# Workshop: Users, Security, App Layers and Web API

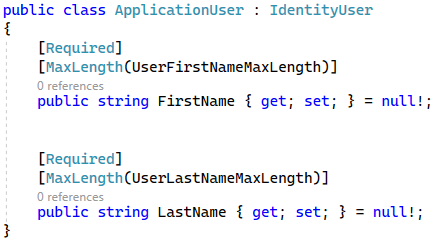
Workshop for the ["ASP.NET Advanced" course @ SoftUni](https://softuni.bg/trainings/4369/asp-net-advanced-february-2024)

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**.

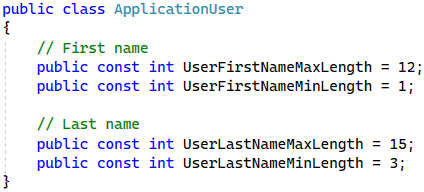
## Add Custom User

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

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

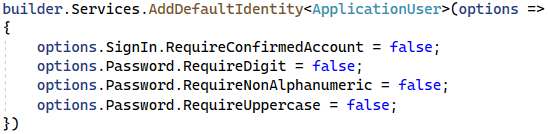


The **maximum and minimum lengths** of properties are part of the DataConstants **class**:



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

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

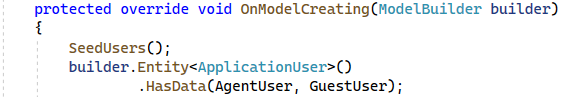


Next, modify the HouseRentingDbContext **class**, which should **inherit the** IdentityDbContext with **ApplicationUser**:



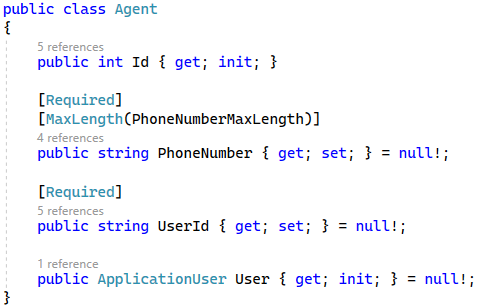
Don't forget to change IdentityUser to User on **all places**. Also, you should **add first and last names** to the **seeded user records**. The rest of the class looks like this when changed:



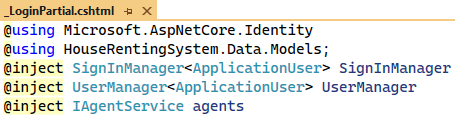




Then, you should **modify the type of the Application**User **property** in the Agent **entity class**:



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



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

|  |
| --- |
| **Add-Migration AddedUserColumns -o Data/Migrations** |

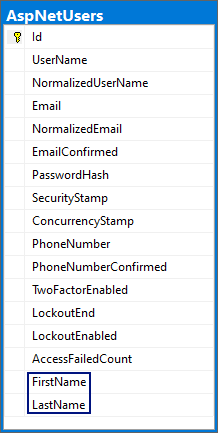
The migration should be created successfully. Look at it – it should have code for **adding** the "FirstName" and "LastName" **columns and the new seeded data** .

Apply the migrating by updating the database with:

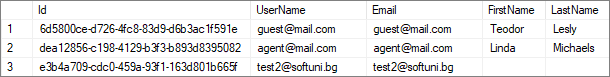
|  |
| --- |
| **Update-Database** |

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

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



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



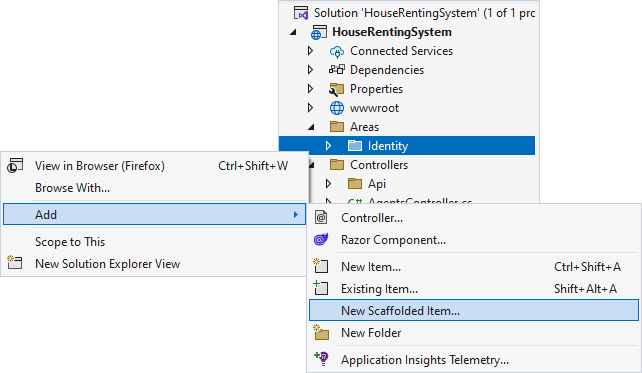
## Modify Registration and Login

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

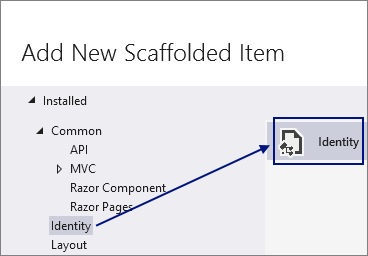
### Scaffold Identity

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

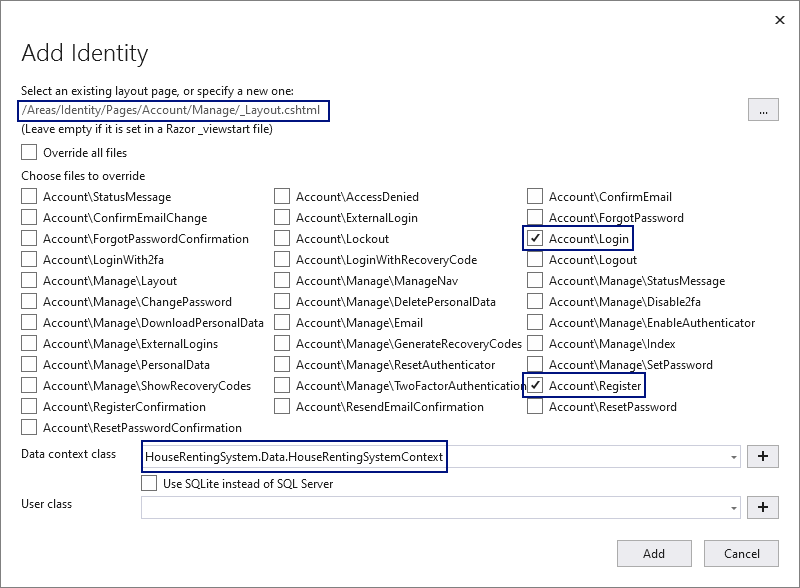
The **scaffolded pages** will be part of the "/Areas/Identity" **folder**. To scaffold, **right-click** on the "Identity" **folder** and choose [Add] 🡪 [New Scaffolded Item…]:



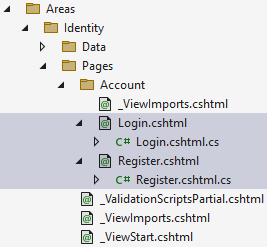
On the next step, go to the [Identity] **tab** and **choose its only option**:



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



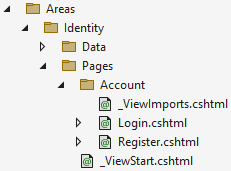
Click the [Add] **button** and examine the **scaffolded pages** in the "/Areas/Identity" **folder**:



Note that the **generated pages** are **Razor pages**. They have two files – one with extension .cshtml and one with .cshtml.cs. The Login.cshtml and Register.cshtml **files** are Razor pages and the logic behind them is in the Login.cshtml.cs and Register.cshtml.cs **files**. The LoginModel and RegisterModel classes hold the **logic** behind the pages. They have OnGetAsync(…) and OnPostAsync(…) **methods**, which are responsible for **handling requests** to the page.

You can now **clear some generated files** and modify others. First, you can delete the new Data folder, as you won't need it. You can also **move the namespaces** from the "\_ViewImports.cshtml" **file** in the "/Areas/Identity/Pages" **folder** to the "\_ViewImports.cshtml" **file** in the "/Areas/Identity/Pages/  
Account" **folder**:

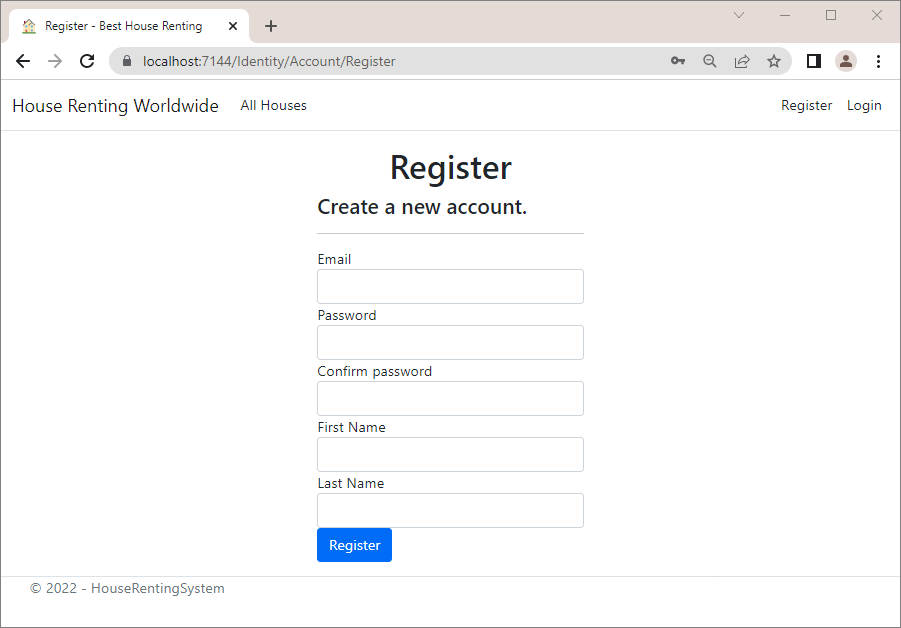
**Delete the classes** in the "Areas" **folder**, which are not in the "Account" **folder**. **Leave** only the \_ViewStart.cshtml **file** – others are **unnecessary**. The left classes should be the following:



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

### Modify the "Register" Page

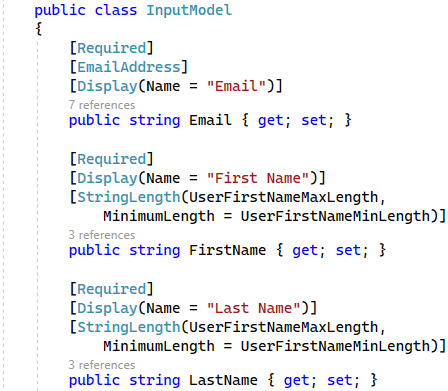
Now we want to **modify** **our** "Register" **page**. It should **not have external logins**, but should have **fields for first** **and last names**. It should look like this:



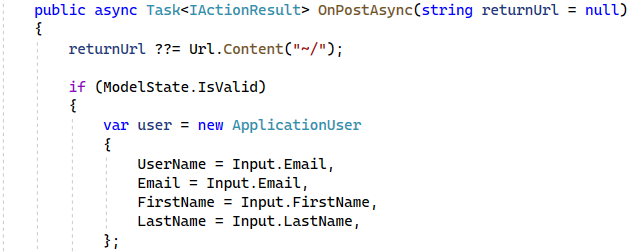
Go to the Register.cshtml **file** to **clear the unnecessary view code**. We want **to remove the section for** **registering with an external provider**. Also, we need to **add text fields** for the **first and last names** of the user. The "**Register.cshtml"** file should look like show below. Also,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



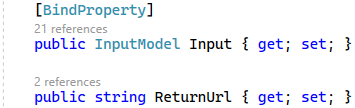
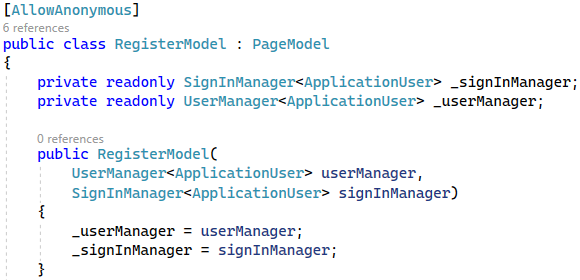
Now we will add the "FirstName" and "LastName" **properties** to the InputModel in the RegisterModel **class**. Open the "Register.cshtml.cs" **file** and do it like this:

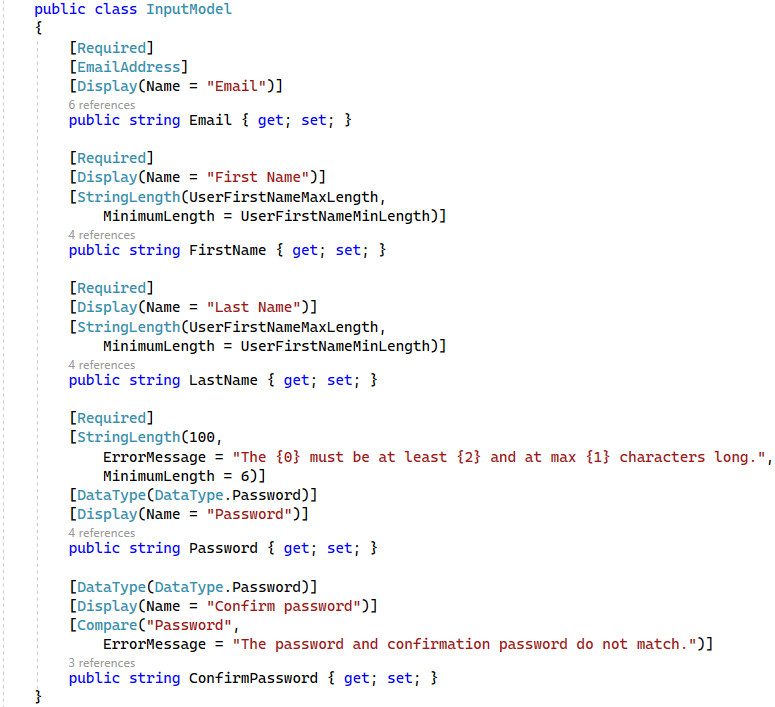


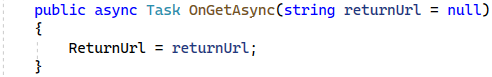
As we **added the properties**, it is important to use them when **creating a user to fill the database columns**. To do this, **modify** the OnPostAsync(…) **method** like this:

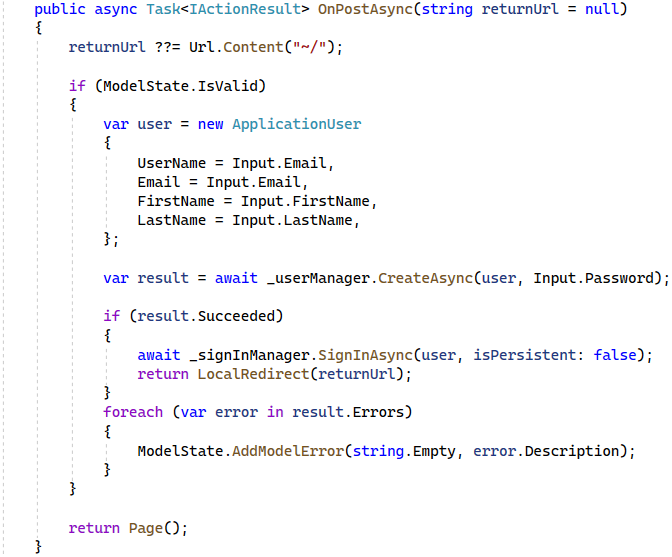


Now it is time to **clear** the RegisterModel **class** from things we won't use like **external logins**, **email sender**, etc. Your class should look like this:









Now **open** the "Register" **page** in the browser. It should look as shown on the beginning of the task. Try to **register** **a new user**. The registration should be **successful** and the user should **appear in the database**. They should also have a **first** and **last names**:



### Modify the "Login" Page

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

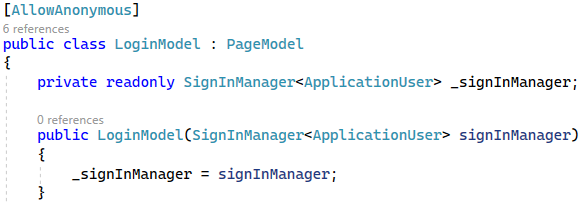
Graphical user interface, application

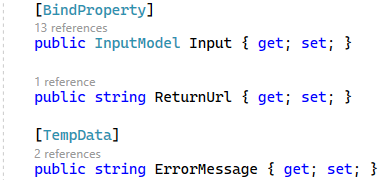
Description automatically generated

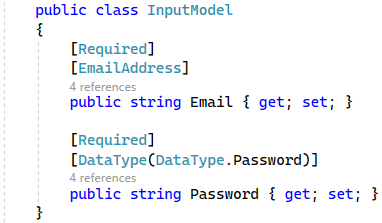
Go to the "Login.cshtml" **view** and **remove links for email confirmation**, **external login** and **forgotten password**. Also, add the needed classes to **center the page content**. The code should look like this:

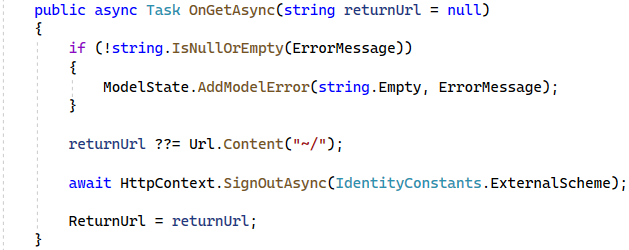


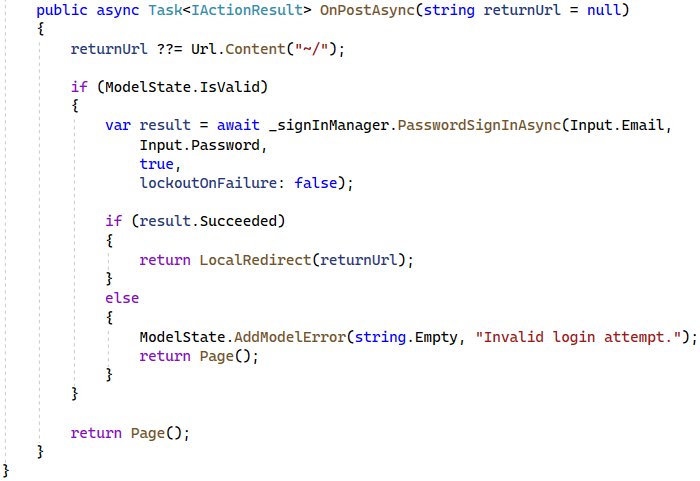
Now **clear the** LoginModel **class** in the "Login.cshtml.cs" **file** as shown below:











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

## Display New User Data

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

### Step 1: Display User Names in the Navigation Bar

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



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



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

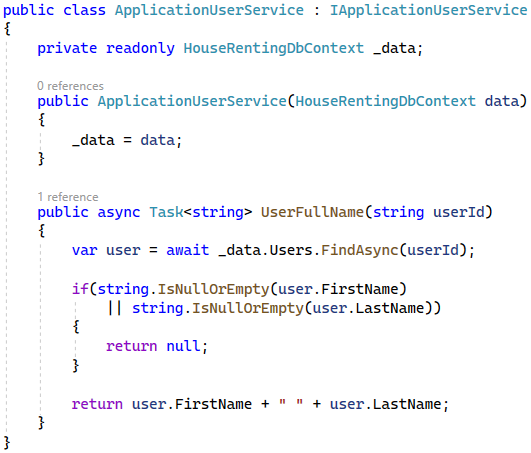
If you want **all of your users to have first and last names** you can **delete and create the database again** or **edit them** **manually** through SSMS (but this is **bad practice** and you should **not** do it in real projects).

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

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



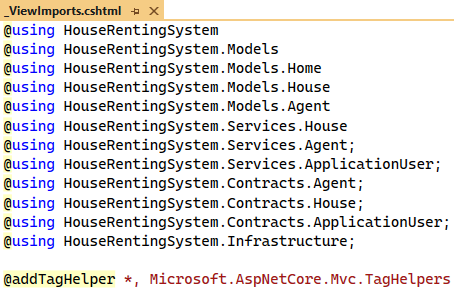
The UserService **class** should **implement the above interface method**. When the **user has names**, they should be returned as a string. When they don't, null **should be returned**. Write the class like this:



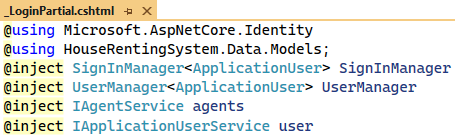
**Add the service** in **Program class** so that you will be able to use it:

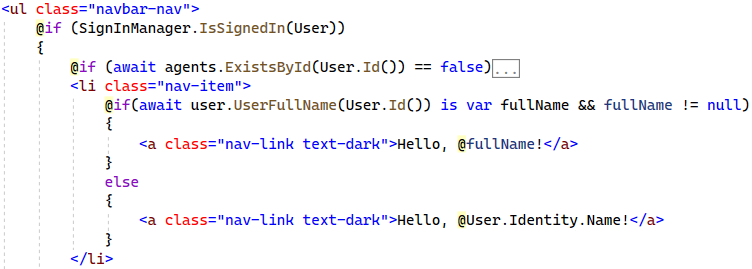


As we want to **inject the service in a view**, we should go to the \_ViewImports.cshtml **file** and **add the class** **namespace**:



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



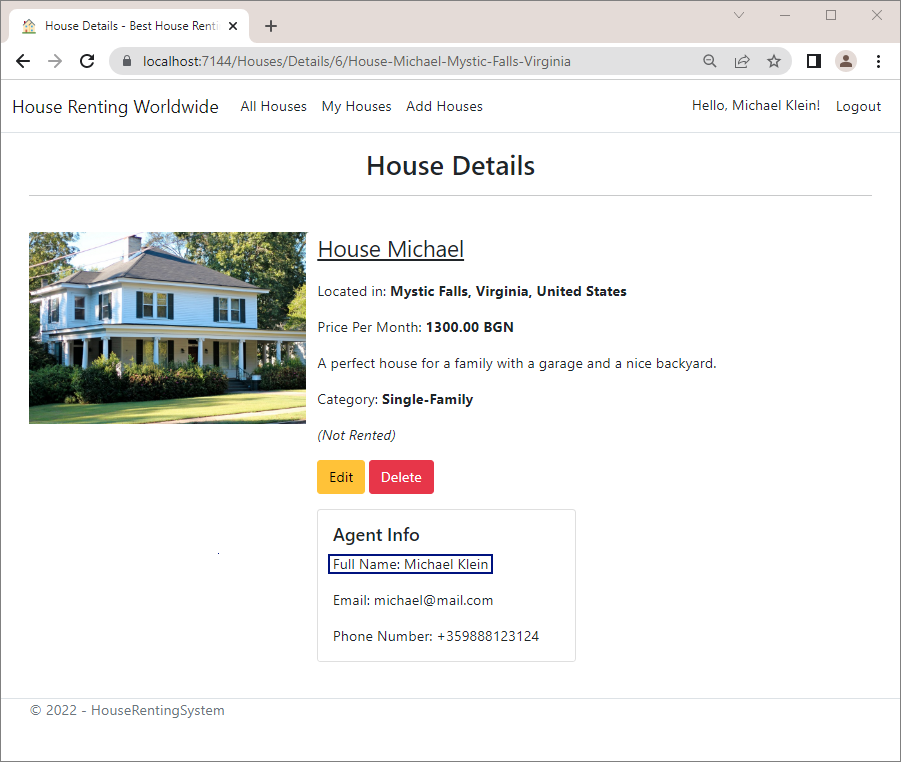


Now **run the app** and make sure that the **navigation bar displays the correct message** – with **first and last names** or with **username**, depending on **the available user data**.

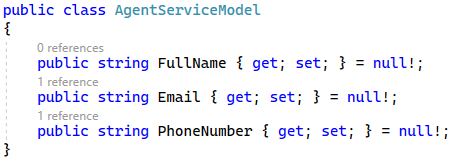
### Step 2: Display Agent Names

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

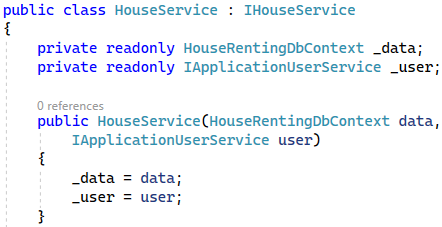
For example, if our **new user** "**Michael Klein**" becomes an **agent** and **adds a house**, the house's "Details" **page** will be the following:



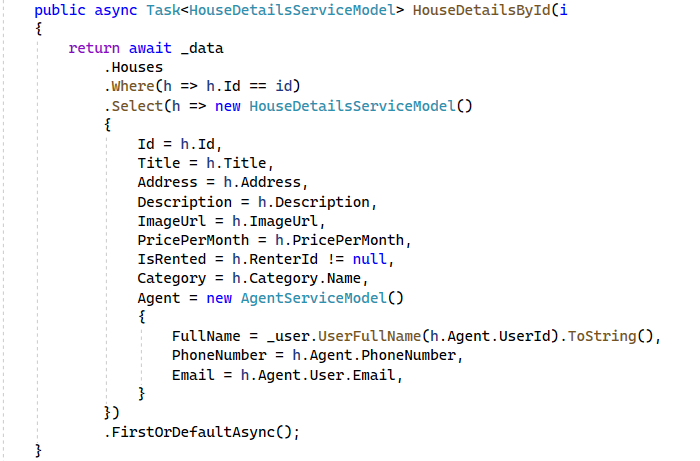
First, to **display the names** we need to add a FullName **property** to the AgentServiceModel:



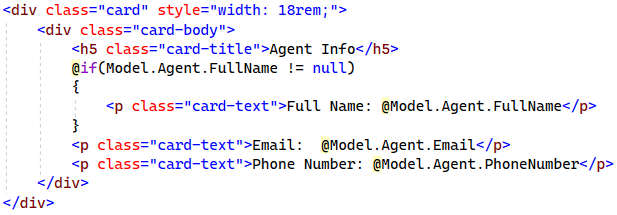
Next, we need to **modify the service model**, which builds the HouseDetailsServiceModel for the "Details" **page**. Use the IUserService to **get the full name of the agent**. To do this, you should first **inject the service** in the HouseService **class** like this:



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



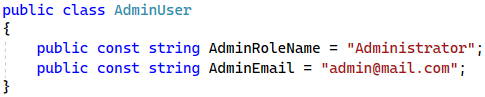
At the end, **modify** the "Details.cshtml" **view** to **display the names** if the FullName **property is not** NULL:



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

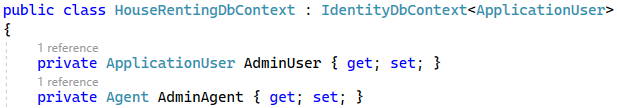
## Seed Administrator

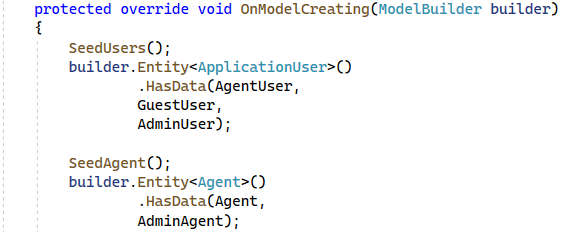
In this task we will see how to **seed an administrator** in our database. Before that, however, let's **create a class with a constant** for the **administrator role name and email**, as we are going to need it. Create the AdminUser **class** and add the **constants** like this:



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

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





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

|  |
| --- |
| AdminUser = new ApplicationUser()  {  Id = "bcb4f072-ecca-43c9-ab26-c060c6f364e4",  Email = AdminEmail,  NormalizedEmail = AdminEmail,  UserName = AdminEmail,  NormalizedUserName = AdminEmail,  FirstName = "Great",  LastName = "Admin"  };  AdminUser.PasswordHash = hasher.HashPassword(AgentUser, "admin123"); |

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

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

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

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

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

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

|  |
| --- |
| Update-Database |



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

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



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

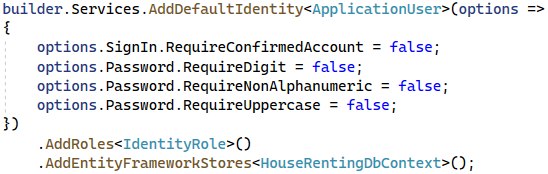


Now look at the "Agents" **table**. There should be the **new agent**, who is also our Admin:



## Add Role for Admin

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



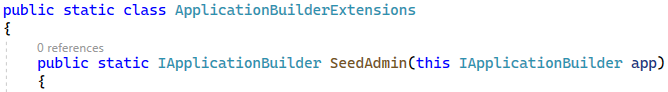
Our **roles** will be of type IdentityRole, which is the **default for ASP.NET Core**.

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

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



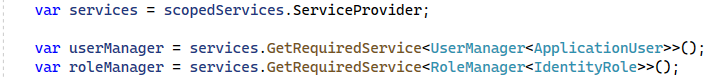
The ApplicationBuilderExtensions **class** will have a single method for **seeding an administrator**. To **extend** **the** IApplicationBuilder, the method should **accept it as a parameter**, **modify it** and then **return it**:



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



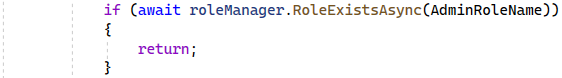
Then, **get your services** through the ServiceProvider like this:



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



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



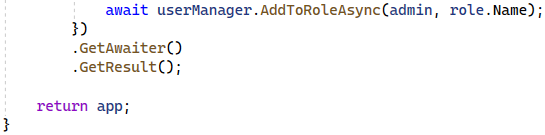
It the **role does not exist**, create it as an IdentityRole:



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



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



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



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

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



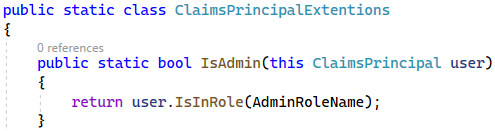
## Modify Admin Access

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

* Admin can **add houses** and **edit** and **delete** not only their houses, but these of other users, too
* Admin can **rent houses**, which are **not rented already**
* Admin can **leave a house**, which is **rented only by them**. They cannot leave houses, which are rented by another user
* Admin is already an agent, so they **cannot become agents** again

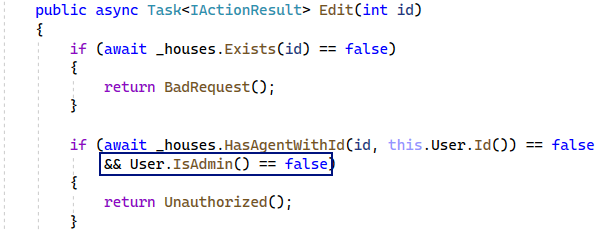
In addition, the "My Houses" **page** will **show the created houses** of the Admin for now. Later, we will create a **separate page** for the Admin to see the **houses they added** and those that **they rented**.

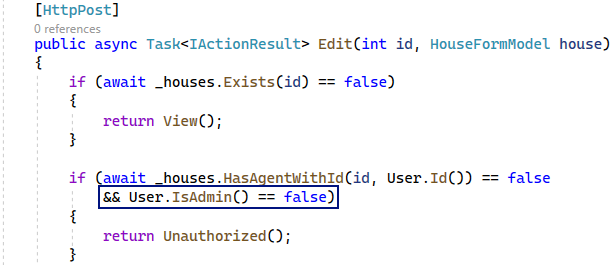
To begin with, we need to **create a method**, which **returns a** bool **whether the current user is the** Admin **or not**, so that we can make **validations**. To do this, we need to **check the role of the current user**. Go to the ClaimsPrincipalExtensions **class** and **add the following method**,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

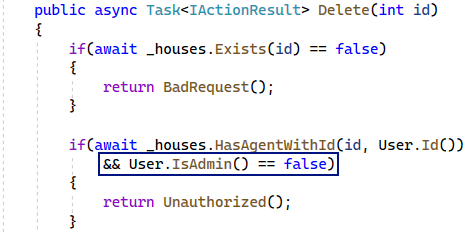


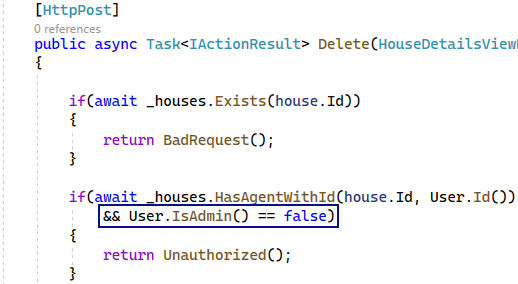
Now use it in the HouseController to **give special access** to pages when the **current user is the** Admin. As the Admin **is an agent**, we do not need to make modifications to the functionality for adding a house (it is already working). However, we should **modify the** "edit" and "delete" **functionalities**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



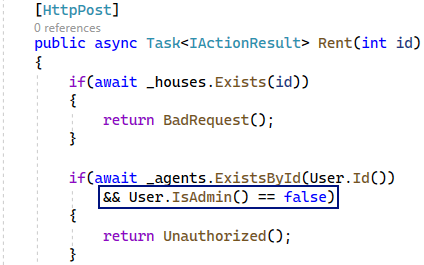








**Modify** the Rent() **method**, too. The Admin should be able to **rent free houses**. Do it like this:



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

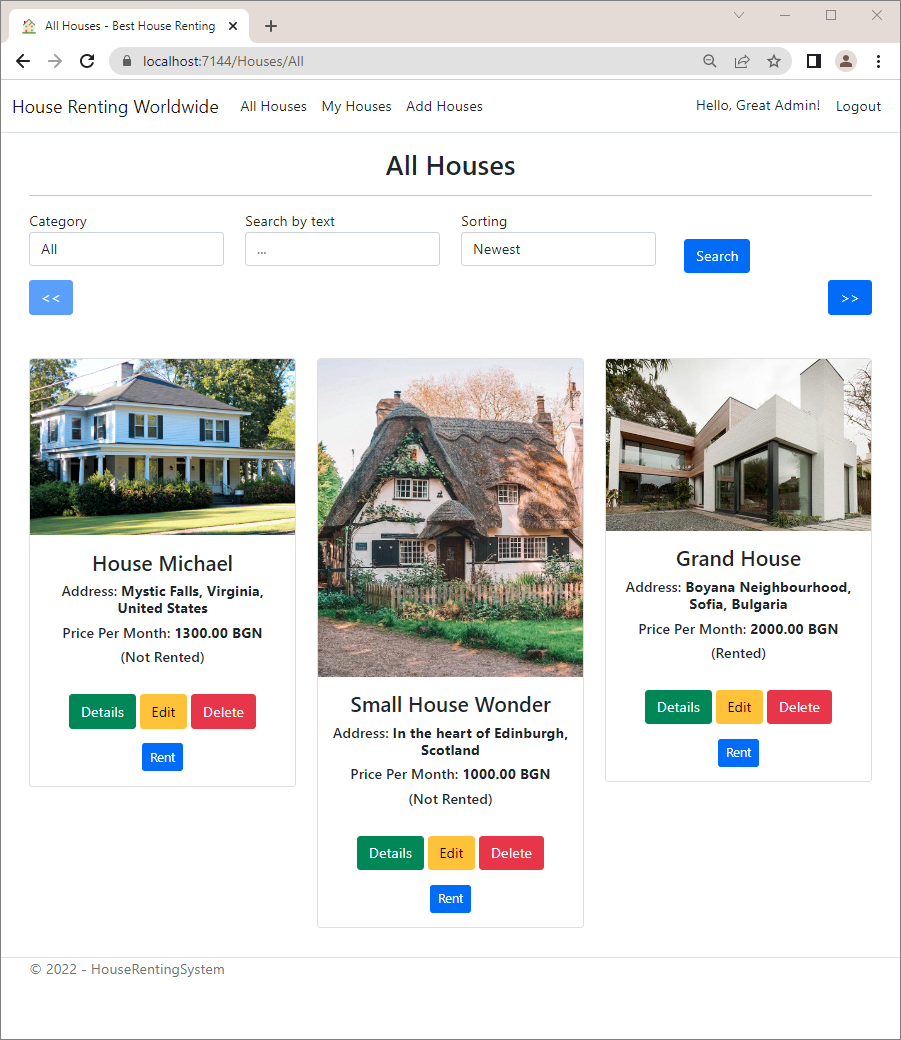


Now **modify** **the** Details.cshtml **view** in the same way:

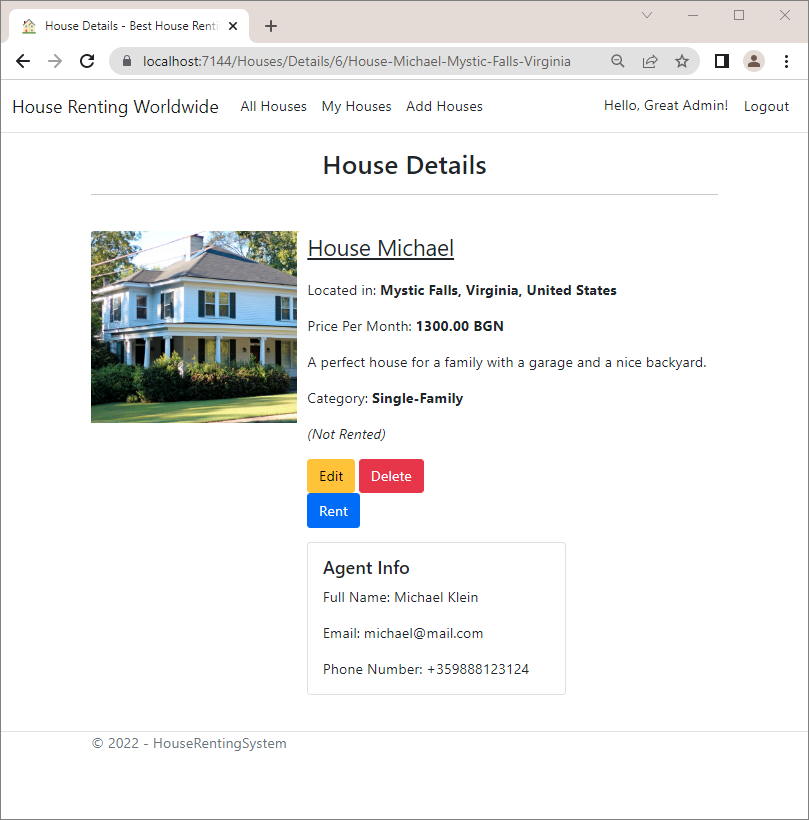


Be **careful with the** **use of brackets** on the above screenshots!

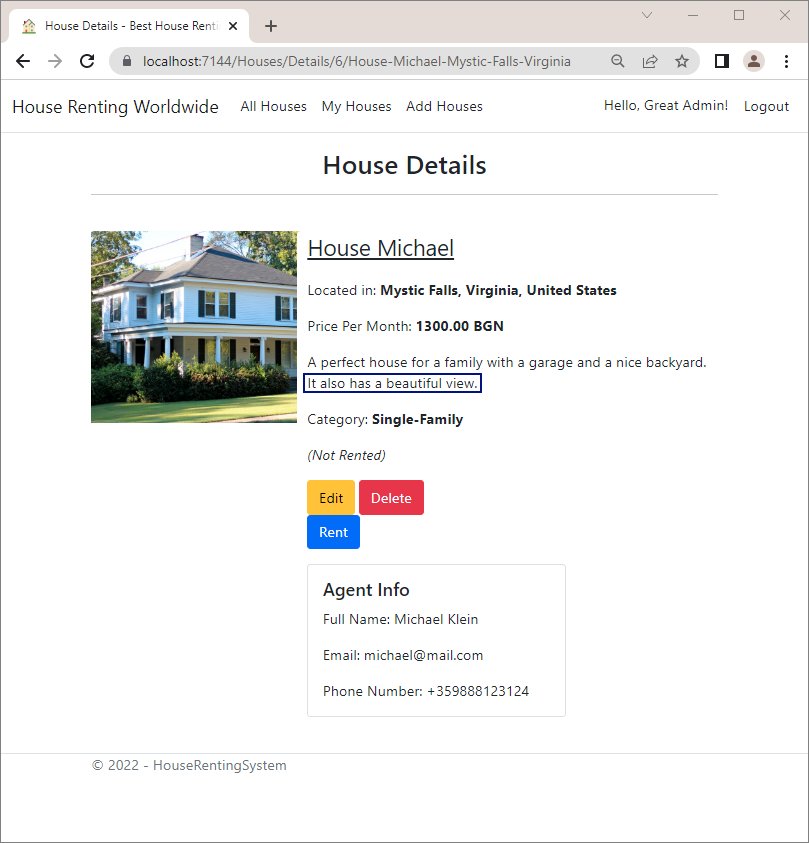
**Run the app** and **log in with the** Admin **credentials**. Go to the "All Houses" and you should see the [Details], [Edit] and [Delete] **buttons** on **each house**. You should also see the [Rent] **button** on houses, which are **not** **rented**:



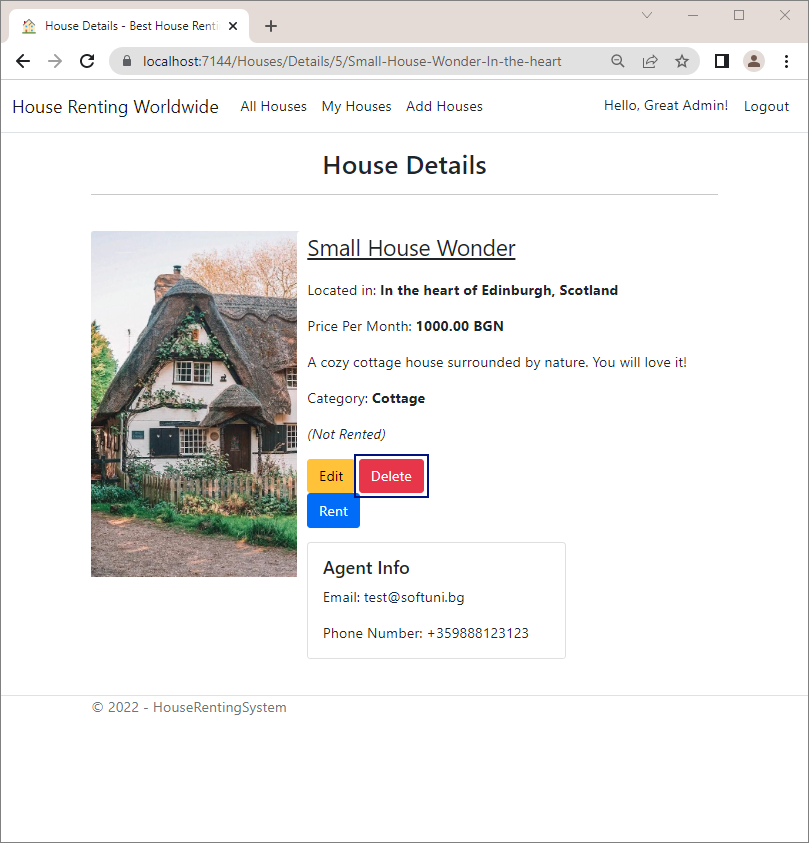
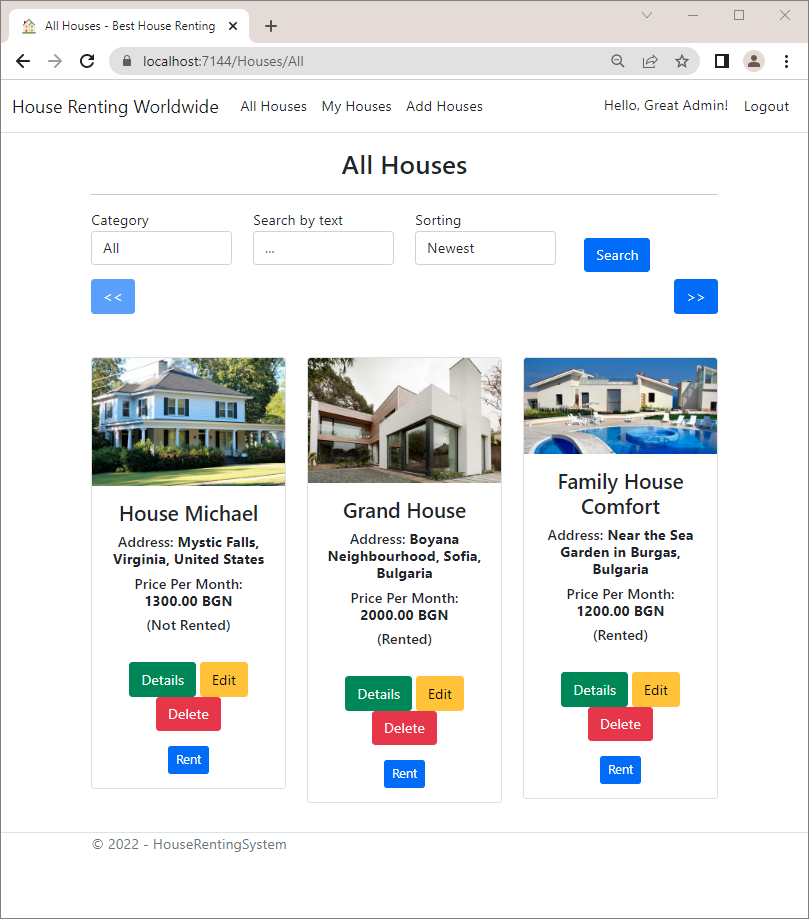
Look at the "Details" **page** of **any house** and make sure the **buttons are present again**:



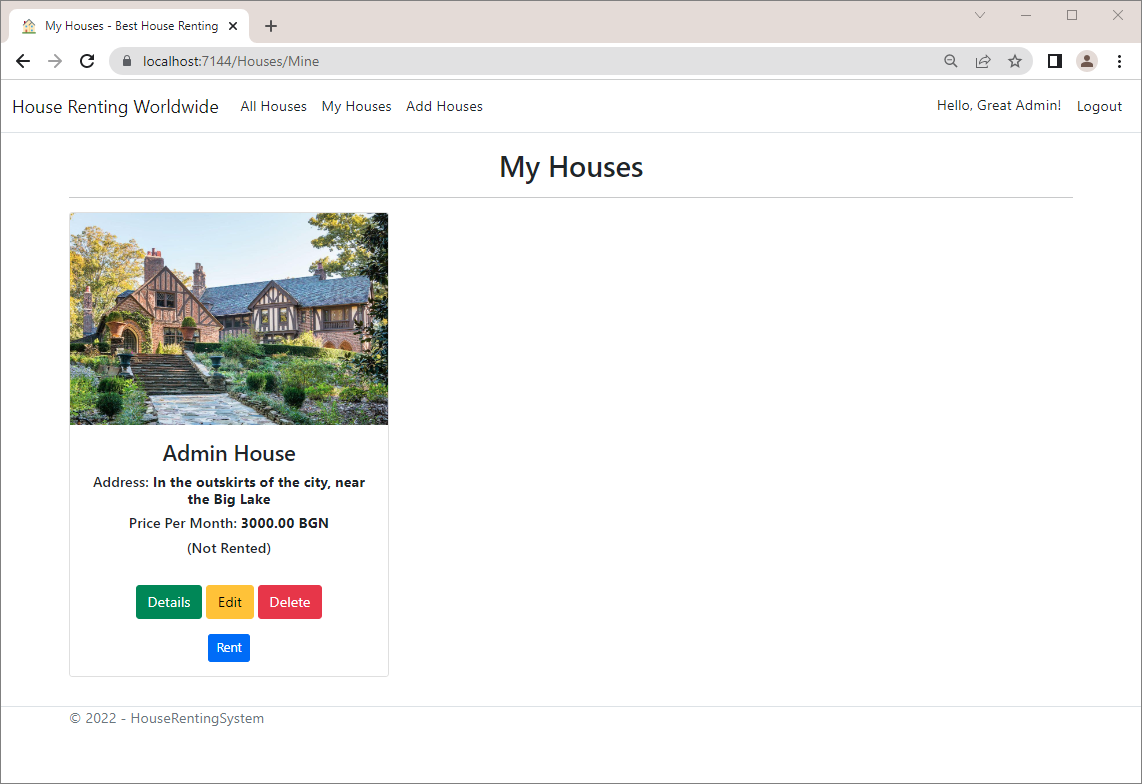
**Try to edit a house** of **another user**. It should be successful:



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

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



## Separate to Multiple Projects

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

In this case, we will have the following projects:

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

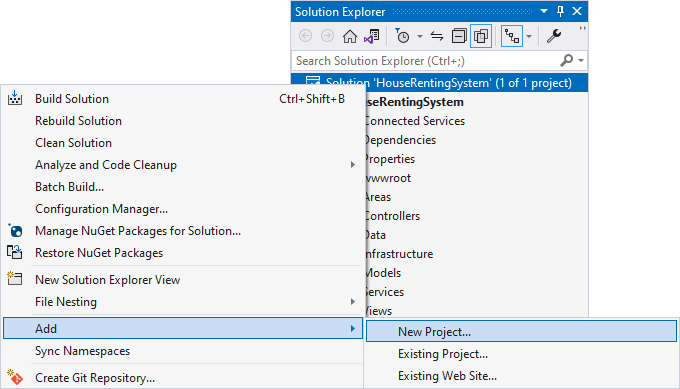
When done, the solution will look like this:

Graphical user interface, application

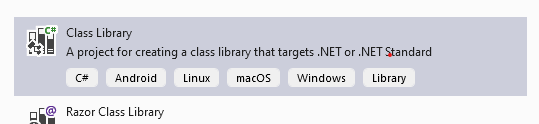
Description automatically generated

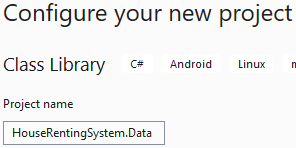
### Step 1: Create a Services + Data Projects

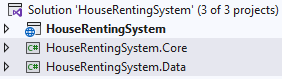
First, we will separate our **services and** **data layers** from the main project, as they do **not have dependencies** on other classes, but depend only on each other. We will move them to **class library** projects.  
To **create a class library**, **right-click** on the "HouseRentingSystem" **solution**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



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



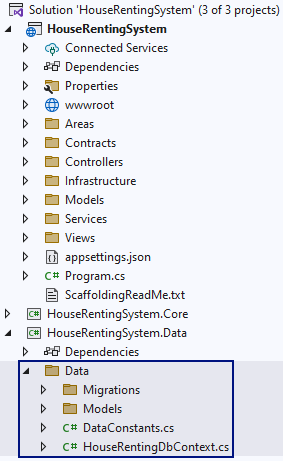




The **projects are created** and you can **delete the** Class1.cs **files**, as we won't need them.

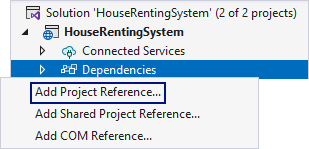
#### Move the Data Layer Classes

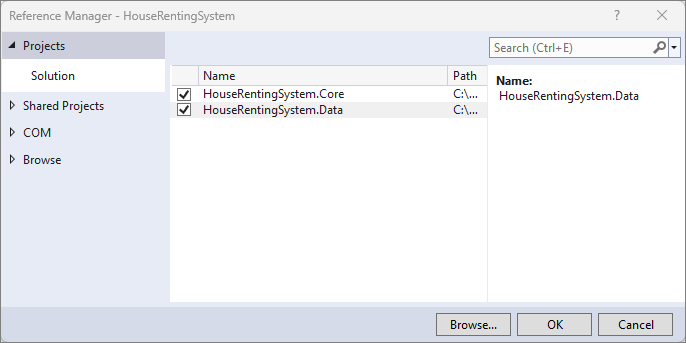
Now let's **move the** "Data" **folder** with its classes from the "HouseRentingSystem" **project** to the "HouseRentingSystem.Data" **project**:

 🡪 

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

First, let's make the "HouseRentingSystem" **project** **depend** on the new "HouseRentingSystem.Core" and "**HouseRentingSystem.Data**"**project**, as it **needs the data classes**. To do this, **right-click** on [Dependencies] 🡪 [Add Project Reference] and **select** **the** "HouseRentingSystem.Core" and "HouseRentingSystem.Data" **projects** like this:

 🡪



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

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

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



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

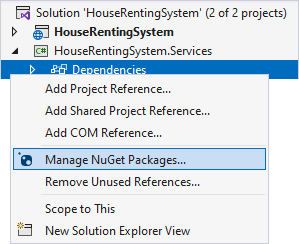


The "Migrations" **folder classes** should be part of this **namespace**:

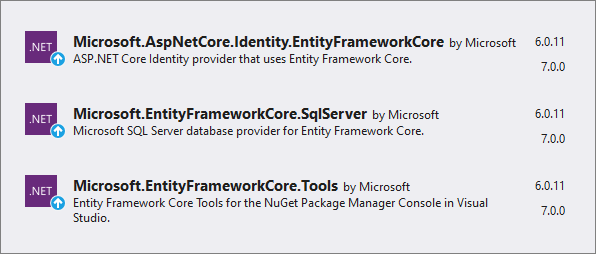


You should also **change the references to the classes** with the modified namespace. When you **build the project**, you should have **no errors**, **related to the namespaces**.

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



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



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

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

#### Move the Service Layer Classes

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

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

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

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

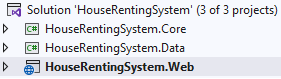
Note that all **errors** that have left are because of the HouseSorting **class**, because it is used in both projects, but the **new project doesn't depend on the old one**, as then we will have a **circular dependency**, which is **not allowed**. For this reason, let's **move the** HouseSorting **class**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

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

### Step 2: Rename the Web Project

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

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



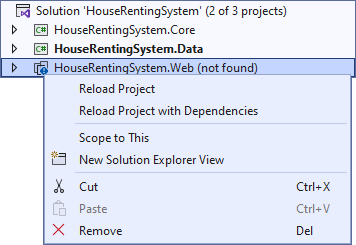
Then, **change the namespace** **of all classes i**n the project to be relevant to the new project name.

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

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

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

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



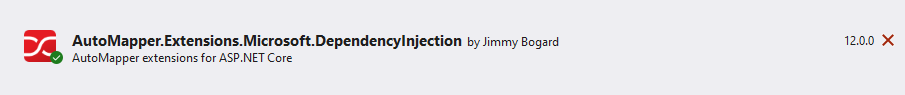
Right-click on the HouseRentinSystem solution and choose **[Add]** → **[Existing Project]**.

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

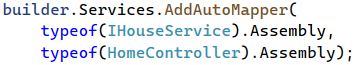
## Add AutoMapper

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

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



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



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

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



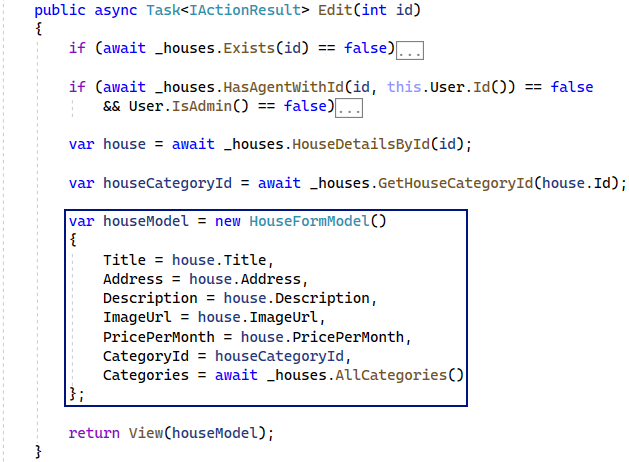
The ControllerMappingProfile **class** should be in a **folder** "Infrastructure" in the "HouseRentingSystem.Web" **project**:

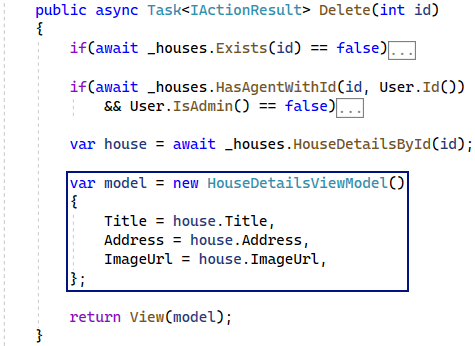


### Step 1: Mapping in Controller Methods

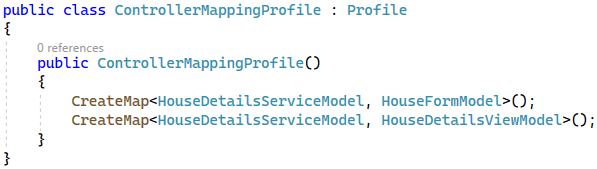
Go to the "HouseRentingSystem.Web" **project** and **examine the controller methods** for manual mapping between objects. In our case, we have such **mappings in the** Edit(int id) **and** Delete(int id) **methods** of the HouseController:



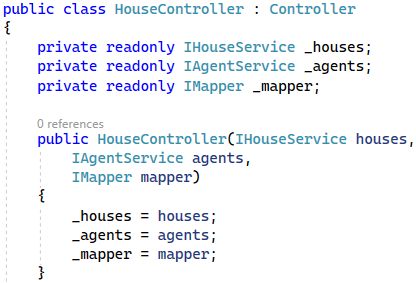




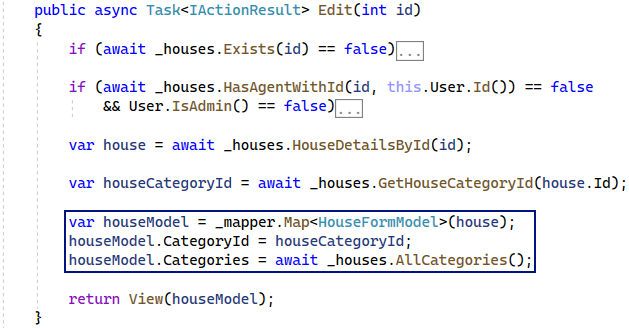
In the above methods, we need to **map the** HouseDetailsService **model to** HouseFormModel and HouseDetailsViewModel. Go to the ControllerMappingProfile **class** and **create the mappings** in the **constructor** like this:

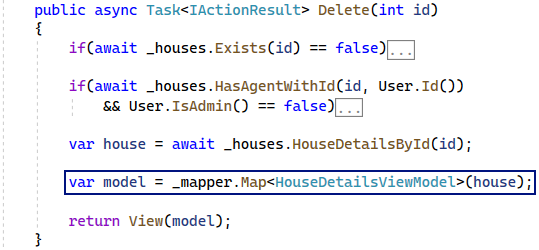


Now go back to the HousesController **class** and **inject the mapper** through the **constructor** and **assign it to a** **field**:

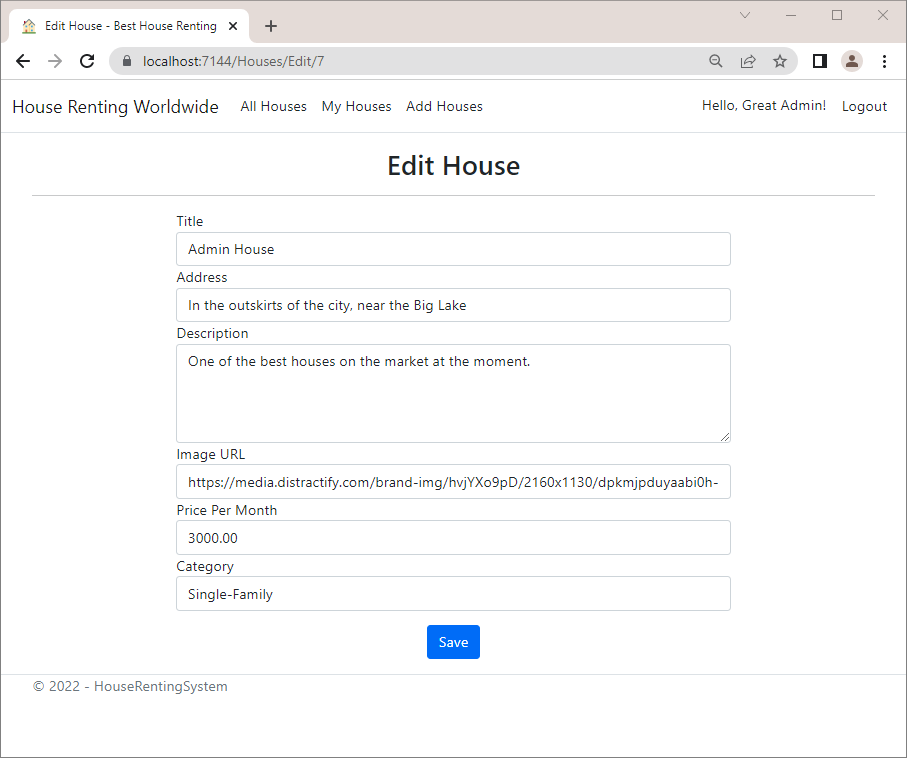


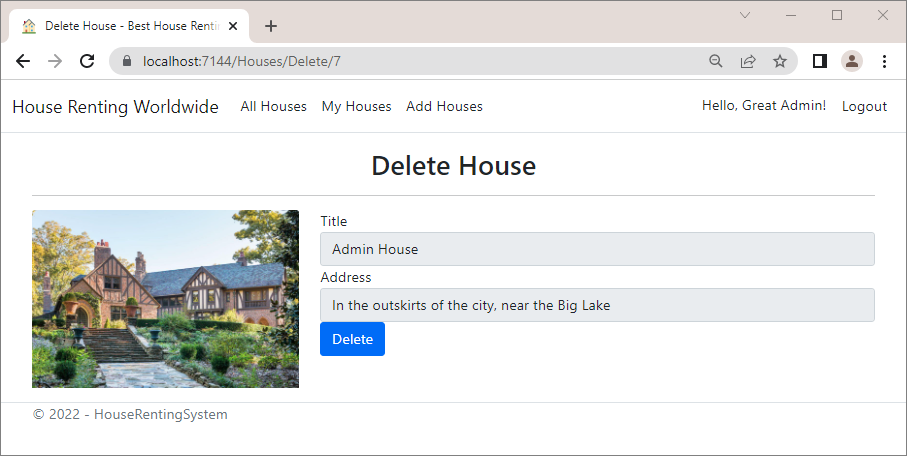
**Modify the methods to use the mapper**. Use the Map() **method** with the **result model type** and the **object to be** **mapped**. Note that some **properties cannot be mapped directly** because their **names in objects are not the same**, so we should take care of them **manually**. Do it like this:





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





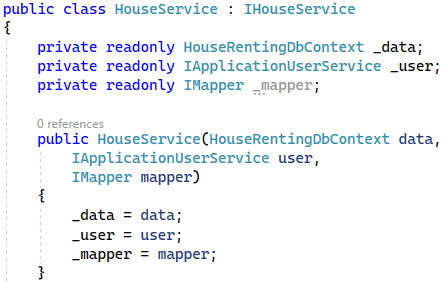
Now let's **create mapping** in the **service methods**, as well.

### Step 2: Mapping in Service Methods

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

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

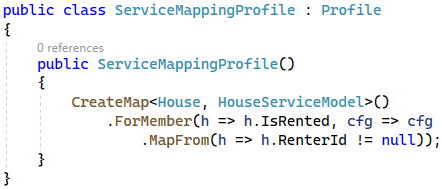
Next, you should **inject the mapper** in the HouseService **class**:



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

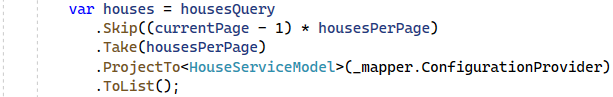
#### All(…) Method

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

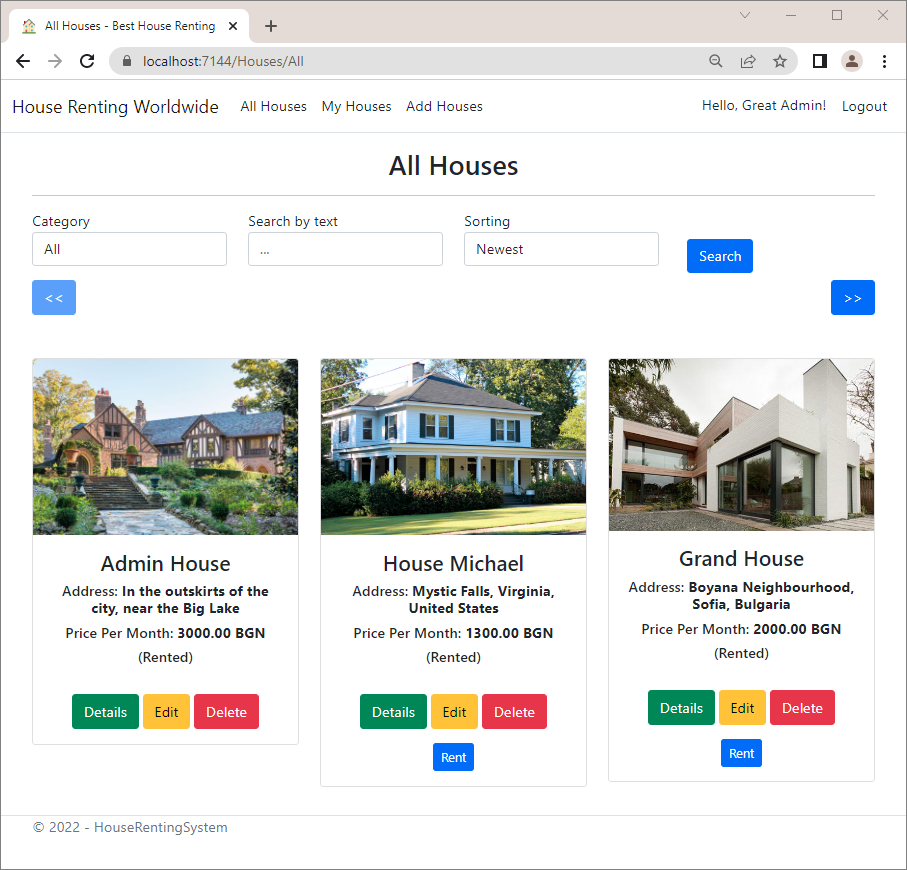


Go back to the All(…) **method** of the HouseService **class** and **use the** ProjectTo() **method** to **map objects** in a **LINQ** **query**. Note that you should **provide the method with the mapper configurations**, not the mapper itself. Do it as shown below:



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

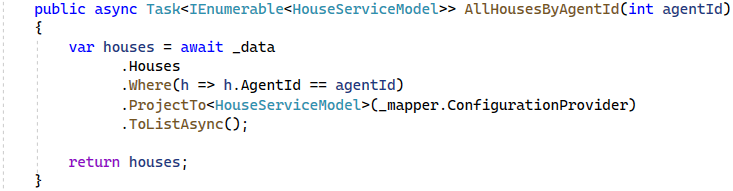
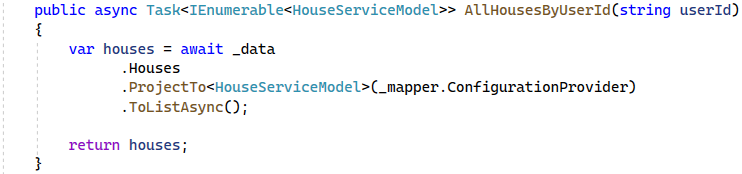


### AllHousesByAgentId(…) and AllHousesByUserId(…) Methods

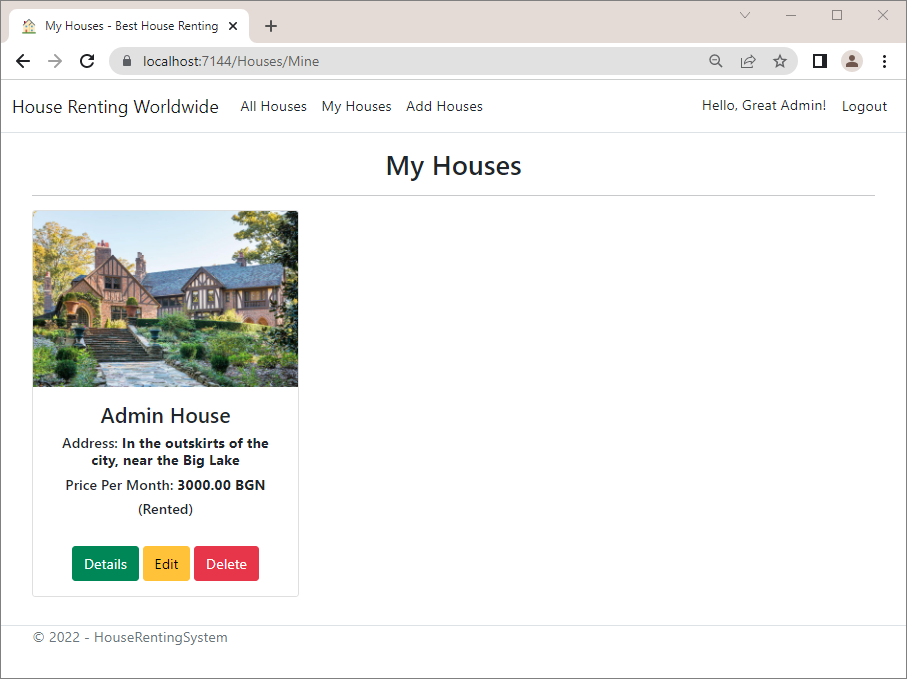
The AllHousesByAgentId(…) and AllHousesByUserId(…) methods use the private ProjectToModel(…) **method** to **map models of type** House **to** HouseServiceModel. However, we won't need that method anymore, as our mapping will be improved with AutoMapper.

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

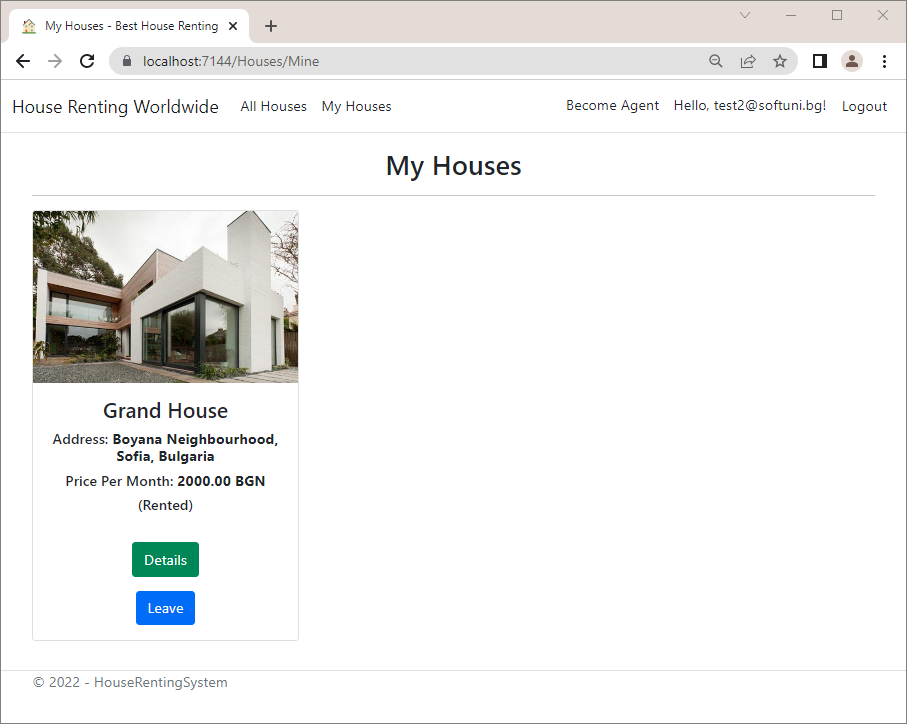


**Run the app** **and navigrate to the "**MyHouses**" page** with the Admin. The page should display the **admin's created** **houses** correctly:



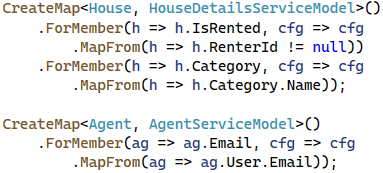
Then, **log in with another user**, who should see the **houses they rented**:



### HouseDetailsById(…) Method

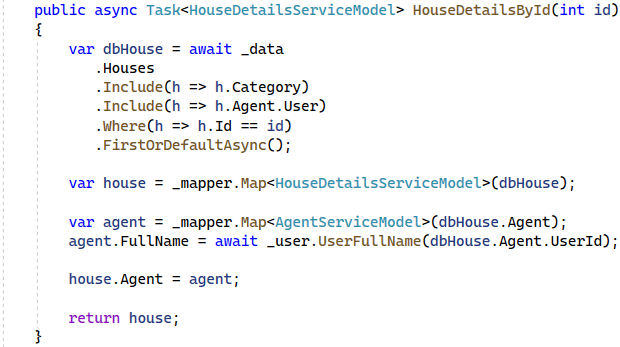
The HouseDetailsById(…) **method** has **two mappings** in one – **from** House **to** HouseDetailsServiceModel and **from** Agent **to** AgentServiceModel. In addition, some **properties mapping needs to be configured**. For example, the Category **property** of the HouseDetailsServiceModel should have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

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

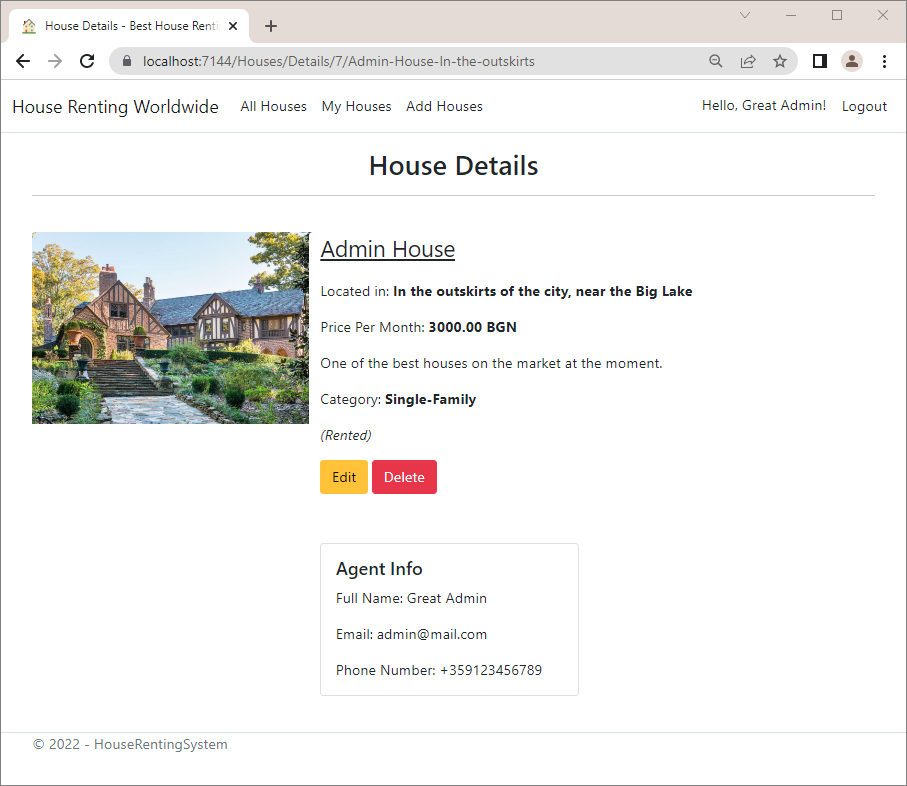


Then, **modify the** HouseDetailsById(…) **method** to **get the house** from the database by **including the necessary** **properties** for mapping and **map the objects**. Note that the **agent's full name** **should be set separately**, as it is **produced by a service method**:





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



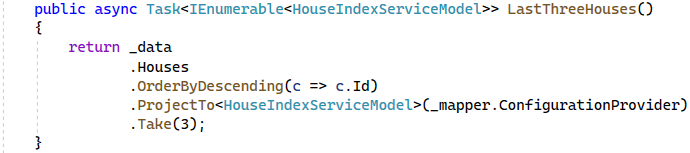
### LastThreeHouses() Method

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

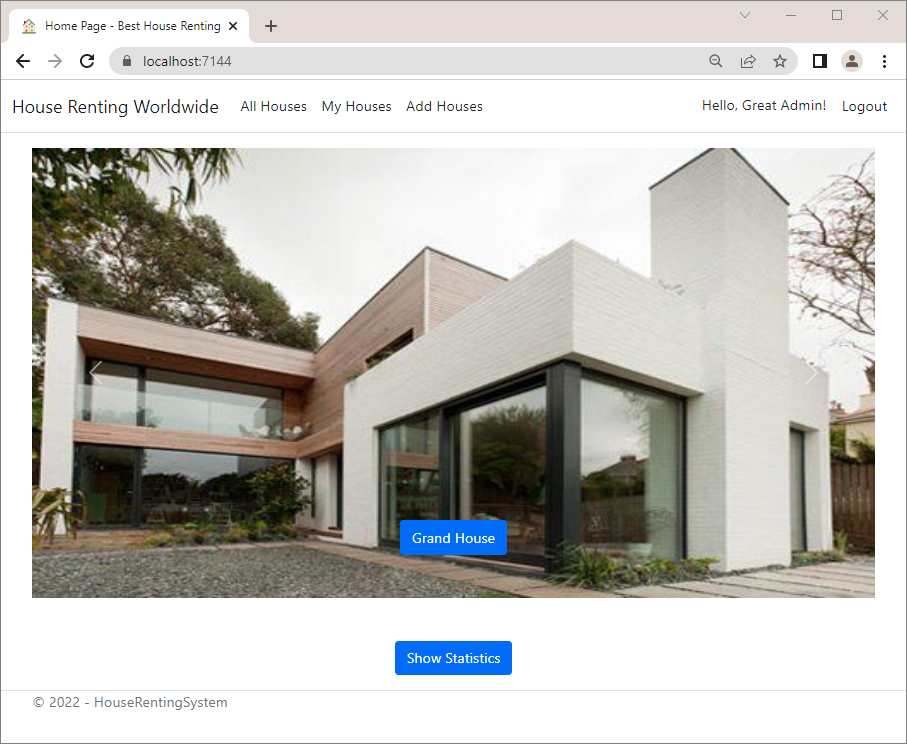


Write the LastThreeHouses() **method** like this:

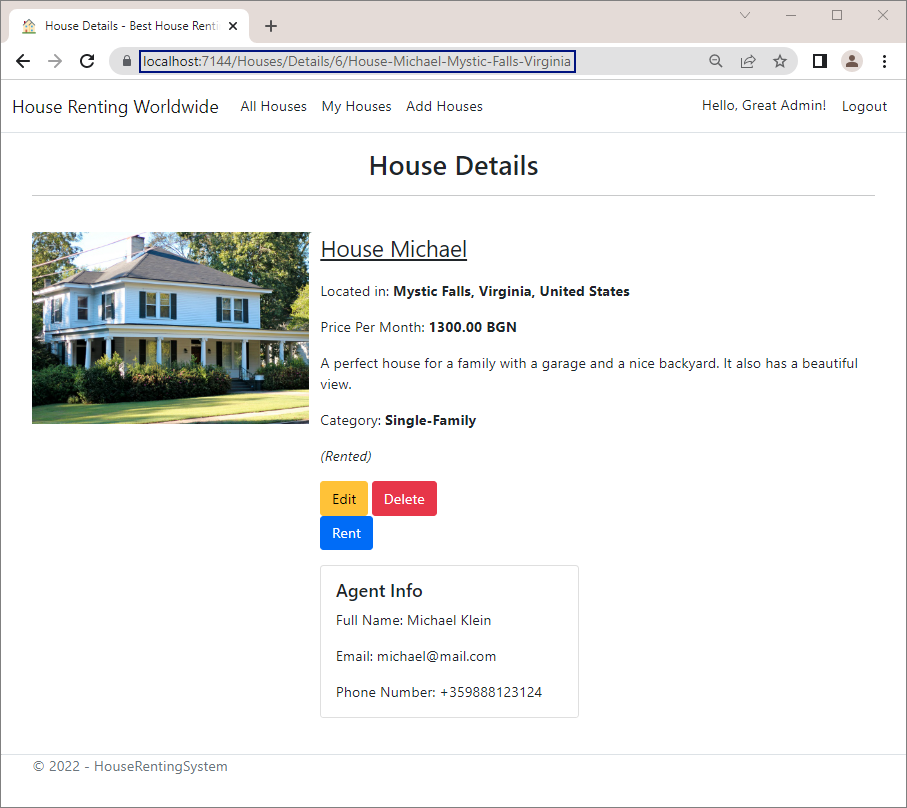




**Run the app** and **examine the** "Home" **page** with the houses. It should display them as before:



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



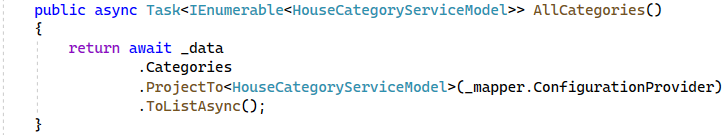
### AllCategories() Method

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

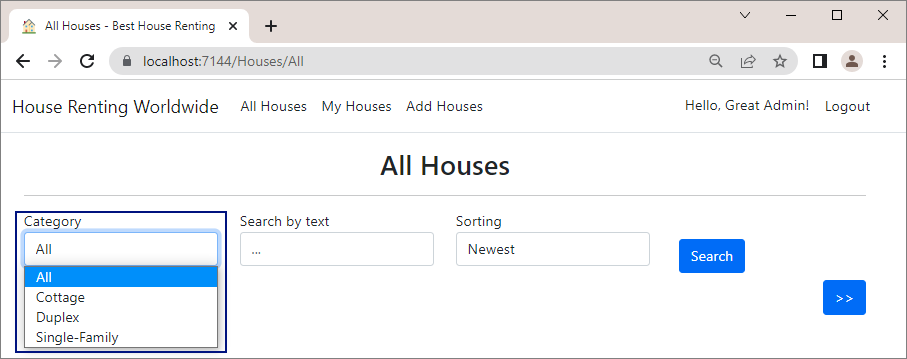


The AllCategories() **method** should be modified like this:





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



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