null)
    {
        return false;
    }

    if (agent.UserId != currentUserId)
    {
        return false;
    }

    return true;
}
public async Task<int> GetHouseCategoryId(int houseId)
{
    return await _data.Houses.FindAsync(houseId).CategoryId;
}
```

The **"Edit.cshtml" view** should **render** the **"_HouseFormPartial.cshtml" partial view** and should look like this:

Edit.cshtml

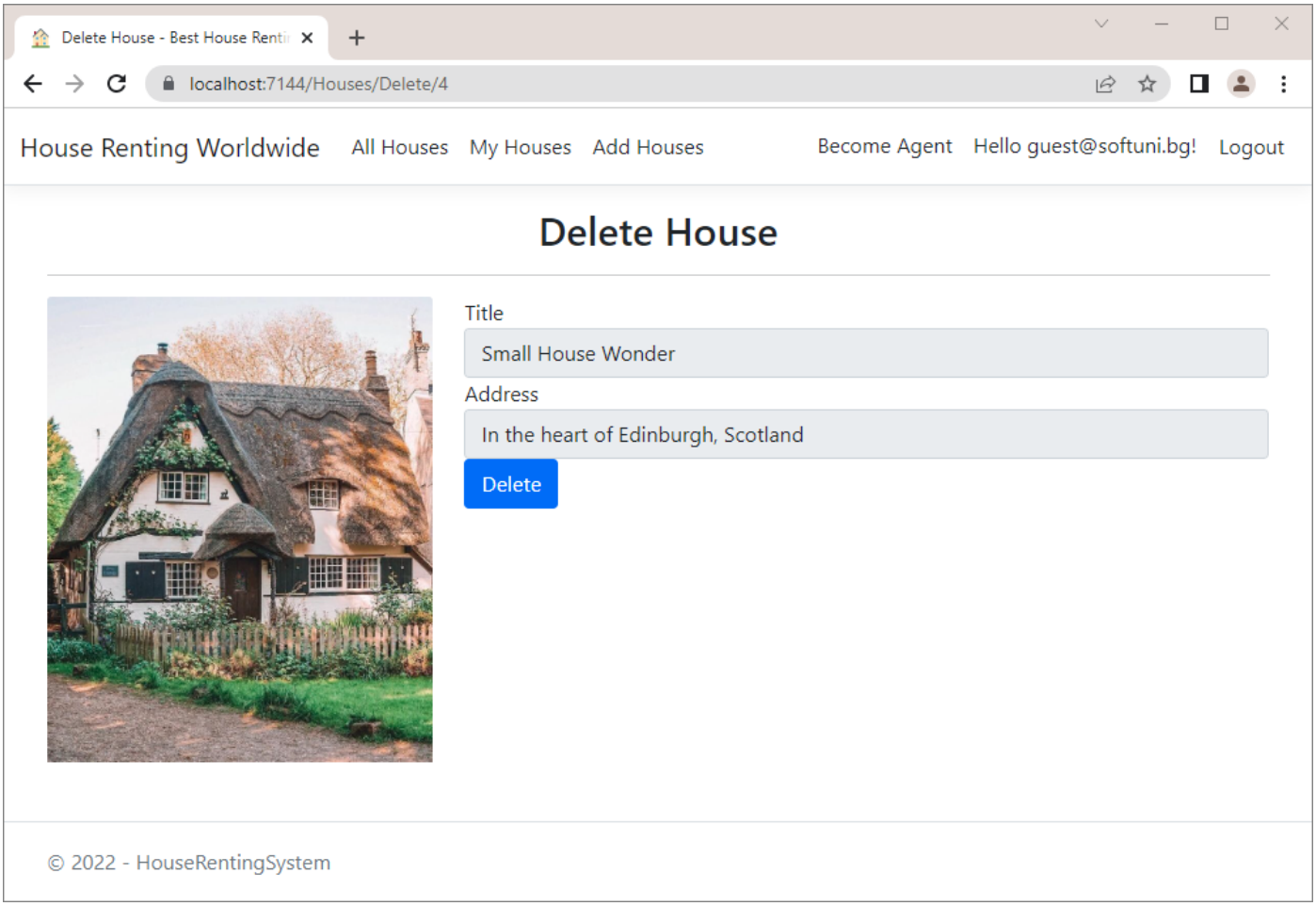
```
@model HouseFormModel

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

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

<partial name="_HouseFormPartial" model="@Model" />

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

Try the "Edit" functionality in the browser. After **successful edit**, you should be **redirected** to the "**Details**" **page** and the **changes should be saved** to the **database**, as well.



19. Implement "Delete House" page

The "**Delete**" **functionality** is next to be implemented in our app. The page should look like this:



First, modify the **`IHouseServices`** interface and the **`HouseService`** class by adding the **`Delete(int houseId)`** service method. The method is quite simple – it **finds by id the house** that we want to **delete** and **removes** it from the **database**.

```
Task Delete(int houseId);

public async Task Delete(int houseId)
{
    var house = await _data.Houses.FindAsync(houseId);

    _data.Remove(house);
    await _data.SaveChangesAsync();
}
```

Before modifying the **`Delete(int id)`** and **`Delete(HouseDetailsViewModel model)`** methods in the **`HouseController`**, we should **add properties** to the **`HouseDetailsViewModel`** view model, that we created at the beginning of this workshop.

It should look like this:

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

    2 references
    public string Title { get; set; } = null!;

    2 references
    public string Address { get; set; } = null!;

    1 reference
    public string ImageUrl { get; set; } = null!;
}
```

Now, our next step is to modify the **`Delete(int id)`** method and the **`Delete(HouseDetailsViewModel model)`** method in the **`HouseController`**.

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

    if(await _houses.HasAgentWithId(id, User.Id()))
    {
        return Unauthorized();
    }

    var house = await _houses.HouseDetailsById(id);

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

    return View(model);
}
```

In the **`HouseController`** **class** write the **`Delete(HouseViewModel model)`** **method**, which should **delete a given house** if it **exists** and if the **current user is its agent**. Then, the user should be **redirected** to the "**`All Houses`**" **page**.

```
[HttpPost]
0 references
public async Task<IActionResult> Delete(HouseDetailsViewModel house)
{

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

    if(await _houses.HasAgentWithId(house.Id, User.Id()))
    {
        return Unauthorized();
    }

    _houses.Delete(house.Id);

    return RedirectToAction(nameof(All));
}
```

Our final step is to implement the "**Delete.cshtml**" view. Add a **form for deleting a house**. Note that the user should **not be able to edit the house data**, e.g. the **<input> tags** should be **disabled**.

You can copy the code from here:

```
@model HouseDetailsViewModel

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

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

<form asp-action="Delete">
    <div class="row">
        <div class="col-md-4">
            <img class="card-img-top" src="@Model.ImageUrl" alt="House Image">
        </div>
        <div class="col-md-8">
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" disabled="disabled" />
            </div>
            <div class="form-group">
                <label asp-for="Address" class="control-label"></label>
                <input asp-for="Address" class="form-control" disabled="disabled" />
            </div>
            <div class="form-group">
                <input type="submit" value="Delete" class="btn btn-primary" />
            </div>
        </div>
    </div>
</form>
```

Test the "**Delete**" functionality in the browser and check if the changes are saved to the database.

20. Implement "Rent House" functionality

The next functionality we shall implement is for **renting a house**. Modify the **`Rent(int id)` method** in the **`HouseController` class**, which should be invoked on a "**POST**" **request** to "**`/Houses/Rent/{id}`**".

Before that, add the following methods to the **`IHouseService`** interface and the **`HouseService`** class:

```
Task<bool> IsRented(int id);

0 references
Task<bool> IsRentedByUserWithId(int houseId, string userId);

0 references
void Rent(int houseId, string userId);
```

The first two methods perform checks regarding the house that we want to rent. If a user wants to **rent a house**, make sure that a **house with a given `id` exists** in the database and that the **house is not already rented**.

```
public async Task<bool> IsRented(int id)
{
    var house = await _data.Houses.FindAsync(id);
    var result = house.RenterId != null;
    return result;
}

public async Task<bool> IsRentedByUserWithId(int id, string userId)
{
    var house = await _data.Houses.FindAsync(id);

    if(house == null)
    {
        return false;
    }

    if (house.RenterId != userId)
    {
        return false;
    }

    return true;
}
```

Then, **set the house `RenterId`** in the **database** to the **current user `id`** and **save it**.

```
public async Task Rent(int houseId, string userId)
{
    var house = await _data.Houses.FindAsync(houseId);

    house.RenterId = userId;
    _data.SaveChangesAsync();
}
```

After that, it's time to modify the **`Rent(int id)`** method in the **`HouseController`**. 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()))
    {
        return Unauthorized();
    }

    if (await _houses.IsRented(id))
    {
        return BadRequest();
    }

        _houses.Rent(id, User.Id());

    return RedirectToAction(nameof(All));
}
```

Now try out the "**rent**" **functionality**. To do this, **log in as a user**, who is **not an agent**, and go to "**/Houses/Rent/{id}**" with the **house id** in the **URL**. The **id** should be of a **house**, which is **not already rented.** If the **rent is successful**, you should be **redirected** to the "**My Houses**" **page** and the **rented House** should have a **RenterId** the **database**:

21. Implement "Leave House" functionality

The last functionality we are going to implement today is the "**leave house**" **functionality**. It should also be **invoked** on a "**POST**" **request** to "**/Houses/Leave/{id}**".

First, implement service methods to the **IHouseService** interface and the **HouseService** class. They should look like this:

```
Task Leave(int houseId);

public async Task Leave(int houseId)
{
    var house = await _data.Houses.FindAsync(houseId);

    house.RenterId = null;
    _data.SaveChangesAsync();
}
```

Modify the **Leave(int id) method** of the **HouseController class**. Note that for a house to be **left** it should **exist** and **be rented and the current user** should be **the one who rented the house**. If these requirements are satisfied, set the **RenterId** of the **House record** to **NULL** the **database** and **redirect** to the "**My Houses**" **page**.

The **controller action** should look like this:

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

    if (await _houses.IsRentedByUserWithId(id, User.Id()))
    {
        return Unauthorized();
    }

    await _houses.Leave(id);

    return RedirectToAction(nameof(Mine));
}
```

Try to **leave the house** you rented on the previous task by going to "**`/Houses/Leave/{id}`**" with the **house `id`**. If it is successful, you should be again **redirected** to the "**`My Houses`**" **page** and the **`House` record** should have a **`NULL`** value on **`RenterId`.** Don't forget to test the **model errors** on the "**`Become Agent`**" **page** – a **user with rented houses** should **not be able to become an agent**.