Workshop: Areas, Cache and Testing

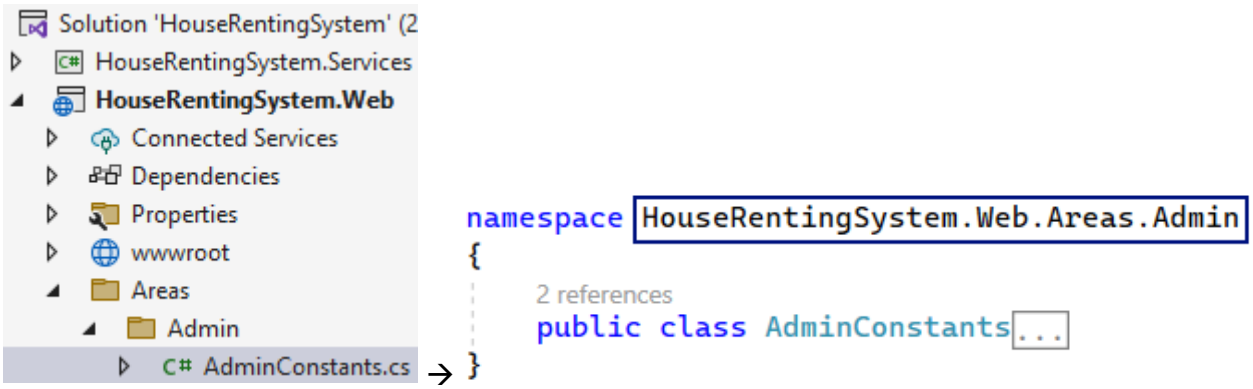
Workshop for the "ASP.NET Advanced" course @ SoftUni

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

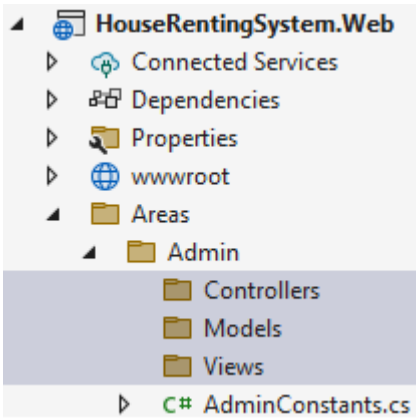
1. Introduce Admin Area

In this task we will **create an admin area**. The area will group the app **functionalities**, which are **accessible only to the Admin**, in a **separate MVC structure**. Classes from the area will have **different namespaces** from the main MVC structure and their methods will be **invoked on different URLs**. Let's see how to do this.

First, create an "**Admin**" **folder** in the "**Areas**" **folder** in the "**HouseRentingSystem.Web**" **project**. Also, it is a good idea to **move the AdminConstants class** to this folder, as this is the admin's space in the class structure. The **class** should have a **changed namespace**. It should look like this:



Create "**Controllers**", "**Models**" and "**Views**" **folders** in the "**Admin**" **folder**. They will hold **area's MVC classes**:

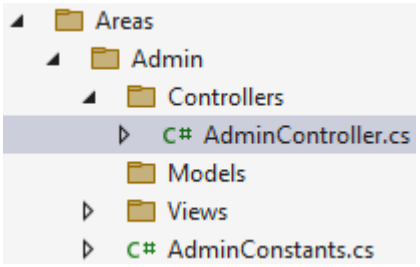


Now let's **create a base controller class** for the **controllers of the admin area**. The **AdminController** should be **accessed only by the Admin** (in its role) and should **set the area name for all child controller classes**.

The **area name** is defined in the **AdminConstants class**:

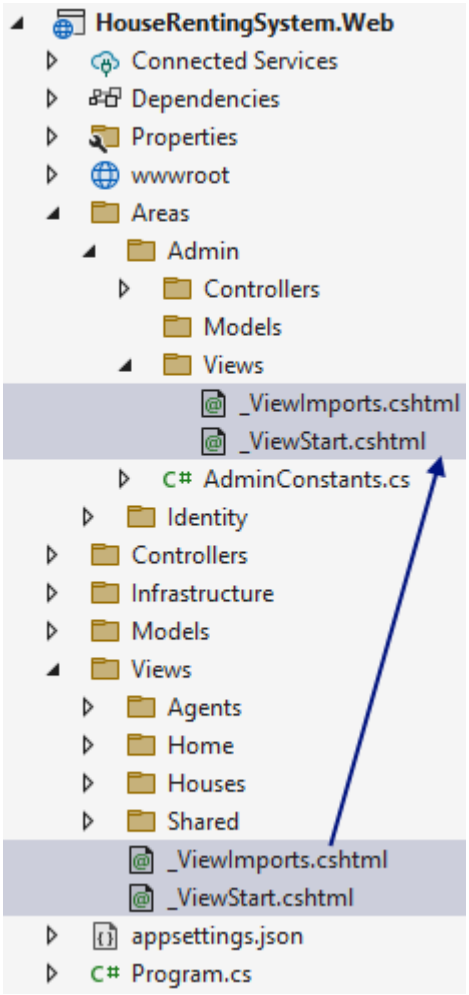
```
public class AdminConstants
{
    public const string AreaName = "Admin";
    public const string AdminRoleName = "Administrator";
    public const string AdminEmail = "admin@mail.com";
}
```

The **`AdminController`** should **inherit the base `Controller` class** and look like this:




```
[Area(AreaName)]
[Authorize(Roles = AdminRoleName)]
0 references
public class AdminController : Controller
{
}
```

Before we create the controllers, let's pay attention to the "**`Views`**" **folder**. In it, we should have the "**`_ViewImports.cshtml`**" and "**`_ViewStart.cshtml`**" **partial views** to render the area views properly. **Copy the files** from the main "**`Views`**" **folder** and **leave them** as they are for now. We will **change and clear them** later, if needed:



The final step before we start implementing **admin area** pages is to **add a new route in** "**`Program.cs`**" file:

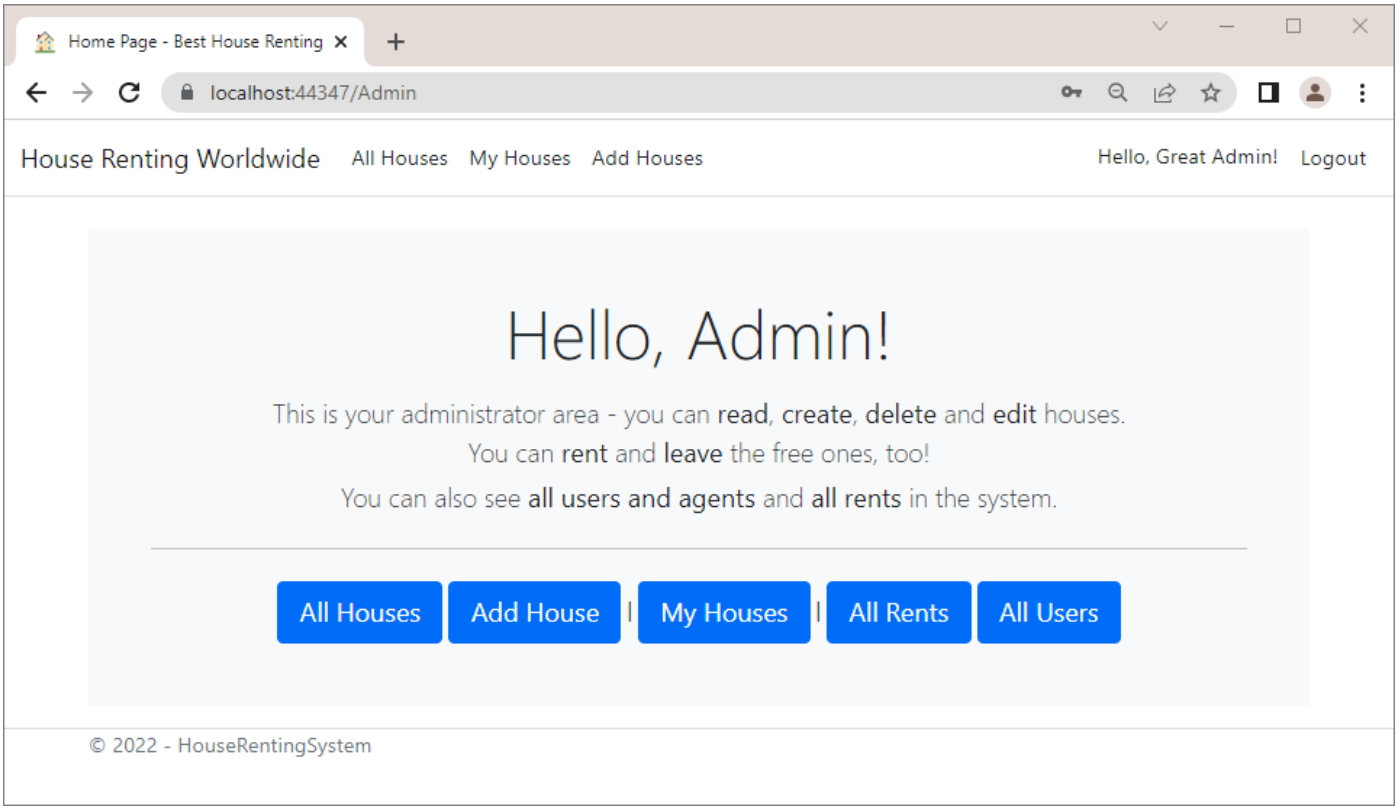
```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "Areas",
        pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");
    endpoints.MapControllerRoute(
        name: "House Details",
        pattern: "/Houses/Details/{id}/{information}",
        defaults: new { Controller = "House", Action = "Details" });

    endpoints.MapDefaultControllerRoute();
    endpoints.MapRazorPages();
});
```

The **controller route** we have added will form **URLs** like "**`/Admin/`**" for the "**Home**" **page** and "**`/Admin/Users/All`**" for **accessing a page with all users**, which is accessed only by the **Admin**.

Step 1: Create Admin Pages

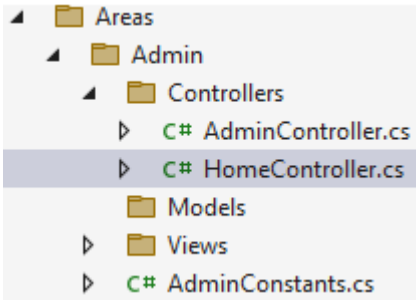
"Home" Page

The "**Home**" **page** of the **admin area** is **accessed on** "**`/Admin/`**" and should look like this:



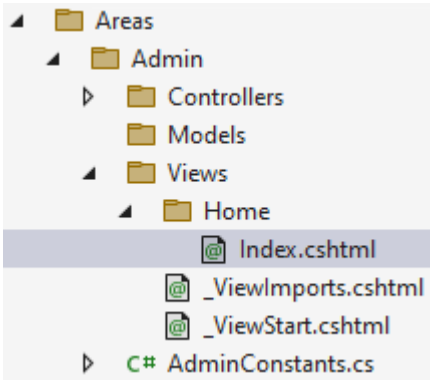
To create the "**Home**" **page**, **create a `HomeController` class**, which should **inherit the `AdminController` base class**. Note that **controller names of areas** may be the same as those from the main MVC structure of the app. It does not matter, as they are **accessed depending on the area name**.

The new **`HomeController`** should have an **`Index()` method**, which **returns a view**:




```
public class HomeController
    : AdminController
{
    0 references
    public IActionResult Index()
        => View();
}
```

Now **create the "`Index.cshtml`" view** for the above **controller action** in the "**`/Areas/Admin/Views/Home`**" **folder**. It should look like this:




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

<div class="mt-4 p-5 bg-light text-center">
    <h2 class="display-4">Hello, Admin!</h2>
    <p class="lead mb-1 mt-3">
        This is your administrator area - you can <b>read</b>, <b>create</b>,
        <b>delete</b> and <b>edit</b> houses.
        <br />You can <b>rent</b> and <b>leave</b> the free ones, too!
    </p>
    <p class="lead mt-1">
        You can also see <b>all users and agents</b> and <b>all rents</b> in the
system.
    </p>
    <hr class="my-4">
    <a asp-area="" asp-controller="House" asp-action="All"
       class="btn btn-primary btn-lg">All Houses</a>
    <a asp-area="" asp-controller="House" asp-action="Add"
       class="btn btn-primary btn-lg">Add House</a>
    <span>  |  </span>
    <a asp-area="Admin" asp-controller="House" asp-action="Mine"
       class="btn btn-primary btn-lg">My Houses</a>
    <span>  |  </span>
    <a asp-area="Admin" asp-controller="Rent" asp-action="All"
       class="btn btn-primary btn-lg">All Rents</a>
    <a asp-area="Admin" asp-controller="User" asp-action="All"
       class="btn btn-primary btn-lg">All Users</a>
</div>
```

Now, if you **run the app** and **log in with the Admin**, you will see the "**Home**" **page for all users**, as **all users are redirected to it** after login.

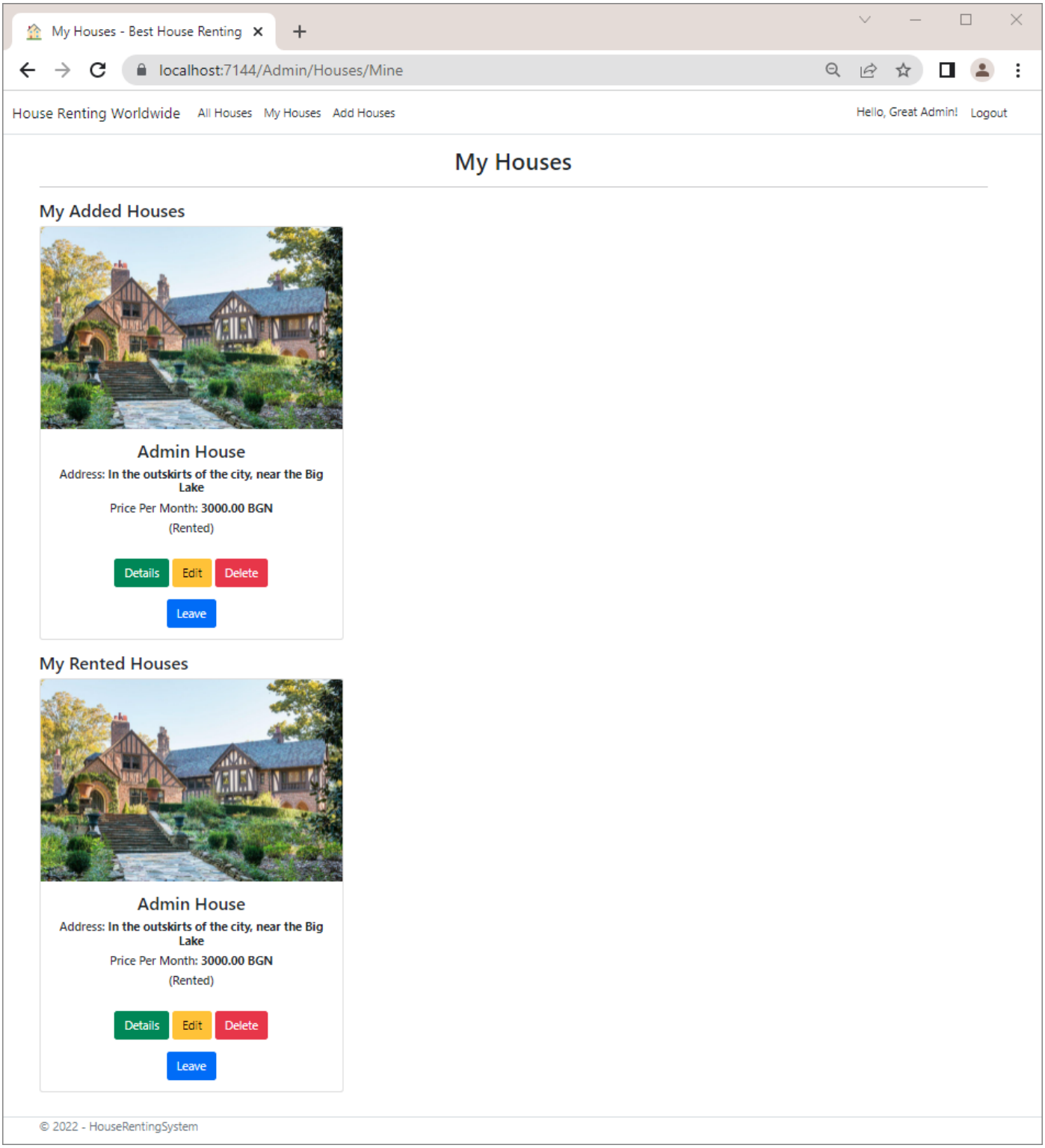
To change that and display the **admin** "**Home**" **page**, go to the **`Index()` method** of the **`HomeController`** in the "**`HouseRentingSystem.Web`**" **project**. The method should **redirect to** "**`/Admin/`**" when the **current user is the `Admin`**:

```
public class HomeController : Controller
{
    private readonly IHouseService _houses;
    1 reference
    public HomeController(IHouseService houses)...

    0 references
    public IActionResult Index()
    {
        if (User.IsInRole(AdminRoleName))
        {
            return RedirectToAction("Index", "Home", new { area = "Admin" });
        }

        var houses = _houses.LastThreeHouses();
        return View(houses);
    }
```

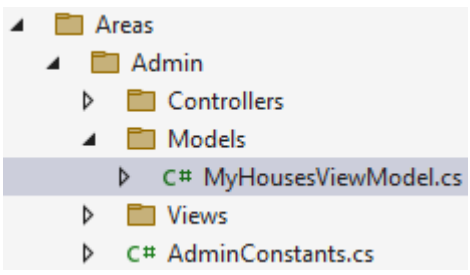
Now **run the app** and make sure that the **correct page is displayed** when you **log in with the `Admin`**. However, when you **click on the "Home" page buttons**, most of them **do not open existing pages** yet. Let's continue with implementing the rest of the pages to fix this problem.

"My Houses" Page

The "**`My Houses`**" **page** of the **`Admin`** will show the **houses they created** and the **houses they rented**. The page is shown on the picture below:



To **implement the above page**, first **create a model** in the "**`/Areas/Admin/Models`**" **folder**, which should have the **added and the rented houses by the admin**, as a **collection of type `HouseServiceModel`**. It should be the following:



→

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

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

Then, **create a `HouseController`** in the **`Admin` area** with a **`Mine()` method**. **Inject services** in the controller and use them to **pass the above model to a view**:

```
public class HouseController : AdminController
{
    private readonly IHouseService _houses;
    private readonly IAgentService _agents;

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

    0 references
    public IActionResult Mine()
    {
        var myHouses = new MyHousesViewModel();

        var adminUserId = User.Id();
        myHouses.RentedHouses = _houses.AllHousesByUserId(adminUserId);

        var adminAgentId = _agents.GetAgentId(adminUserId);
        myHouses.AddedHouses = _houses.AllHousesByAgentId(adminAgentId);

        return View(myHouses);
    }
}
```

Create the "**`Mine.cshtml`**" **view** in the "**`/Admin/Views/Houses`**" **folder**, which should **accept a `MyHousesViewModel`** and **pass each house to the** "**`_HousePartial.cshtml`**" **partial view**. Note that we should use the **`Html.PartialAsync()` HTML helper method**, as the **partial view is from another project**. The view should look like this:

```
@model MyHousesViewModel

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

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

```
<hr />

<h4>My Added Houses</h4>

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

<div class="row">
    @foreach (var house in Model.AddedHouses)
    {
        @await Html.PartialAsync("~/Views/House/_HousePartial.cshtml", house)
    }
</div>

<h4>My Rented Houses</h4>

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

<div class="row">
    @foreach (var house in Model.RentedHouses)
    {
        @await Html.PartialAsync("~/Views/House/_HousePartial.cshtml", house)
    }
</div>
```

Don't forget that you should **add the `MyHousesViewModel` class namespace** to the "**`_ViewImports.cshtml`**" **file** to use the model.

Run the app and make sure that the "**`My Houses`**" **page** is displayed correctly on the "**`/Admin/House/Mine`**" **URL**.

Try to **click on the `[Details]`** or any other **button under a house**. You should see that they **return an error**. The reason for this is that they **form an URL with the current area name** but our methods for these functionalitites are **not part of the admin area**.

Because of this problem we should **modify the `<a>` tags of buttons in views**. They should have the **`asp-area` tag helper**, so that they **redirect to the correct controller actions**. **Add tag helpers** to the "**`_HousePartial.cshtml`**" **partial view** to **specify the area** (which is the default one without a name) like this:

_HousePartial.cshtml

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

<div class="col-md-4">
    <div class="card mb-3">
        <img class="card-img-top" src="@Model.ImageUrl" alt="House Image">
        <div class="card-body text-center">
            <h4>@Model.Title</h4>
            <h6>Address: <b>@Model.Address</b></h6>
            <h6>...
            <h6>(@(Model.IsRented ? "Rented" : "Not Rented"))</h6>
            <br />
            <a asp-area="" asp-controller="House" asp-action="Details"
               asp-route-id="@Model.Id"
               asp-asp-route-information="@Model.GetInformation()"
               class="btn btn-success">Details</a>
```

```
@if (await houses.HasAgentWithId(Model.Id, User.Id()) || User.IsAdmin())
{
    <a asp-area="" asp-controller="House" asp-action="Edit"
       asp-route-id="@Model.Id" class="btn btn-warning">Edit</a>
    <a asp-area="" asp-controller="House" asp-action="Delete"
       asp-route-id="@Model.Id" class="btn btn-danger">Delete</a>
}

@if (!Model.IsRented && await agents.ExistsById(User.Id()) == false || User.IsAdmin())
{
    <form class="input-group-sm" asp-controller="House" asp-area=""
        asp-action="Rent" asp-route-id="@Model.Id" method="post">
        <input class="btn btn-primary" type="submit" value="Rent" />
    </form>
}
else if (await houses.IsRentedByUserWithId(Model.Id, User.Id()))
{
    <form asp-area="" asp-controller="House" asp-action="Leave"
        asp-route-id="@Model.Id" method="post">
        <input class="btn btn-primary" type="submit" value="Leave" />
    </form>
}
```

Now **all buttons redirect to the correct pages**, which are in the **default area**. However, the "**My Houses**" **link** in the **navigation bar** will always point to "**/House/Mine**" and **not** "**/Admin/House/Mine**", independently of the user. For this reason, we should go to the common **HouseController** and **modify the Mine() method to redirect to the admin area** if the **current user is the Admin**. Do it like this:

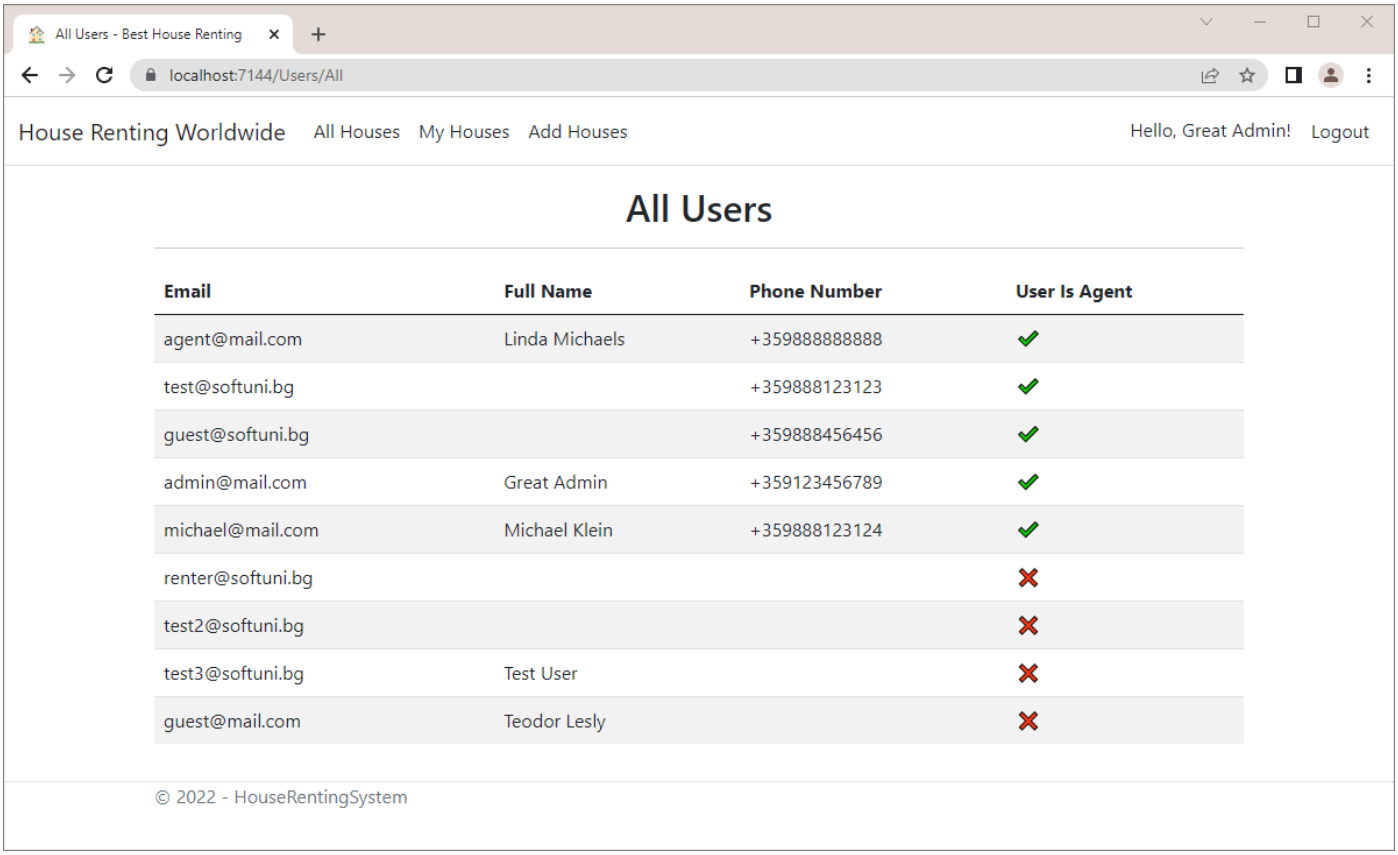
```
if (User.IsInRole(AdminRoleName))
{
    return RedirectToAction(actionName: "Mine",
        controllerName: "House", new { area = "Admin" });
}
```

Run the app again and **log in with a user**, who is **not the Admin**. You should be **redirected to the** "**My Houses**" **page** when you **click on its link in the navigation bar**:

Now **log in with the Admin**. The **navigation link** should access the "**My Houses**" **page** on "**/Admin/Houses/Mine**":

"All Users" Page

The "**All Users**" **page** should be **visible only to the Admin** and **display all app users and if they are agents or not**. It should be **accessible on** "**/Users/All**":



In the current case we will need a **service method to return all users** from the database. **Define an `All()` method** in the **`IUserService` interface** from the "**`HouseRentingSystem.Core`**" **project**:

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

    0 references
    Task<IEnumerable<UserServiceModel>> All();
}
```

Note that the above **method** returns a **collection of type `UserServiceModel`**, which is part of the "**`/User/Models`**" **folder**, and has the following **properties**:

```
public class UserServiceModel
{
    0 references
    public string Email { get; set; } = null!;

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

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

The **`UserService` class** should **implement the above `All()` method**. Do not forget that you need to **inject `AutoMapper`** before that. Write the class like this:

```
public class UserService : IUserService
{
    private readonly HouseRentingDbContext _data;
    private readonly IMapper _mapper;

    0 references
    public UserService(HouseRentingDbContext data, IMapper mapper)
    {
        _data = data;
        _mapper = mapper;
    }

    1 reference
    public async Task<IEnumerable<UserServiceModel>> All()
    {
        var allUsers = new List<UserServiceModel>();

        var agents = await _data
            .Agents
            .Include(ag => ag.User)
            .ProjectTo<UserServiceModel>(_mapper.ConfigurationProvider)
            .ToListAsync();

        allUsers.AddRange(agents);

        var users = await _data
            .Users
            .Where(u => !_data.Agents.Any(ag => ag.UserId == u.Id))
            .ProjectTo<UserServiceModel>(_mapper.ConfigurationProvider)
            .ToListAsync();

        allUsers.AddRange(users);

        return allUsers;
    }
```

Be **careful with the mappings**. They should be **added to the `ServiceMappingProfile` class**. Note that **users don't have phone numbers** and, in this case, the **`PhoneNumber` model property should contain an empty string**. The **mappings** should be the following:

```
CreateMap<Agent, UserServiceModel>()
    .ForMember(us => us.Email, cfg => cfg.MapFrom(ag => ag.User.Email))
    .ForMember(us => us.FullName, cfg => cfg
        .MapFrom(ag => ag.User.FirstName + " " + ag.User.LastName));

CreateMap<ApplicationUser, UserServiceModel>()
   .ForMember(us => us.PhoneNumber, cfg => cfg.MapFrom(us => string.Empty))
   .ForMember(us => us.FullName, cfg => cfg
        .MapFrom(us => us.FirstName + " " + us.LastName));
```

Next, create a **`UserController`** in the **admin area with an `All()` method**, which should **use the service method** we just created and **return a view**. Don't forget to set the **controller action route** to "**`/User/All`**":

```
public class UserController : AdminController
{
    private readonly IUserService _users;

    0 references
    public UserController(IUserService users)
    {
        _users = users;
    }

    [Route("User/All")]
    0 references
    public async Task<IActionResult> All()
    {
        var users = await _users.All();
        return View(users);
    }
}
```

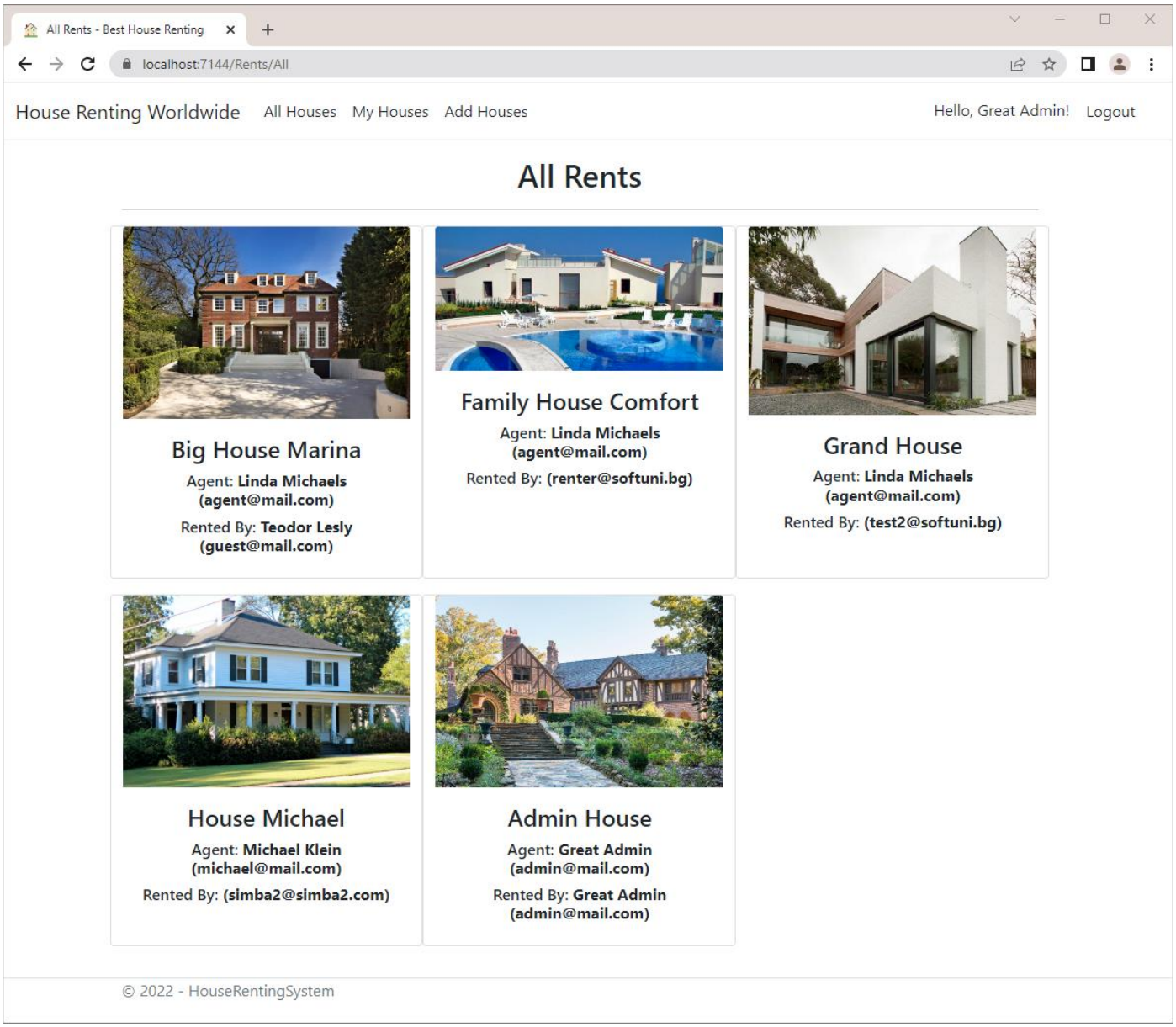
Finally, **create an `All.cshtml` view** in the "**`/Admin/Views/User`**" **folder**. You can copy the code from here:

```
@model IEnumerable<UserServiceModel>

@{
    ViewBag.Title = "All Users";
}

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

<table class="table table-striped">
    <thead>
        <tr>
            <th><label>Email</label></th>
            <th><label>Full Name</label></th>
            <th><label>Phone Number</label></th>
            <th><label>User Is Agent</label></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var user in Model)
        {
            <tr>
                <td><label>@user.Email</label></td>
                <td><label>@user.FullName</label></td>
                <td><label>@user.PhoneNumber</label></td>
                <td>
                    @if (user.PhoneNumber != string.Empty)
                    {
                        <label>✔</label>
                    }
                    else
                    {
                        <label class="text-center">✖</label>
                    }
                </td>
            </tr>
        }
```

```
    </tbody>
</table>
```

"All Rents" Page

The "**All Rents**" **page** will **display rented houses** with **information about the house agent and renter**. It should be **accessed on** "**/Rent/All**" by the **Admin** and look like this:



To implement the above page, we will need to **access the house renter data**. For this reason, we will first **add a Renter property** to the **House entity** in the "**Data**" **folder** of the "**HouseRentingSystem.Data**" **project**. It should be of **type ApplicationUser**:

```
public ApplicationUser Renter { get; set; } = null!;
```

Then, let's **create a service with a service model to return all rents**. Go to the "**HouseRentingSystem.Core**" **project** and **add a new folder** "**Rent**" to the "**Contracts**", "**Services**" and "**Models**" folders. **Create a IRentService interface** with an **All() method**, which should be **implemented in a RentService class**. The method will **return a collection of type RentServiceModel**. All classes are shown below:

```
public class RentServiceModel
{
    0 references
    public string HouseTitle { get; set; } = null!;
    0 references
    public string HouseImageURL { get; set; } = null!;
    0 references
    public string AgentFullName { get; set; } = null!;
    0 references
    public string AgentEmail { get; set; } = null!;
    0 references
    public string RenterFullName { get; set; } = null!;
    0 references
    public string RenterEmail { get; set; } = null!;
}

public interface IRentService
{
    0 references
    Task<IEnumerable<RentServiceModel>> All();
}

public class RentService : IRentService
{
    private readonly HouseRentingDbContext _data;
    private readonly IMapper _mapper;

    0 references
    public RentService(HouseRentingDbContext data,
        IMapper mapper)
    {
        _data = data;
        _mapper = mapper;
    }

    1 reference
    public async Task<IEnumerable<RentServiceModel>> All()
    {
        return await _data
            .Houses
            .Include(h => h.Agent.User)
            .Include(h => h.Renter)
            .Where(h => h.RenterId != null)
            .ProjectTo<RentServiceModel>(_mapper.ConfigurationProvider)
            .ToListAsync();
    }
}
```

Note that we should **include the agent user** and the **renter of the house** when we **extract data from the database**, so that we **use them for mapping**.

Before we continue, don't forget to **register the service in `Program` class** of the "**`HouseRentingSystem.Web`**" **project**:

```
builder.Services.AddTransient<IRentService, RentService>();
```

Go back to the "**`HouseRentingSystem.Services`**" **project** and **add the mapping from `House` to `RentServiceModel`** to the **`ServiceMappingProfile` class**. Note that you should **map all properties**, as their **names don't match**:

```
CreateMap<House, RentServiceModel>()
 .ForMember(h => h.HouseTitle, cfg => cfg.MapFrom(h => h.Title))
 .ForMember(h => h.HouseImageURL, cfg => cfg.MapFrom(h => h.ImageUrl))
 .ForMember(h => h.AgentFullName, cfg => cfg
      .MapFrom(h => h.Agent.User.FirstName + " " + h.Agent.User.LastName))
 .ForMember(h => h.AgentEmail, cfg => cfg.MapFrom(h => h.Agent.User.Email))
 .ForMember(h => h.RenterFullName, cfg => cfg
      .MapFrom(h => h.Renter.FirstName + " " + h.Renter.LastName))
 .ForMember(h => h.RenterEmail, cfg => cfg.MapFrom(h => h.Renter.Email));
```

Next, in the web project, **add a `RentController` class** with an **`All()` method**, which **uses the above service method** and **returns a view**:

```
public class RentController : AdminController
{
    private readonly IRentService _rents;
    private readonly IMemoryCache _cache;

    0 references
    public RentController(IRentService rents,
        IMemoryCache cache)
    {
        _rents = rents;
        _cache = cache;
    }

    [Route("Rent/All")]
    0 references
    public async Task<IActionResult> All()
    {
        var rents = await _rents.All();
        return View(rents);
    }
}
```

Finally, add the **`RentServiceModel` namespace** to the "**`_ViewImprots.cshtml`** **file** and **create a folder** "**`Rent`**" in the "**`/Admin/Views`**" **folder**, which will have an **`All.cshtml` file**. It should be the following:

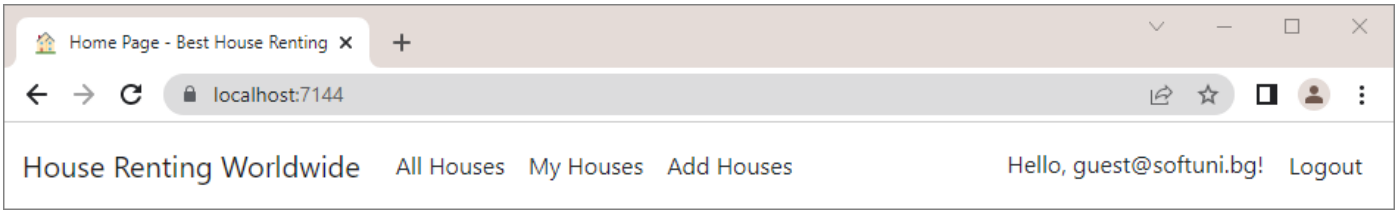
```
@model IEnumerable<RentServiceModel>

@{
    ViewBag.Title = "All Rents";
}

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

<div class="row">
    @foreach (var rent in Model)
    {
        <div class="card mb-3 col-4">
            <img class="card-img-top" src="@rent.HouseImageURL" alt="House Image">
            <div class="card-body text-center">
                <h4>@rent.HouseTitle</h4>
                <h6>Agent: <b>@rent.AgentFullName (@rent.AgentEmail)</b></h6>
                <h6>Rented By: <b>@rent.RenterFullName (@rent.RenterEmail)</b></h6>
            </div>
        </div>
    }
</div>
```

Run the app, **navigate to the** "**`All Rents`**" **page** on "**`/Rent/All`**" and make sure that **rents are displayed correctly**.

In addition, it is a good idea to **clear the `_ViewImports.cshtml` file** in "**`/Areas/Admin/Views`**" **from unused namespaces**.

Step 2: Modify Layout

In this task, we want to **modify our navigation bar links**. When the **user is not the `Admin`**, their links should be the following (they are **not changed**):



However, when the **`Agent` is logged-in**, their **navigation bar should have additional links** in a **dropdown menu**:

To do this, we will create a **special layout view** for the **admin pages**. **Copy the "`_Layout.cshtml`" view** and **rename** it to "**`_AdminLayout.cshtml`**". **Move it to a** "**`/Areas/Admin/Views/Shared`**" **folder**.

Modify the **`_AdminLayout.cshtml` view** by **adding a dropdown with links** like this:

_AdminLayout.cshtml

```
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">House Renting Worldwide</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse"...
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item dropdown">
                            <div>
                                <a class="btn dropdown-toggle" href="#" role="button"
                                   id="dropdownMenuLink" data-bs-toggle="dropdown" aria-expanded="false">
                                    All
                                </a>
                                <ul class="dropdown-menu" aria-labelledby="dropdownMenuLink">
                                    <li>
                                        <a class="dropdown-item nav-link text-dark" asp-area=""
                                           asp-controller="House" asp-action="All">All Houses</a>
                                    </li>
                                    <li>
                                        <a class="dropdown-item nav-link text-dark" asp-area=""
                                           asp-controller="Rent" asp-action="All">All Rents</a>
                                    </li>
                                    <li>
                                        <a class="dropdown-item nav-link text-dark" asp-area=""
                                           asp-controller="User" asp-action="All">All Users</a>
                                    </li>
                                </ul>
                            </div>
                        </li>
                        <li class="nav-item">...
                        <li class="nav-item">...
                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
```

Now go to the "**_ViewStart.cshtml" file** in the "**/Admin/Views**" **folder** and **set the layout** with the new file:

_ViewStart.cshtml

```
@{
    Layout = "_AdminLayout";
}
```

Now we should **modify the "_ViewStart.cshtml" file** in the main "**Views**" **folder**, as the "**All Houses**" **page** uses it. Do it as shown below, so that the **layout is changed**, **depending on the user**:

_ViewStart.cshtml

```
@{
    if(this.User.IsAdmin())
    {
        Layout = "~/Areas/Admin/Views/Shared/_AdminLayout.cshtml";
    }
    else
    {
        Layout = "_Layout";
    }
}
```

Run the app and **look at the navigation bar** when you **log in with the Admin** and **with another user**. Links should **redirect you to the correct pages**.

2. Cache Pages

In this task, we want to **cache our "`All Users`" and "`All Rents`" pages**, as this will improve the overall **app's performance**. We will use **cache stored in the memory**, which is the simplest cache. The **`IMemoryCache` interface** will allow us to **refresh a given page on a set amount of time**, no matter if there is a **new data to be extracted from the database**.

To do this, we should first **register the cache service** in **`Program` class** of the "**`HouseRentingSystem.Web`" project**:

```
builder.Services.AddMemoryCache();
```

Be careful where you **place the registered cache service** between other services.

Next, go to the **`AdminConstants` class** and **add cache keys for the two pages**. We need them, as they are **unique identifiers for each object in the cache**. Do it like this:

```
public class AdminConstants
{
    public const string AreaName = "Admin";
    public const string AdminRoleName = "Administrator";
    public const string AdminEmail = "admin@mail.com";
    public const string UsersCacheKey = "UsersCacheKey";
    public const string RentsCacheKey = "RentsCacheKey";
}
```

Step 1: Cache the "All Users" Page

Now we will cache the "**`All Users`" page** first. To do this, go to the **`UserController` class** in the "**`/Areas/Admin/Controllers`" folder** and **inject the cache service**:

```
public class UserController : AdminController
{
    private readonly IUserService _users;
    private readonly IMemoryCache _cache;

    0 references
    public UserController(IUserService users, IMemoryCache cache)
    {
        _users = users;
        _cache = cache;
    }
```

Then, in the **`All()` method**, let's first check whether a **cache with the cache key and the users is already created**:

```
[Route("User/All")]
0 references
public async Task<IActionResult> All()
{
    var users = _cache
        .Get<IEnumerable<UserServiceModel>>(UsersCacheKey);
```

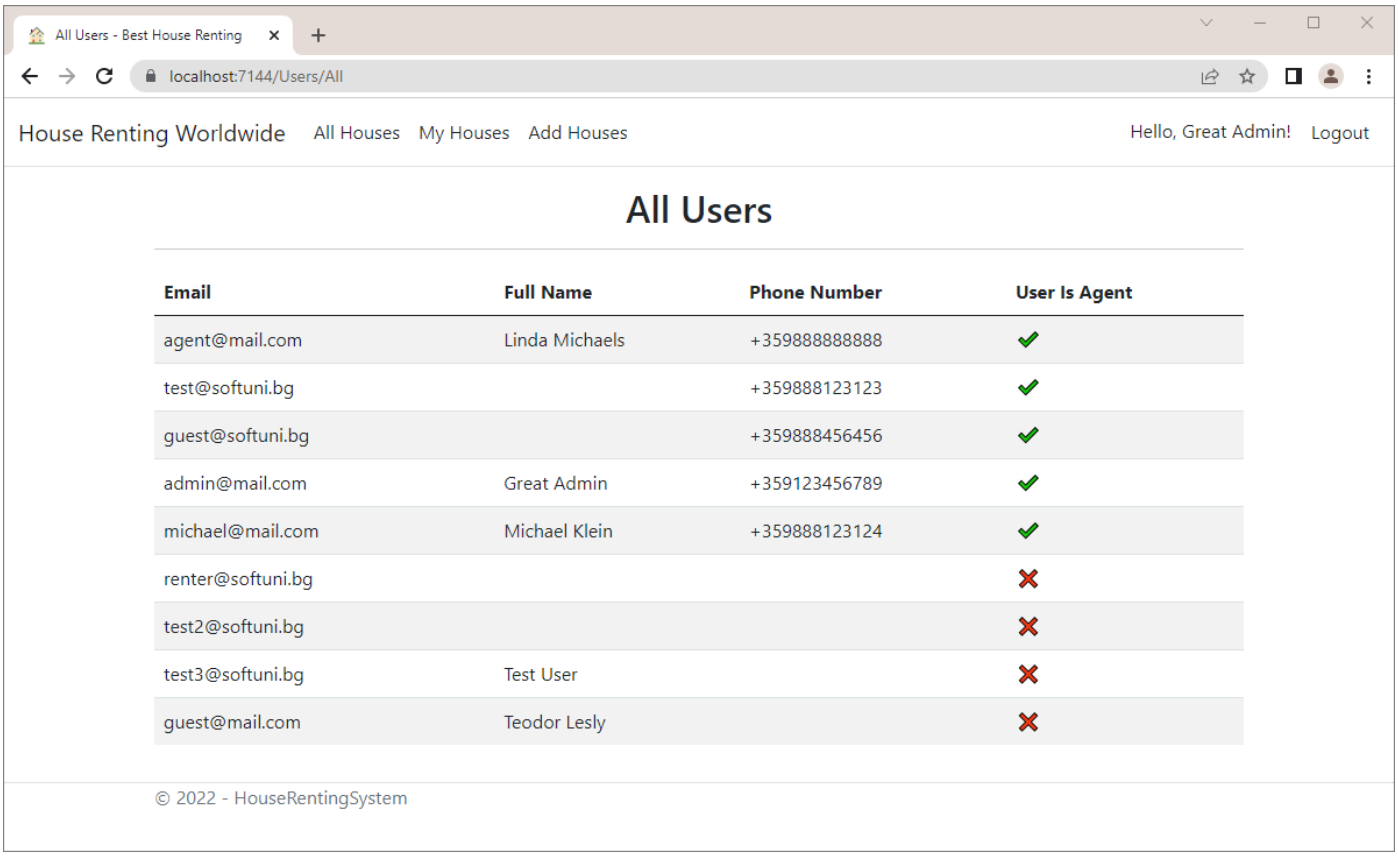
Then, if the **cache is not created or empty**, **create a cache with the cache key**, **users** and **an expiration of 5 minutes**:

```
    public async Task<IActionResult> All()
    {
        var users = _cache
            .Get<IEnumerable<UserServiceModel>>(UsersCacheKey);

        if (users == null)
        {
            users = await _users.All();

            var cacheOptions = new MemoryCacheEntryOptions()
               .SetAbsoluteExpiration(TimeSpan.FromMinutes(5));

            _cache.Set(UsersCacheKey, users, cacheOptions);
        }
```

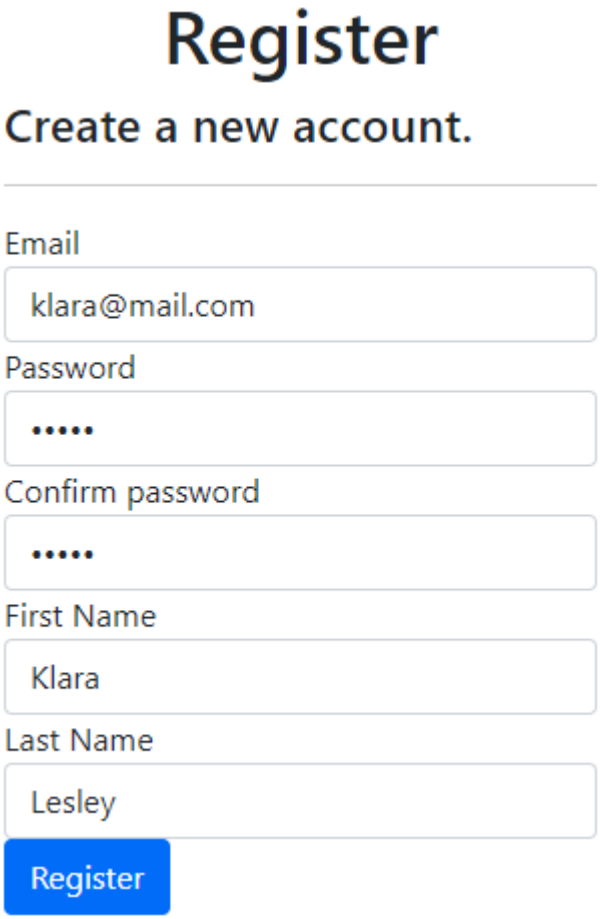
At the end, **return a view** with the **users**:

```
        return View(users);
    }
}
```

Now **run the app** in the browser, **log in with the `Admin`** and **navigate to the "`All Users`" page**:



Then, **register a new user**:



After a **successful registration**, **log in** with the **`Admin` again** and **look at the "`All Users`" page** – it should **not have the newly-created user** yet. Note that the **users cache is saved in the server's memory**, so you should **not rebuild or refresh the app**!

Wait for 5 minutes after the registration and **refresh the "`All Users`" page**. Then it should **display the new user**.

Step 2: Clear the Cache when Creating a New User

Now our "**`All Users`" page** is **refreshed on every 5 minutes**, no matter if there is a **new user in the database or not**. However, this is **not appropriate**, as the **`Admin` should not wait** to see new users. For this reason, we need to **clear the cache** when a **new user is created** successfully.

To do this, go to the **`Register.cshtml.cs` class** in the "**`/Areas/Identity/Pages/Account`" folder** and **inject the memory cache interface** through the **constructor**:

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

    0 references
    public RegisterModel(
        UserManager<ApplicationUser> userManager,
        SignInManager<ApplicationUser> signInManager,
        IMemoryCache cache)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _cache = cache;
    }
```

Then, **clear the cache** when the **user is created successfully** in the **`OnPostAsync(…)` method**:

```
            if (result.Succeeded)
            {
                await _signInManager.SignInAsync(user, isPersistent: false);
                _cache.Remove(AdminConstants.UsersCacheKey);
                return LocalRedirect(returnUrl);
            }
```

Now **test the new modification**. Go to the "**`All Users`**" **page** with the **`Admin`** and look at it:

Register a new user and make sure that they are **displayed immediately** on the "**`All Users`**" **page**:

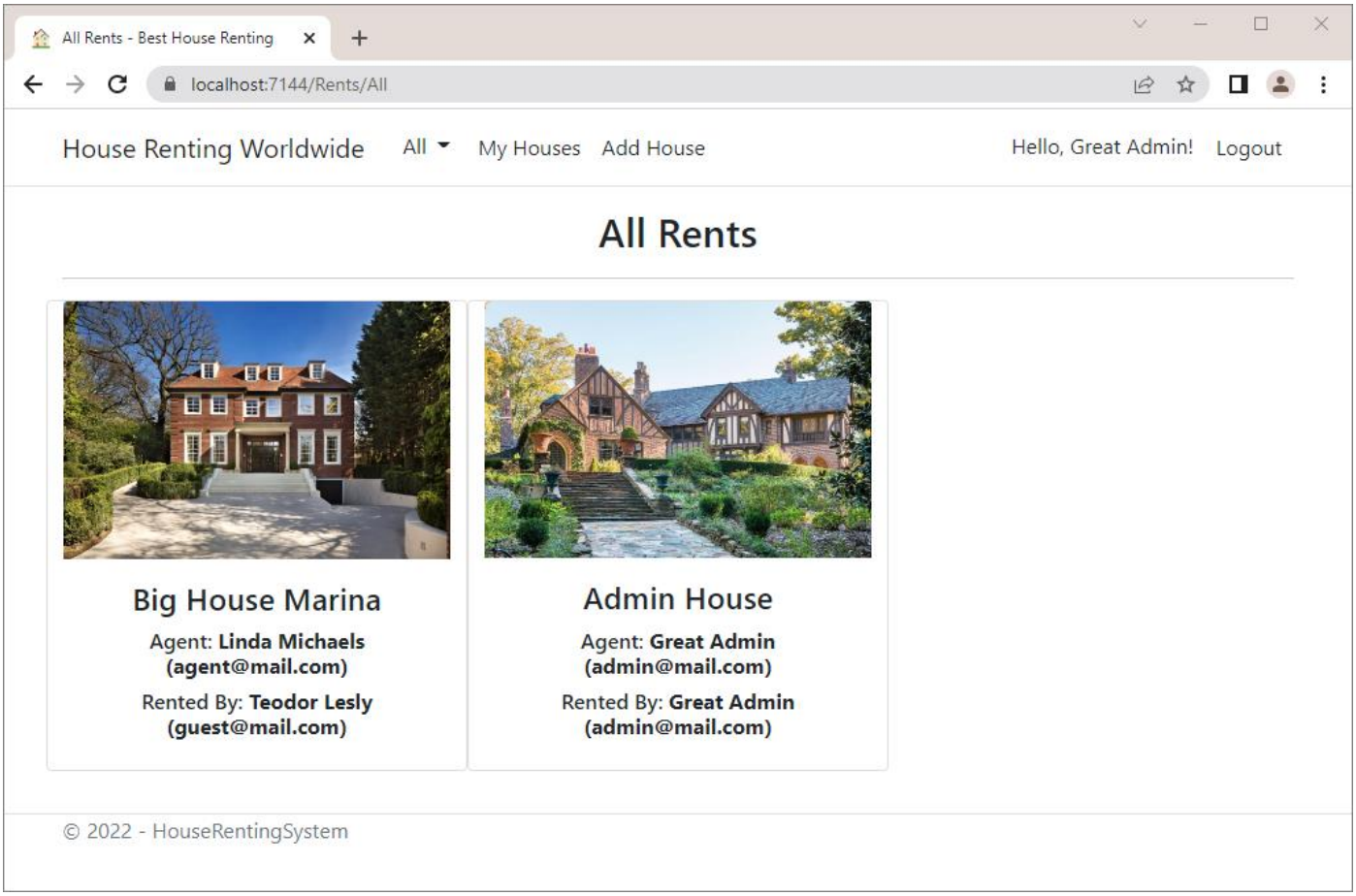
Now the **caching** of the users' page is **done correctly**.

Step 3: Cache the "All Rents" Page

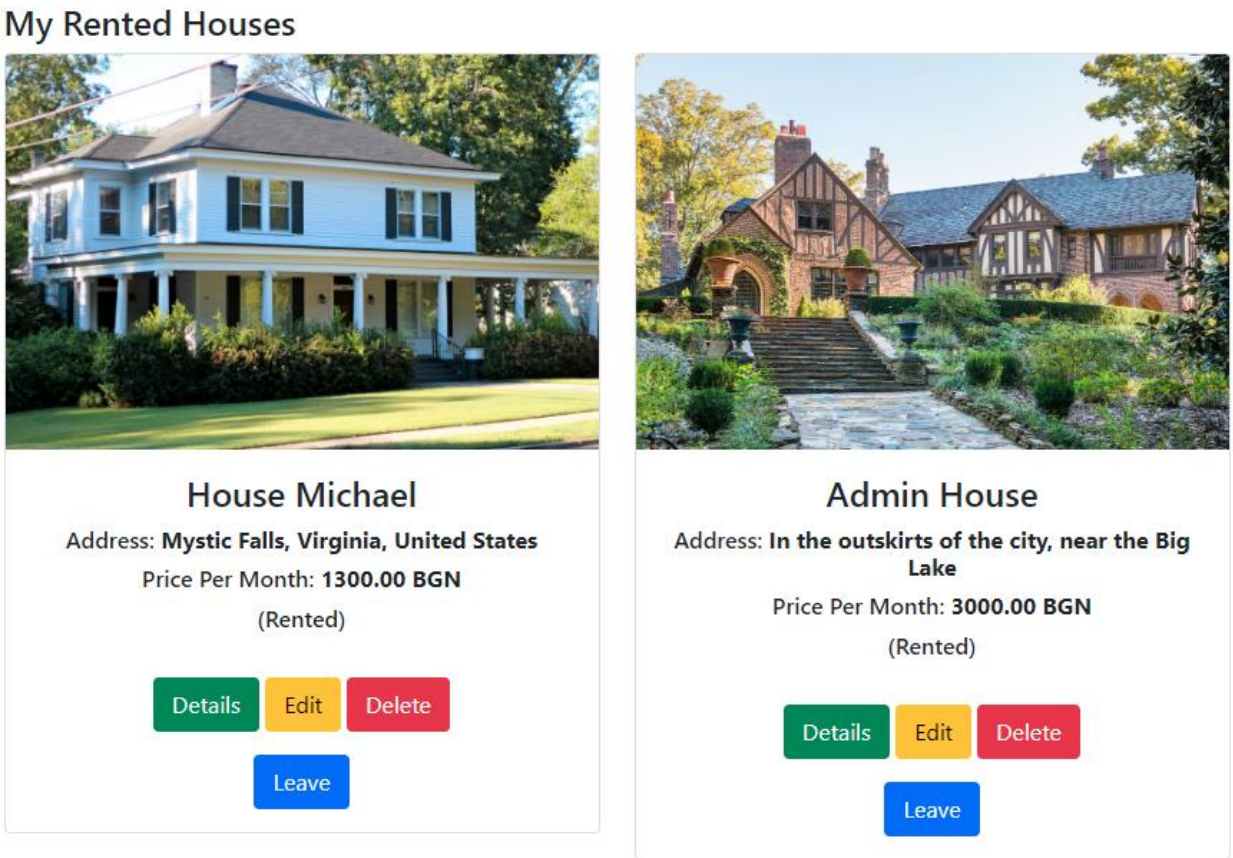
Now let's **cache the** "**`All Rents`**" **page** in the same way. Go to the **`RentController`** and **inject the service**. Then, **create a cache with a 5-minute expiration** and the **rents**. Do it like this:

```
public class RentController : AdminController
{
    private readonly IRentService _rents;
    private readonly IMemoryCache _cache;

    0 references
    public RentController(IRentService rents,
        IMemoryCache cache)
    {
        _rents = rents;
        _cache = cache;
    }

    [Route("Rent/All")]
    0 references
    public async Task<IActionResult> All()
    {
        var rents = _cache
            .Get<IEnumerable<RentServiceModel>>(RentsCacheKey);

        if (rents == null)
        {
            rents = await _rents.All();

            var cacheOptions = new MemoryCacheEntryOptions()
               .SetAbsoluteExpiration(TimeSpan.FromMinutes(5));

            _cache.Set(RentsCacheKey, rents, cacheOptions);
        }

        return View(rents);
    }
}
```

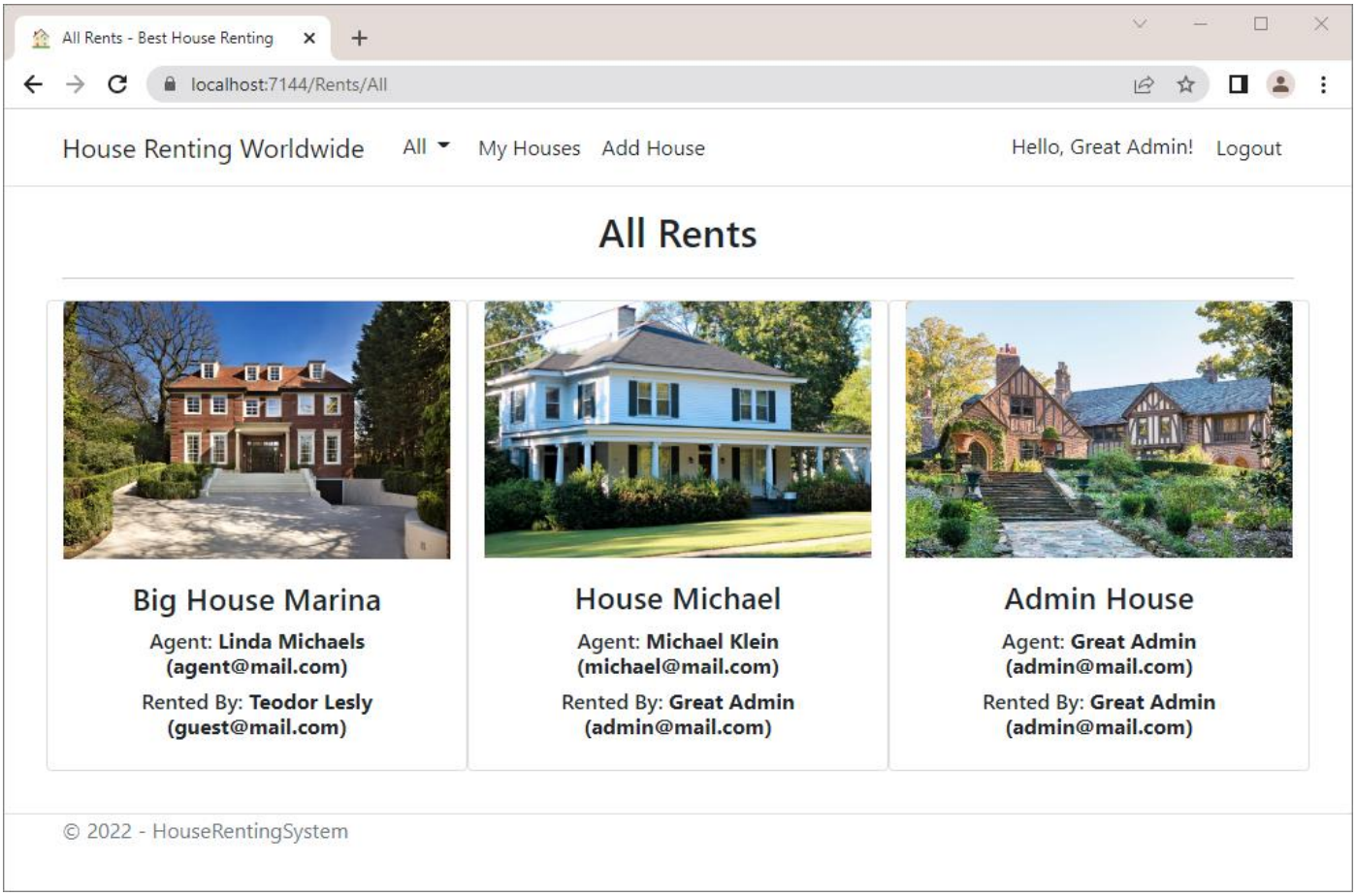
Run the app and **navigate to the** "**`All Rents`**" **page** with the **`Admin`**. You should see **all rented houses**:



Rent a house of your choice, for example the "**`Michael`**" **house**. It should be **displayed on the** "**`My Houses`**" **page**, but **not on the** "**`All Rents`**" **page** yet:



Wait for 5 minutes and make sure that the **rented house is displayed on the** "**`All Rents`**" **page** after **refresh**:



Now we have **cached pages in our app**, which are **refreshed on every 5 minutes**. However, let's again clear the cache when a house is rented or left, so that the page always displays the correct houses.

Go to the **`HouseController`** in the "**`/Controllers`**" **folder** and **inject the memory cache** through the **constructor**. Then, **clear the cache** in the **`Rent()`** and **`Leave()` methods**, when a **house is rented or left** successfully. Do it as shown below:

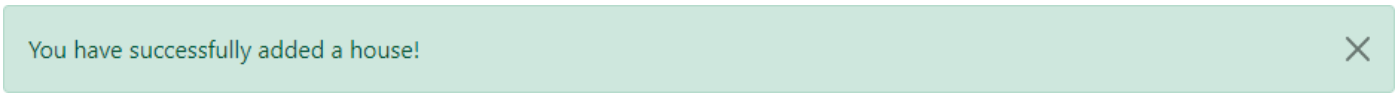
```
public class HouseController : Controller
{
    private readonly IHouseService _houses;
    private readonly IAgentService _agents;
    private readonly IMapper _mapper;
    private readonly IMemoryCache _cache;

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

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

        if (await _agents.ExistsById(User.Id())
            && User.IsAdmin() == false)...

        if (await _houses.IsRented(id))...

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

        _cache.Remove(RentsCacheKey);

        return RedirectToAction(nameof(All));
    }

    [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);

        _cache.Remove(RentsCacheKey);

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

Now try to **rent or leave a house** and make sure that this action result is **displayed** on the "**`All Rents`**" **page** without any delay.

3. Add TempData

`TempData` is used to **transfer data from view to controller** or **controller to view**. In this task, we want to **use a cookie-based `TempData`** to **show messages to the user** after **successful adding**, **editing**, **deleting**, **renting** and **leaving a house** and for **becoming an agent**. For example, the message for **successful adding of a new house** looks like this:



Note that these messages are shown only **once on a particular page** and **disappear when the page is refreshed**.

To **show these messages**, we will have to **add a message to `TempData`** in the **controllers** and **display these messages in views**. To display these messages on **any page** that we are redirected to (**only once**, as this is how **`TempData`** works), we will **add an alert** to **the** "**`_AdminLayout.cshtml`**" and "**`_Layout.cshtml`**" **views before `@RenderBody()`** like this:

_AdminLayout.cshtml

```
@inject IAgentService agents

<!DOCTYPE html>
<html lang="en">
<head>...
<body>
    <header>...
    <div class="container">
        <main role="main" class="pb-3">
            @if (TempData["message"] != null)
            {
                <div class="alert alert-success alert-dismissible fade show"
                 role="alert">
                    @TempData["message"]
                    <button type="button" class="btn-close"
                        data-bs-dismiss="alert"
                        aria-label="Close">
                        <span aria-hidden="true">&times;</span>
                    </button>
                </div>
            }
            @RenderBody()
        </main>
    </div>
    <footer class="border-top footer text-muted">...

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

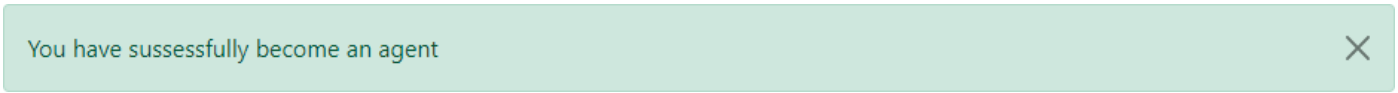
_Layout.cshtml

```
@inject IAgentService agents

<!DOCTYPE html>
<html lang="en">
<head>...
<body>
    <header>...
    <div class="container">
        <main role="main" class="pb-3">
            @if (TempData["message"] != null)
            {
                <div class="alert alert-success alert-dismissible fade show">
                    @TempData["message"]
                    <button type="button" class="btn-close"
                        data-bs-dismiss="alert">
                        <span aria-hidden="true">&times;</span>
                    </button>
                </div>
            }
            @RenderBody()
        </main>
    </div>
    <footer class="border-top footer text-muted">...

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The above **code in the views displays** the **TempData message** (if it **exists**) in an **alert**. Now let's **add the messages** themselves.

Step 1: Message for Becoming an Agent

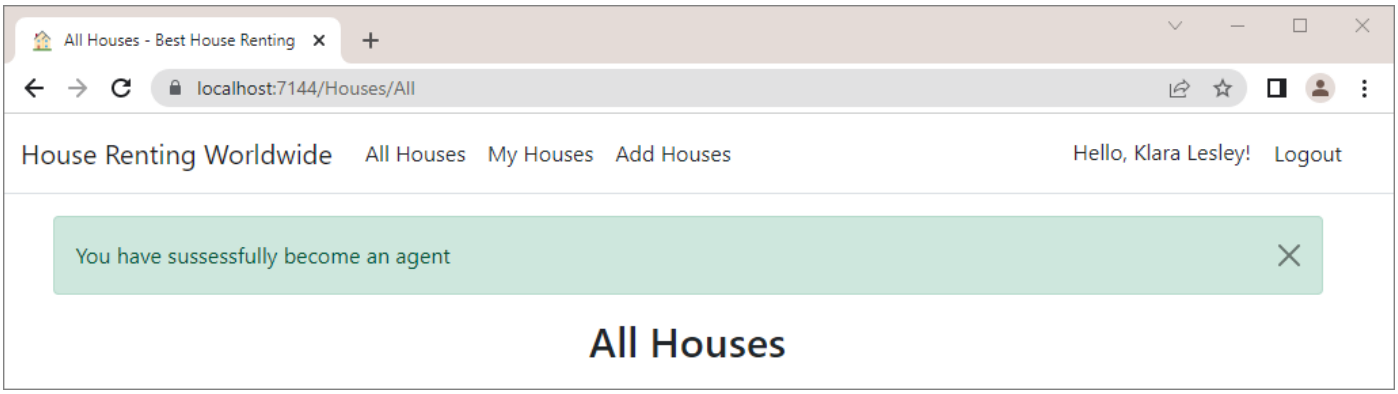
Start with **adding a TempData message** to be shown when the **user becomes an agent**. It should be the following:



To do this, go to the **AgentController** in the "**HouseRentingSystem.Web**" **project**. In the **Become(BecomeAgentFormModel model)** we should **add a message**, which will be **displayed in the view**. **Set the message** just **before you return an action result**:

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

    if (await _agents.ExistsById(userId))...

    if (await _agents.UserWithPhoneNumberExists(model.PhoneNumber))...

    if (await _agents.UserHasRents(userId))...

    if (!ModelState.IsValid)...

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

    TempData["message"] = "You have successfully become an agent";

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

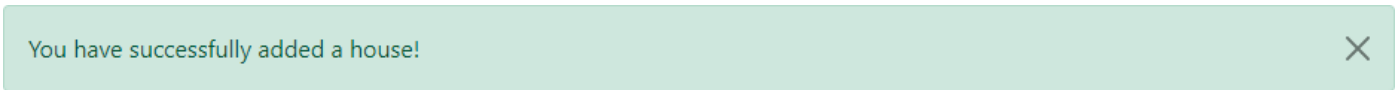
As you know, the **user is redirected to the "`All Houses`" page** after **successfully becoming an agent**. **Run the app** and **make the new user we created** (*Klara Lesley*) **become an agent**. When done, the "`All Houses`" **page** should **display the message** we added:



If you **refresh the page**, you will see that the **alert disappears**, as the **`TempData` value** is already **read and empty**.

Step 2: Message for Adding a House

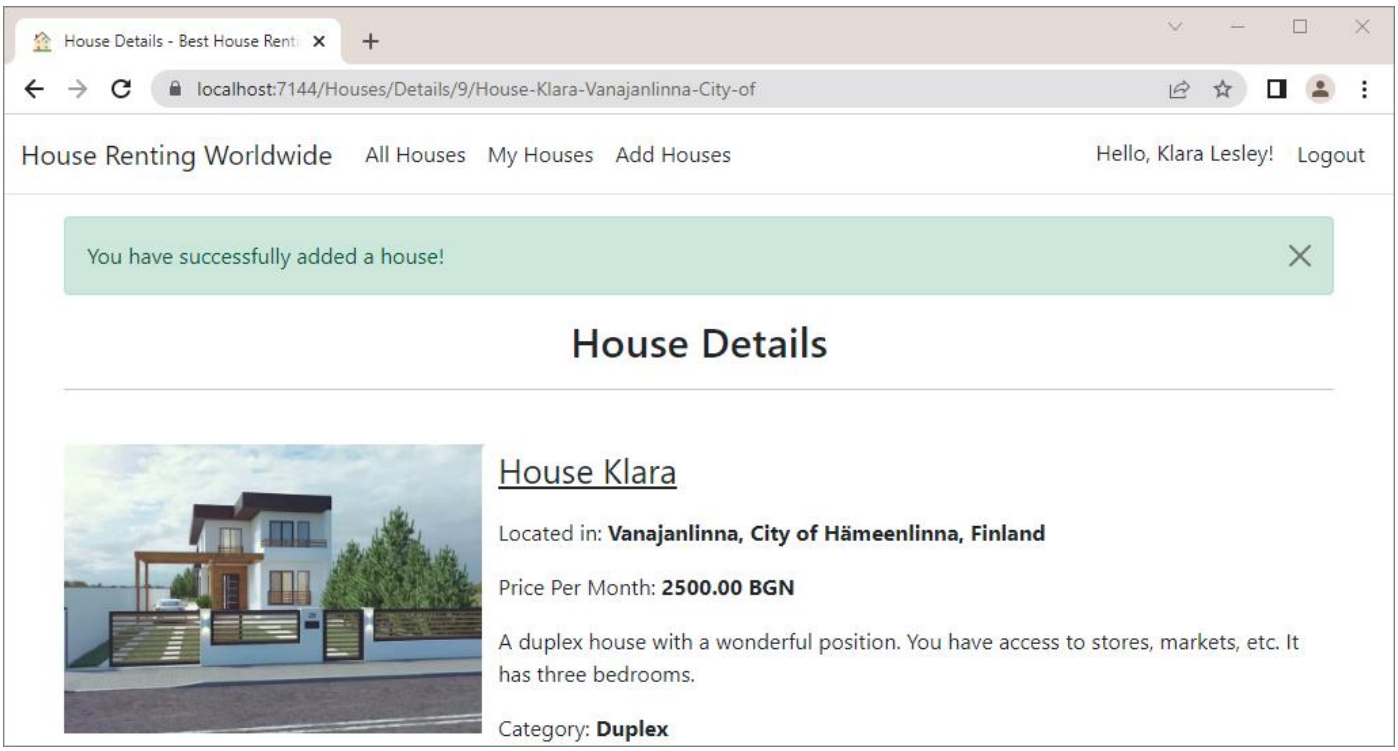
When an **agent adds a house successfully**, they should see the following message:



Go to the **`Add(HouseFormModel model)` method** of the **`HouseController`** and **assign a value to `TempData`** like this:

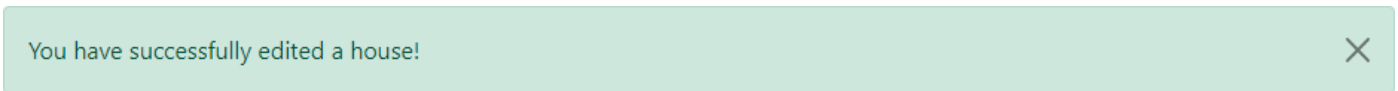
```
[HttpPost]
0 references
public async Task<IActionResult> Add(HouseFormModel model)
{
    if (await _agents.ExistsById(User.Id()) == false)...

    if (await _houses.CategoryExists(model.CategoryId) == false)...

    if (!ModelState.IsValid)...

    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);

    TempData["message"] = "You have successfully added a house!";

    return RedirectToAction(nameof(Details),
        new { id = newHouseId, information = model.GetInformation() });
}
```

Run the app and **create a new house**. When the **creation is successful**, the following **alert with the message** should be shown on the "**House Details**" **page**:



Step 3: Message for Editing a House

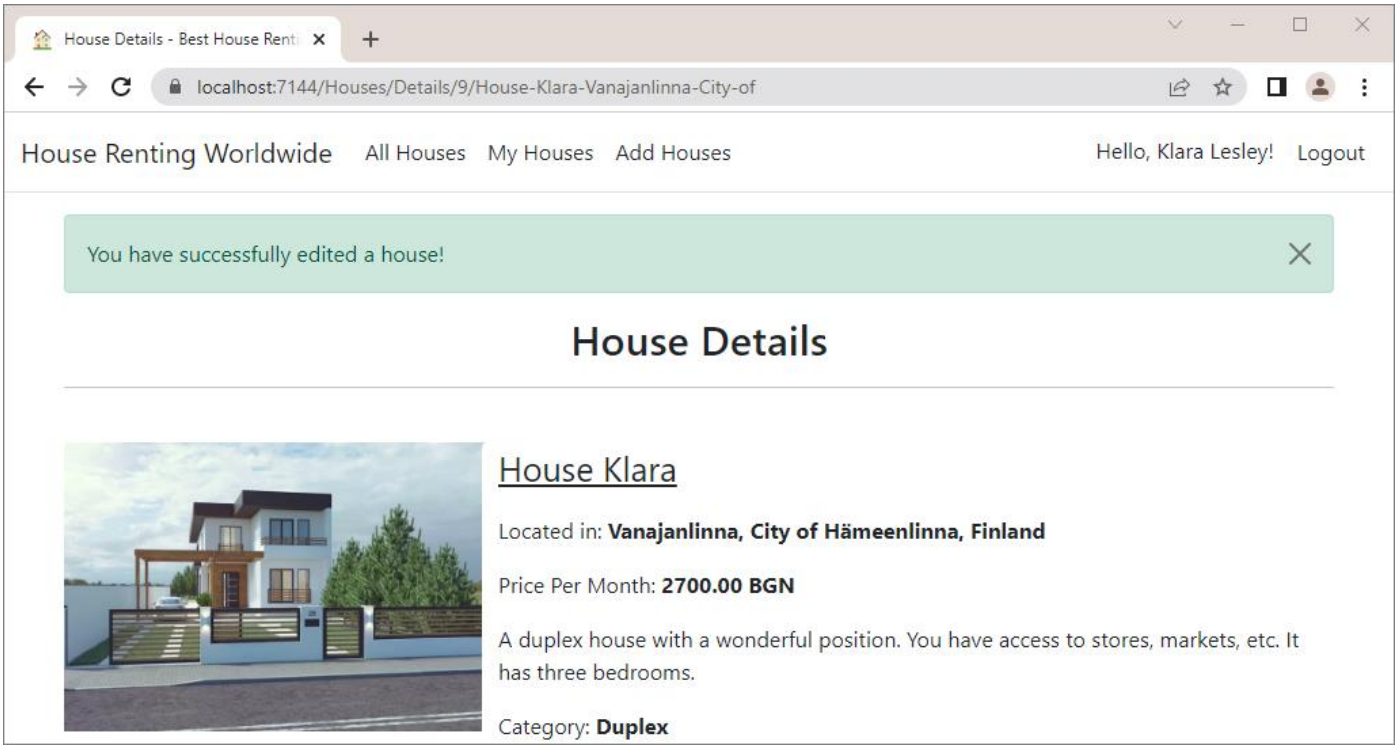
The next **message with TempData** we will show is on **successfully editing a house**:



Add the **TempData message** in the **Edit(int id, HouseFormModel model) method** of the **HousesController**:

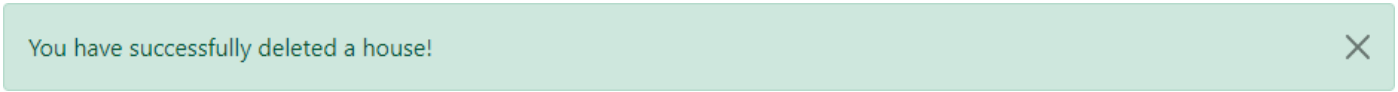
```
TempData["message"] = "You have successfully edited a house!";
```

Run the browser, **edit a house** and make sure the **message is displayed correctly** on the "**House Details**" **page**:



Step 4: Message for Deleting a House

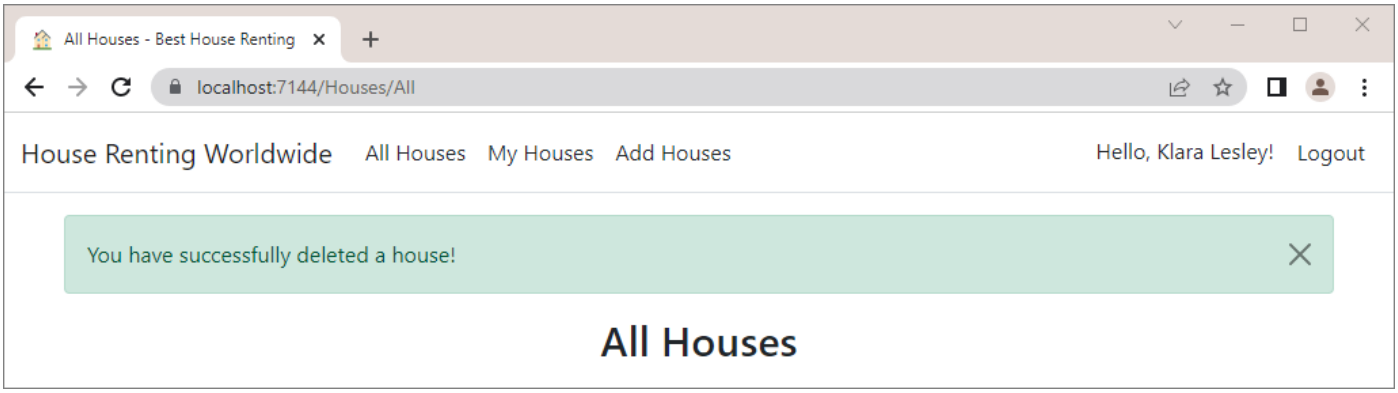
The **message for successfully deleting a house** is the following:



Assign a value to **`TempData`** in the **`Delete(HouseDetailsViewModel model)` method** of the **`HouseController`**:

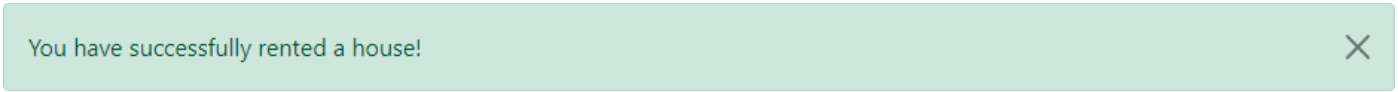
```
TempData["message"] = "You have successfully deleted a house!";
```

Delete a house and make sure the **message is displayed** on the "**`All Houses`**" **page**:



Step 5: Message for Renting a House

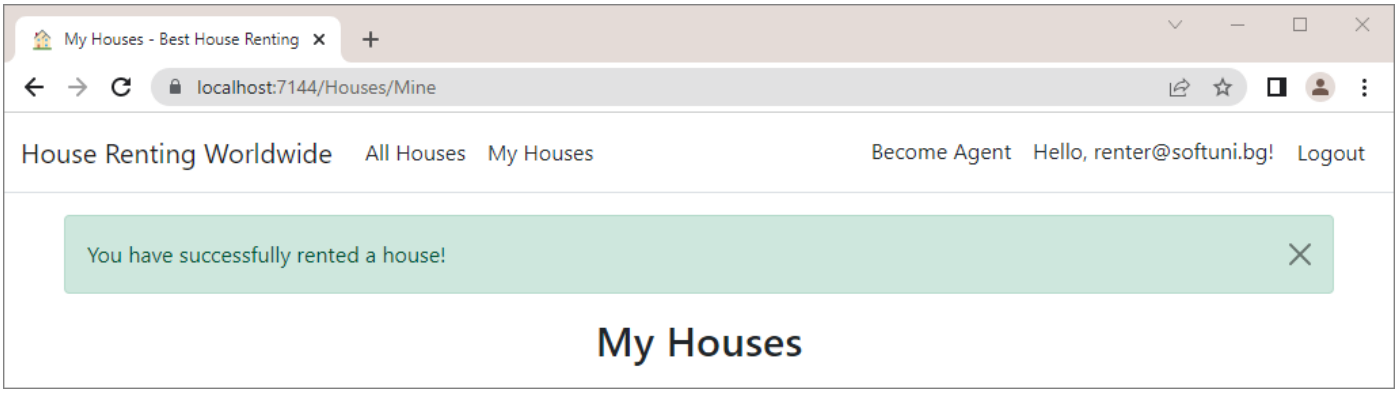
The next message we will show is when a **user rents a house**:



Write the following code to the **`Rent(int id)` method** in the **`HouseController` class**:

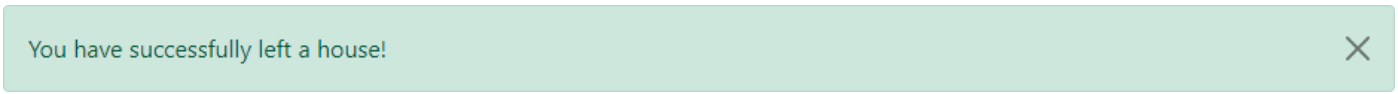
```
TempData["message"] = "You have successfully rented a house!";
```

Rent a house and see if the **message is displayed correctly** on the "**My Houses**" **page after the redirect**:



Step 6: Message for Leaving a House

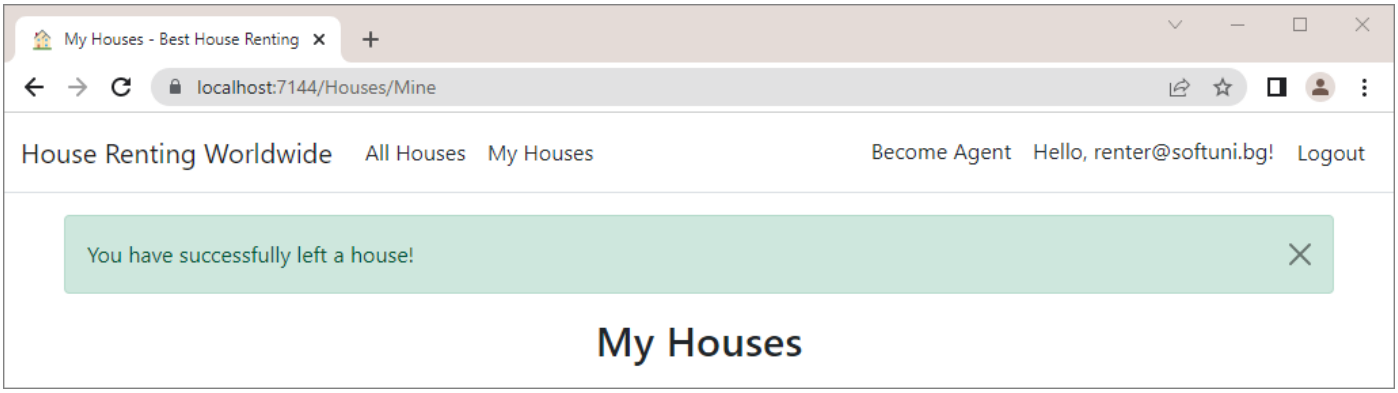
The last message we should display is when a **user leaves a rented house**. It should look like this:



Assign a **TempData value** in the **Leave(int id) method** of the **HouseController**:

```
TempData["message"] = "You have successfully left a house!";
```

Try to **leave a house** and make sure the **message is displayed** on the "**My Houses**" **page** like this:



Now we have **added success messages** for user actions with **TempData** and improved our app design to be more **user-friendly**.

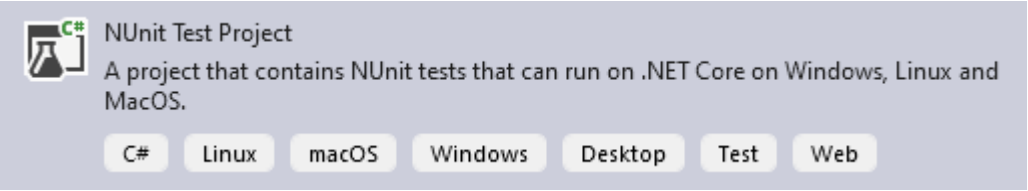
4. Write Unit Tests

Those were the **last steps for developing our** "**HouseRentingSystem**" **app**. Now, we will see how to **write tests** for the app's functionalities we already implemented.

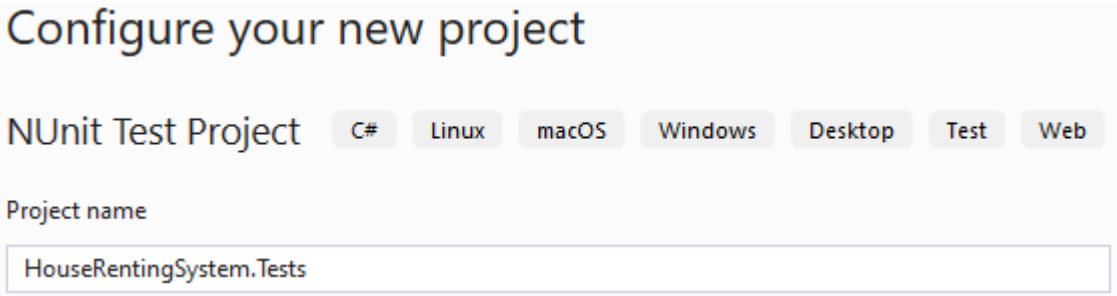
Step 1: Create a Test Project for Unit Tests

In this task, we will show you **how to write unit tests**. For them, we will need to **mock the database and AutoMapper**, as we should isolate and focus on the code being tested and not on the behavior or state of **external dependencies**. In our case, we will **write tests only for positive test cases**, because we do not intend to test the app exhaustively, but to learn how to create tests. However, in **real projects you should write positive and negative test cases**.

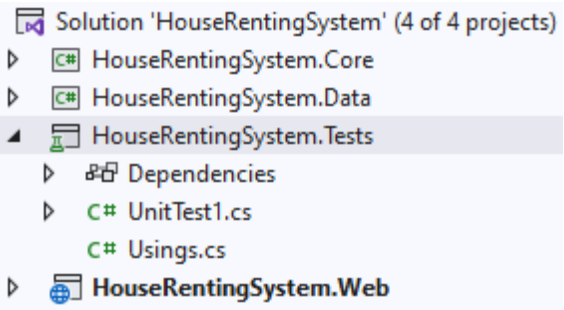
First, let's **create a test project** in our "**HouseRentingSystem**" **solution**. **Right-click** on the solution and choose **[Add] → [New Project]** and select the "**NUnit Test Project**" **template**:



Name the project "**`HouseRentingSystem.Tests`**":

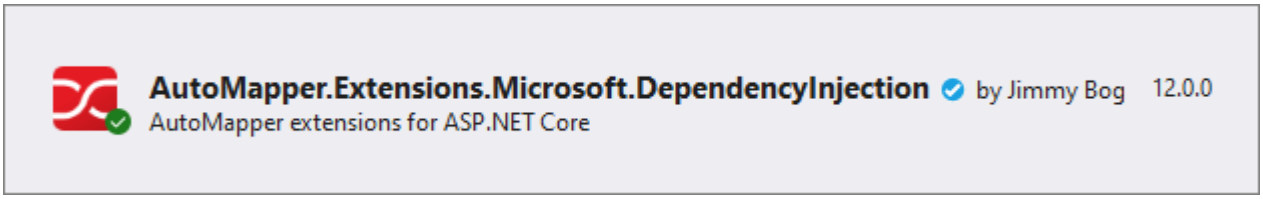


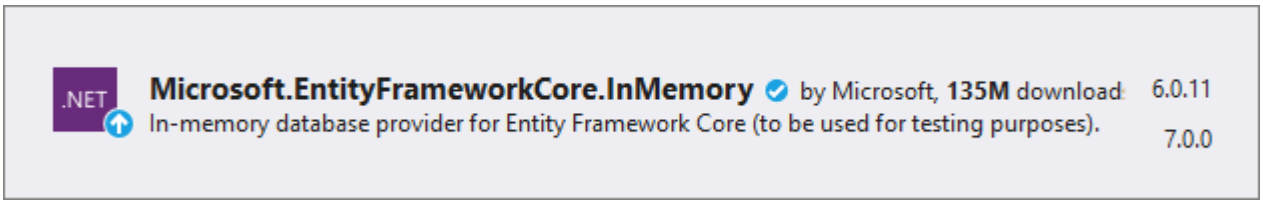
The **created project** looks like this:



You can **delete** the **`UnitTest1.cs` file**, as we will create our test classes.

The default **`NUnit` packages** are **already installed** in the project. However, you should install some additional ones. For example, the **mocked database** will be an **in-memory database**, so we need the **`Microsoft.EntityFrameworkCore.InMemory` package**. We also need the **`AutoMapper.Extensions.Microsoft.DependencyInjection` package**, as we will create an **`AutoMapper` mock**. Be careful with **package versions**! The additional packages you need are these:





One more thing we should do is to make our **tests project dependent on the tested projects**. For now, we want to **test the data and service layer**, so the "**`HouseRentingSystem.Tests`**" **project** should **depend** on the "**`HouseRentingSystem.Core`**" and "**`HouseRentingSystem.Data`**" ones.

Now our project is ready, so we can start **adding different test classes**.

Step 2: Modify the HouseRentingDbContext Class

The **`HouseRentingDbContext` class** is **responsible for our database**. However, in our tests, we want to use a **separate database**, which is **not our production one**. For this reason, we will **mock the db `context`** and **create an in-memory database for testing**.

To do this, however, we should first **modify our `HouseRentingDbContext` class**, so that we can **mock it correctly**. First, we should **modify the class constructor** – the **in-memory database we will create is non-relational**, so we **cannot migrate** it. We should only **make sure that it is created**, so the **constructor** should use a **different method**, depending on the database, like this:

```
public HouseRentingDbContext(DbContextOptions<HouseRentingDbContext>
    options) : base(options)
{
    if (Database.IsRelational())
    {
        Database.Migrate();
    }
    else
    {
        Database.EnsureCreated();
    }
}
```

Next, we should **modify our database seed** – our **testing database should not be seeded**, as the production one, but will have **different data**. For this reason, **accept a `bool` parameter through the constructor**, which defines whether the **database should be seeded or not**. If it is **`false`**, **do not invoke the seed methods** in the **`OnModelCreating(…)` method**. Do it like this:

```
public class HouseRentingDbContext : IdentityDbContext<ApplicationUser>
{
    private bool _seedDb;

    public HouseRentingDbContext(DbContextOptions<HouseRentingDbContext>
        options, bool seed = true) : base(options)
    {
        if (Database.IsRelational())...
        else...

        _seedDb = seed;
    }
```

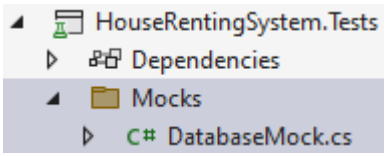
```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder
        .Entity<House>()
        .HasOne(h => h.Category)
        .WithMany(c => c.Houses)
        .HasForeignKey(h => h.CategoryId)
        .OnDelete(DeleteBehavior.Restrict);

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

    if (_seedDb)
    {
        SeedUsers();
        builder.Entity<ApplicationUser>()
                .HasData(AgentUser,
                GuestUser,
                AdminUser);

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

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

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

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

Now let's see how to **mock the database** for our tests.

Step 3: Create a Database Mock

In this task, we will **create a mocked database**. The reason for this is that our **service classes need a db `context`**, but we **cannot work with the production database**. However, it is important to **unit-test our business logic** and its **interaction with a database**. That's why we need to **create an artificial database for testing** purposes only.

Our **mocked database** will be an **in-memory database**, which will be **filled with data**, which tests need. To begin with, create a "**`Mocks`**" **folder** in the "**`HouseRentingSystem.Tests`**" **project**, where all our **mock classes** will be. In it, create the **`DatabaseMock` class**:



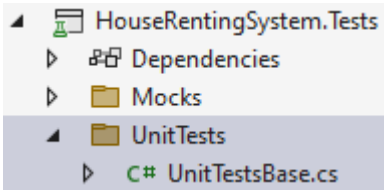

```
public class DatabaseMock
{
}
```

Make the class **static** and **add a single property `Instance`**. This property will have only a **`getter`** and will **return a new `HouseRentingDbContext` instance**, which will **create a not seeded**, **in-memory database** with a **unique name** every time:

```
public static class DatabaseMock
{
    0 references
    public static HouseRentingDbContext Instance
    {
        get
        {
            var dbContextOptions = new DbContextOptionsBuilder<HouseRentingDbContext>()
                .UseInMemoryDatabase("HouseRentingInMemoryDb"
                    + DateTime.Now.Ticks.ToString())
                .Options;

            return new HouseRentingDbContext(dbContextOptions, false);
        }
    }
}
```

Step 4: Create a Base Tests Class

Before we create our test classes and tests, let's **create a base tests class**, which should **instantiate the mocked database and seed some data in it**. Create the **`UnitTestsBase` class** in a "**`UnitTests`**" **folder** (for our unit test classes) in the "**`HouseRentingSystem.Tests`**" **project**:



In the class, **use the mocked database instance** and **seed a renter**, **an agent**, **a rented house with a category** and a **non-rented house**. Create a **`SetUpBase()` method**, which has the **`[OneTimeSetUp]` attribute** and will be **invoked before all tests**:

```
public class UnitTestsBase
{
    protected HouseRentingDbContext _data;
    protected IMapper _mapper;

    [OneTimeSetUp]
    0 references
    public void SetUpBase()
    {
        _data = DatabaseMock.Instance;
        _mapper = MapperMock.Instance;
        SeedDatabase();
    }
```

Then, **create properties for the seeded data** and **seed the database** in the **`SeedDatabase()` method**. Note that we are **not obliged to add values to all entity properties**, as our **in-memory database has no restrictions**:

```
public ApplicationUser Renter { get; private set; }
20 references
public Agent Agent { get; private set; }
12 references
public House RentedHouse { get; private set; }
private void SeedDatabase()
{
    Renter = new ApplicationUser()
    {
        Id = "RenterUserId",
        Email = "rent@er.bg",
        FirstName = "Renter",
        LastName = "User"
    };
    _data.Users.Add(Renter);
    Agent = new Agent()
    {
        PhoneNumber = "+359111111111",
        User = new ApplicationUser()
        {
            Id = "TestUserId",
            Email = "test@test.bg",
            FirstName = "Test",
            LastName = "Tester"
        }
    };
    _data.Agents.Add(Agent);
    RentedHouse = new House()
    {
        Title = "First Test House",
        Address = "Test, 201 Test",
        Description = "This is a test description. This is a test description. This is a test description.",
        ImageUrl = "https://www.bhg.com/thmb/0Fg0imFSA6HVZMS2DFWPvjbYDoQ=/1500x0/filters:no_upscale():max_by
        Renter = Renter,
        Agent = Agent,
        Category = new Category() { Name = "Cottage" }
    };
    _data.Houses.Add(RentedHouse);

    var nonRentedHouse = new House()
    {
        Title = "Second Test House",
        Address = "Test, 204 Test",
        Description = "This is another test description. This is another test description.",
        ImageUrl = "https://images.adsttc.com/media/images/629f/3517/c372/5201/650f/1c7f/large_jpg/hyde-park
        Renter = Renter,
        Agent = Agent,
        Category = new Category() { Name = "Single-Family" }
    };

    _data.Houses.Add(nonRentedHouse);
    _data.SaveChanges();
}
```

You can copy the **ImageUrl**s from here or choose your own:

https://www.bhg.com/thmb/0Fg0imFSA6HVZMS2DFWPvjbYDoQ=/1500x0/filters:no_upscale():max_bytes(150000):strip_icc()/white-modern-house-curved-patio-archway-c0a4a3b3-aa51b24d14d0464ea15d36e05aa85ac9.jpg
https://images.adsttc.com/media/images/629f/3517/c372/5201/650f/1c7f/large_jpg/hyde-park-house-robeson-architects_1.jpg?1654601149

At the end, create the **TearDownBase() method**, which should have the **[OneTimeTearDown] attribute** and will be **executed after all tests**. It should only **dispose of the database** like this:

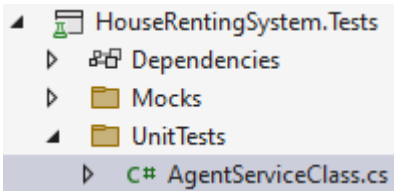
```
    [OneTimeTearDown]
    0 references
    public void TearDownBase()
       => _data.Dispose();
}
```

Now all **test classes** we create should **inherit the UnitTestsBase class** to **access the created database** with its **seeded data**.

Step 5: Create AgentService Tests

The **AgentService unit tests** we will create will test the **service method functionalities**, which will interact with our **mocked database**.

First, **create the AgentServiceTests class**:



Then, use the **[TestFixture] attribute** to **mark the class as a test class**. Also, it should **inherit the base tests class** and **use its db context** to **instantiate the AgentService class**, which we want to test, in its **SetUp() method**:

```
[TestFixture]
1 reference
public class AgentServiceTests : UnitTestsBase
{
    private IAgentService _agentService;

    [OneTimeSetUp]
    0 references
    public void SetUp()
        => _agentService = new AgentService(_data);
```

Now let's **examine the methods** of the **AgentService class**, which we should test. It is important to look at methods, because you can sometimes find out that you have **missed something or something is wrong**.

For example, in the **AgentService** class we can notice that the **UserWithPhoneNumberExists() method's name** is **not suitable** and meaningful enough, as it returns whether an **agent with the given phone number exists**. For this reason, let's **rename the method** to "**AgentWithPhoneNumberExists**":

```
public interface IAgentService
{
    2 references
    Task<bool> UserWithPhoneNumberExists(string phoneNumber);
```

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

Another thing we can notice is that the **`UserHasRents(…)` method's place is not in the `AgentService` class**, as it is **connected to users**, not to agents. For this reason, let's **move the method** to the **`UserService` class** and use it where needed. Don't forget to **modify the `IAgentService` and `IUserService` interfaces**, as well. All **modifications** are shown below:

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

```
public class AgentService : IAgentService
{
    2 references
    public async Task<bool> UserHasRents(string userId)
    {
        return await _data.Houses.AnyAsync(h => h.RenterId == userId);
    }
```

```
public interface IUserService
{
    0 references
    Task<bool> UserHasRents(string userId);
```

```
public class UserService : IUserService
{
    1 reference
    public async Task<bool> UserHasRents(string userId)
        => await _data.Houses.AnyAsync(h => h.RenterId == userId);
```

You should also **modify the `AgentController`** and its **`Become(…)` method** to use the above method:

```
public class AgentController : Controller
{
    private readonly IAgentService _agents;
    private readonly IUserService _users;

    0 references
    public AgentController(IAgentService agents,
        IUserService users)
    {
        _agents = agents;
        _users = users;
    }
```

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

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

After we have **examined the `AgentService` class**, go back to the **`AgentServiceTests` test class** and let's **write test methods**. We have already **instantiated the service and db `context` classes**, so we have everything for our tests.

Test the GetAgentId(…) Method

Start with testing the **`GetAgentId(…)` service method**, which **returns the id of an agent with a given user id**, if it **exists**. Create a **testing method** with the **`Act-Arrange-Assert` pattern** and the **`[Test]` attribute** like this:

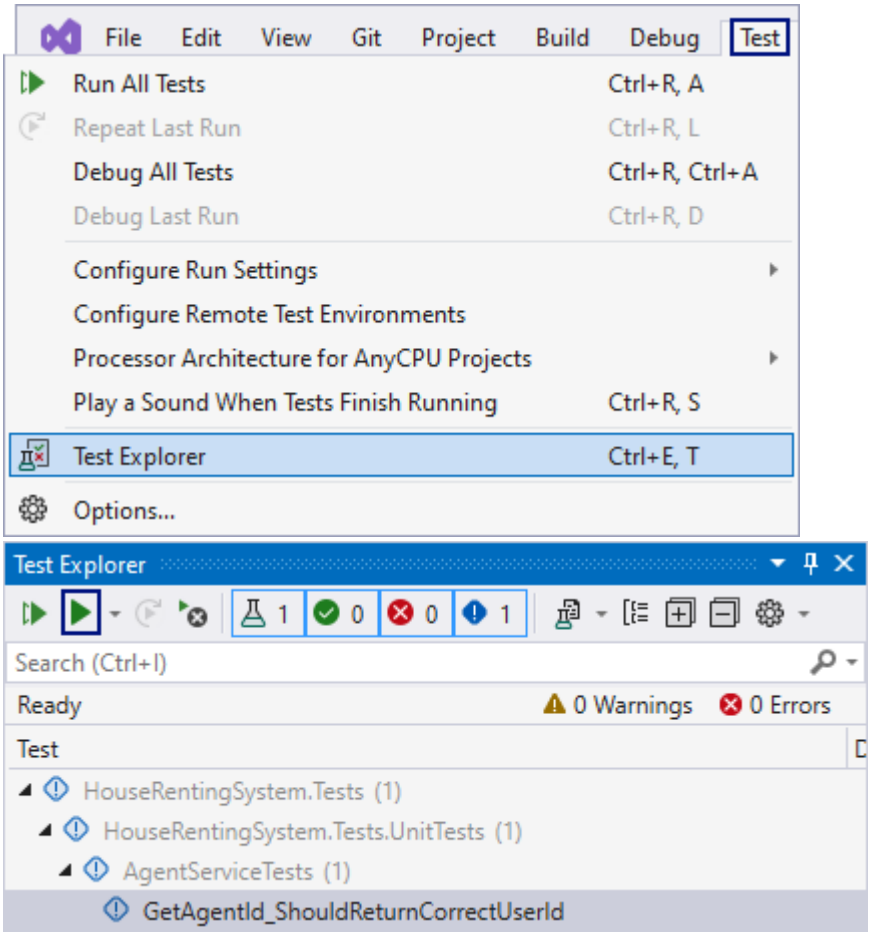
```
[TestFixture]
0 references
public class AgentServiceTests : UnitTestsBase
[Test]
0 references
public void GetAgentId_ShouldReturnCorrectUserId()
{
    // Arrange

    // Act: invoke the service method with valid id

    // Assert a correct id is returned
}
```

We already **have an `Agent` in our database**, which we will **use in for this test**. So, we should **directly invoke the `GetAgentId(…)` service method** and **get its returned value**. At the end, we should **assert that the returned user id** is the same as this in the **`UserId` property of the agent**. Write the test like this:

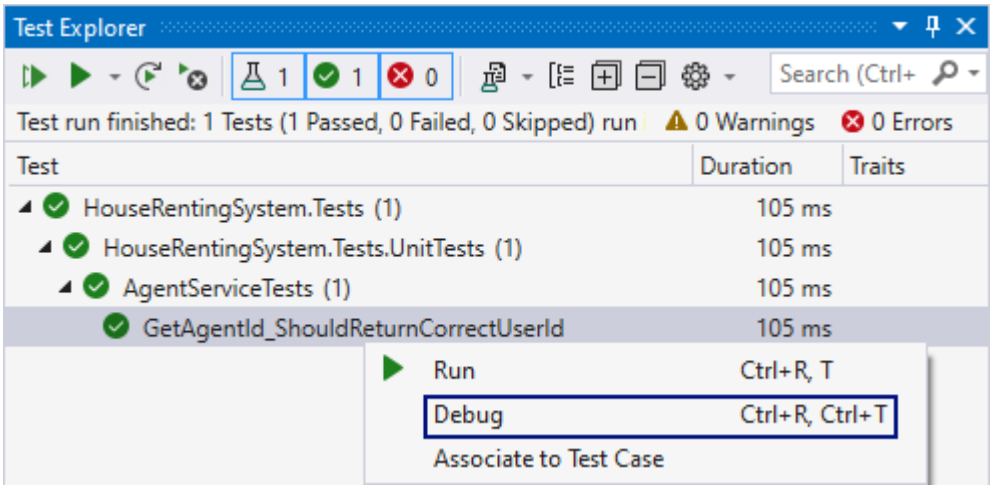
```
[Test]
0 references
public async Task GetAgentId_ShouldReturnCorrectUserId()
{
    // Arrange

    // Act: invoke the service method with valid id
    var resultAgentId = _agentService.GetAgentId(Agent.UserId);

    // Assert a correct id is returned
    Assert.That(Convert.ToInt32(resultAgentId), Is.EqualTo(Agent.Id));
}
```

Note that in the **`Assert.AreEqual(Object expected, Object actual)` method** you should first place the **expected value**, then the **actual one**, so that you can have a **correct exception message** in case of a **wrong test result**.

Our test is ready. Now open the **`Test Explorer`** from **`[Test]`** → **`[Test Explorer]`** and **run the test** with the **arrow button**. It should be successful:



Note that if you have **any trouble with a test**, you can **add a breakpoint** and **debug it** by **right-clicking** on the test and **choosing `[Debug]`**:

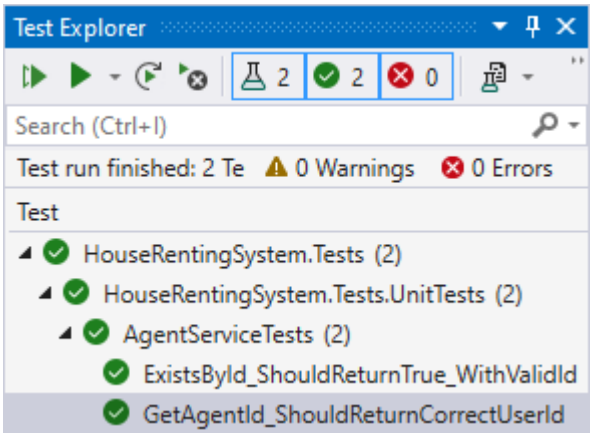


Test the ExistsById(...) Method

Our **next test** will be for the **`ExistsById(...)` service method**. In the test, we should **invoke the method** and make sure that it **returns `true` when we test it with a valid user id**. Do it like this:

```
[Test]
0 references
public async Task ExistsById_ShouldReturnTrue_WithValidIdAsync()
{
    // Arrange

    // Act: invoke the service method with valid agent id
    var result = await _agentService.ExistsById(Agent.UserId);

    // Assert the method result is true
    Assert.IsTrue(result);
}
```

Run all tests with the **double arrow button** and make sure that they are **all sucessful**:

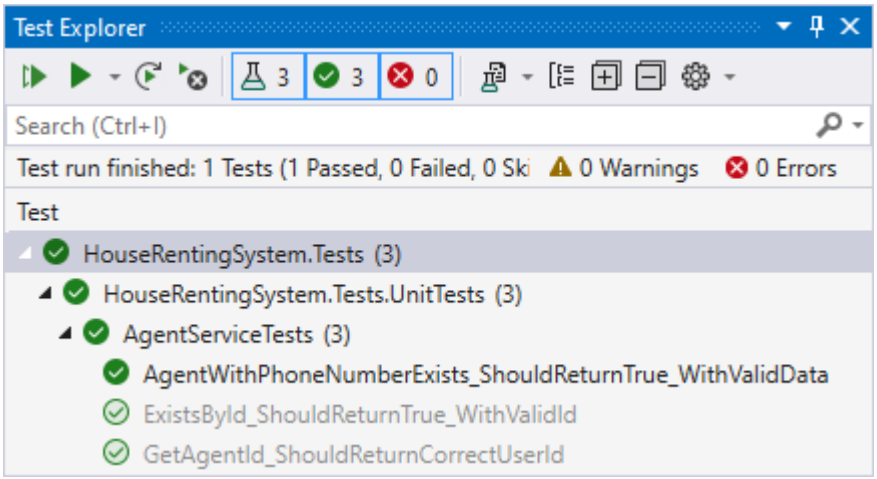


Test the UserWithPhoneNumberExists(…) Method

To test the **`UserWithPhoneNumberExists(…)` method** we should **invoke it** with an **already-used phone number** and make sure that it **returns `true`**, meaning that an agent with this phone number already exists (as we added it to our database). Write the test as shown below:

```
[Test]
0 references
public async Task AgentWithPhoneNumberExists_ShouldReturnTrue_WithValidData()
{
    // Arrange

    // Act: invoke the service method with valid agent phone num
    var result = await _agentService.UserWithPhoneNumberExists(Agent.PhoneNumber);

    // Assert the method result is true
    Assert.IsTrue(result);
}
```

Run the test and make sure it is successful:



Test the Create(…) Method

The last and most difficult method from the **`AgentService` class** is the **`Create(…)` method** for **adding an agent** to the database. To **test** this method, we should first **get the number of agents** in the database **before the creation:**

```
    [Test]
    0 references
    public async Task CreateAgent_ShouldWorkCorrectly()
    {
        // Arrange: get all agents' current count
        var agentsCountBefore = _data.Agents.Count();
```

Then, **invoke the `Create(…)` method**. Note that it will **create the agent in the database**, no matter if there is an **agent with the given user id** already – the **method does not check** this. Do it like this:

```
        // Act: invoke the service method with valid data
        await _agentService.Create(Agent.UserId, Agent.PhoneNumber);
```

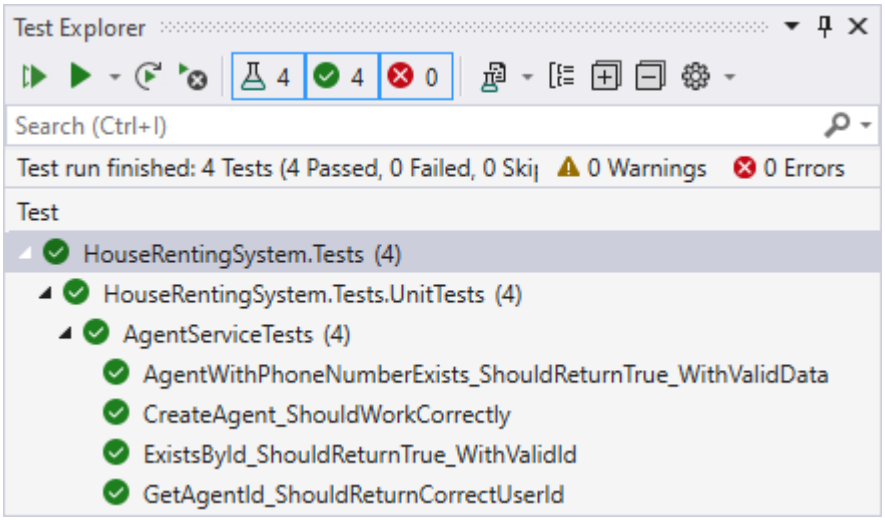
Finally, **make the needed test assertions**. First, make sure that the **agents' count has increased by 1** with the addition of the **new `Agent`**:

```
        // Assert the agents' count has increased by 1
        var agentsCountAfter = _data.Agents.Count();
        Assert.That(agentsCountAfter, Is.EqualTo(agentsCountBefore + 1));
```

Then, **use other service methods** (we assume they work correctly) to **get the newly-added agent** from the database. Then, make sure its **data is correct**:

```
        // Assert a new agent was created in the db with correct data
        var newAgentId = await _agentService.GetAgentId(Agent.UserId);
        var newAgentInDb = await _data.Agents.FindAsync(newAgentId);
        Assert.IsNotNull(newAgentInDb);
        Assert.That(newAgentInDb.UserId, Is.EqualTo(Agent.UserId));
        Assert.That(newAgentInDb.PhoneNumber, Is.EqualTo(Agent.PhoneNumber));
    }
}
```

Run all tests for the **`AgentService` class** together:

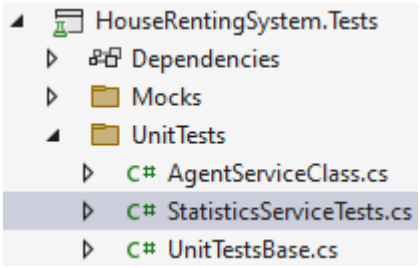


Continue with **writing test for other service classes** or, if you want, you can add some **more tests** for the **`AgentService` class** with **invalid data**.

Step 6: Create StatisticsService Tests

Now let's create **a test** for the only method of the **`StatiticsService` class** – the **`Total()` method**, which **returns the number of all houses and the number of all rented houses** in the database. **Examine the method** before you test it.

Now create the **`StatisticsServiceTests` class** in the "**`HouseRentingSystem.Tests`**" **project**. In it, add a **`SetUp()` method** which should **instantiate the service class**:




```
[TestFixture]
0 references
public class StatisticsServiceTests : UnitTestsBase
{
    private IStatisticService _statisticsService;

    [OneTimeSetUp]
    0 references
    public void SetUp()
        => _statisticsService = new StatisticService(_data);
```

Now **write the test method**, which should **test the `Total()` service method**. First, **invoke the method**, then make sure that it **returns the correct houses and rents count**. The test should be the following:

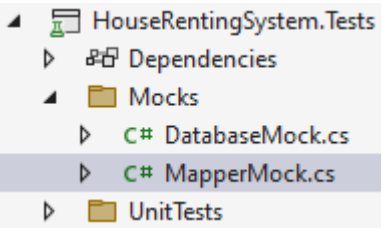
```
    [Test]
    0 references
    public async Task Total_ShouldReturnCorrectCounts()
    {
        // Arrange

        // Act: invoke the service method
        var result = _statisticsService.Total();

        // Assert the returned result is not null
        Assert.IsNotNull(result);

        // Assert the returned houses' count is correct
        var housesCount = _data.Houses.Count();
        Assert.That(result.TotalHouses, Is.EqualTo(housesCount));

        // Assert the returned rents' count is correct
        var rentsCount = _data.Houses.Where(h => h.RenterId != null).Count();
        Assert.That(result.TotalRents, Is.EqualTo(rentsCount));
    }
}
```

Step 7: Create a Mapper Mock

The next service class we want to test is the **`RentService` class**. However, we should first **create a mocked mapper**, as the service class and its method depend on it.

For this reason, let's **create a `MapperMock` class** in the "**`Mocks`**" **folder** of the "**`HouseRentingSystem.Tests`**" **project**. It should have a **single property `Instance`** with a **`getter`**. It should **return a mapper**, which uses the **mappings from the `ServiceMappingProfile` class** in the "**`HouseRentingSystem.Services`**" **project**.

Create and write the class like this:




```
public static class MapperMock
{
    1 reference
    public static IMapper Instance
    {
        get
        {
            var mapperConfiguration = new MapperConfiguration(config =>
            {
                config.AddProfile<ServiceMappingProfile>();
            });

            return new Mapper(mapperConfiguration);
        }
    }
}
```

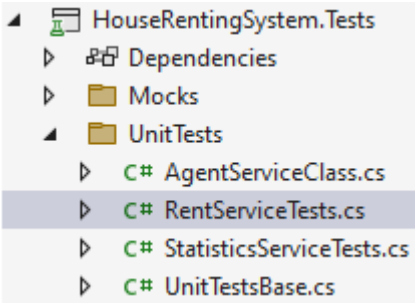
Instantiate the mapper in the **`UnitTestsBase` class (**if you haven't**)** to **use it in child classes** like this:

```
public class UnitTestsBase
{
    protected HouseRentingDbContext _data;
    protected IMapper _mapper;

    [OneTimeSetUp]
    0 references
    public void SetUpBase()
    {
        _data = DatabaseMock.Instance;
        _mapper = MapperMock.Instance;
        SeedDatabase();
    }
```

Now you can **inject the mapper** when **instantiating service classes**.

Step 8: Create RentService Tests

In the **`RentServiceTests` class** we will **write tests for the `RentService` class**. **Examine the services class** and its **methods** for testing. Then, create the **`RentServiceTests` class** and write its **`SetUp()` method** to **instantiate the service class**. Do it as shown below:



→

```
[TestFixture]
0 references
public class RentServiceTests : UnitTestsBase
{
    private IRentService _rentService;

    [OneTimeSetUp]
    0 references
    public void SetUp()
        => _rentService = new RentService(_data, _mapper);
```

Test the **`All()` service method** by invoking it. The method should **return only the rented house** as a **service model** with **correct property values**. The test may look like this:

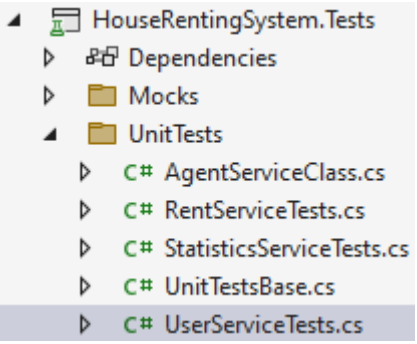
```
    [Test]
    0 references
    public async Task All_ShouldReturnCorrectData()
    {
        // Arrange

        // Act: invoke the service method
        var result = _rentService.All();

        // Assert the result is not null
        Assert.IsNotNull(result);

        // Assert the returned rents' count is correct
        var rentedHousesInDb = _data.Houses
            .Where(h => h.RenterId != null);
        Assert.That(result.ToList().Count(), Is.EqualTo(rentedHousesInDb.Count()));

        // Assert a returned rent's data is correct
        var resultHouse = result.ToList()
            .FindAsync(h => h.HouseTitle == RentedHouse.Title);
        Assert.IsNotNull(resultHouse);
        Assert.AreEqual(Renter.Email, resultHouse.RenterEmail);
        Assert.AreEqual(Renter.FirstName + " " + Renter.LastName,
            resultHouse.RenterFullName);
        Assert.AreEqual(Agent.User.Email, resultHouse.AgentEmail);
        Assert.AreEqual(Agent.User.FirstName + " " + Agent.User.LastName,
            resultHouse.AgentFullName);
    }
}
```

Run all tests together.

Note that our last test is successful because the **mocked mapper** we created is **working correctly**.

Step 9: Create UserService Tests

Tests for the **`UserService`** should be created in the **`UserServiceTests` class**. It should **inherit the base tests class** and **instantiate the service class** in its **`SetUp()` method**:



→

```
[TestFixture]
0 references
public class UserServiceTests : UnitTestsBase
{
    private IUserService _userService;

    [OneTimeSetUp]
    0 references
    public void SetUp()
        => _userService = new ApplicationUserService(_data, _mapper);
```

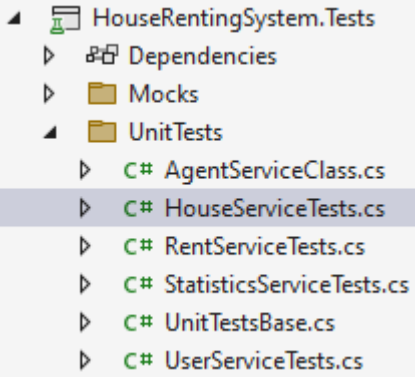
Next, **examine the methods** of the **`UserService` class** and **write tests** for each of them. Tests may be the following:

```
    [Test]
    0 references
    public async Task UserHasRents_ShouldReturnTrue_WithValidData()
    {
        // Arrange

        // Act: invoke the service method with valid renter id
        var result = await _userService.UserHasRents(Renter.Id);

        // Assert the retunred result is true
        Assert.IsTrue(result);
    }

    [Test]
    0 references
    public async Task UserFullName_ShouldReturnCorrectResult()
    {
        // Arrange

        // Act: invoke the service method with valid renter id
        var result = await _userService.UserFullName(Renter.Id);

        // Assert the returned result is correct
        var renterFullName = Renter.FirstName + " " +
            Renter.LastName;
        Assert.That(result, Is.EqualTo(renterFullName));
    }

    [Test]
    0 references
    public async Task All_ShouldReturnCorrectUsersAndAgents()
    {
        // Arrange

        // Act: invoke the service method
        var result = await _userService.All();

        // Assert the returned users' count is correct
        var usersCount = _data.Users.Count();
        var resultUsers = result.ToList();
        Assert.That(resultUsers.Count(), Is.EqualTo(usersCount));

        // Assert the returned agents' count is correct
        var agentsCount = _data.Agents.Count();
        var resultAgents = resultUsers.Where(us => us.PhoneNumber != "");
        Assert.That(resultAgents.Count(), Is.EqualTo(agentsCount));

        // Assert a returned agent data is correct
        var agentUser = resultAgents
            .FirstOrDefault(ag => ag.Email == Agent.User.Email);
        Assert.IsNotNull(agentUser);
        Assert.That(agentUser.PhoneNumber, Is.EqualTo(Agent.PhoneNumber));
    }
}
```

All tests should be **successful**.

Step 10: Create HouseService Tests

Tests for the **`HouseService` class** will be in the **`HouseServiceTests` class**:



The **class's `SetUp()` method** should **instantiate the `UserService` and `HouseService` classes**, as shown below:

```
[TestFixture]
0 references
public class HouseServiceTests : UnitTestsBase
{
    private IUserService userService;
    private IHouseService houseService;

    [OneTimeSetUp]
    0 references
    public void SetUp()
    {
        this.userService = new UserService(this.data, this.mapper);
        this.houseService = new HouseService(this.data, this.userService, this.mapper);
    }
```

Now you should try to **write tests for all service methods by yourself**. Tests are **similar** to the ones we already created for other service classes. **Write all tests by yourself**. However, note that you should **add some new `House` records** to the **database** for **testing the `Edit()`, `Delete()`, `Rent()` and `Leave()` methods**, so that **tests do not depend on each other**. If you have any difficulties, **look at the code for the tests** below:

```
    [Test]
    0 references
    public async Task AllCategoryNames_ShouldReturnCorrectResult()
    {
        // Arrange

        // Act: invoke the service method
        var result = await _houseService.AllCategoriesNames();

        // Assert the returned categories' count is correct
        var dbCategories = _data.Categories;
        Assert.That(result.Count(), Is.EqualTo(dbCategories.Count()));

        // Assert the returned categories' are correct
        var categoryNames = dbCategories.Select(c => c.Name);
        Assert.That(categoryNames.Contains(result.FirstOrDefault()));
    }
```

```
[Test]
0 references
public async Task AllHousesByAgentId_ShouldReturnCorrectHouses()
{
    // Arrange: add a valid agent id to a variable
    var agentId = Agent.Id;

    // Act: invoke the service method with valid agent id
    var result = await _houseService.AllHousesByAgentId(agentId);

    // Assert the returned result is not null
    Assert.IsNotNull(result);

    // Assert the returned houses' count is correct
    var housesInDb = _data.Houses
        .Where(h => h.AgentId == agentId);
    Assert.That(result.Count(), Is.EqualTo(housesInDb.Count()));
}
```

```
[Test]
0 references
public async Task AllHousesByUserId_ShouldReturnCorrectHouses()
{
    // Arrange: add a valid renter id to a variable
    var renterId = Renter.Id;

    // Act: invoke the service method with valid renter id
    var result = await _houseService.AllHousesByUserId(renterId);

    // Assert the returned result is not null
    Assert.IsNotNull(result);

    // Assert the returned houses' count is correct
    var housesInDb = _data.Houses
        .Where(h => h.RenterId == renterId);
    Assert.That(result.Count(), Is.EqualTo(housesInDb.Count()));
}
```

```
[Test]
0 references
public async Task Exists_ShouldReturnCorrectTrue_WithValidId()
{
    // Arrange: get a valid rented house id
    var houseId = RentedHouse.Id;

    // Act: invoke the service method with the valid id
    var result = await _houseService.Exists(houseId);

    // Assert the returned result is true
    Assert.IsTrue(result);
}
```


```
[Test]
0 references
public async Task HouseDetailsById_ShouldReturnCorrectHouseData()
{
    // Arrange: get a valid rented house id
    var houseId = RentedHouse.Id;

    // Act: invoke the service method with the valid id
    var result = await _houseService.HouseDetailsById(houseId);

    // Assert the returned result is not null
    Assert.IsNotNull(result);

    // Assert the returned result data is correct
    var houseInDb = _data.Houses.Find(houseId);
    Assert.That(result.Id, Is.EqualTo(houseInDb.Id));
    Assert.That(result.Title, Is.EqualTo(houseInDb.Title));
}

[Test]
0 references
public async Task AllCategories_ShouldReturnCorrectCategories()
{
    // Arrange

    // Act: invoke the service method
    var result = await _houseService.AllCategories();

    // Assert the returned categories' count is correct
    var dbCategories = _data.Categories;
    Assert.That(result.Count(), Is.EqualTo(dbCategories.Count()));

    // Assert the returned categories are correct
    var categoryNames = dbCategories.Select(c => c.Name);
    Assert.That(categoryNames.Contains(result.FirstOrDefault().Name));
}
```

```
[Test]
0 references
public async Task Create_ShouldCreateHouse()
{
    // Arrange: get the houses current count
    var housesInDbBefore = _data.Houses.Count();

    // Arrange: create a new House variable with needed data
    var newHouse = new House()
    {
        Title = "New House",
        Address = "In a Galaxy far far away...",
        Description = "On a very hot sandy planet, in the outskirts of the capital city",
        ImageUrl = "https://www.pexels.com/photo/house-lights-turned-on-106399/"
    };

    // Act: invoke the service method with neccessary valid data
    var newHouseId = _houseService.Create(newHouse.Title,
        newHouse.Address, newHouse.Description, newHouse.ImageUrl, 2200.00M, 1, Agent.Id);

    // Assert the houses' current count has increased by 1
    var housesInDbAfter = _data.Houses.Count();
    Assert.That(housesInDbAfter, Is.EqualTo(housesInDbBefore + 1));

    // Assert the new house is created with correct data
    var newHouseInDb = _data.Houses.Find(newHouseId);
    Assert.That(newHouseInDb.Title, Is.EqualTo(newHouse.Title));
}
```

```
[Test]
0 references
public async Task HasAgentWithId_ShouldReturnTrue_WithValidId()
{
    // Arrange: get valid rented house's renter and agent ids
    var houseId = RentedHouse.Id;
    var userId = RentedHouse.Agent.User.Id;

    // Act: invoke the service method with valid ids
    var result = await _houseService.HasAgentWithId(houseId, userId);

    // Assert the returned result is true
    Assert.IsTrue(result);
}
```

```
[Test]
0 references
public async Task Edit_ShouldEditHouseCorrectly()
{
    // Arrange: add a new house to the database
    var house = new House()
    {
        Title = "New House for Edit",
        Address = "Sofia",
        Description = "This house is a test house that must be edit
        ImageUrl = "https://www.pexels.com/photo/house-lights-turne
    };

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

    // Arrange: create a variable with the changed address
    var changedAddress = "Sofia, Bulgaria";

    // Act: invoke the method with valid data and changed address
    await _houseService.Edit(house.Id, house.Title, changedAddress,
        house.Description, house.ImageUrl, house.PricePerMonth,
        house.CategoryId);

    // Assert the house data in the database is correct
    var newHouseInDb = await _data.Houses.FindAsync(house.Id);
    Assert.IsNotNull(newHouseInDb);
    Assert.That(newHouseInDb.Title, Is.EqualTo(house.Title));
    Assert.That(newHouseInDb.Address, Is.EqualTo(changedAddress));
}
```

```
[Test]
0 references
public async Task Delete_ShouldDeleteHouseSuccessfully()
{
    // Arrange: add a new house to the database
    var house = new House()
    {
        Title = "New House for delete",
        Address = "Sofia",
        Description = "This house is a test house that must be delete
        ImageUrl = "https://www.pexels.com/photo/house-lights-turned-
    };

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

    // Arrange: get the current houses' count
    var housesCountBefore = _data.Houses.Count();

    // Act: invoke the service method with valid id
    await _houseService.Delete(house.Id);

    // Assert the returned houses' cound has decreased by 1
    var housesCountAfter = _data.Houses.Count();
    Assert.That(housesCountAfter, Is.EqualTo(housesCountBefore - 1));

    // Assert the house is not present in the db
    var houseInDb = await _data.Houses.FindAsync(house.Id);
    Assert.IsNull(houseInDb);
}

[Test]
0 references
public async Task IsRented_ShouldReturnCorrectTrue_WithValidId()
{
    // Arrange: get a valid rented house id
    var houseId = RentedHouse.Id;

    // Act: invoke the service method with valid id
    var result = await _houseService.IsRented(houseId);

    // Assert the returned result is true
    Assert.IsTrue(result);
}
```

```
[Test]
0 references
public async Task Rent_ShouldRentHouseSuccessfully()
{
    // Arrange: add a new house to the db
    var house = new House()
    {
        Title = "New House for rent",
        Address = "A little to the left from the middle of nowhere",
        Description = "This house is a test house that must be rented",
        ImageUrl = "https://www.pexels.com/photo/house-lights-turned-on-106399/"
    };

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

    // Arrange: get a valid renter id
    var renterId = Renter.Id;

    // Act: invoke the service method with valid ids
    await _houseService.Rent(house.Id, renterId);

    // Assert the house has correct data in the db
    var newHouseInDb = _data.Houses.Find(house.Id);
    Assert.IsNotNull(newHouseInDb);
    Assert.That(renterId, Is.EqualTo(house.RenterId));
}

[Test]
0 references
public async Task Rent_ShouldRentHouseSuccessfully()...

[Test]
0 references
public async Task Leave_ShouldRentHouseSuccessfully()
{
    // Arrange: add a new house to the db
    var house = new House()
    {
        Title = "New House for leave",
        RenterId = "TestRenterId",
        Address = "Somewhere in the middle of nowhere",
        Description = "This house is a test house that must be left",
        ImageUrl = "https://www.pexels.com/photo/house-lights-turned-on-106399/"
    };

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

    // Act: invoke the service method with valid id
    await _houseService.Leave(house.Id);

    // Assert the returned result is not null
    Assert.IsNull(house.RenterId);

    // Assert the house has correct data in the db
    var newHouseInDb = await _data.Houses.FindAsync(house.Id);
    Assert.IsNotNull(newHouseInDb);
    Assert.IsNull(newHouseInDb.RenterId);
}
```

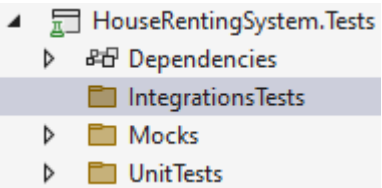
```
    [Test]
    0 references
    public async Task LastThreeHouses_ShouldReturnCorrectHouses()
    {
        // Arrange

        // Act: invoke the service method
        var result = await _houseService.LastThreeHouses();

        // Assert the retuned houses count is correct
        var housesInDb = _data.Houses
            .OrderByDescending(h => h.Id)
            .Take(3);
        Assert.That(result.Count(), Is.EqualTo(housesInDb.Count()));

        // Assert a retuned house's data is correct
        var firstHouseInDb = housesInDb
            .FirstOrDefault();

        var firstResultHouse = result.FirstOrDefault();
        Assert.That(firstResultHouse.Id, Is.EqualTo(firstHouseInDb.Id));
        Assert.That(firstResultHouse.Title, Is.EqualTo(firstHouseInDb.Title));
    }
}
```

We have now **18 test method** for the **`HouseService` class**. **Run them all together** with other tests. They should all be **succesful**.

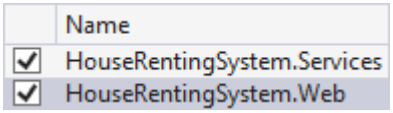
5. Write Integration Tests

In this task, we will show you **examples of integration tests**. In our case, the **integration tests** should **test controllers' functionality**, as **controller methods combine service methods**, tested in unit tests. We will write only **two easy tests**.

First, create a new "**`IntegrationTests`**" **folder** in the "**`HouseRentingSystem.Tests`**" **project**, where we should keep our **integration test classes:**



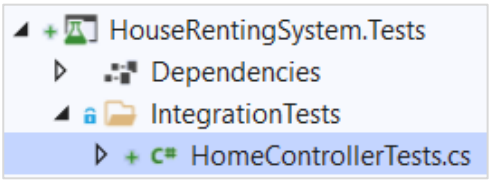
As we will **test the controllers** from the "**`HouseRentingSystem.Web`**" **project**, we should make "**`HouseRentingSystem.Tests`**" **depend** on it:



Step 1: HomeController Test

The **`HomeController`** has **two methods** – **`Index()`** and **`Error()`**. In our case, we will **test only the `Error()` method**, because the **`Index()`** needs a **more complex mocking**.

Create the **HomeControllerTests class**, which should **instantiate the HomeController** in its **SetUp() method**. Pass a **null value**, instead of an **IHouseService instance** to the **controller constructor**, as we **don't use service methods** in the **Error() method**:




```
public class HomeControllerTests
{
    private HomeController homeController;

    [OneTimeSetUp]
    0 references
    public void SetUp()
        => this.homeController = new HomeController(null);
```

Now **create a test** for the **Error() method**, which should check **whether the method returns a view**. Write it like this:

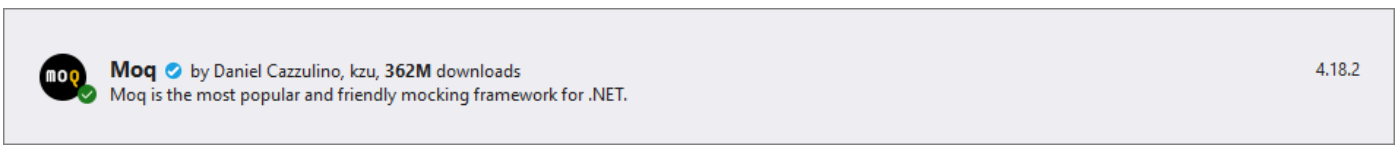
```
[Test]
0 references
public void Error_ShouldReturnCorrectView()
{
    // Arrange: assign a valid status code to a variable
    var statusCode = 500;

    // Act: invoke the controller method with valid data
    var result = this.homeController.Error(statusCode);

    // Assert the returned result is not null
    Assert.IsNotNull(result);

    // Assert the returned result is a view
    var viewResult = result as ViewResult;
    Assert.IsNotNull(viewResult);
}
```

Step 2: StatisticsApiController Test

Now we will **test the GetStatistics() method** of the **StatisticsApiController class**, as the controller **depends** only on the **StatisticsService class**.

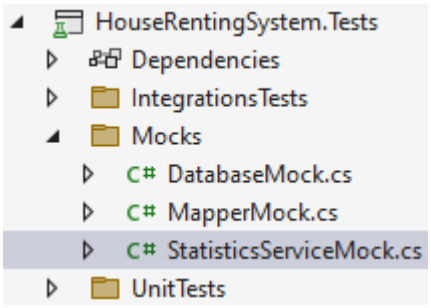
The **StatisticsService class** will be **mocked**. However, it will be **different from the DatabaseMock and MapperMock mock classes**.

For the **StatisticsService class mock**, we will **need the "Moq" NuGet package**. **Install** it:



Now create the **StatisticsServiceMock class** in the "**Mocks**" **folder** of the "**HouseRentingSystem.Tests**" **project**. The **mock class** will have a **property with a getter**, which should **return the mocked service instance**. It should also **set hardcoded models**, which will be returned when a **service method is invoked**.

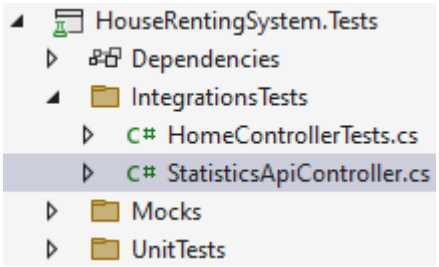
The **StatisticsService class** has a **single method to be mocked** and the **mock class** should look like this:



→

```
public class StatisticsServiceMock
{
    0 references
    public static IStatisticsService Instance
    {
        get
        {
            var statisticsServiceMock = new Mock<IStatisticsService>();

            statisticsServiceMock
                .Setup(s => s.Total())
                .Returns(new StatisticsServiceModel()
                {
                    TotalHouses = 10,
                    TotalRents = 6
                });

            return statisticsServiceMock.Object;
        }
    }
}
```

Now use the **service class mock** in **StatisticsApiController tests**. Create the **StatisticsApiControllerTests class** and **instantiate the controller** with the **mocked service**:



→

```
public class StatisticsApiControllerTests
{
    private StatisticsApiController statisticsController;

    [OneTimeSetUp]
    0 references
    public void SetUp()
        => this.statisticsController =
            new StatisticsApiController(StatisticsServiceMock.Instance);
```

The **test** for the **`GetStatistics()` method** should check whether it **returns a correct result**. Write it like this:

```
[Test]
0 references
public void GetStatistics_ShouldReturnCorrectCounts()
{
    // Arrange

    // Act: invoke the service method
    var result = this.statisticsController.GetStatistics();

    // Assert the returned result counts are correct
    Assert.NotNull(result);
    Assert.That(result.TotalHouses, Is.EqualTo(10));
    Assert.That(result.TotalRents, Is.EqualTo(6));
}
```

Now we have learnt how to **write both unit and integration tests**. However, you can **add more integration tests** and try to **test more complex controller methods**, if you want.

6. Check Code Coverage

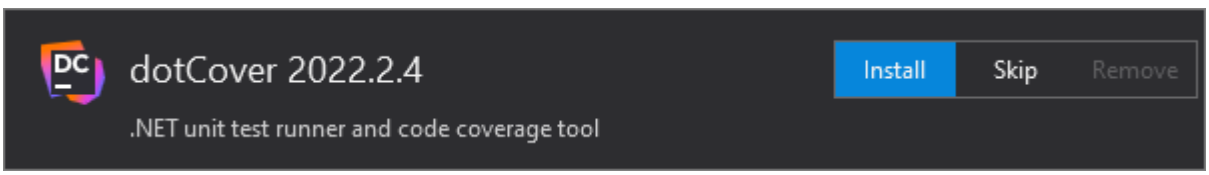
Code coverage is the **percentage of code** which is **covered by automated tests**. We want to check it to understand if the **tests we have are enough** to cover **most of our app functionalities**, if not all.

To **check the code coverage** in **`Visual Studio`**, we should either have **`VS Enterprise`**, or we should use the **`dotCover` tool** of **`JetBrains`**. Let's see how to **use the tool**, as VS Enterprise is not free.

Step 1: Download and Install the dotCover Tool

First, **download `dotCover`** from the **`JetBrains's`** site or from the following **link**: https://www.jetbrains.com/dotcover/. You should just **click on the `[Download]` button** and on the **downloaded `.exe` file** to **open it**:

When you **open the file**, you should **choose to install `dotCover`** on your **current VS version**:



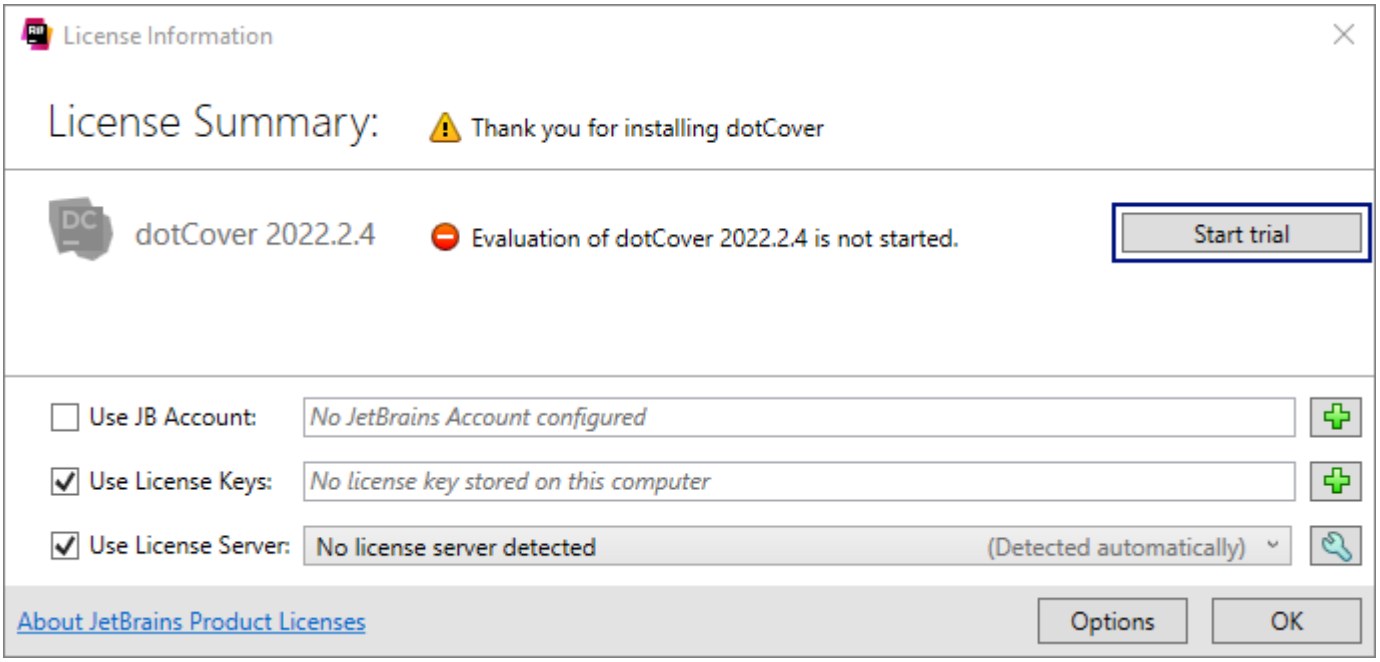
Check the option **`[I have read and accept the license agreement]`** and click on the **`[Next]`** button.

On the next window, you should just **click the `[Install]` button**. Note that you will **also install `ReSharper`**, as **`dotCover` needs it**:

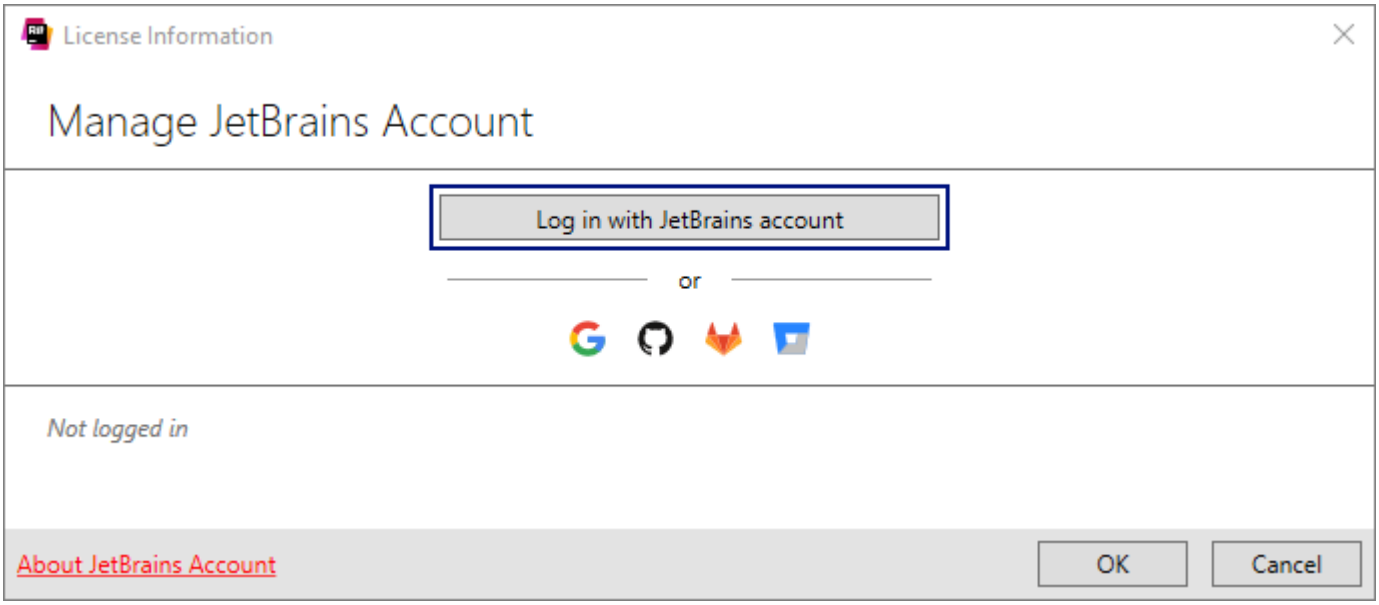
Wait for the installation to finish and **exit the window**. Now **close and open `Visual Studio`** again.

In **VS**, go to **`[Extensions]`** → **`[ReSharper]`** → **`[Start ReSharper]`**.

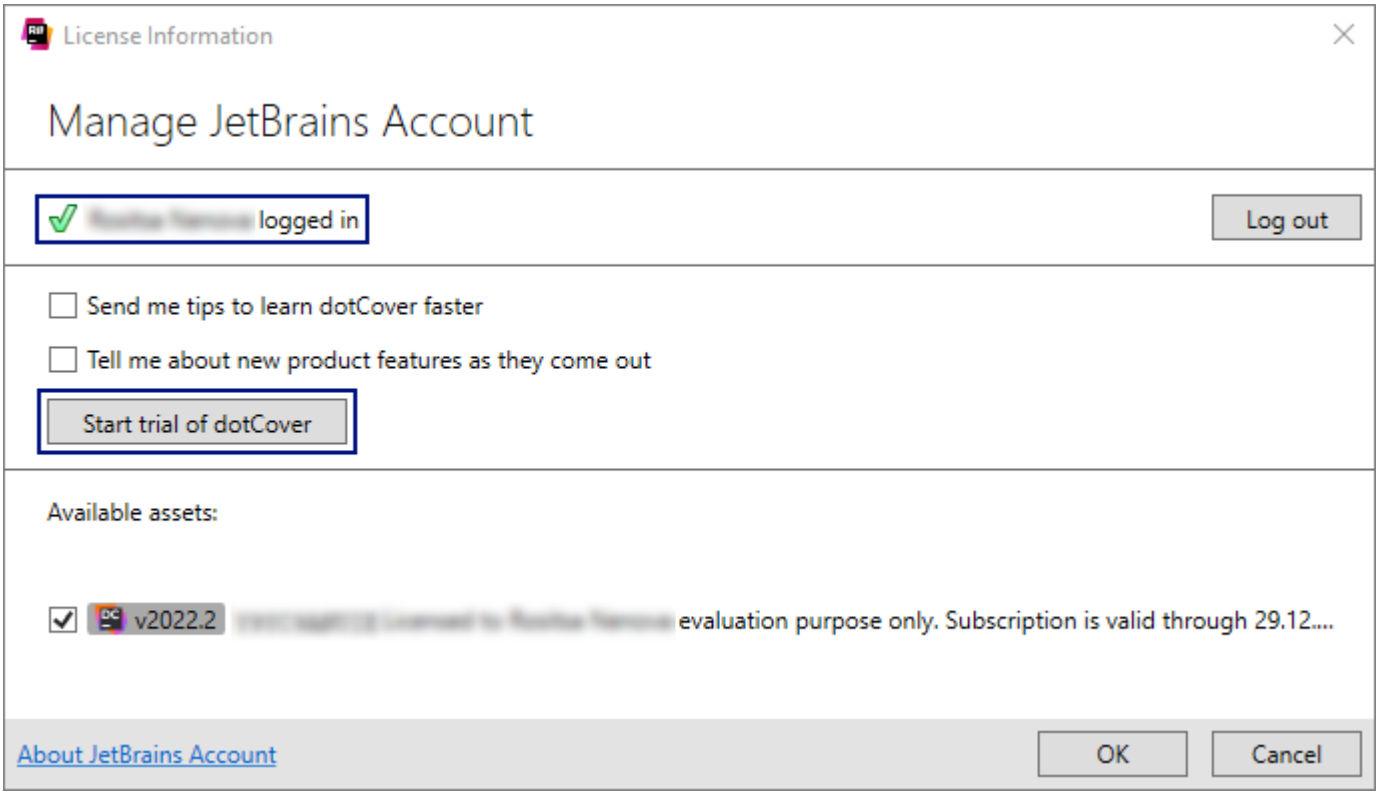
If you **don't see the above buttons**, **wait** for VS to show them or **restart** it. If a **user agreement window** appears, **scroll down** and **click on `[I Accept]`** and **`[OK]` buttons** on the first two windows. On the **next window**, **click on `[Start trial]`** on **`dotCover`** and **proceed**:



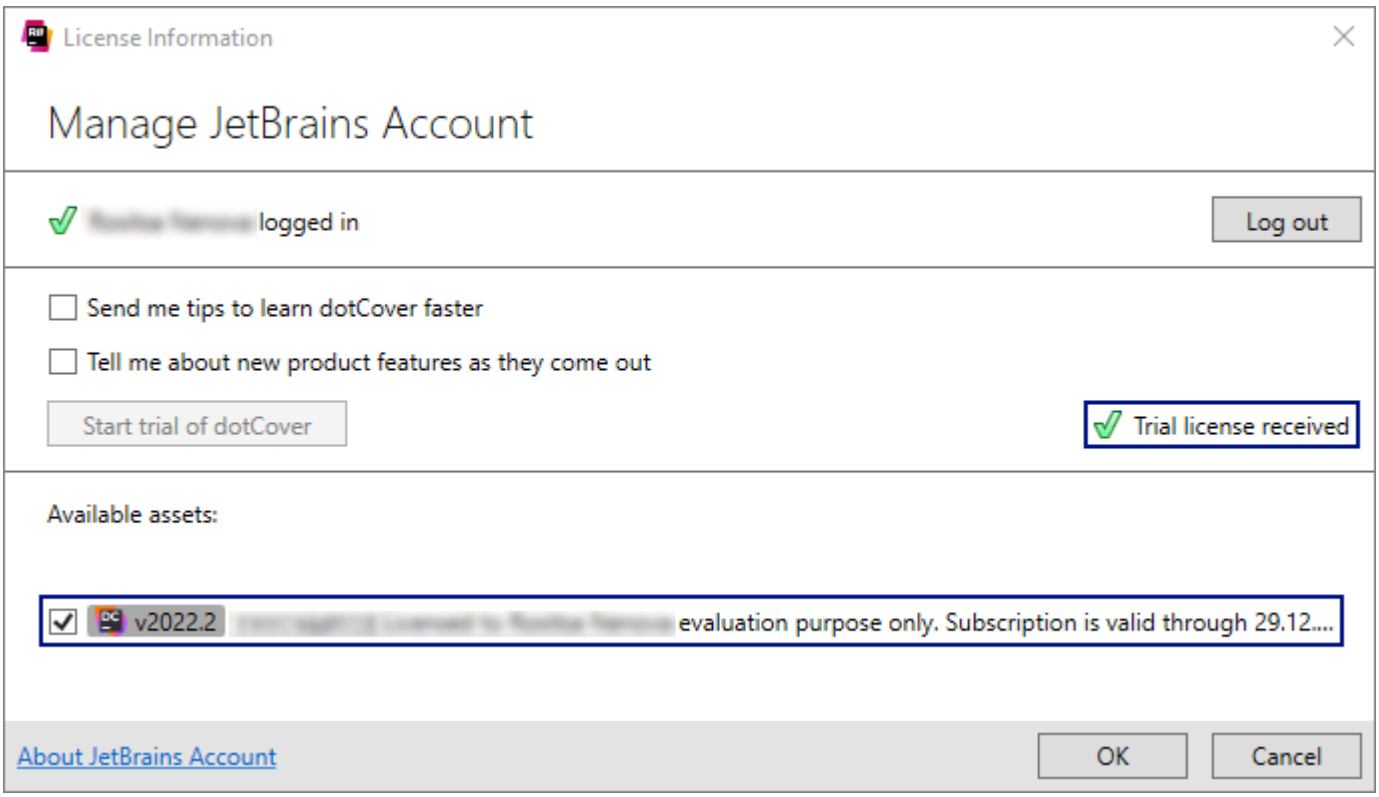
Then, **click on the `[Log in with JetBrains account]` button** and **follow the instructions** to **log in** or **create an account**:



When done, the window should **display that you are logged-in**. Then, **click on `[Start trial of dotCover]`**:



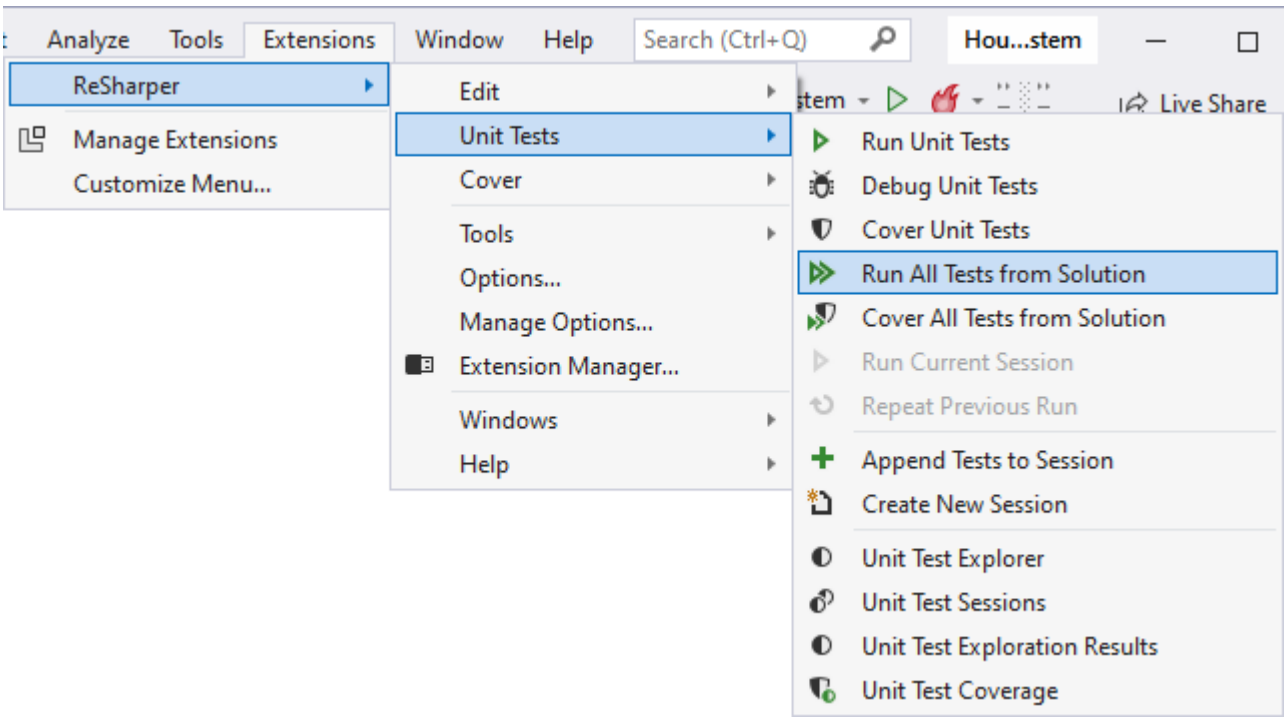
You **trial** should be **started successfully**:

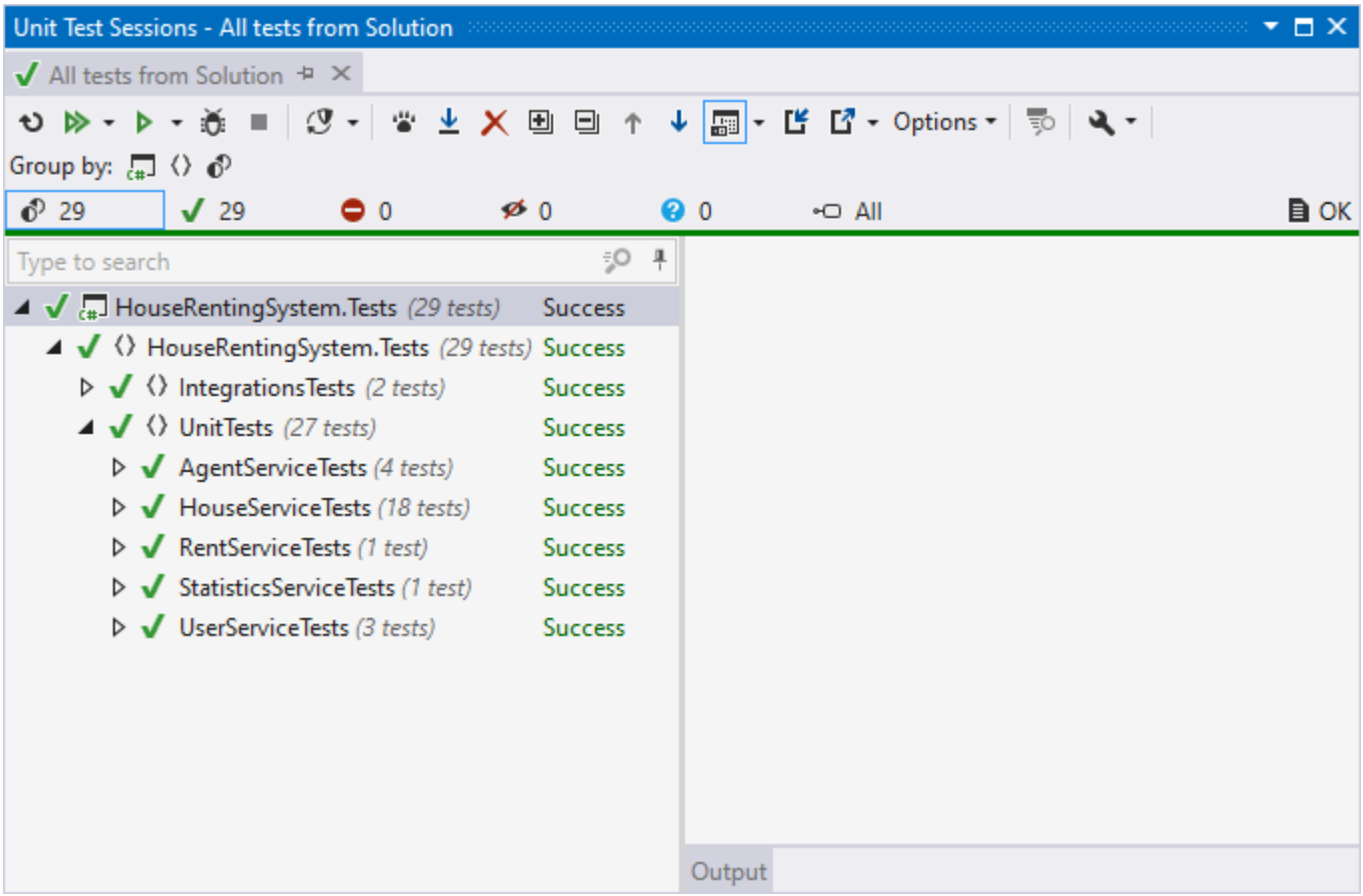


Finally, **click on the `[OK]` button** on this and on the next window.

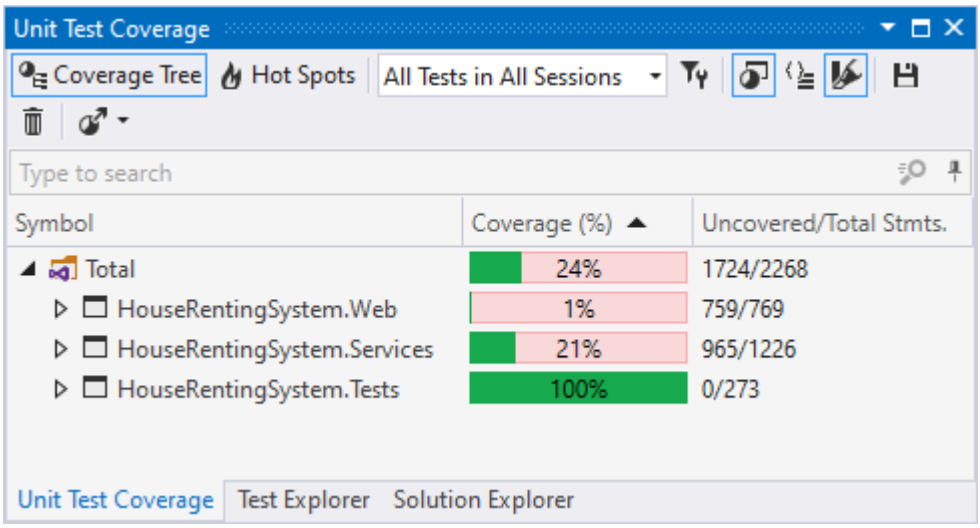
Step 2: Check the Code Coverage

Now you can go **back to `Visual Studio`** and **click on `[Extensions]` → `[ReSharper]` → `[Unit Tests]`** and choose **`[Cover All Tests from Solution]`**. This will **run all tests** on a new window:



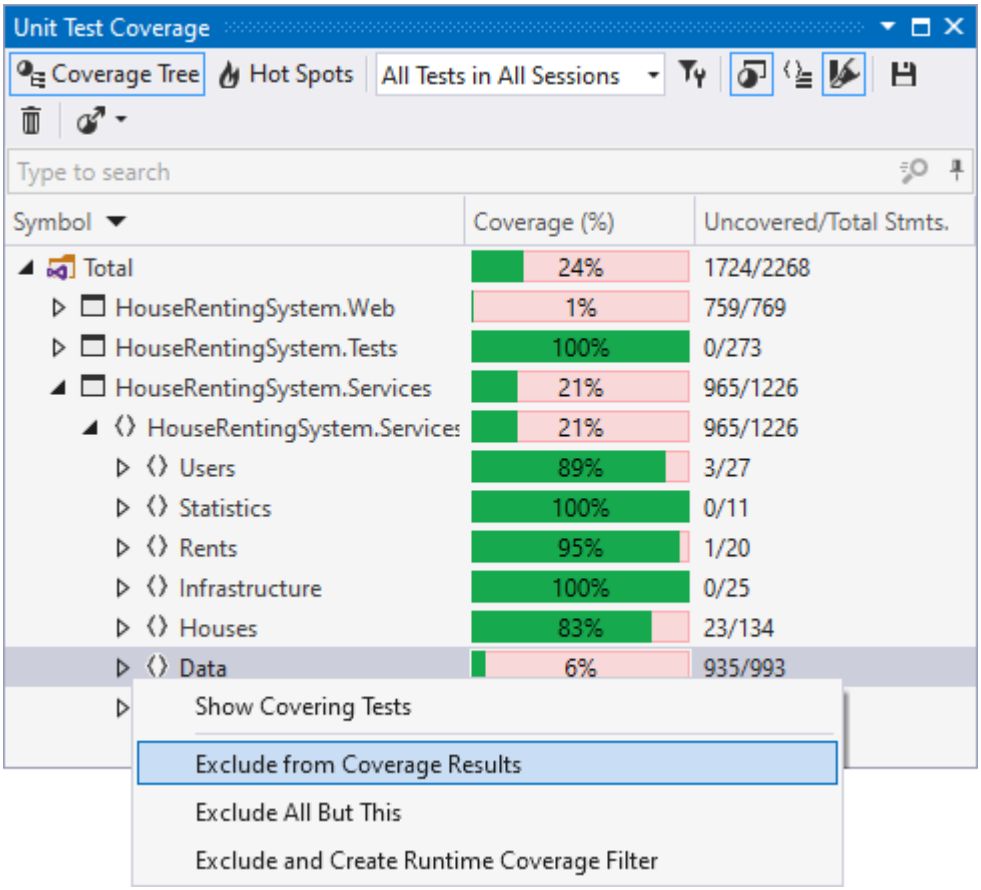


All tests should be **successful**. In order to see the **unit test coverage of tests**, click on **`[Extensions]` → `[ReSharper]` → `[Unit Tests]` → `[Unit Tests Coverage]`**:

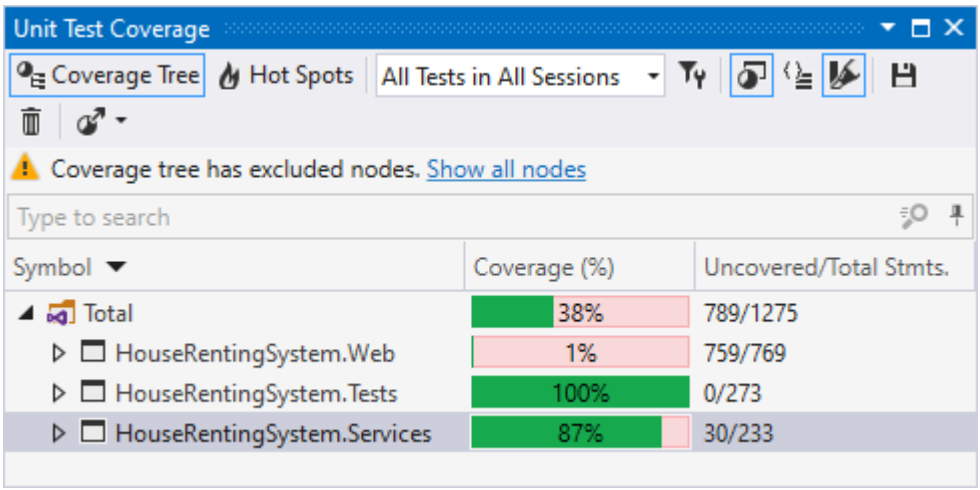


Look at the **coverage** of the "**`HouseRentingSystem.Services`**" **project** with our **services**, as it is the important one for us. You can see that the **coverage percentage** for the "**`Data`**" **folder classes** is small and it **decreases our overall percentage**.

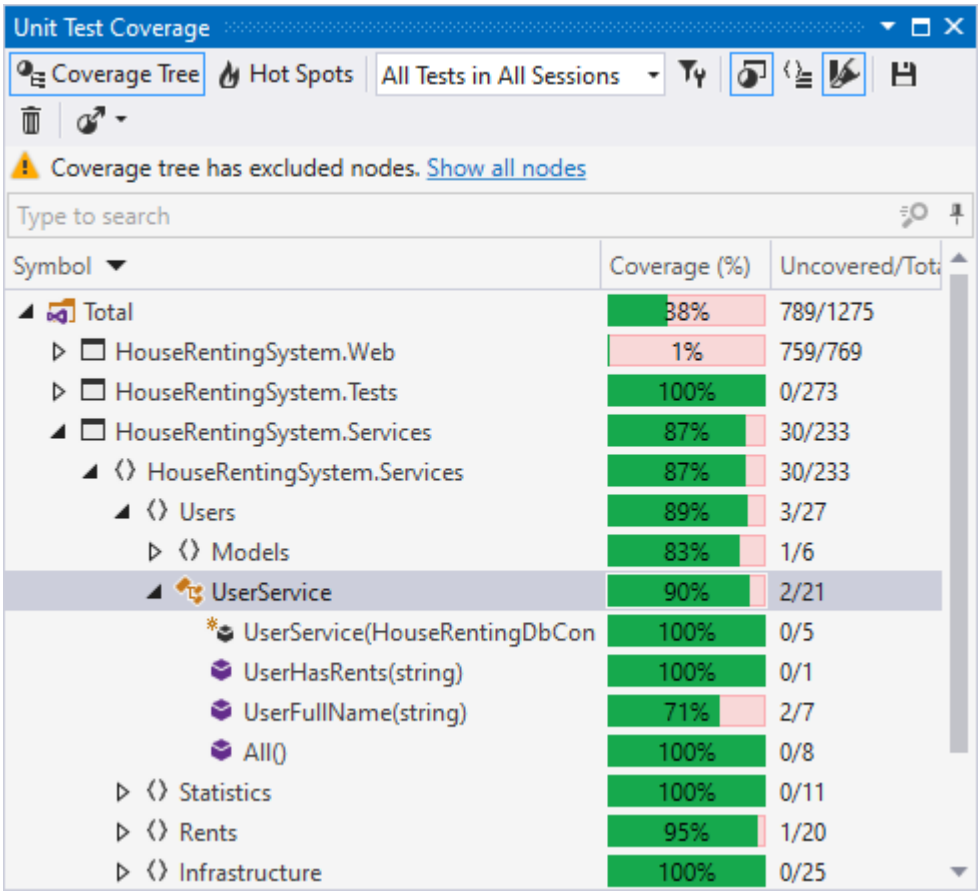
However, our aim was **not to test the data layer** but mostly the **services**, so you can **exclude the** "**`Data`**" **folder test coverage** by **right-clicking** on "**`Data`**" → **`[Exclude from Coverage Results]`**:



Now the **coverage** of the "**`HouseRentingSystem.Services`**" **project** is the one we should be looking for:



If you need to **write more tests** to **reach this percent**, you can examine more carefully which **service classes are tested less** than others and **write tests for them**:



Make sure that you have **written enough tests** for your app. If you **cannot reach the required code coverage** with the **positive test cases** for each service method, you can **add some negative tests cases**.