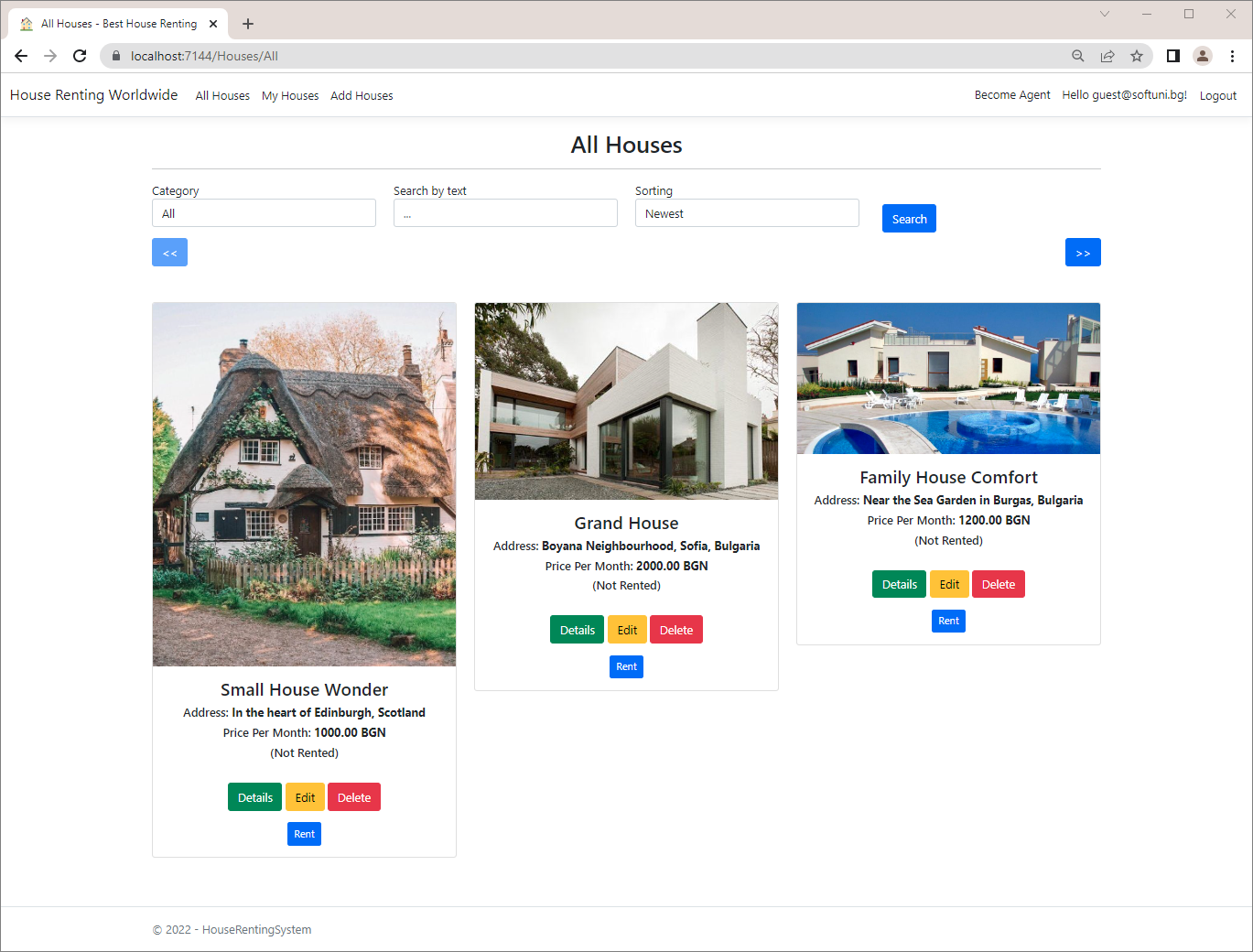
# Workshop: Services and Error Pages

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

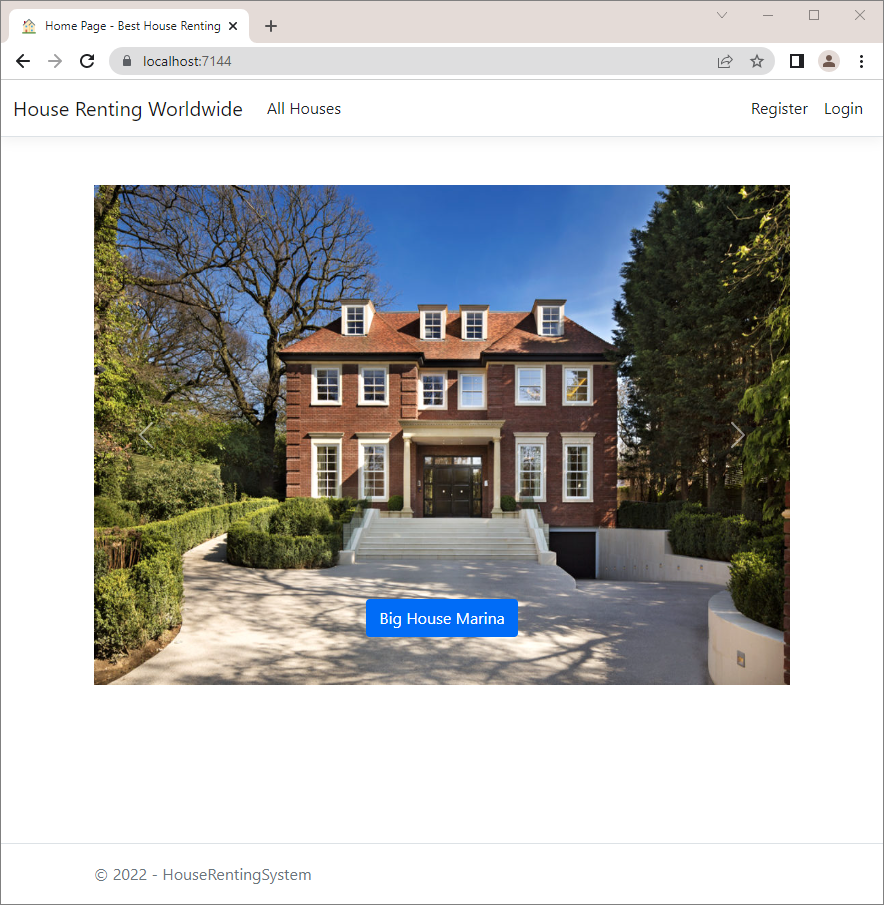
The "House Renting System" **ASP.NET Core MVC App** is a Web application for **house renting**. Users can look at **all** **houses** with their **details**, **rent a house** and look at **their rented houses**. They can also **become** Agents. Agents can **add houses**, see their **details** and **edit** 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**.

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



## Implement the "Home" Page

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



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

Graphical user interface, application

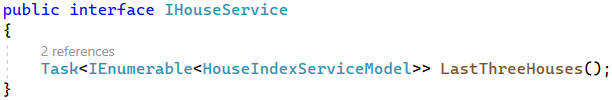
Description automatically generated

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

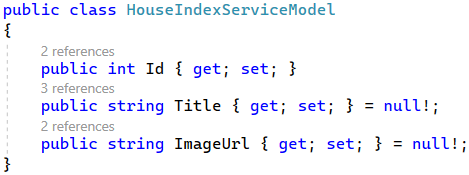
The "Home" **page** is accessed on "/" and **invokes** the Index() **method** of the HomeController **class**. We should **modify** the HomeController class with its I**ndex()** method to use services. The **houses** **collection** for the **view** (for the houses slideshow) will be implemented by using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

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

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

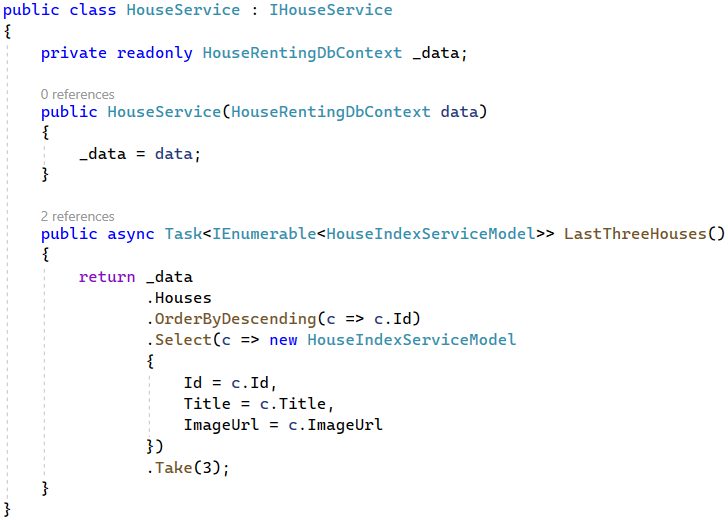


Create a new model HouseIndexServiceModel to be returned by the **service method**:



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

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

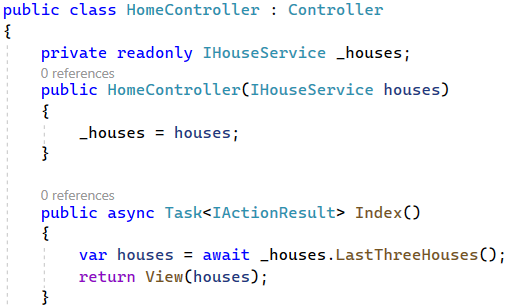


It's very important to remember to go to the "**Program.cs**" file and add the service like this:



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

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



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

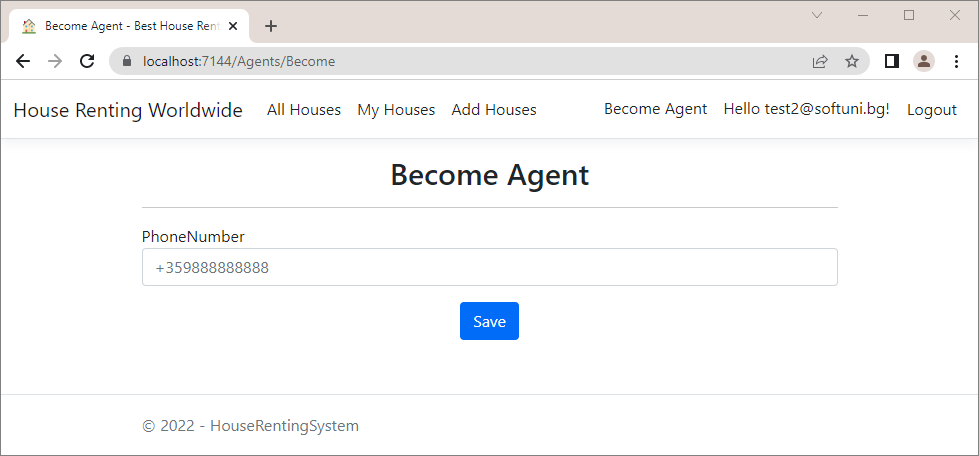
|  |
| --- |
| @model IEnumerable<HouseIndexServiceModel>  @{  ViewData["Title"] = "Home Page";  var houses = Model.ToList();  }  @if (!houses.Any())  {  <div class="mt-4 p-5 bg-light">  <h1 class="display-4">Welcome to the best place for renting houses!</h1>  <p class="lead">  There are no houses in the system currently available  so why don't you just add one?  </p>  <hr class="my-4">  <p class="lead">  @if (User.Identity.IsAuthenticated)  {  <**a** **asp-controller**="House" **asp-action**="Add" class="btn btn-primary btn-lg"  role="button">Add House</**a**>  }  </p>  </div>  }  <div class="mb-5"></div>  <div id="carouselExampleControls" class="carousel slide" data-bs-ride="carousel">  <div class="carousel-inner">  @for (int i = 0; i < houses.Count(); i++)  {  var house = houses[i];  <div class="carousel-item @(i == 0 ? "active" : string.Empty)">  <img class="d-block w-100" style="height:500px"  src="@house.ImageUrl" alt="@house.Title">  <div class="carousel-caption d-none d-md-block">  <h5>  <**a** class="btn btn-primary" **asp-controller**="House" **asp-action**="Details"  **asp-route-id**="@house.Id"> @house.Title</**a**>  </h5>  </div>  </div>  }  </div>  <button class="carousel-control-prev" type="button" data-bs-target="#carouselExampleControls" data-bs-slide="prev">  <span class="carousel-control-prev-icon" aria-hidden="true"></span>  <span class="visually-hidden">Previous</span>  </button>  <button class="carousel-control-next" type="button" data-bs-target="#carouselExampleControls" data-bs-slide="next">  <span class="carousel-control-next-icon" aria-hidden="true"></span>  <span class="visually-hidden">Next</span>  </button>  </div> |

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

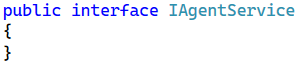
## Implement Agents

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

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



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

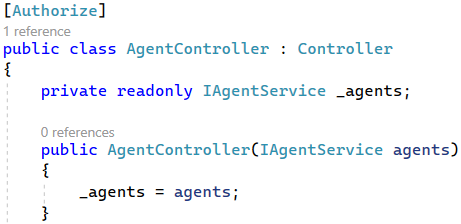




Now go to **Program.cs** and **add the service** like this:

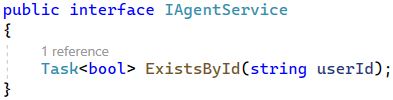


Now go to the AgentController and **i****nject the service through the constructor**, so that we can **use the service** **methods** when we create them. **Create a property** for the service and **set it** like this:

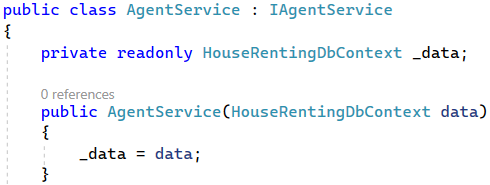


We want to **write the business logic in a service method**. Start with the Become() controller **method** – we want it to **invoke a service method** to **check whether the currently logged-in user already** **exists as an agent**. The **service method**, on the other hand, will **accept parameters**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

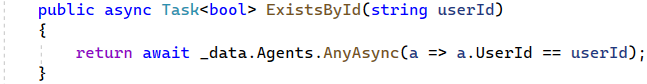
Go to the IAgentService **class** and **declare a method**, which accepts an **user id** and **returns a** bool, depending on whether an **agent with the user id exists**:



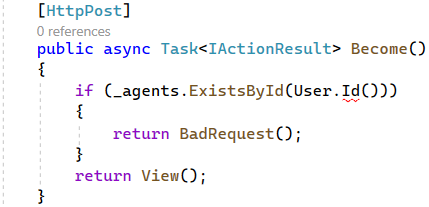
Now **implement the method** in the AgentService **class**. Before that, however, we need to **add** the db context **through the constructor** and **assign it to a property**, so that we can **obtain data from the database**. Do it as shown below:



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

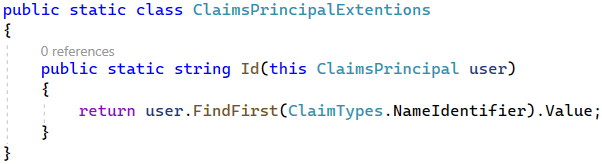


Go to AgentController and **modify** the Become() **method** to **use the service**:



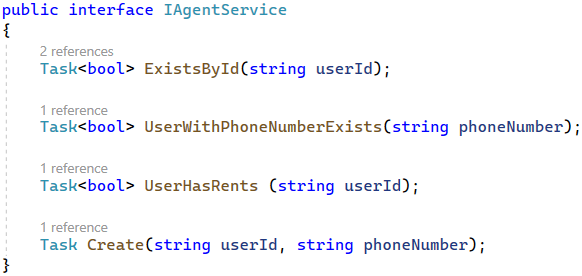
As you can see, the **ClaimsPrincipal** does not contain a definition for **Id**, so we'll have to add it. To **get the current** **user id**, let's create a **separate class** called ClaimsPrincipalExtensions in a **new folder** "Infrastructure". The reason for this is that we will need the id in **several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

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



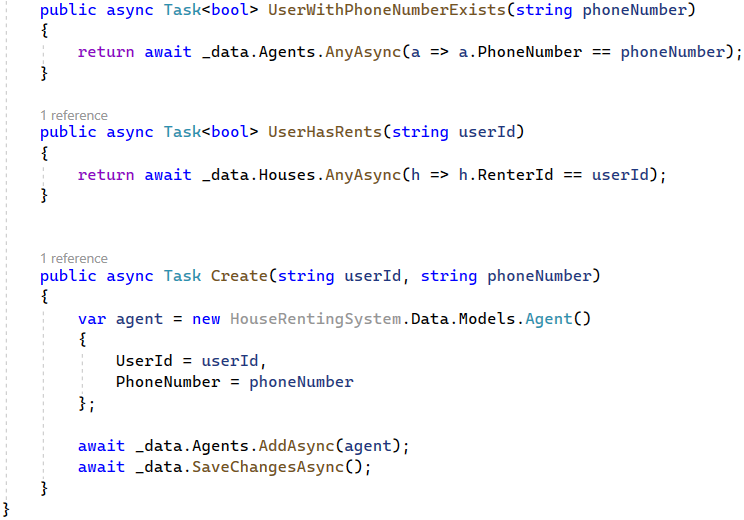
Now let's use **service methods** in the Become(BecomeAgentFormModel model) **controller method**, as well. In it, we **check whether the current user exists as an agent**, **whether an agent with a phone number exists** and whether the **user has any rents in the system**. Then, we create a **new**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

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

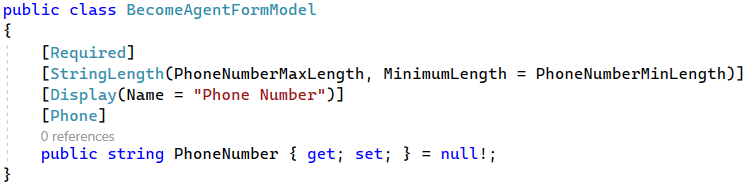


**Implement the methods** in the AgentService **class** like shown below. First, we need to check if the **current user is not already an agent**. This is done again for more security. Then we check if the user has any rents and if another agent with that phone number exists.

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



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

****

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

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

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

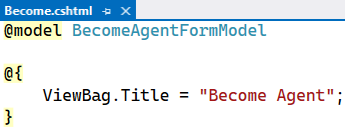
We have to **add some other validations** for **becoming an agent**, using the ModelState **class**. If a user tries to **become an agent** with a **phone number of another agent**, we should display an **error**. Also, a **user** **should have** **no current rents** to become an agent. Then, the ModelState.IsValid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



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

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

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



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

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

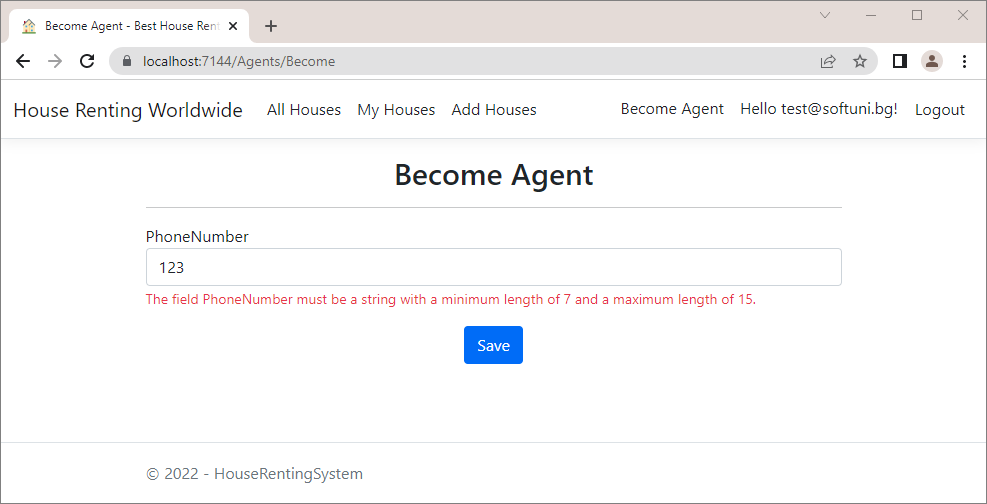
|  |
| --- |
| @model BecomeAgentFormModel  @{  ViewBag.Title = "Become Agent";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  @if (!ViewData.ModelState.IsValid && ViewData.ModelState["Error"] != null)  {  <div class="alert alert-danger" role="alert">  @ViewData.ModelState["Error"].Errors.First().ErrorMessage  </div>  }  <div class="row">  <div class="col-sm-12 offset-lg-2 col-lg-8 offset-xl-3 col-xl-6">  <**form** method="post">  <div class="form-group">  <**label** **asp-for**="PhoneNumber"></**label**>  <**input** **asp-for**="PhoneNumber" class="form-control" placeholder="+359888888888">  <**span** **asp-validation-for**="PhoneNumber" class="small text-danger"></**span**>  </div>  <div class="text-center">  <input class="btn btn-primary mt-3 " type="submit" value="Save" />  </div>  </**form**>  </div>  </div>  @section Scripts {  <**partial** **name**="\_ValidationScriptsPartial" />  } |

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

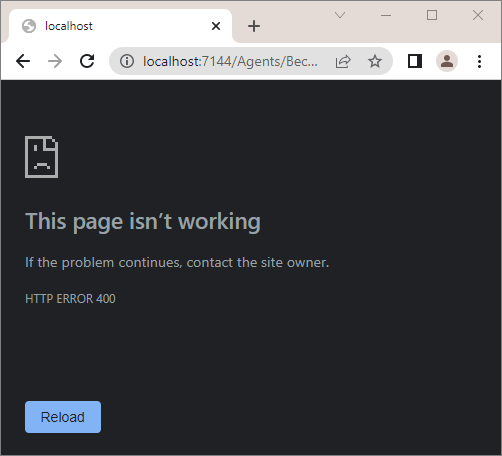
At the end, we will **define a section** in our Become **view**, which will **invoke** the "\_ValidationScriptsPartial.cshtml" **partial view**. It checks whether the **entered form data is valid** **before the form is submitted**. We will see how this is done later.

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

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



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

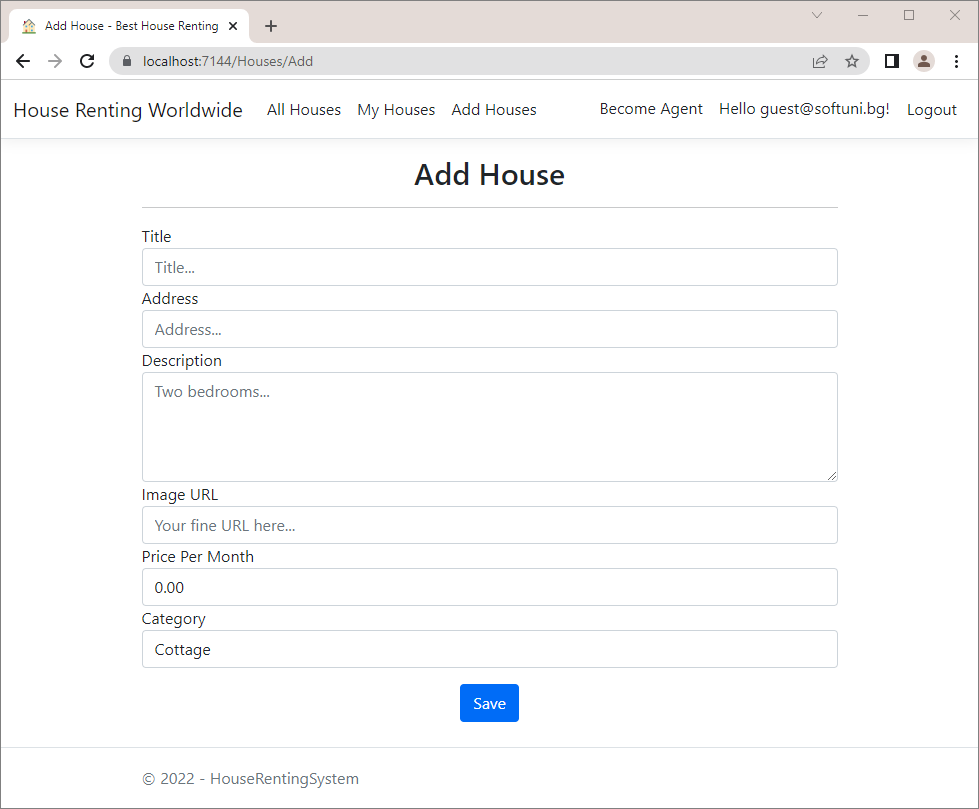


Look at the "Agents" **table** in the **database** in SQL Server Management Studio. It should have the new Agent **record**:

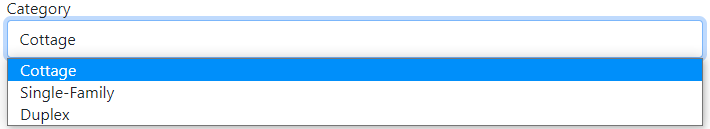


## Implement "Add House" page

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

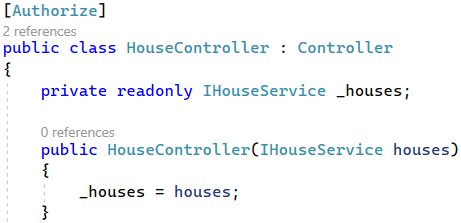


The user should **fill in the** "Add House" **form** with **valid data**, **choose a category** and press [Submit]. Note that available **categories should be displayed in the dropdown menu** like this:



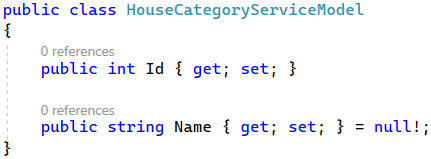
The **form data** should be used for creating a new House **record** in the **database** and the user should be **redirected** to the "Details" **page** of the **newly-created house**.

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



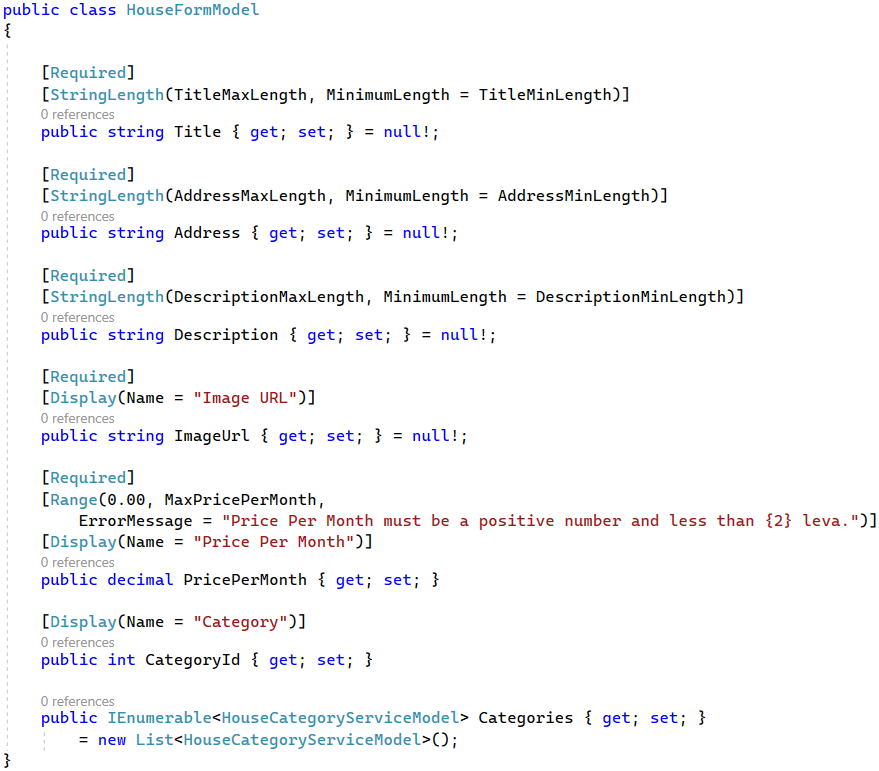
For this controller action, we need to **add a single service method** to **get all categories from the database** and **return them as a collection**.

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

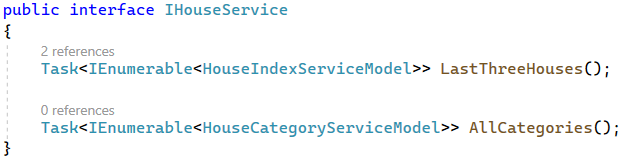


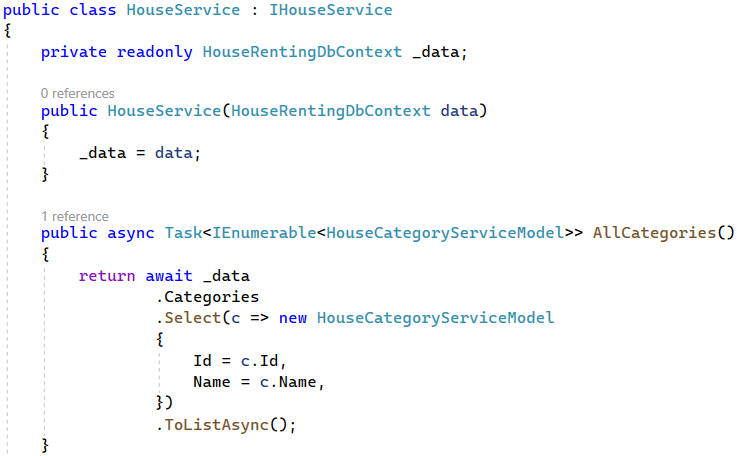
Now we should modify the HouseFormModel, as well. This class should have **properties** for **title**, **address**, **description**, **image** **URL**, **price** **per** **month**, **category** **id** and a **collection for categories**. As this is a model for **submitting a form**, we should **add restrictions**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

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

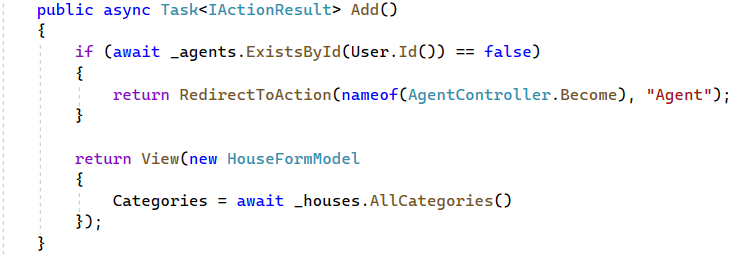


Now, implement the **service method** for **getting all categories** like this:

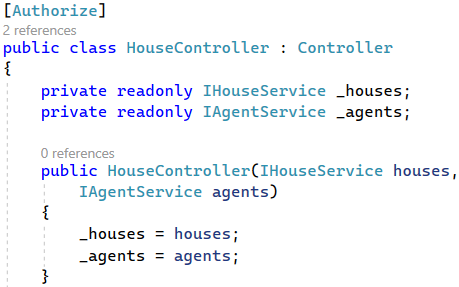
****



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



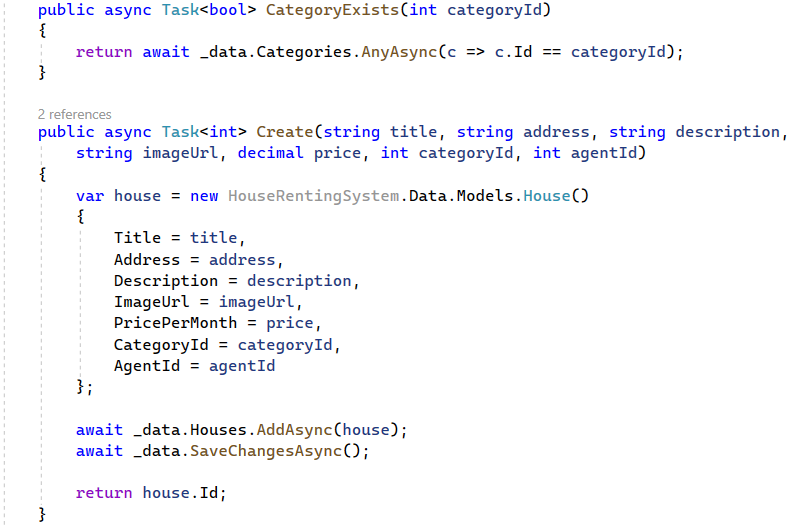
Don't forget to **accept the** IAgentService **through the constructor**:



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

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



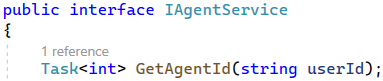


As you can see, the Create(…) **service methods** returns the **new house id** because our **controller** needs it. The Add(HouseFormModel model) **controller** **methods** should **use service methods** as shown below:

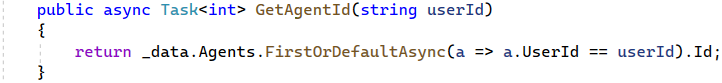


You can see that our **IAgentService** interface doesn't have a **GetAgentId method,** so our next task is to **implement** it**.**

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



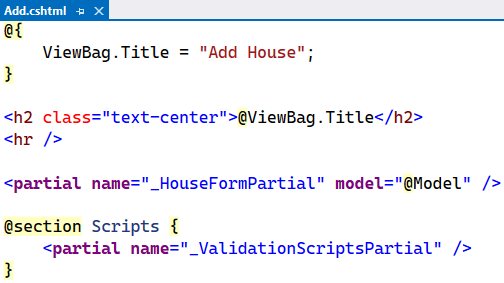
Next, go to the **AgentService** class and write the **service method**:



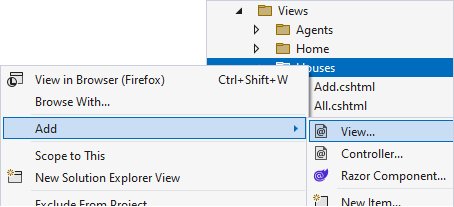
The last for completing this functionality is implementing the view. We have already created the "**Add.cshtml**" file in the **/Views/Houses** folder. We have to **add** the **HouseFormModel** to the view.

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

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

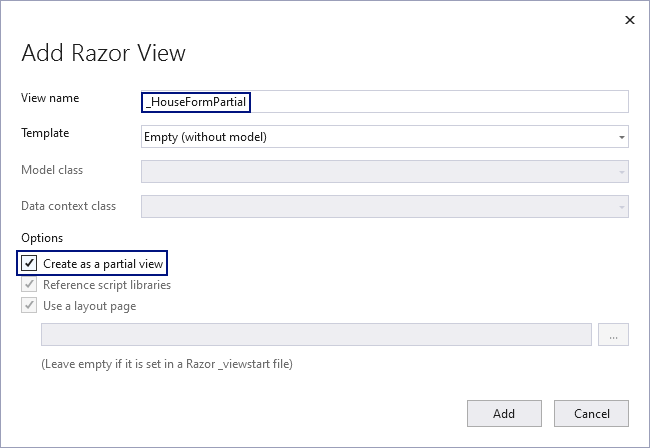


Create the "\_HouseFormPartial.cshtml" **partial view**. To do this, **right-click** on the "/Views/House" **folder** and on [Add] 🡪 [View] and **choose** [Razor View]:





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



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

|  |
| --- |
| @model HouseFormModel  <div class="row">  <div class="col-sm-12 offset-lg-2 col-lg-8 offset-xl-3 col-xl-6">  <**form** method="post">  <div class="form-group">  <**label** **asp-for**="Title"></**label**>  <**input** **asp-for**="Title" class="form-control" placeholder="Title...">  <**span** **asp-validation-for**="Title" class="small text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="Address"></**label**>  <**input** **asp-for**="Address" class="form-control" placeholder="Address...">  <**span** **asp-validation-for**="Address" class="small text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="Description"></**label**>  <**textarea** **asp-for**="Description" rows="4" class="form-control" placeholder="Two bedrooms..."></**textarea**>  <**span** **asp-validation-for**="Description" class="small text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="ImageUrl"></**label**>  <**input** **asp-for**="ImageUrl" class="form-control" placeholder="Your fine URL here...">  <**span** **asp-validation-for**="ImageUrl" class="small text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="PricePerMonth"></**label**>  <**input** **asp-for**="PricePerMonth" class="form-control">  <**span** **asp-validation-for**="PricePerMonth" class="small text-danger"></**span**>  </div>  <div class="form-group">  <**label** **asp-for**="CategoryId"></**label**>  <**select** **asp-for**="CategoryId" class="form-control">  @foreach (var category in Model.Categories)  {  <**option** **value**="@category.Id">@category.Name</**option**>  }  </**select**>  <**span** **asp-validation-for**="CategoryId" class="small text-danger"></**span**>  </div>  <div class="text-center">  <input class="btn btn-primary mt-3" type="submit" value="Save" />  </div>  </**form**>  </div>  </div> |

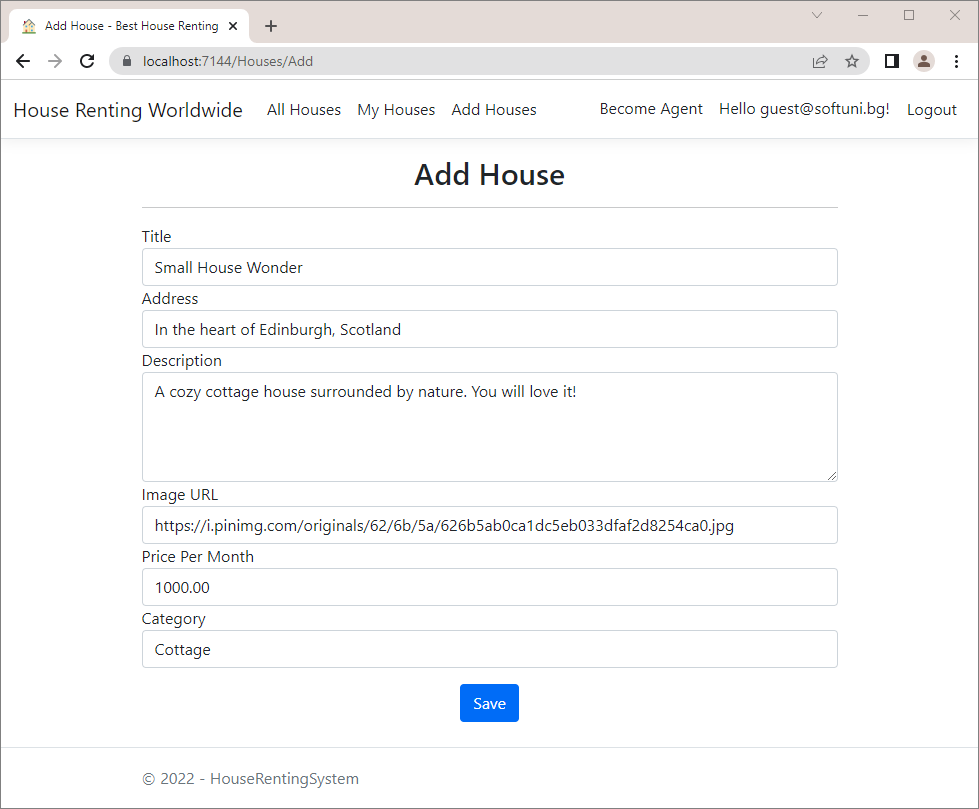
Note that we used a foreach **loop** in a **Razor view** to iterate through the Categories **collection** in the **model**. Also, look at the <label> **tags** of the **form** – they use the **class property name**. However, our properties have names like "ImageURL", "CategoryId", etc., which should **not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

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

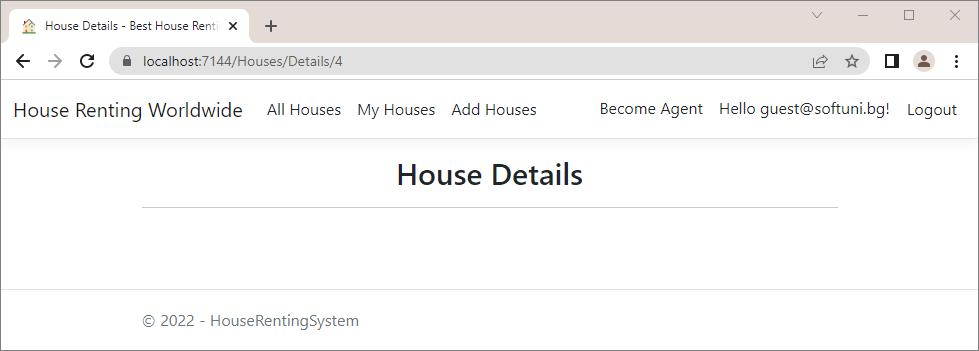
Try filling in some **invalid data** – you should see that **errors appear**. Then, **fill in valid data** and you should be **redirected** to the "Details" **page** **of the new house**:

You can get the above image URL from here:

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



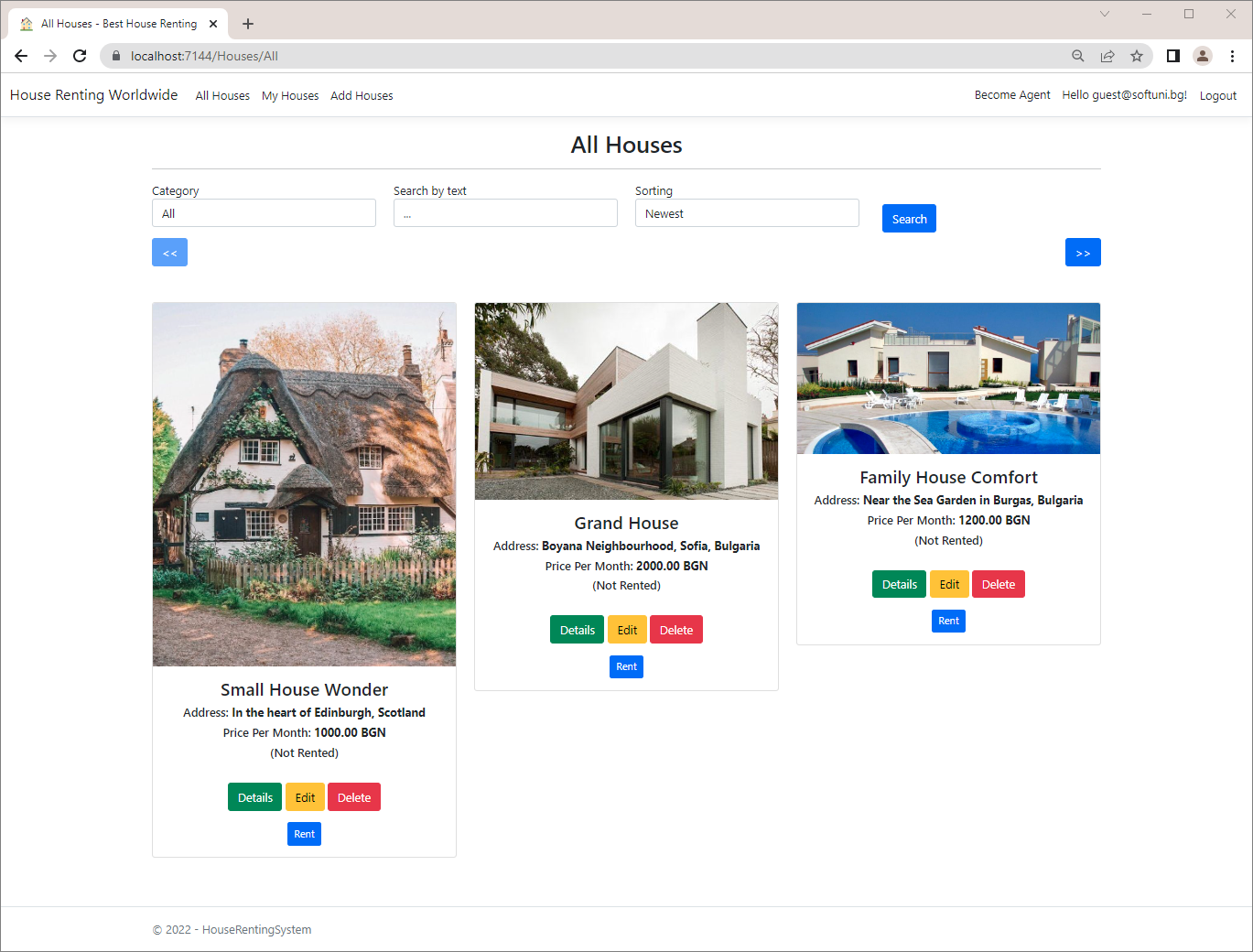
Upon pressing the [**Save**] button, you are redirected to the "**House Details**" page, but we have not yet implemented this functionality. We will do this later in the workshop, because first we must implement the "**All Houses**" page, which will give us access to the "**Edit** **House**" and "**Delete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



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

## Implement "All Houses" page

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

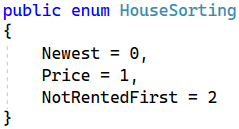


Start with the All([FromQuery] AllHousesQueryModel query) **controller method** – we want it to **invoke a service method** and **pass the result model to a view**. The service method, on the other hand, will accept **parameters** **from the** **controller action** and use them to **return a model**.

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

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

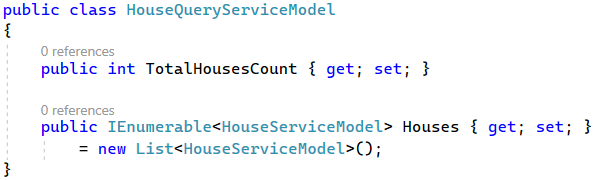
First, let's add an enum **class** for the **house sorting**. Create the HouseSorting **class** in the "Infrastructure" **folder**. The class should have the **options for sorting** – we can **sort houses by newest**, by **price** or by **not-rented first**. The enum **class** looks like this:



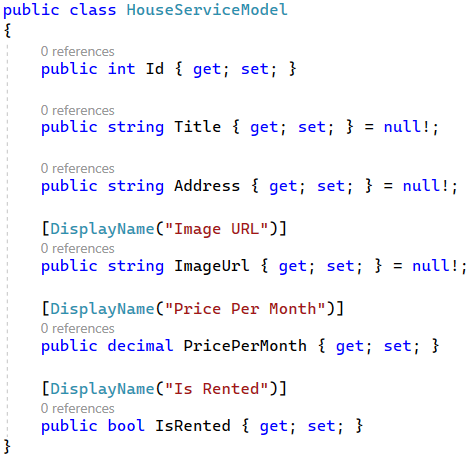
Now, to implement the above-described **workflow** in the controller method, let's first **create the service methods**. Go to the IHouseService **class** and **declare a method for getting all houses and their count** and one for **getting the house categories**.

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

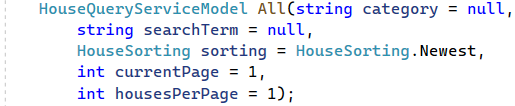
Create the HouseQueryServiceModel in a separate **folder** called "Models" in "/Services/Houses" like this:



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

****

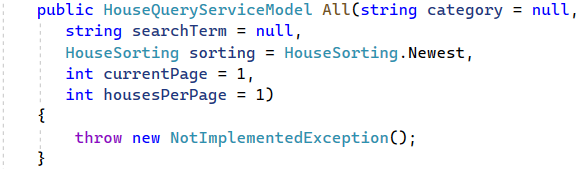
**Define** the All() **method** like this:



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



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





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

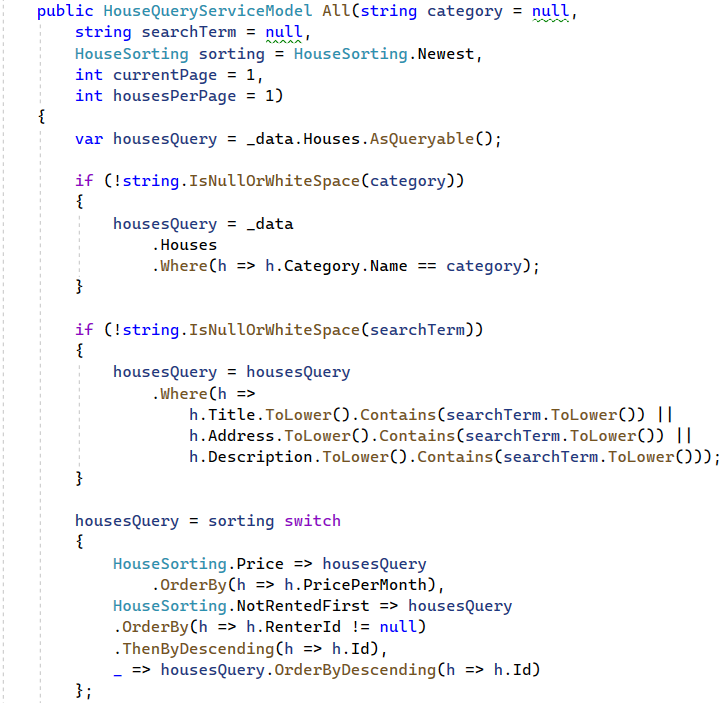
We get the **houses from the model collection** and **convert** **it** to IQueryable, so that we can **perform LINQ operations** on it.

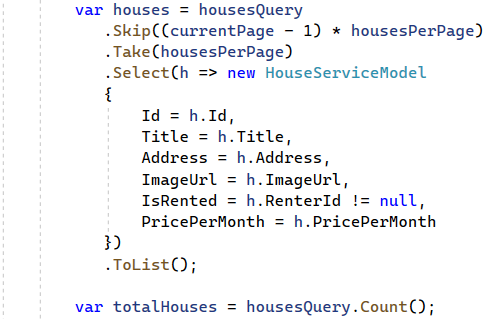
Then, we check whether we need to **filter houses by a category** and **get only the houses from the given category**. If we have a **search term**, we should get only the House **entities**, which **contain** it in their **title**, **address** or **description**. **Searching** should be **case-insensitive**.

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

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

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

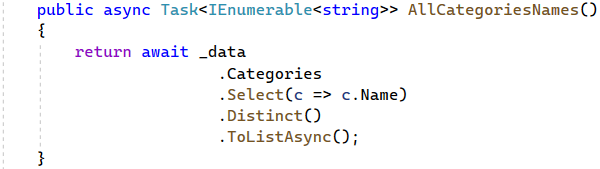




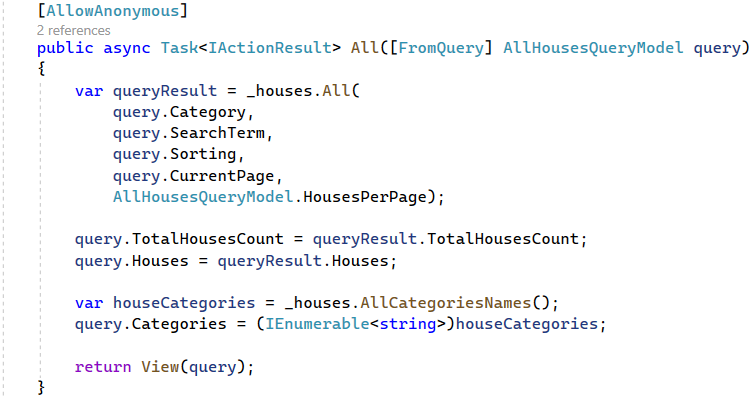
At the end, **return** a HouseQueryServiceModel with the **needed properties**:



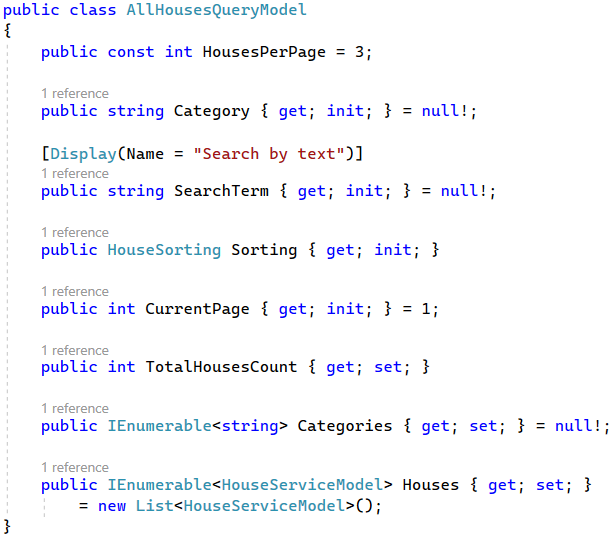
**Implement** the AllCategoriesNames() **method**, which should only **return the categories names**:



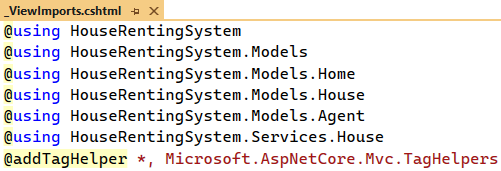
Now go to the All([FromQuery] AllHousesQueryModel query) **method** in the HouseController **class** and implement it to **use the service methods** we created and **return a view with the model:**



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



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



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

Graphical user interface, application

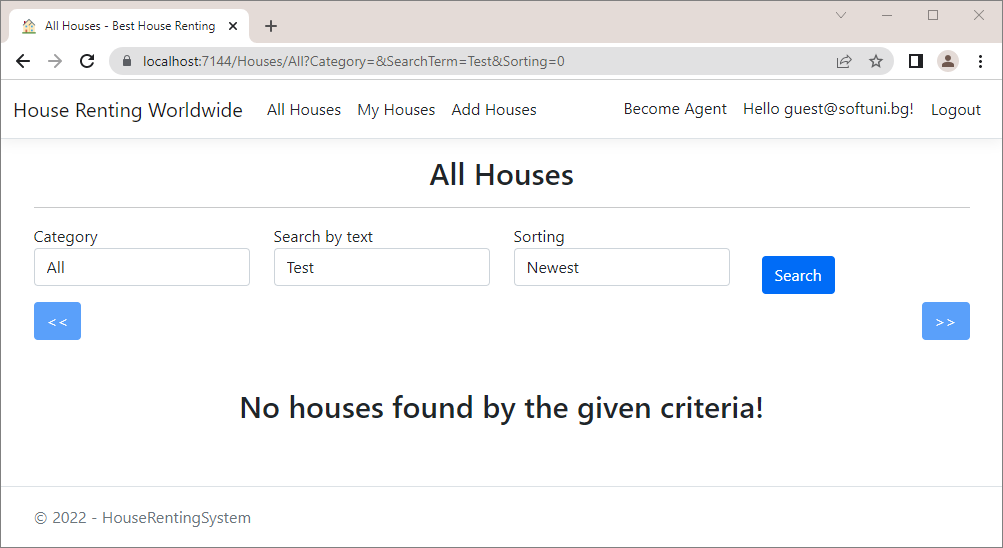
Description automatically generated

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





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



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

|  |
| --- |
| @model AllHousesQueryModel  @{  ViewBag.Title = "All Houses";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  <**form** method="get">  <div class="row">  <div class="form-group col-md-3 d-flex justify-content-between">  <div class="form-group">  <**label** **asp-for**="Category"></**label**>  <**select** **asp-for**="Category" class="form-control">  <**option** **value**="">All</**option**>  @foreach (var category in Model.Categories)  {  <**option** **value**="@category">@category</**option**>  }  </**select**>  </div>  </div>  <div class="form-group col-md-3">  <**label** **asp-for**="SearchTerm"></**label**>  <**input** **asp-for**="SearchTerm" class="form-control" placeholder="...">  </div>  <div class="form-group col-md-3">  <div class="form-group">  <**label** **asp-for**="Sorting"></**label**>  <**select** **asp-for**="Sorting" class="form-control">  <**option** **value**="0">Newest</**option**>  <**option** **value**="1">Lowest price first</**option**>  <**option** **value**="2">Not rented first</**option**>  </**select**>  </div>  </div>  <div class="col-md-3">  <div class="form-group mt-4 p-2">  <input type="submit" value="Search" class="btn btn-primary" />  </div>  </div>  </div>  </**form**>  @{  var previousPage = Model.CurrentPage - 1;  if (previousPage < 1)  {  previousPage = 1;  }  var maxPage = Math.Ceiling((double)Model.TotalHousesCount /  AllHousesQueryModel.HousesPerPage);  }  <div class="row mb-5">  <div class="col-md-6 d-grid gap-2 d-md-flex justify-content-md-start">  <**a** class="btn btn-primary @(Model.CurrentPage == 1 ? "disabled" : string.Empty)"  **asp-controller**="House"  **asp-action**="All"  **asp-route-currentPage**="@previousPage"  **asp-route-category**="@Model.Category"  **asp-route-searchTerm**="@Model.SearchTerm"  **asp-route-sorting**="@((int)Model.Sorting)"><<</**a**>  </div>  @{  var shouldButtonBeDisabled = Model.CurrentPage == maxPage ||  !Model.Houses.Any();  }  <div class="col-md-6 d-grid gap-2 d-md-flex justify-content-md-end">  <**a** class="btn btn-primary  @(shouldButtonBeDisabled ? "disabled" : string.Empty)"  **asp-controller**="House"  **asp-action**="All"  **asp-route-currentPage**="@(Model.CurrentPage + 1)"  **asp-route-category**="@Model.Category"  **asp-route-searchTerm**="@Model.SearchTerm"  **asp-route-sorting**="@((int)Model.Sorting)">>></**a**>  </div>  </div>  @if (!Model.Houses.Any())  {  <h2 class="text-center">No houses found by the given criteria!</h2>  }  <div class="row">  @foreach (var house in Model.Houses)  {  <**partial** **name**="\_HousePartial" **model**="@house" />  }  </div> |

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

Copy the code from here:

|  |
| --- |
| @model HouseServiceModel  <div class="col-md-4">  <div class="card mb-3">  <img class="card-img-top" src="@Model.ImageUrl" alt="House Image">  <div class="card-body text-center">  <h4>@Model.Title</h4>  <h6>Address: <b>@Model.Address</b></h6>  <h6>  Price Per Month:  <b>@String.Format("{0:f2}", Model.PricePerMonth) BGN</b>  </h6>  <h6>(@(Model.IsRented ? "Rented" : "Not Rented"))</h6>  <br />  <a asp-controller="House" asp-action="Details" asp-route-id="@Model.Id"  class="btn btn-success">Details</a>  @if (this.User.Identity.IsAuthenticated)  {  <a asp-controller="House" asp-action="Edit" asp-route-id="@Model.Id"  class="btn btn-warning">Edit</a>  <a asp-controller="House" asp-action="Delete" asp-route-id="@Model.Id"  class="btn btn-danger">Delete</a>  <p></p>  @if (!Model.IsRented)  {  <form class="input-group-sm" asp-controller="House"  asp-action="Rent" asp-route-id="@Model.Id" method="post">  <input class="btn btn-primary" type="submit" value="Rent" />  </form>  }  else  {  <form asp-controller="House" asp-action="Leave"  asp-route-id="@Model.Id" method="post">  <input class="btn btn-primary" type="submit" value="Leave" />  </form>  }  }  </div>  </div>  </div> |

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

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

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



## Implement "My Houses" page

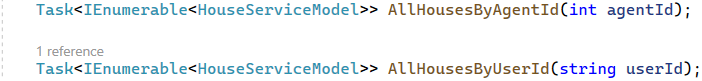
Now it's time to implement the "**My Houses**" page. It will have **different implementations** for **users** and for **agents**. When the **user is not an agent**, they will **see the houses they rented**. When the **user is an agent**, they will see the **houses they created**. The page should look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



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

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

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

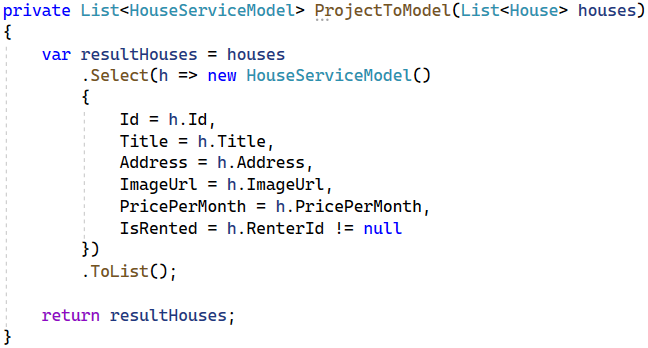


Now let's write **these methods** in the HouseService. We will add a **separate method for projecting a filtered** House **collection** to List<HouseServiceModel>. Note that we should **return a collection of type** HouseServiceModel, as the AllHousesQueryModel requires it.

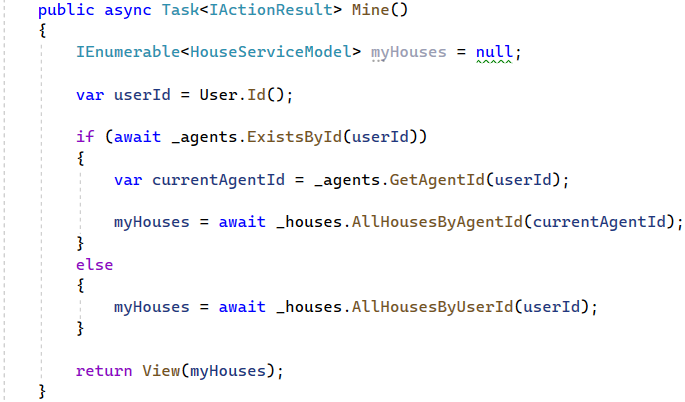
Write the following **methods** in the HouseService **class** like this:



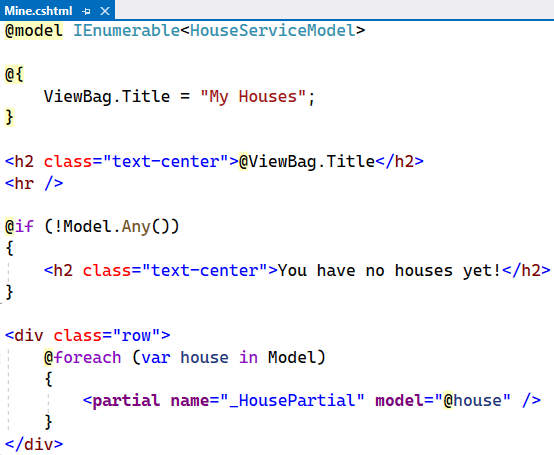
As you can see, we are going to need an additional method in the **HouseService** class, that should look like this:



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



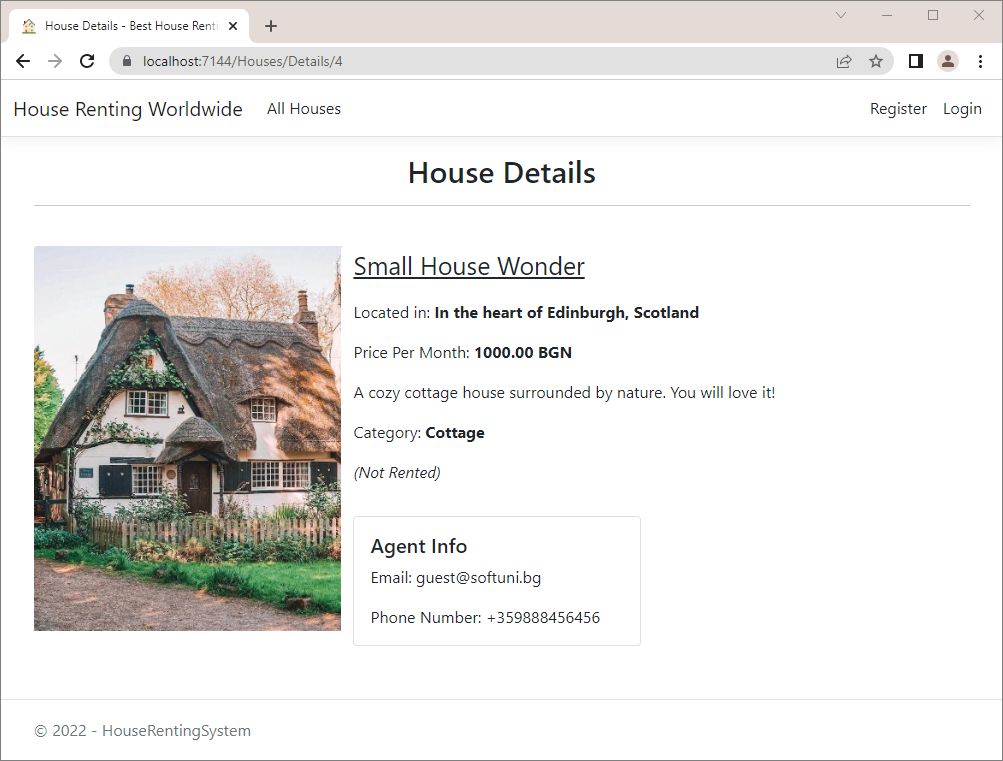
Finally, write the "Mine.cshtml" **view**, which should only **accept a model** and **use** the "\_HousePartial.cshtml" **partial view** to **display each house**.



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

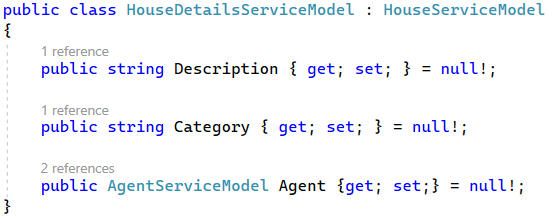
## Implement "Details" page

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

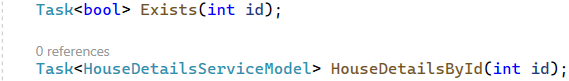


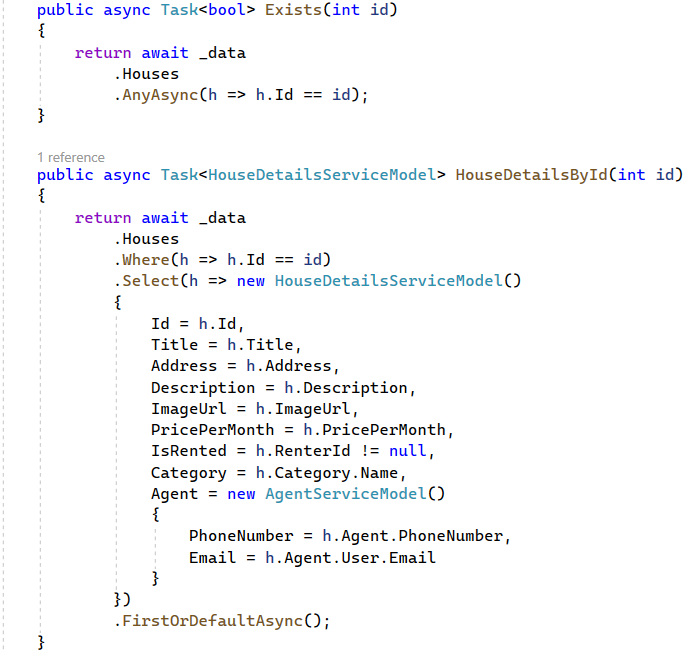
The next method we should **modify** is Details(int id) in the HouseController. It will have logic for **checking** **whether a house with a given id exists** and for **projecting the** House to a HouseDetailsViewModel.

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

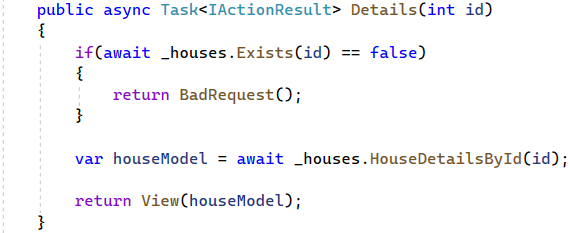


Now let's write the logic of **service methods**. Create the following **service methods** in the IHouseService **interface** and the HouseService **class**. **Get the house from the database** and **project it to** **a** HouseDetailsViewModel.





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



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

|  |
| --- |
| @model HouseDetailsServiceModel  @{  ViewBag.Title = "House Details";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  <div class="container" style="display:inline">  <div class="row">  <div class="col-4">  <img class="card-img-top" style="width: 20rem;"  src="@Model.ImageUrl" alt="House Image">  </div>  <div class="card col-8 border-0">  <p style="font-size:25px;"><u>@Model.Title</u></p>  <p>Located in: <b>@Model.Address</b></p>  <p>  Price Per Month:  <b>@String.Format("{0:f2}", Model.PricePerMonth) BGN</b>  </p>  <p>@Model.Description</p>  <p>Category: <b>@Model.Category</b></p>  <p><i>(@(Model.IsRented ? "Rented" : "Not Rented"))</i></p>  <div class="form-inline">  @if (this.User.Identity.IsAuthenticated)  {  <**a** class="btn btn-warning" **asp-controller**="House" **asp-action**="Edit"  **asp-route-id**="@Model.Id">Edit</**a**>  <**a** class="ml-2 btn btn-danger" **asp-controller**="House" **asp-action**="Delete"  **asp-route-id**="@Model.Id">Delete</**a**>  @if (!Model.IsRented)  {  <**form** class="ml-2" **asp-controller**="House"  **asp-action**="Rent" **asp-route-id**="@Model.Id" method="post">  <input class="btn btn-primary" type="submit" value="Rent" />  </**form**>  }  else  {  <**form** class="ml-2" **asp-controller**="House" **asp-action**="Leave"  **asp-route-id**="@Model.Id" method="post">  <input class="btn btn-primary" type="submit" value="Leave" />  </**form**>  }  }  </div>  <p></p>  <div class="card" style="width: 18rem;">  <div class="card-body">  <h5 class="card-title">Agent Info</h5>  <p class="card-text">Email: @Model.Agent.Email</p>  <p class="card-text">Phone Number: @Model.Agent.PhoneNumber</p>  </div>  </div>  </div>  </div>  </div> |

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

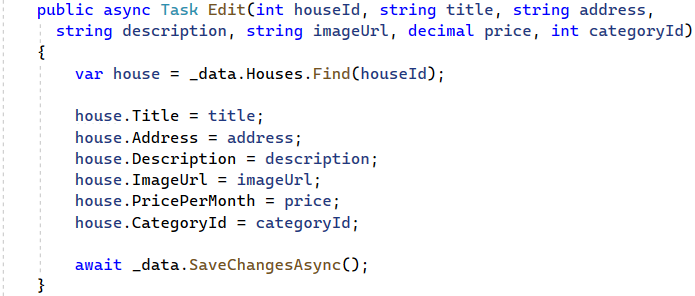
## Implement "Edit House" page

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

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

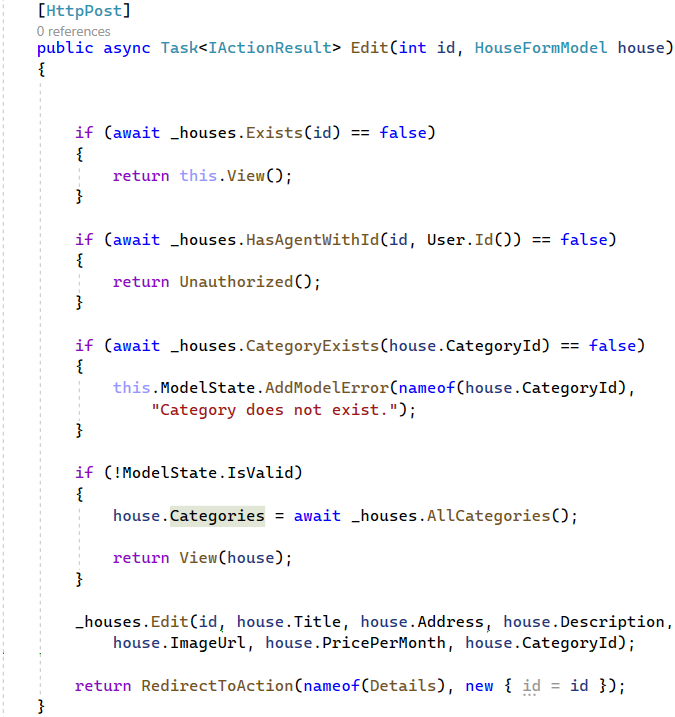


Do that for the **HouseService** class, too. The method should **find by id** the house that we want to **edit**, **change** the needed info and **save** the **changes** to the database.



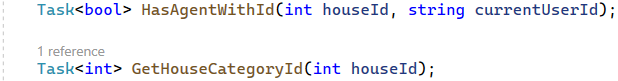
Next, we need to modify the **Edit(int id)** and … controller methods from the **HouseController**. Do it like this:

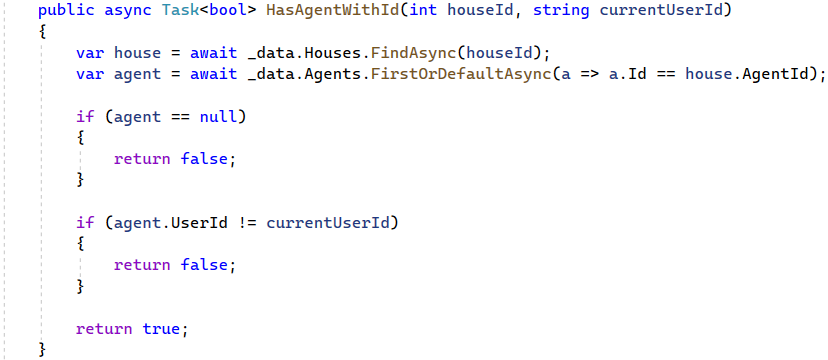


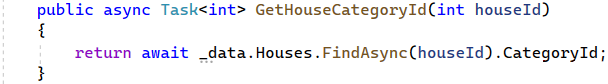


As you can see, we need some additional methods that checks if the agent that created the house, is the currently signed in agent and another one that returns the category of a house.

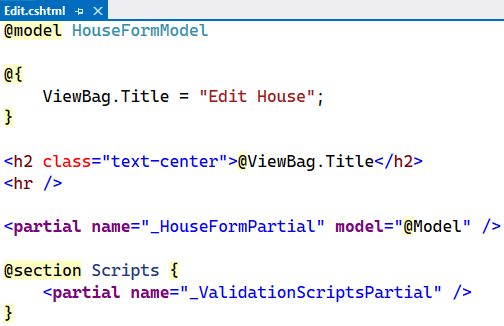
Write them in the **IHouseService** interface and in the **HouseService** class like shown below:



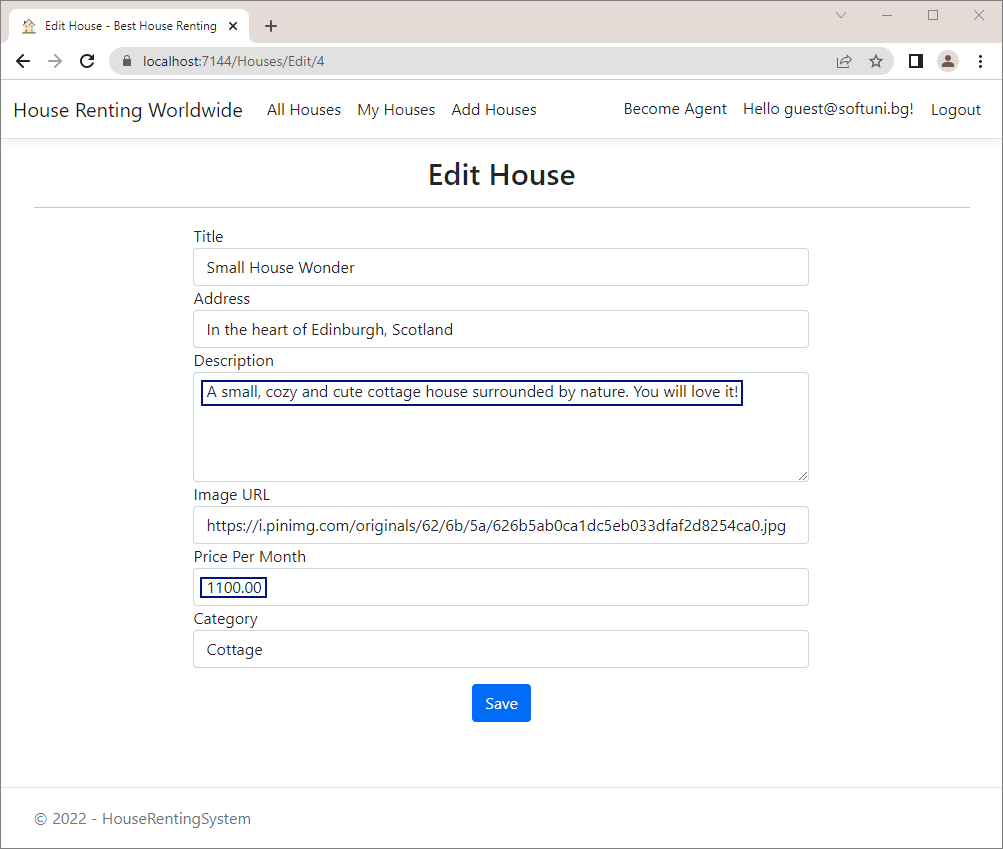




The "Edit.cshtml" **view** should **render** the "\_HouseFormPartial.cshtml" **partial view** and should look like this:

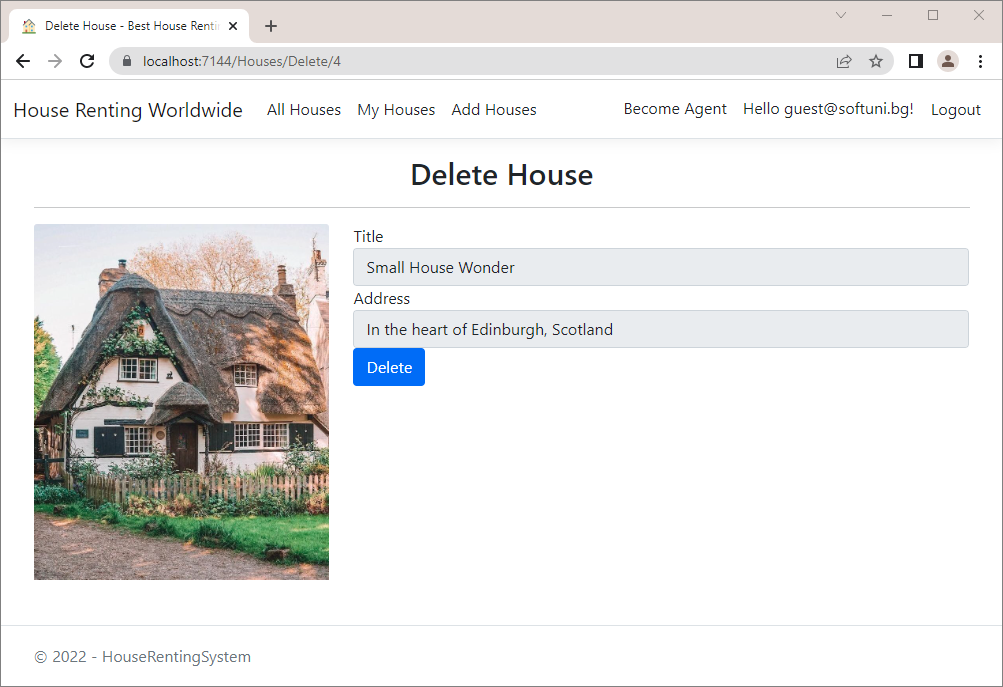


Try the "Edit" functionality in the browser. After **successful edit**, you should be **redirected** to the "Details" **page** and the **changes should be saved** to the **database**, as well.

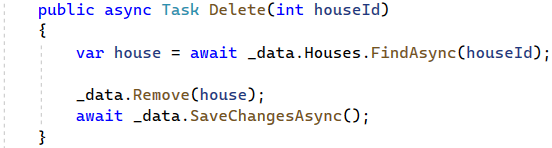


## Implement "Delete House" page

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

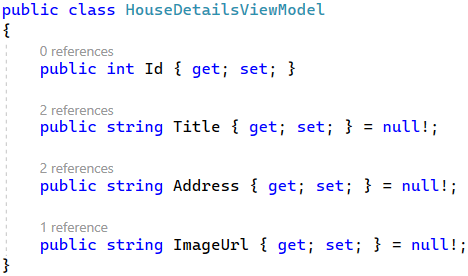
 First, modify the **IHouseServices** interface and the **HouseService** class by adding the **Delete(int houseId)** service method. The method is quite simple – it **finds** **by id the house** that we want to **delete** and **removes** it from the **database**.



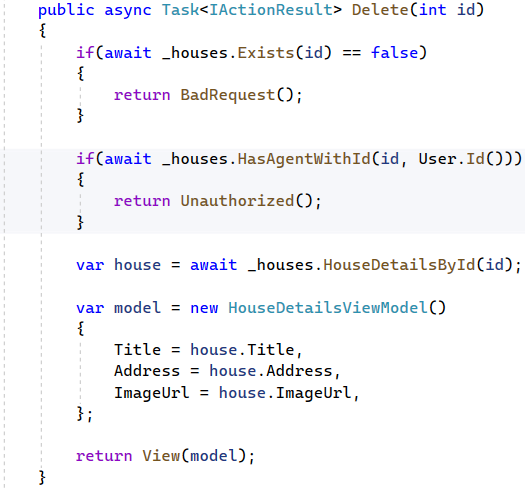


Before modifying the **Delete(int id)** and **Delete(HouseDetailsViewModel** **model)** methods in the **HouseController,** we should **add** **properties** to the **HouseDetailsViewModel** view model, that we created at the beginning of this workshop.

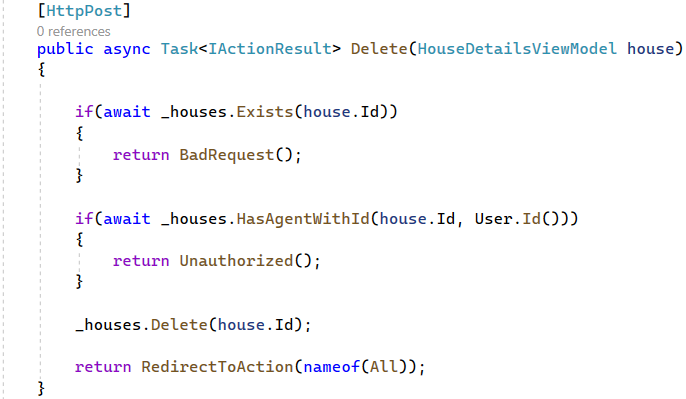
It should look like this:



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



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



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

You can copy the code from here:

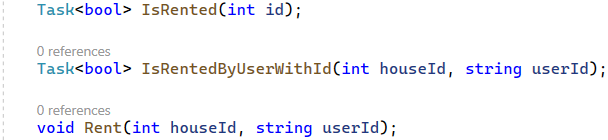
|  |
| --- |
| @model HouseDetailsViewModel  @{  ViewBag.Title = "Delete House";  }  <h2 class="text-center">@ViewBag.Title</h2>  <hr />  <**form** **asp-action**="Delete">  <div class="row">  <div class="col-md-4">  <img class="card-img-top" src="@Model.ImageUrl" alt="House Image">  </div>  <div class="col-md-8">  <div class="form-group">  <**label** **asp-for**="Title" class="control-label"></**label**>  <**input** **asp-for**="Title" class="form-control" disabled="disabled" />  </div>  <div class="form-group">  <**label** **asp-for**="Address" class="control-label"></**label**>  <**input** **asp-for**="Address" class="form-control" disabled="disabled" />  </div>  <div class="form-group">  <input type="submit" value="Delete" class="btn btn-primary" />  </div>  </div>  </div>  </**form**> |

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

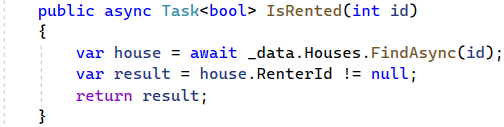
## Implement "Rent House" functionality

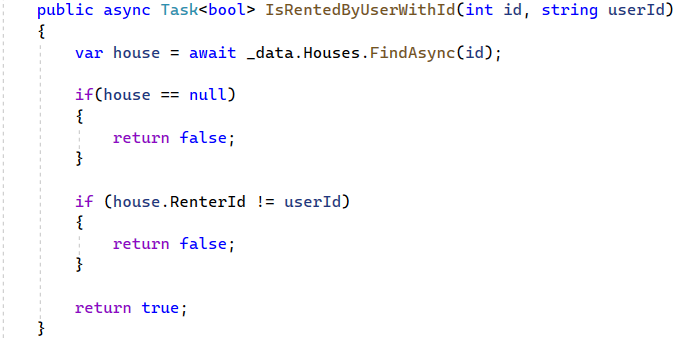
The next functionality we shall implement is for **renting a house**. Modify the Rent(int id) **method** in the HouseController **class**, which should be invoked on a "POST" **request** to "/Houses/Rent/{id}".

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

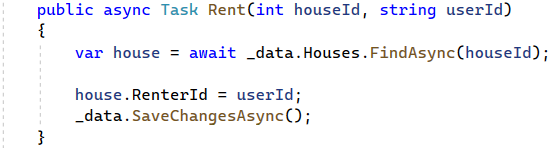


The first two methods perform checks regarding the house that we want to rent. If a user wants to **rent a house**, make sure that a **house with a given** id **exists** in the database and that the **house is not already rented**.

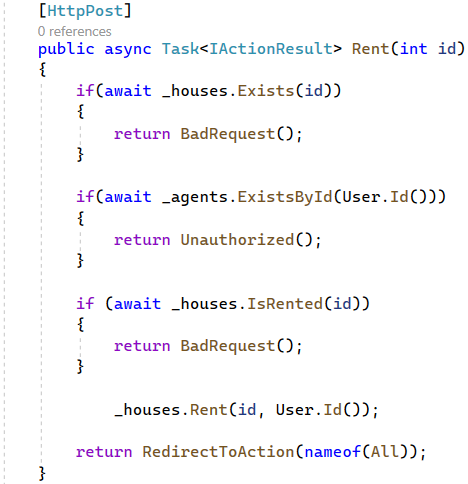




Then, **set the house** RenterIdin the **database** to the **current user** id and **save it**.



After that, it's time to modify the **Rent(int id)** method in the **HouseController**. Do it like this:



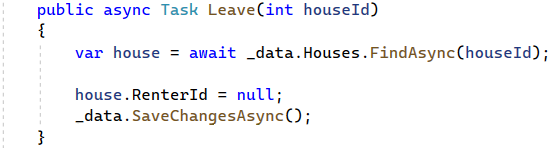
Now try out the "**rent**" **functionality**. To do this, **log in as a** **user**, who is **not an agent**, and go to "/Houses/Rent/{id}" with the **house** id in the **URL**.The **id** should be of a **house**, which is **not already rented.** If the **rent is successful**, you should be **redirected**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

## Implement "Leave House" functionality

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

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





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

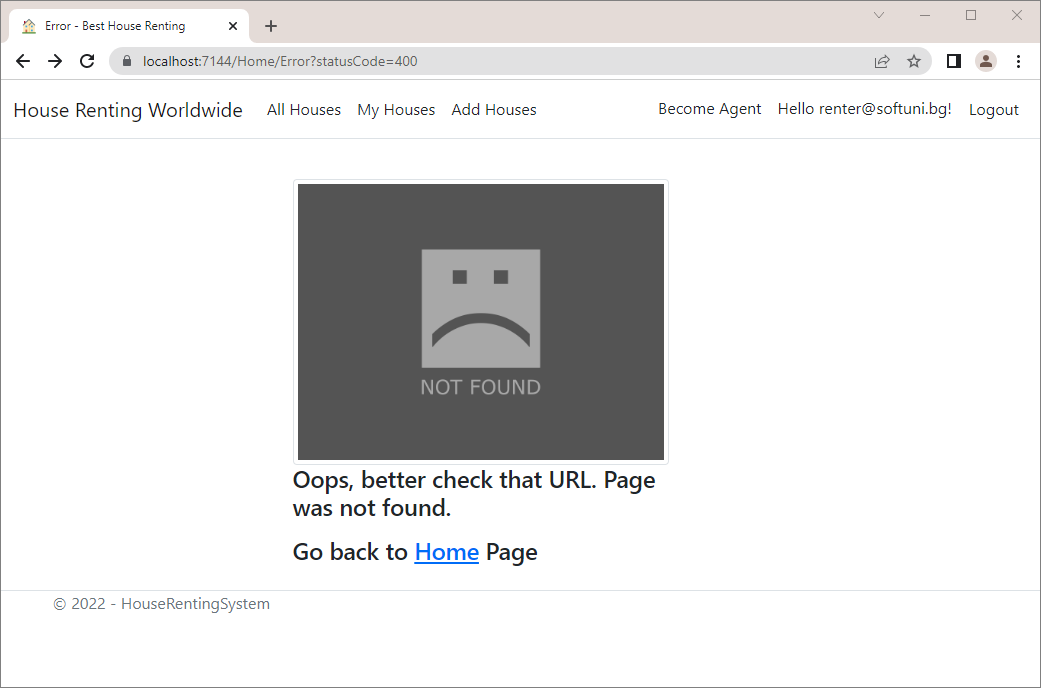
The **controller action** should look like this:

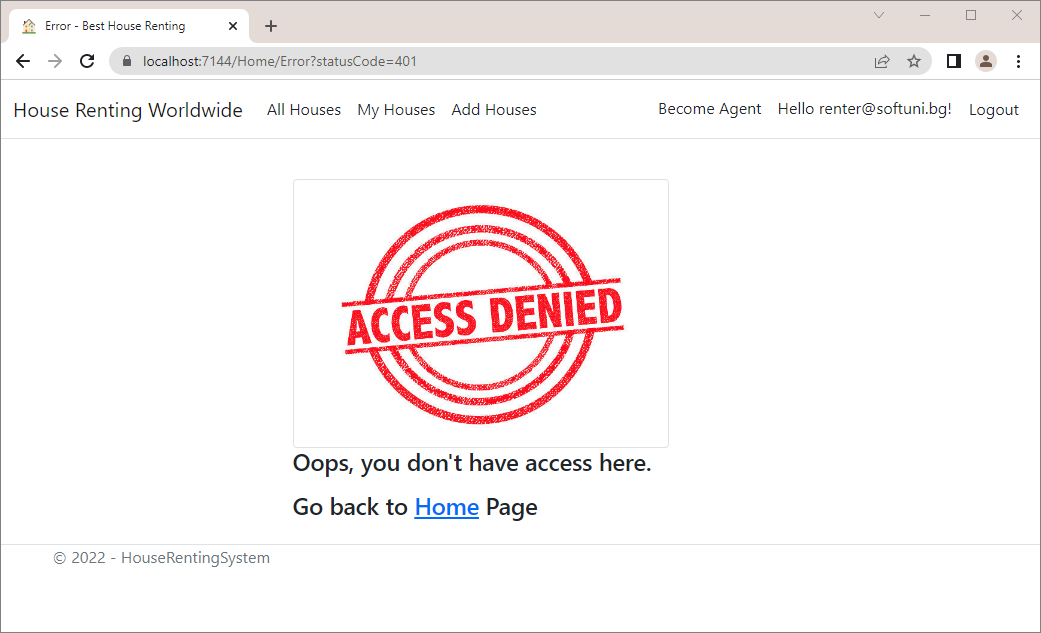


Try to **leave the house** you rented on the previous task by going to "/Houses/Leave/{id}" with the **house** id. If it is successful, you should be again **redirected** to the "My Houses" **page** and the House **record** should have a NULL value on RenterId. Don't forget to test the **model errors**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

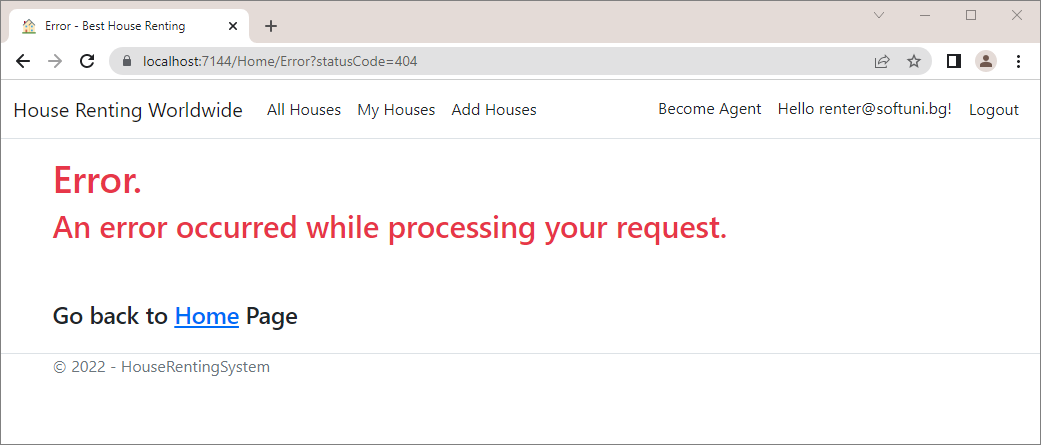
## Implement Custom Error Pages

In this task, we will see how to **implement custom error pages** for "400 Bad Request" and "401 Unauthorized" **errors** in "Production" **environment**, so that errors look more user-friendly. They will look like this:

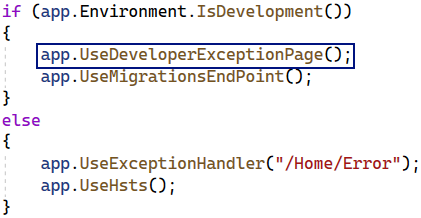




The **error page** for **other status code** will be the following:



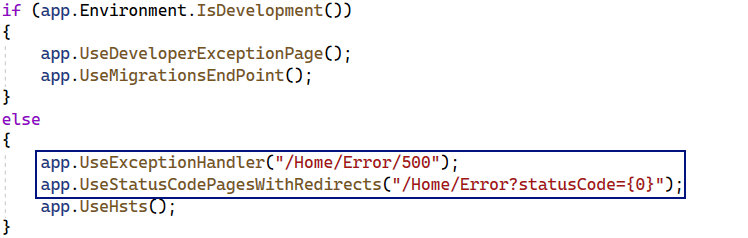
To make **custom error pages**, we should first **modify the app middleware**. Go to the Program.cs:



Note that when we are in "Development" **environment**, we have a **middleware** for using **developer exception** **pages**. When we are not, we **invoke** the Error() **method** of the HomeController **class**, which **returns an error view**.

In our case, however, we want to **set custom error pages** when we are not in "Development", depending on the **error status code**.

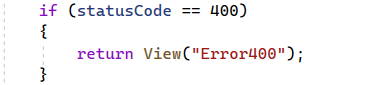
For this reason, we should **change the second middleware**, responsible for **status code pages**. We want to be **redirected** again to "/Home/Error" but also to have the **status code as an** **URL query parameter**. Note that the "500 Internal Server Error" is treated differently than other status code pages, so we should **set a special** **exception handler** for it. Do it like this:



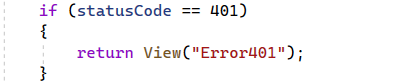
Now our Error() **method** in the HomeController should accept a **status code** like this:



Then, if we have a **status code** "400 Bad Request", we will return the "Error400.cshtml" **view** (which we will create later):



If we have a **status code** "401 Unauthorized", we will return the "Error401.cshtml" **view**:



If none of the above cases happen, we should just return the "Error.cshtml" **view**:



Now let's **create the view files**. In the "/Views/Home" **folder** create the "Error.cshtml", "Error400.cshtml" and "Error401.cshtml" **views**:

**Remove** the "Error.cshtml" file from the "/Views/Shared" **folder**, as we won't need it. Also, **remove** the ErrorViewModel **class** from the "/Models" **folder**:

Write the "Error.cshtml" **view** like this

|  |
| --- |
| @{  ViewData["Title"] = "Error";  }  <h1 class="text-danger">Error.</h1>  <h2 class="text-danger">An error occurred while processing your request.</h2>  <br /><br />  <div class="text-left h4"> Go back to <**a** **asp-controller**="Home" **asp-action**="Index">Home</**a**> Page</div> |

The "Error400.cshtml" **view** should be the following:

|  |
| --- |
| @{  ViewData["Title"] = "Error";  }  <div class="row">  <div class="col-3"></div>  <div class="col-6">  <br>  <div class="row">  <div class="col-10">  <img src="https://www.publicdomainpictures.net/pictures/280000/velka/not-found-image-15383864787lu.jpg" alt="Alternative image" class="img-thumbnail">  <div class="text-left h4">  <p>Oops, better check that URL. Page was not found.</p>  </div>  <div class="text-left h4"> Go back to <**a** **asp-controller**="Home" **asp-action**="Index">Home</**a**> Page</div>  </div>  </div>  </div>  <div class="col-3"></div>  </div> |

Get the **image URL** from here:

|  |
| --- |
| <https://www.publicdomainpictures.net/pictures/280000/velka/not-found-image-15383864787lu.jpg> |

The last view is "Error401.cshtml". Write it like this:

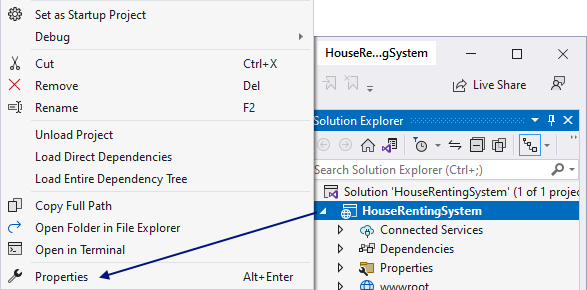
|  |
| --- |
| @{  ViewData["Title"] = "Error";  }  <div class="row">  <div class="col-3"></div>  <div class="col-6">  <br>  <div class="row">  <div class="col-10">  <img src="https://media.istockphoto.com/photos/rubber-stamp-picture-id545992942?k=6&m=545992942&s=612x612&w=0&h=XD1kddQeDrbG9DEQQxGNf6\_3snw4gdFZv8GqSFta6Is=" alt="Alternative image" class="img-thumbnail">  <div class="text-left h4">  <p>Oops, you don't have access here.</p>  </div>  <div class="text-left h4"> Go back to <**a** **asp-controller**="Home" **asp-action**="Index">Home</**a**> Page</div>  </div>  </div>  </div>  <div class="col-3"></div>  </div> |

The **image URL** for this page is the following:

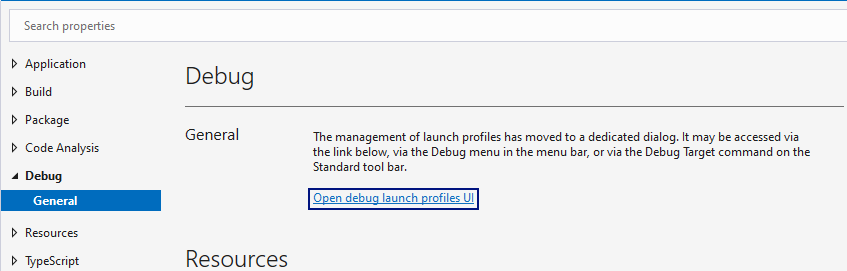
|  |
| --- |
| <https://media.istockphoto.com/photos/rubber-stamp-picture-id545992942?k=6&m=545992942&s=612x612&w=0&h=XD1kddQeDrbG9DEQQxGNf6_3snw4gdFZv8GqSFta6Is=> |

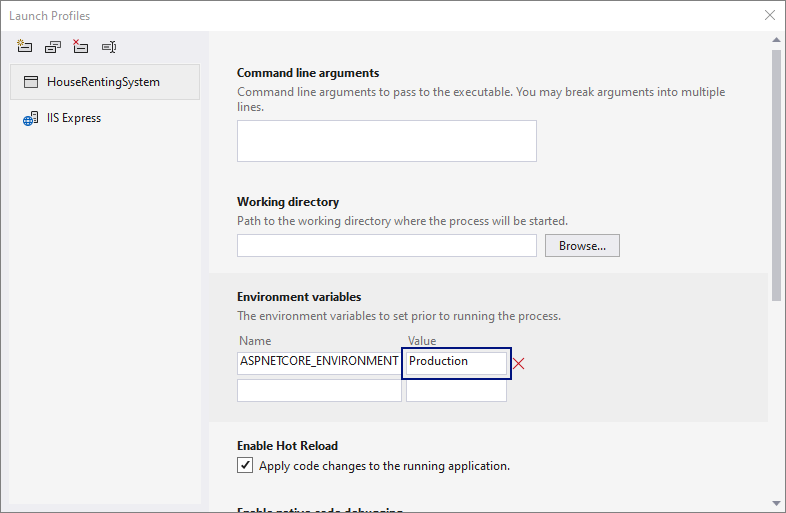
Now let's **change the** **environment** to "Production", so that we can **test the error pages** we created. To do this, we need to **change** the ASPNETCORE\_ENVIRONMENT **variable value**.

**Right-click** on the **project** in **Visual** **Studio** and go to [Properties]:



Then, go to the [Debug] **menu** and change the ASPNETCORE\_ENVIRONMENT **variable value** from "Development" to "Production":





Now **run the app**. Go to a **non-existing page** like <https://localhost:44342/Houses/Details/1000> to see the "400 Bad Request" **error page**. You should be **redirected** to <https://localhost:44342/Home/Error?statusCode=400>.

Test the "401 Unauthorized" **error page** by going to [https://localhost:44342/Houses/Edit/{id}](https://localhost:44342/Houses/Edit/%7bid%7d) with a **house id** **from the database**, but you **should not be the creator of the house**. You should be **redirected** to <https://localhost:44342/Home/Error?statusCode=401>.