Lab: ASP.NET Web API - Online Shop Services

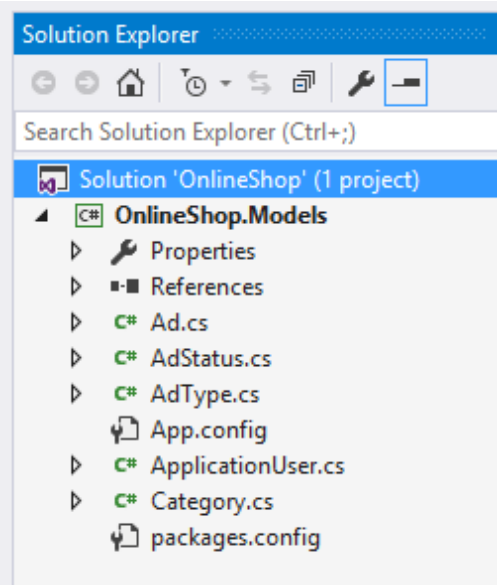
This document defines the lab assignment from the "Web Services and Cloud" Course @ Software University.

Assignment

We're writing a mobile application through which you can **buy/sell** all kinds of things. The Android mobile team is nearly ready with the UI. However, they still need solid **REST services** to consume and test the application.
As part of the back-end team, you're going to write them!

NOTE: Disable AdBlock if you're going to test the services in the browser - it might block some outgoing requests.

1. Define the Data Layer



You are given a solution with a single project, holding the models of your to-be REST application. In short, there are **users**, **ads**, **ad types** and **categories**.

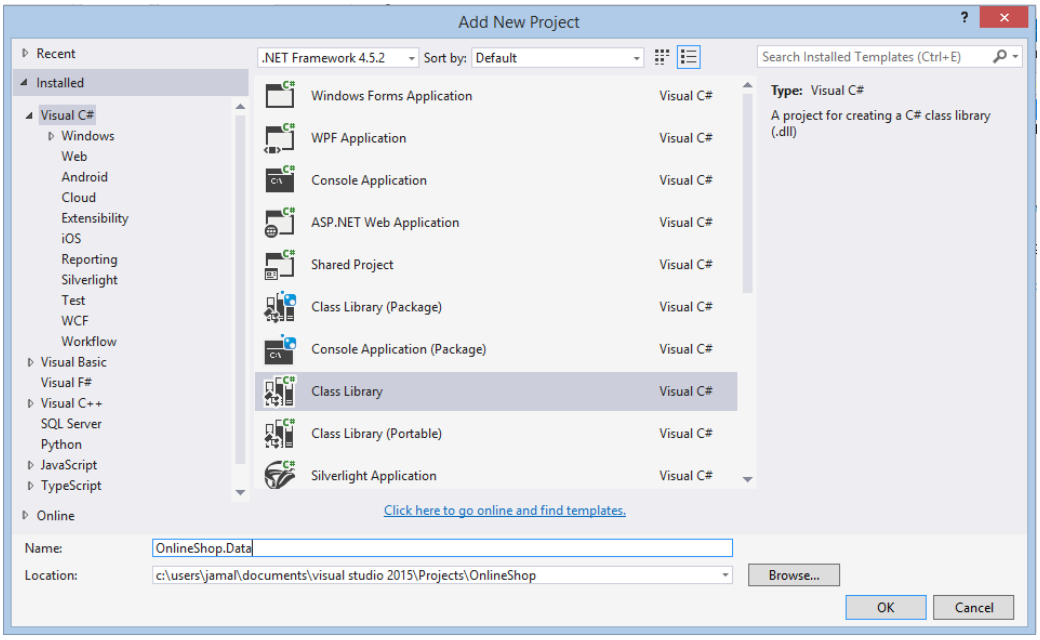
Users can post several ads with a **name**, **description**, **price**, **type** and a **set of categories**.

Ads can have a type (**Normal**, **Premium** or **Diamond**) - each with its specific **price per day**.

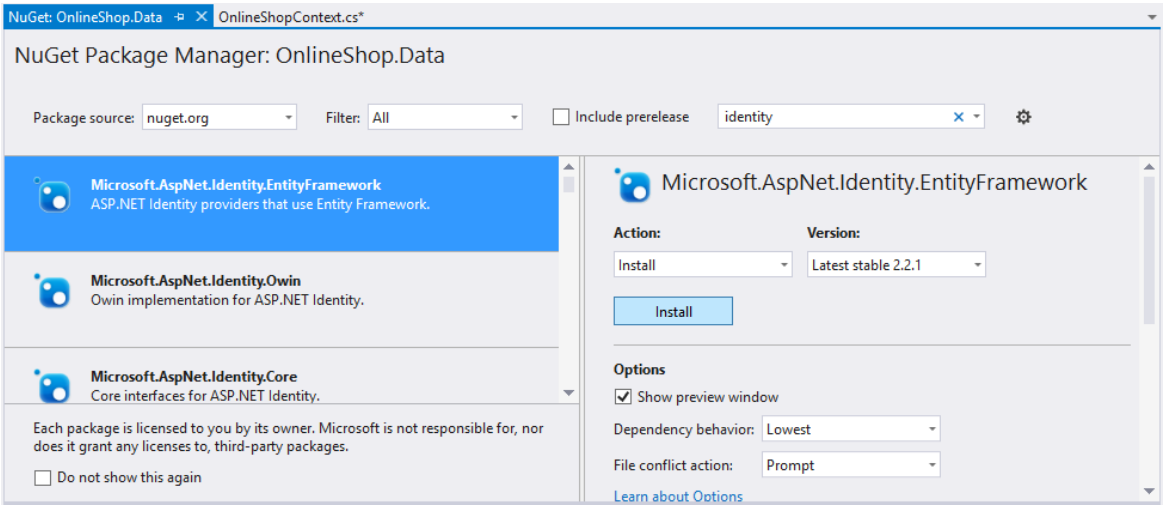
Ads also have a status - **open** or **closed**.

Notice how **ApplicationUser** inherits **IdentityUser** - it's a built-in class in ASP.NET which provides username, password, email and other ready utilities for our user. That way we do not need to write them from scratch.

We've got the models, it's time we define our data layer. Create a new project:



1. Create a new ADO.NET model with **EF Code First** (refer to this lab document if you're having difficulty).
2. Install the **Microsoft.AspNet.Identity.EntityFramework** package from NuGet - it provides functionality for auto-generating user tables (with columns for id, username, password, etc.) in the database.



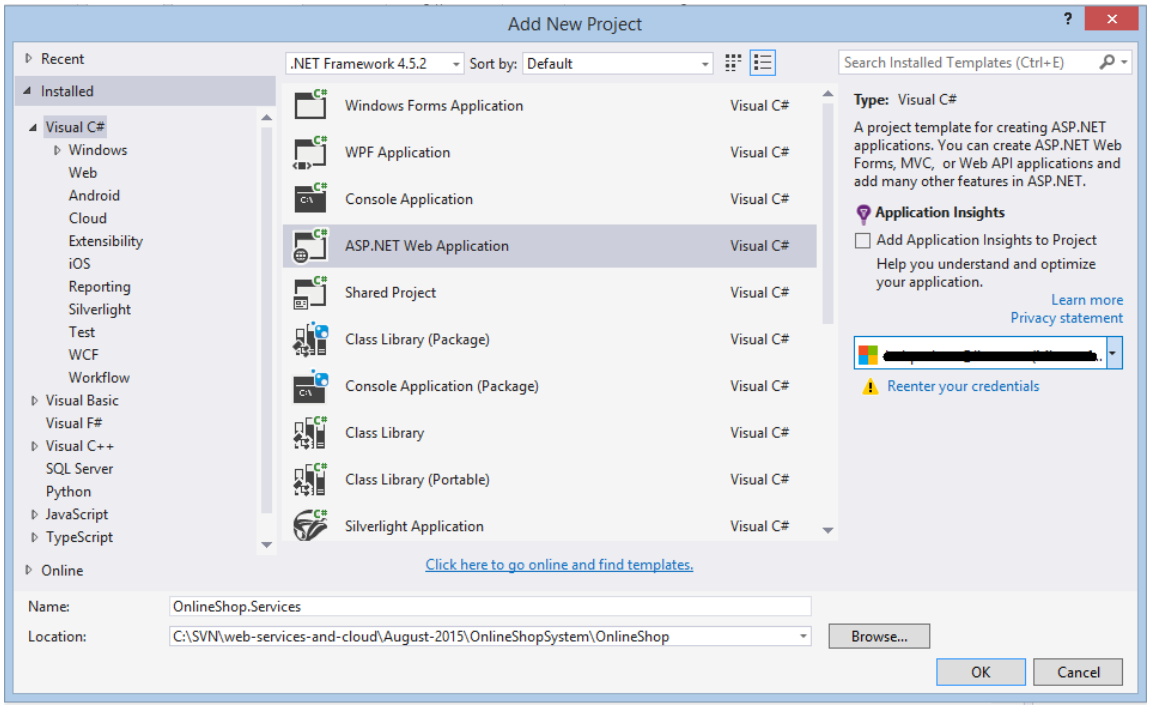
3. Fix the connection string to connect to **localhost**, not (localDB)/v11.
4. Edit the generated OnlineShopContext class (or however you called it) to inherit **IdentityDbContext<T>** where **T** is our application user. That gives our context a **DbSet<ApplicationUser>** out of the box.

```
namespace OnlineShop.Data
{
    using Microsoft.AspNet.Identity.EntityFramework;
    using Models;

    public class OnlineShopContext : IdentityDbContext<ApplicationU>
    {
        public OnlineShopContext()
            : base("OnlineShopContext")
        {
        }

        // TODO: Add DB sets for ads, ad types and categories
    }
}
```

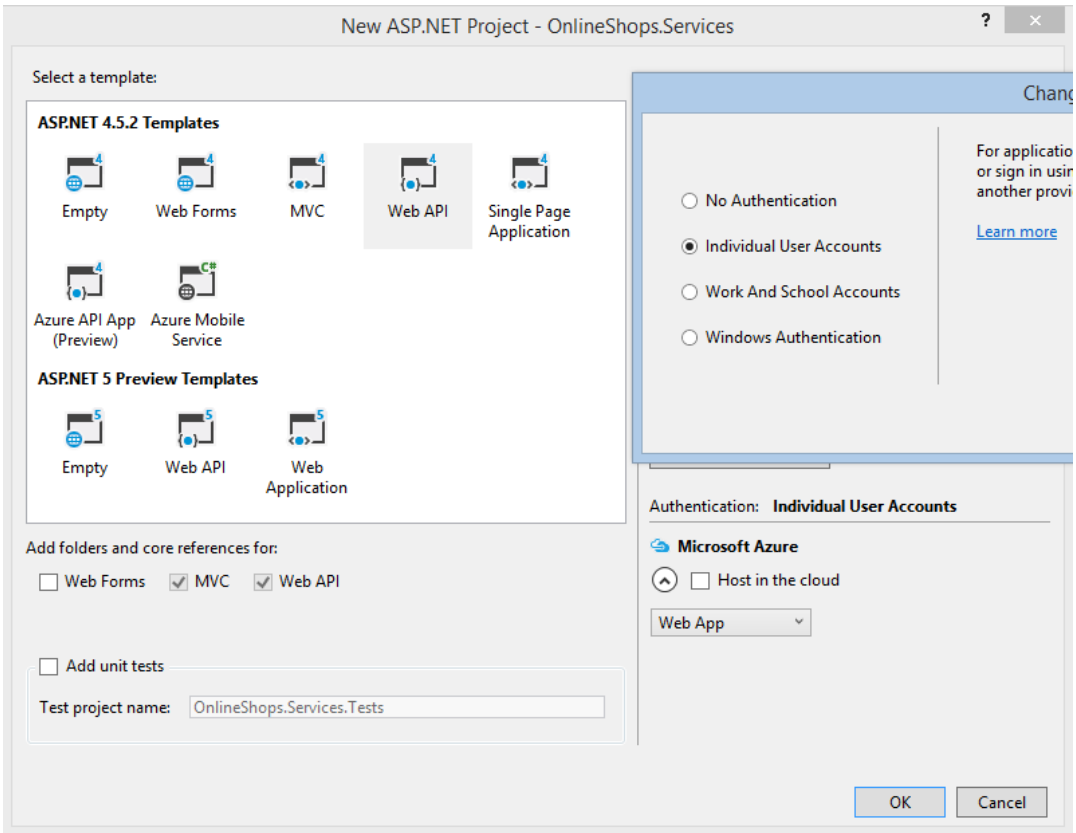
5. Enable EF Code First migrations by running the **Enable-Migrations -EnableAutomaticMigrations** command in the Package Manager Console. Make sure the Data project is selected.
6. Before we start writing any services, go to **Configuration.cs** and seed some sample data into the database. You may use this code: http://pastebin.com/LQKFS00N

2. Create the Services

Let's create a new ASP.NET Web App:

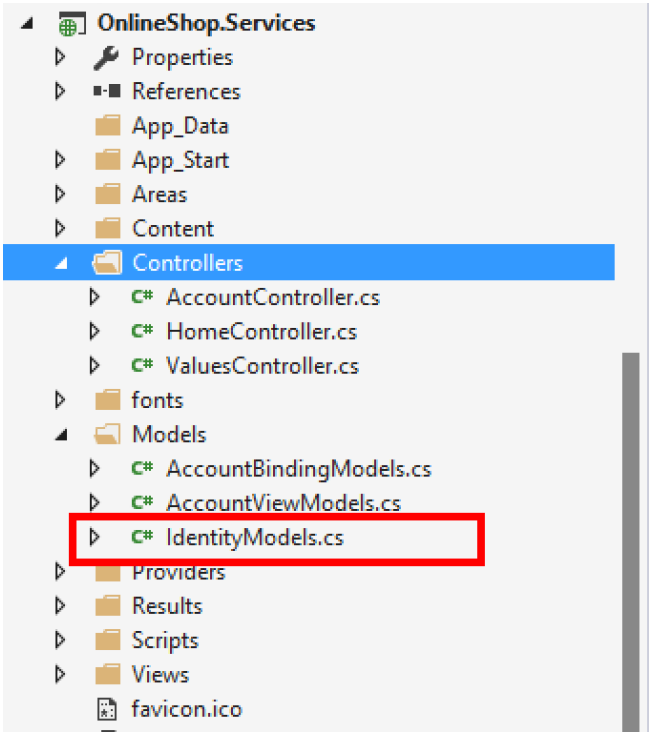


Select Web API with **Individual User Accounts**:

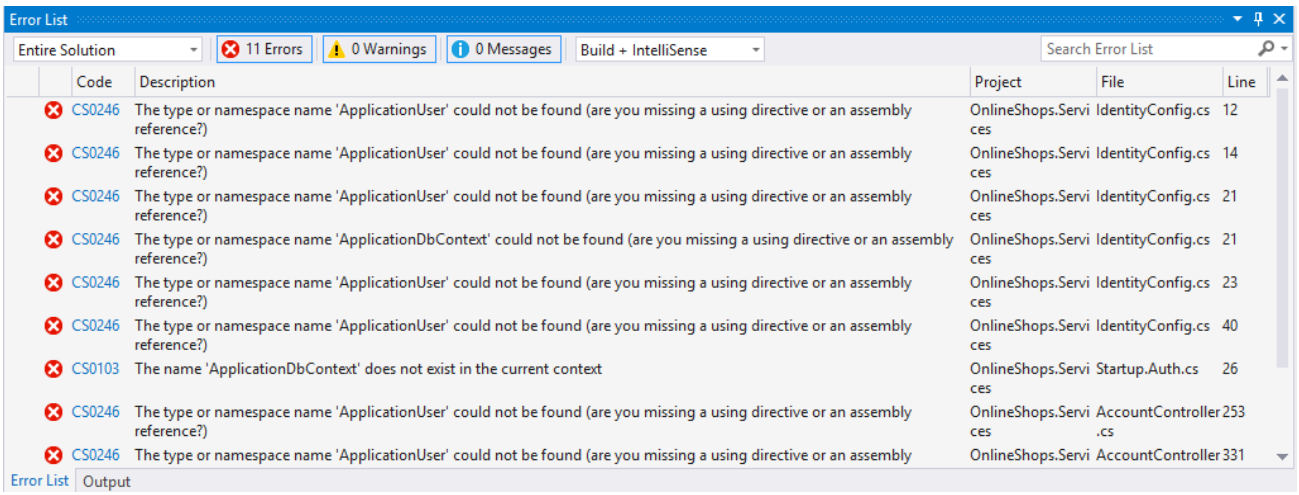


Set it as startup and let's begin.

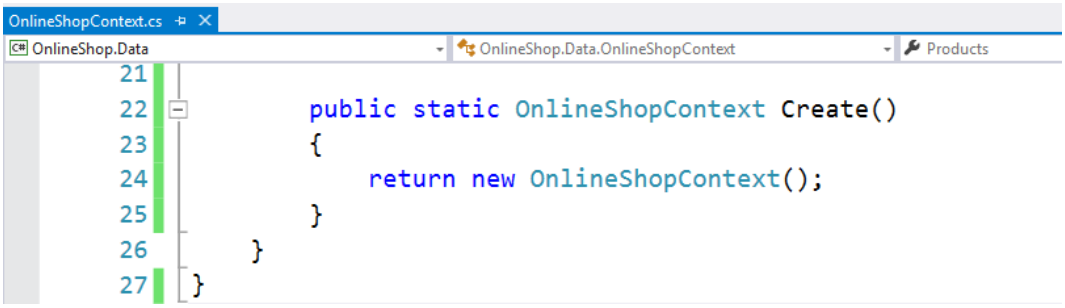
1. We have our usual controllers, models and all other default stuff. There's also an **IdentityModels** class which holds default **ApplicationUser** and **ApplicationDbContext** classes. We have our own (in the Data project), so **delete** this class.



2. Fix all broken references that have just appear. Replace ApplicationDbContext with **OnlineShopContext** and reference **ApplicationUser** from the Models project.

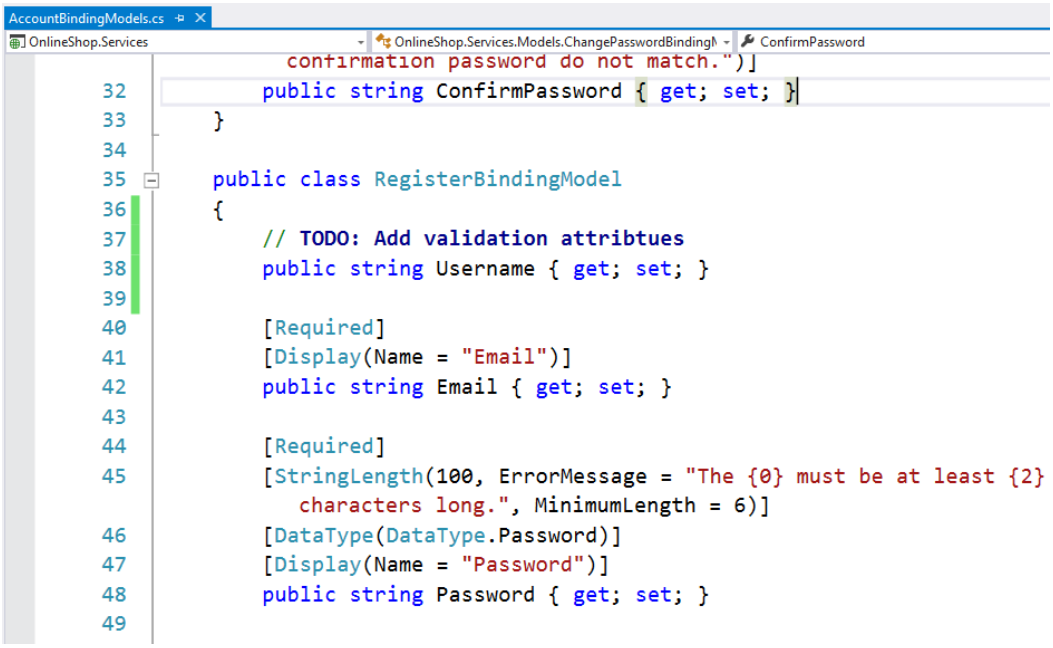


3. In one place Web API calls a static **Create()** method from the context. Let's add it:



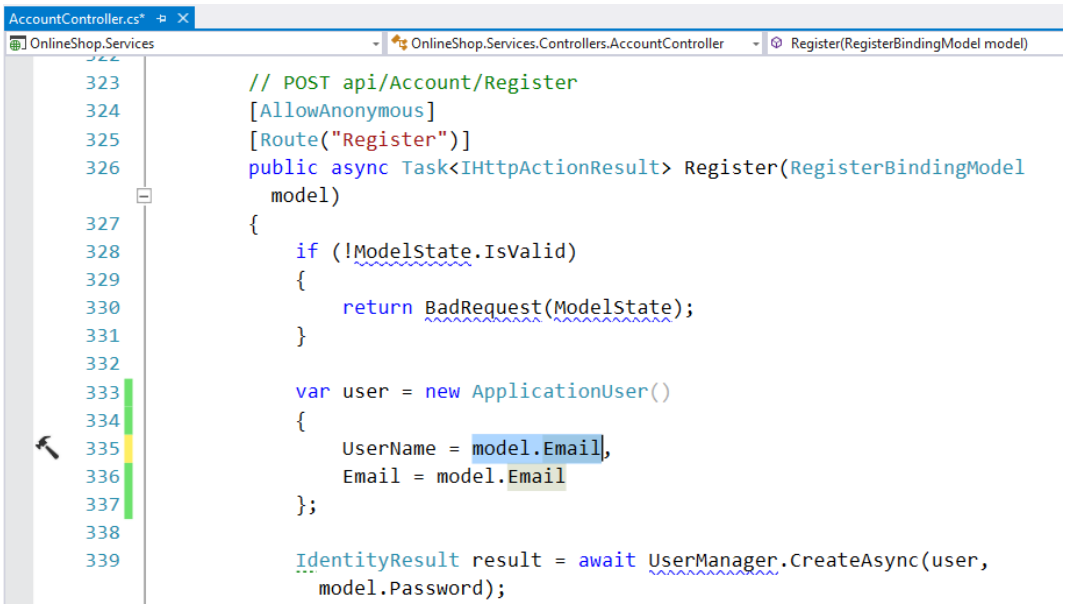
4. Copy the **connection string** from the **Data** project to **Web.config**

5. The default user registration does not need a username - it uses email. Let's change that - go to the **AccountBindingModels.cs** class and add a Username property. Validate it with attributes by making it mandatory and at least 5 symbols long.



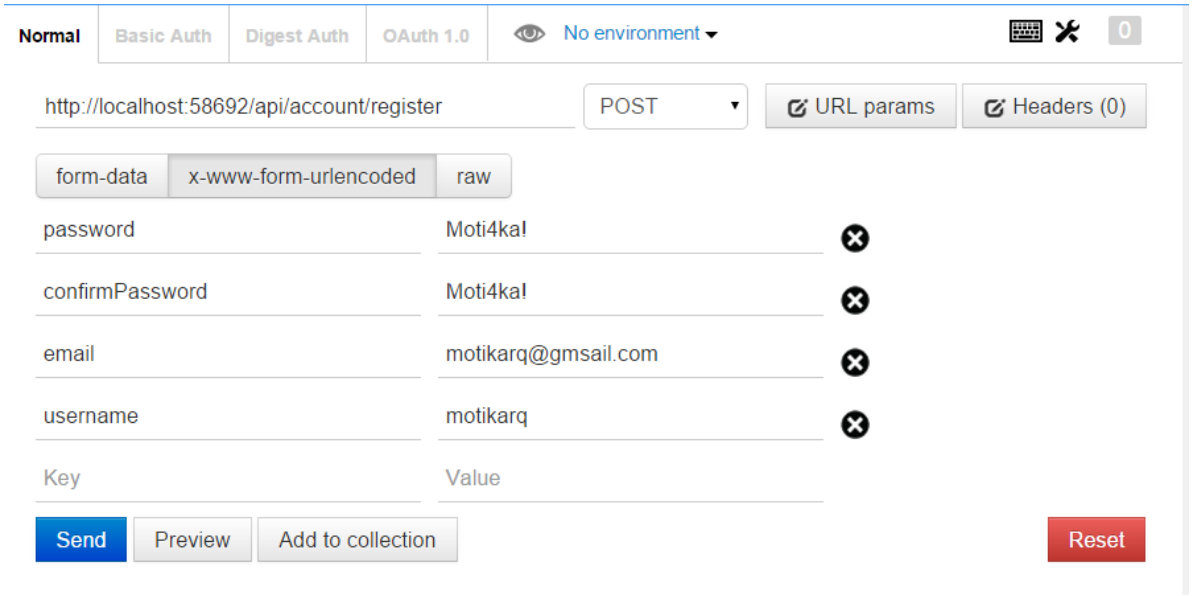
```
AccountBindingModels.cs
OnlineShop.Services | OnlineShop.Services.Models.ChangePasswordBindingM | ConfirmPassword

            confirmation password do not match.")]
32          public string ConfirmPassword { get; set; }
33      }
34
35      public class RegisterBindingModel
36      {
37          // TODO: Add validation attribtues
38          public string Username { get; set; }
39
40          [Required]
41          [Display(Name = "Email")]
42          public string Email { get; set; }
43
44          [Required]
45          [StringLength(100, ErrorMessage = "The {0} must be at least {2}
              characters long.", MinimumLength = 6)]
46          [DataType(DataType.Password)]
47          [Display(Name = "Password")]
48          public string Password { get; set; }
49
```

6. Go to the **Register** action in the **Account controller** and set the **username** from the model to application user username.

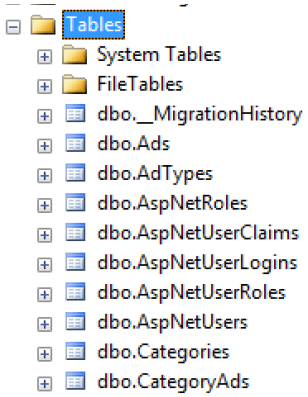


```
AccountController.cs*
OnlineShop.Services | OnlineShop.Services.Controllers.AccountController | Register(RegisterBindingModel model)

323         // POST api/Account/Register
324         [AllowAnonymous]
325         [Route("Register")]
326         public async Task<IHttpActionResult> Register(RegisterBindingModel
              model)
327         {
328             if (!ModelState.IsValid)
329             {
330                 return BadRequest(ModelState);
331             }
332
333             var user = new ApplicationUser()
334             {
335                 UserName = model.Email,
336                 Email = model.Email
337             };
338
339             IdentityResult result = await UserManager.CreateAsync(user,
              model.Password);
```

7. Build the **Services** project. Run it and test the register endpoint.

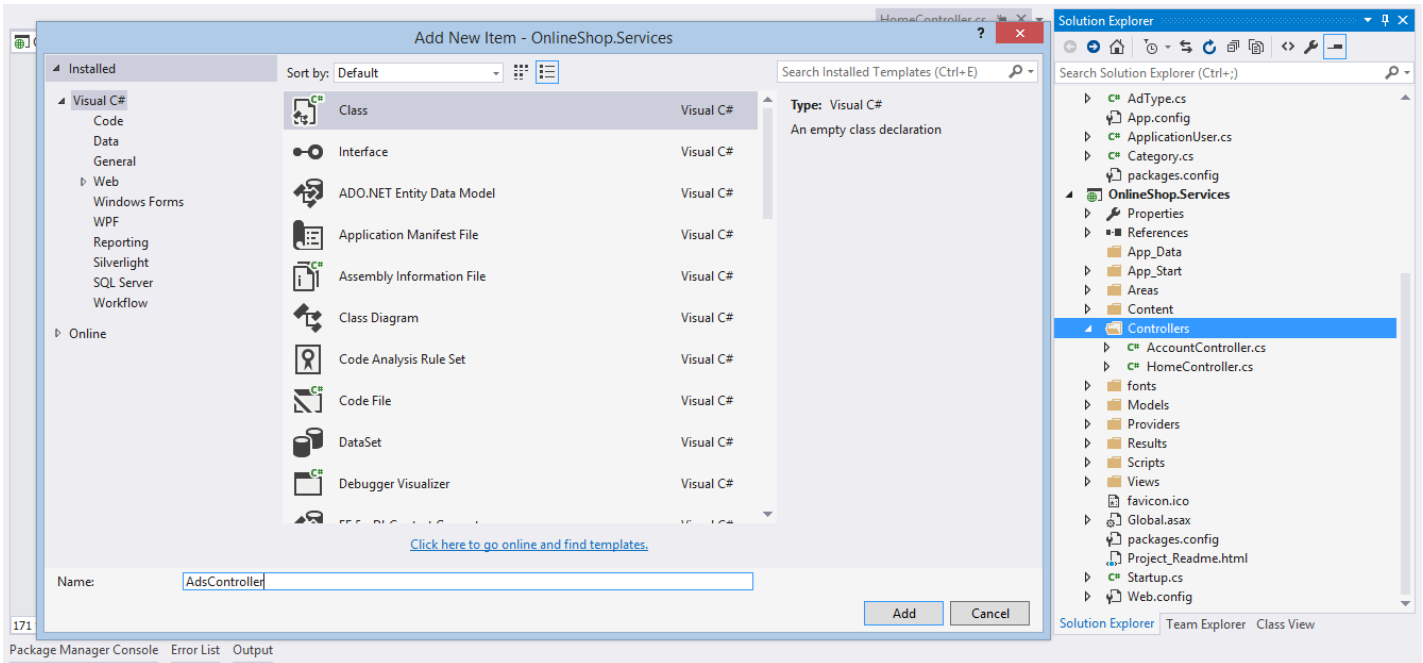


8. Check the database - the following tables should be present:



3. Endpoints

It's time we started writing services for real. There's a sample **ValuesController** which shows how endpoints are made - we can delete it and create our very own **AdsController** and **CategoriesController**.



1. Creating The Controllers

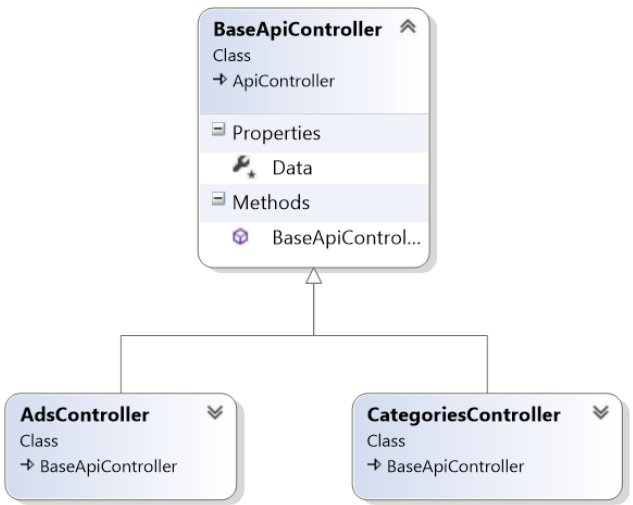
The **AdsController** will manage ads and the **CategoriesController** will manage categories. What they both have in common is that will use **OnlineShopContext** to connect to the database and perform CRUD operations.

In OOP, when 2 or more classes have similar functionality, we **extract it in a base class**.

1. Create a class **BaseApiController** and inherit the built-in **ApiController** class. The BaseApiController will keep the OnlineShopContext as a **protected** property (protected means only the class and its children can access it).
   We create a constructor which takes 1 parameter **OnlineShopContext** and sets it to the property. We also create a second constructor without parameters - it passes an **OnlineShopContext** instance to the other constructor.

```
namespace OnlineShop.Services.Controllers
{
    using System.Web.Http;
    using Data;

    public class BaseApiController : ApiController
    {
        public BaseApiController()
            : this(/* pass new OnlineShopContext instance */)
        {
        }

        public BaseApiController(OnlineShopContext data)
        {
            // TODO: Set data to property
        }

        protected OnlineShopContext Data { get; set; }
    }
}
```

2. Create **AdsController** and **CategoriesController**. Both should inherit our **BaseApiController** class. That way they will have the **OnlineShopContext** property (we basically reuse code this way).



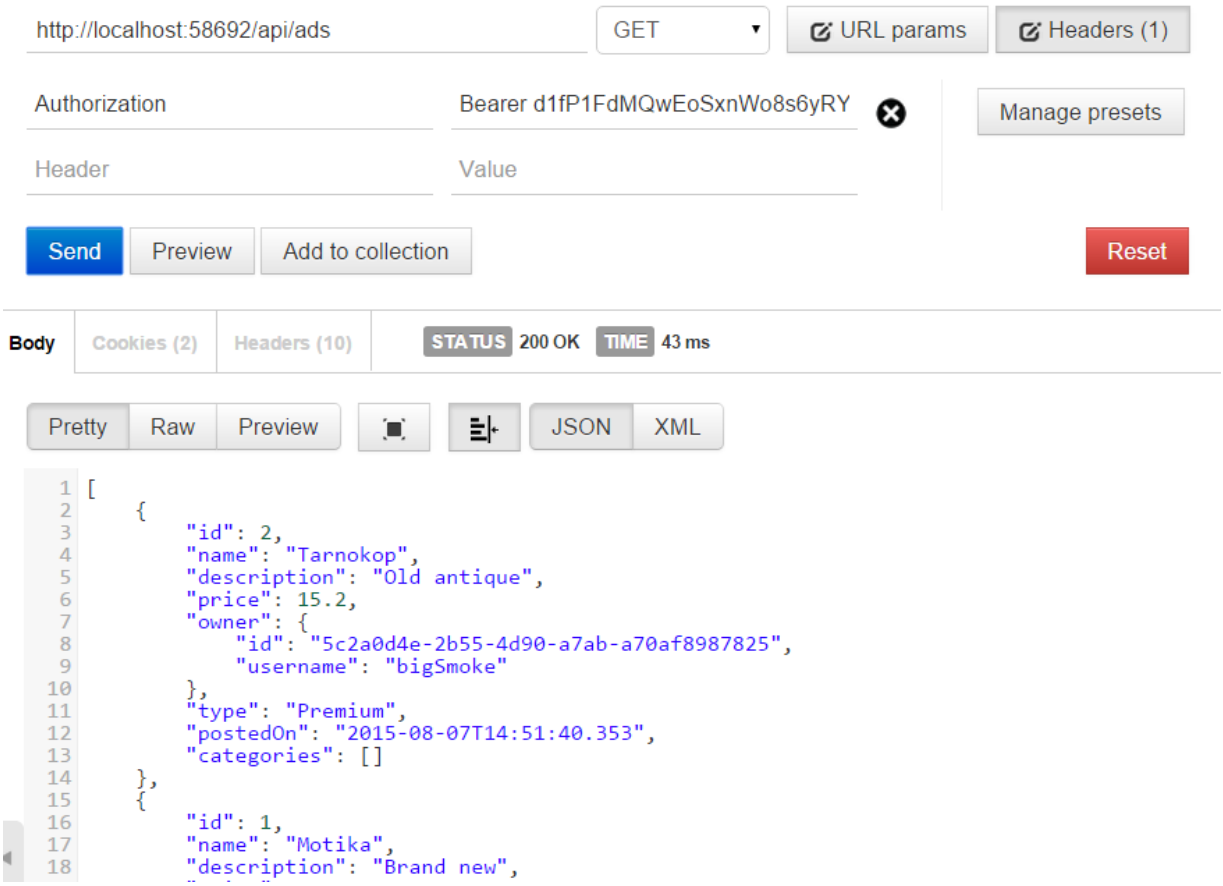
2. Getting Open Ads

Let's start by making an endpoint (**GET `api/ads`**) for getting all **open ads**.

1. Create a method (also called action) in the **AdsController**, called **GetAds()**. The method will return IHttpActionResult - an abstact type which allows us to return **status code + data** together.

```
public class AdsController : BaseApiController
{
    public IHttpActionResult GetAds()
    {
        var ads = this.Data.Ads
            /* Build LINQ query */;

        return this.Ok(/* ads */);
    }
```

2. Get all **open ads**, **ordered** by type (first **diamond**, then **premium**, then **normal**) and then by **date of post** (from most recent to oldest).
3. Select each ad's **id**, **name**, **description**, **price**, **owner id** and **username**, **type**, **date of post** and its **categories** (id and name). The output should be in the following format:



4. If the returned JSON in **PascalCase**, make the following configuration in **WebApiConfig.cs**:

```
WebApiConfig.cs
OnlineShop.Services  |  OnlineShop.Services.WebApiConfig  |  Register(HttpConfiguration config)
26                  defaults: new { id = RouteParameter.Optional }
27              );
28
29              config.Formatters.JsonFormatter
30                  .SerializerSettings.ContractResolver =
31                  new CamelCasePropertyNamesContractResolver();
32          }
33      }
34  }
```

3. Creating New Ads

Let's define an endpoint for creating new ads - **`POST api/ads`**.

1. Create a new action called **CreateAd()** which takes a **CreateAdBindingModel** as parameter. A binding model is a class to which Web API maps request data - that way all data is stored in a single C# object.

```
AdsController.cs
OnlineShop.Services  |  OnlineShop.Services.Controllers.AdsController  |  GetAds()
33
34          public IHttpActionResult CreateAd
              (CreateAdBindingModel model)
35          {
36
37          }
38      }
39  }
```

2. Let's define what data we need when an ad is created - ad **name**, **description**, **type id** (normal, premium, etc.), **price** and a **collection of category IDs**. Create the class in the **OnlineShop.Services.Models** namespace. **Note**: Binding models should contain only data the user sends.

```
public class CreateAdBindingModel
{
    // TODO: Add validation attributes
    public string Name { get; set; }

    public string Description { get; set; }

    public int TypeId { get; set; }

    public decimal Price { get; set; }

    public IEnumerable<int> Categories { get; set; }
}
```

3. Validate if the user is logged (through **this.User.Identity**) and if the binding model is valid (through **this.ModelState**).

```
public IHttpActionResult CreateAd(CreateAdBindingModel model)
{
    var userId = this.User.Identity.GetUserId();
    if (userId == null)
    {
        // TODO: User is not logged in => return a proper
          status code
    }

    if (!this.ModelState.IsValid)
    {
        return this.BadRequest(this.ModelState);
    }

    var ad = new Ad() { // TODO: Set data from model };

    // TODO: Persist changes to DB
    // Return created post with a proper status code
```

4. Perform the following validations:
   - The user has sent data (i.e. there is a binding model)
   - There's **at least 1** category and **no more than 3**
   - The sent **category IDs** and **type ID** are real
5. Create the **Ad** entity. Make sure all **required data** is set. Persist the changes to the database.
6. Copy the return type from the previous action here. It should be something similar:

```
OnlineShop.Services.Controllers.AdsController    CloseAd(int id
this.Data.SaveChanges();

return this.Ok(new
{
    ad.Id,
    ad.Name,
    ad.Description,
    ad.Price,
    Owner = new
    {
        ad.Owner.Id,
        Username = ad.Owner.UserName,
    },
    Type = ad.Type.Name,
    ad.PostedOn,
    Categories = ad.Categories
            .Select(c => c.Name)
});
```

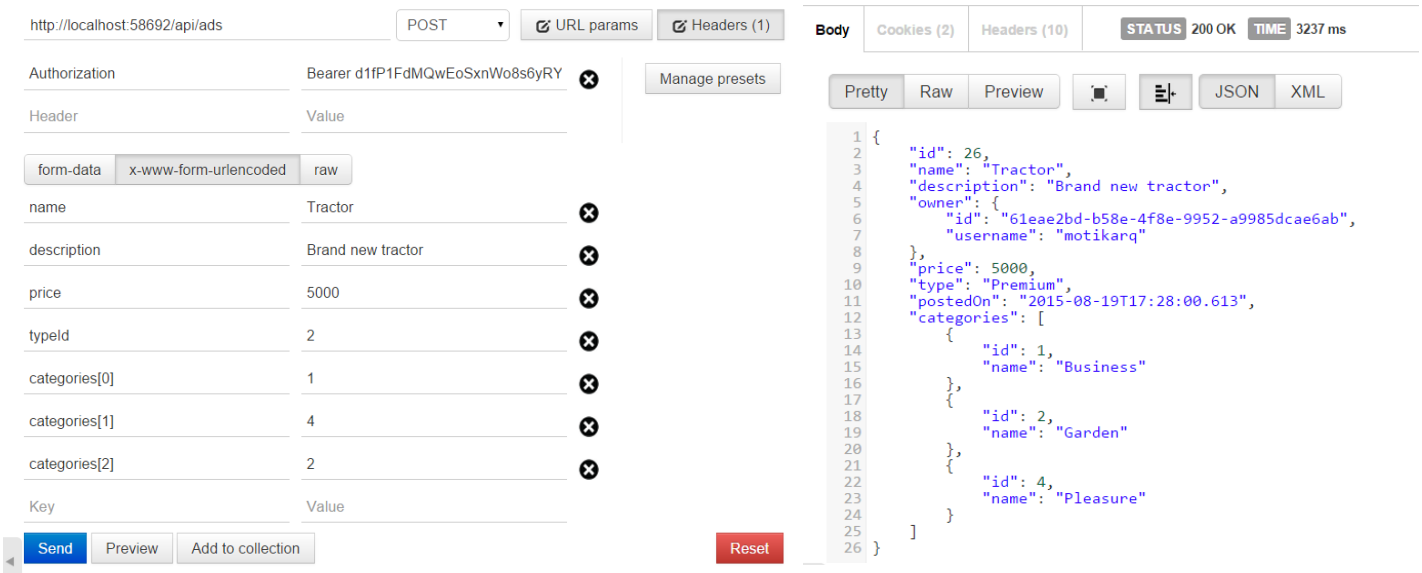
However, this will produce **NullReferenceException**, because the **ad entity** does not have the **Owner** navigation property loaded (it's null).

What must be done is **fetch** the newly created ad from the database by **id**.

We could use the **.Find()** method for this - it's fast but it **returns the entire entity** and we don't want that - we want only certain data from the ad. So we do the following:

```
OnlineShopContext.cs   AdsController.cs*
OnlineShop.Services   OnlineShop.Services.Controllers.AdsController   GetCategories(IEnumerable<int> c
            this.Data.Ads.Add(ad);
            this.Data.SaveChanges();

            var result = this.Data.Ads
                .Where(/* Filter by ad.Id */)
                .Select(/* Same projection */)
                .FirstOrDefault();

            return this.Ok(result);
        }
```

7. We **filter the ads by id** (obviously that returns only 1 ad), **select the necessary data** and take the first with **.FirstOrDefault()**.



8. Last but not least - we obviously want only logged users to post ads. We can achieve this by setting the **[Authorize]** attribute over the **CreateAd()** action or over the entire controller. With this done, we can remove the ~~**if (userId == null)**~~ check we made earlier - now Web API will make sure the user is logged in for us. Make sure you send in the request **Headers** key **Authorization** wth value **Bearer {token}**.

```
[Authorize]
public class AdsController : BaseApiController
{
    [AllowAnonymous]
    public IHttpActionResult GetAds()
    {
```

9. Set the **[AllowAnonymous]** attrbute on actions we do not want authorization for.

3.1. Optimizations

Notice how we are repeating code - both **GetAds()** and **CreateAd()** actions are projecting ads into anonymous types. When code is repeated, we normally extract the code into a method - however we cannot do this here, because a method simply cannot return an anonymous type (it's anonymous, remember? It has not type). That's why we create **view models**.

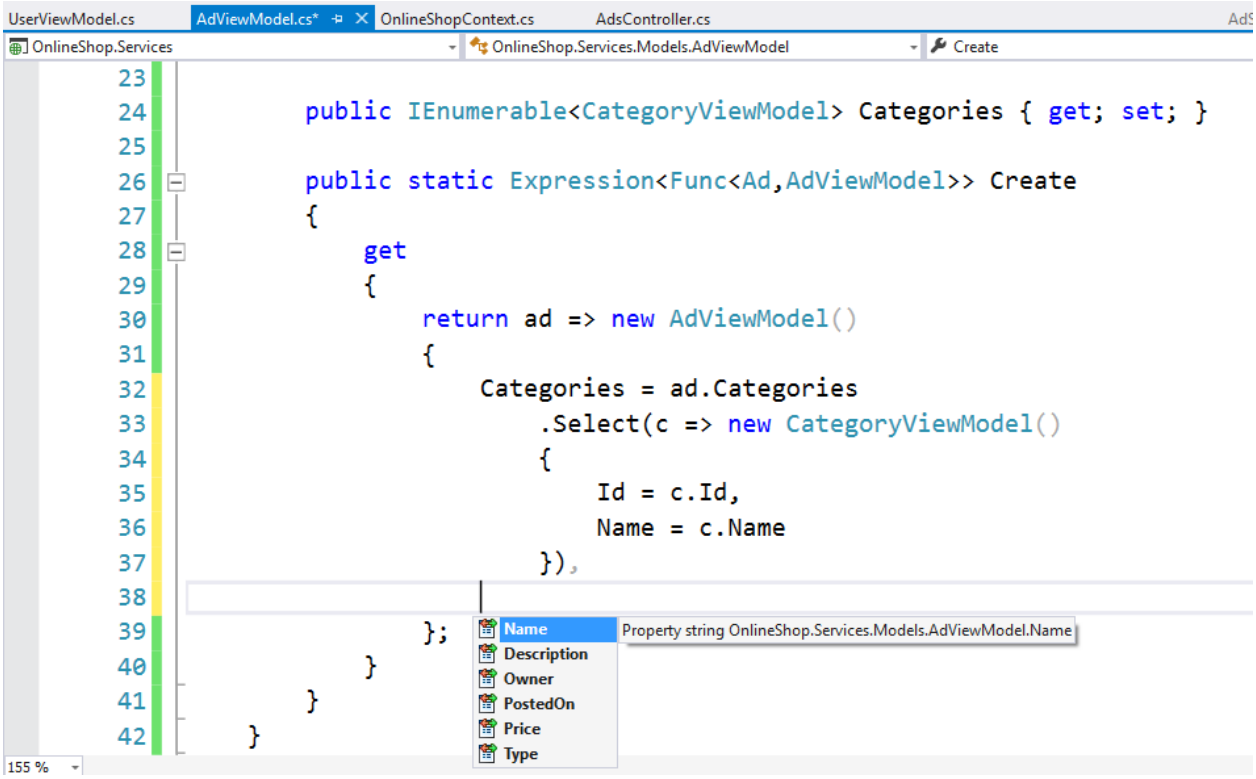
When projecting data we ask the question - what are we projecting? In this case - ads data, categories data and user data (as part of the ad). So we create 3 view models - **AdViewModel**, **UserViewModel**, **CategoryViewModel**. Create them in the **OnlineShop.Services.Models** namespace.

1. Create a **CategoryViewModel** holding **id** and **name**.
2. Create a **UserViewModel** holding **id** and **username**.
3. Create a **AdViewModel** holding all the things we wish to project from an ad - id, name, price, owner (which is a **UserViewModel** holding the **id** and **username** of the ad owner), etc.

```
namespace OnlineShop.Services.Models
{
    public class AdViewModel
    {
        public string Name { get; set; }

        public string Description { get; set; }

        public UserViewModel Owner { get; set; }

        // TODO: Add properties for all of the projected data
    }
}
```

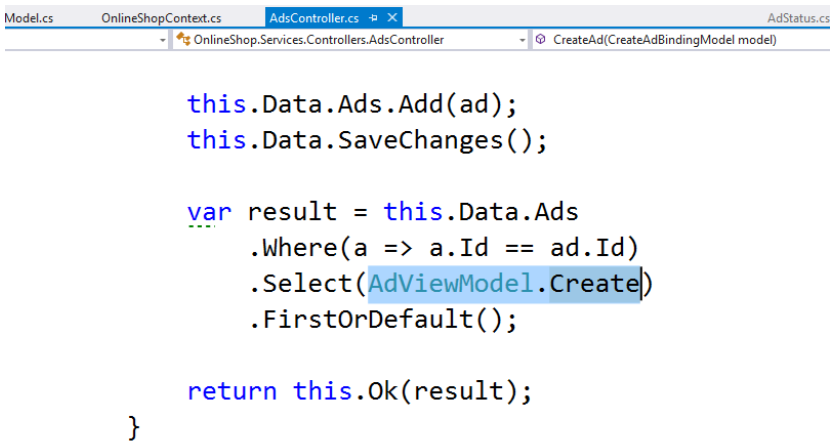
Remember: The view models should hold only the data we wish to project.

4. Now let's create a reusable method we can use to project an **Ad** to an **AdViewModel**. This isn't as simple as it sounds though. The method we're about to create will be passed to **.Select()** in our LINQ query - meaning it will be translated to SQL.
   However, Entity Framework cannot translate any C# method to SQL - it needs to be an **Expression<Func<*>>** - long story short, we need to write something that EF understands how to parse to SQL. It will be **Expression<Func<Ad, AdViewModel>>** - an expression with a method which projects **Ad** into **AdViewModel**.
5. Create a static property with return type **Expression<Func<Ad, AdViewModel>>**. Call it **Create**.

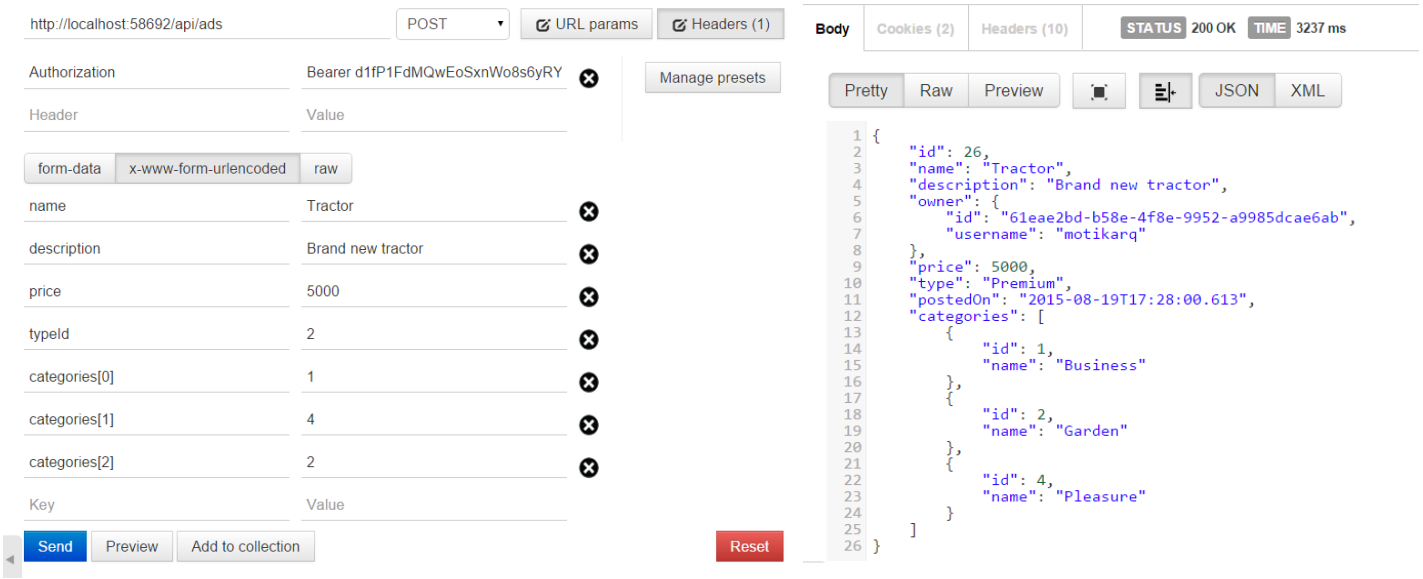


The **Create** property will return a method (a lambda expression) which says "**I accept an Ad and project it into an AdViewModel.**" Transfer all necessary data to the returned **AdViewModel**.
By doing this we make sure EF translates this projection to SQL, rather than materializing the data instead (which would happen if it were a normal method).

6. Pass the **Create** property to **Select()** in the all LINQ queries where we project **Ads** into **AdViewModels**



The result should be the same as before:



7. One last things - it's always a good idea to explicitly tell Web API which HTTP request method the action supports. Set the **[HttpPost]** attribute above the action.

```
[HttpPost]
public IHttpActionResult CreateAd(CreateAdBindingModel model)
{
    if (!this.ModelState.IsValid)
```

Closing Ads

The final endpoints is **PUT `api/ads/{id}/close`** for closing an open ad by id

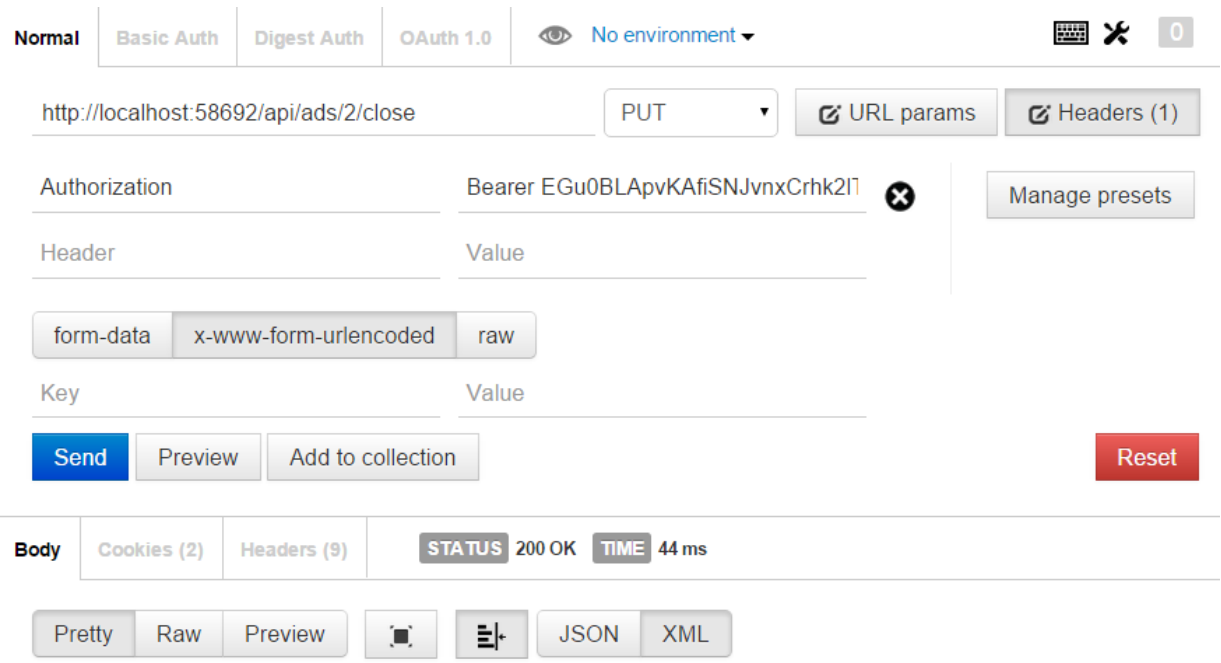
1. Create a **CloseAd()** action which takes **id** as parameter
2. Validate that the **ad exists**
3. Validate that the **logged user** is the **owner** of the ad

```
public IHttpActionResult CloseAd(int id)
{
    Ad ad = null;// TODO: Get ad by id;
    if (ad == null)
    {
        // Ad not found, return a proper response
    }

    string userId = null; // TODO: Get currrently logged user
    if (ad.OwnerId != userId)
    {
        // User is not ad owner, return a proper response
    }
```

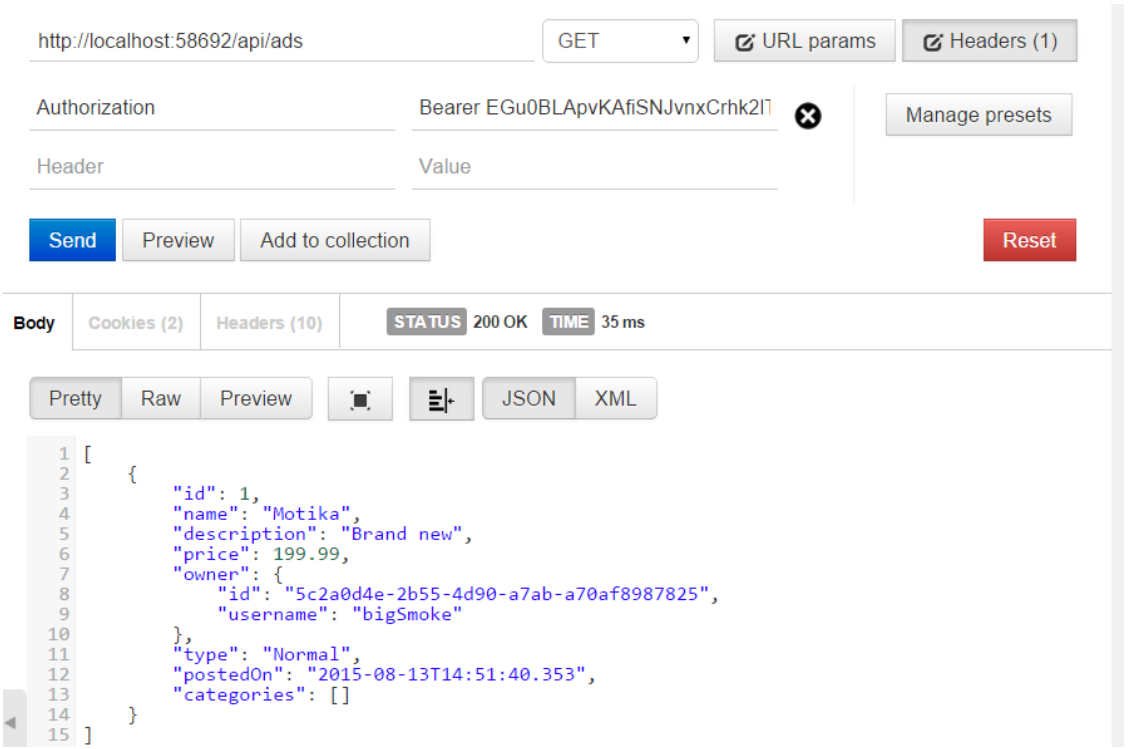
4. If all validations pass, set the ad status to **Closed** and the **ClosedOn** property to the current server date.
5. As of now, the action might be working properly but Web API will not recognize it. We need to explicitly set **the route** it is meant to and the **HTTP request method** it serves. This is done through the attributes **[Route("...")]** and **[Http*]**.

```
// TODO: Allow PUT requests
[Route("api/ads/{id}/close")]
public IHttpActionResult CloseAd(int id)
{
```

6. Test the endpoint as owner of the ad should return **200 OK**:



7. Getting **all open ads** in the future should not show the previously **closed** ads (in this case with **id 2**):




```
[
    {
        "id": 1,
        "name": "Motika",
        "description": "Brand new",
        "price": 199.99,
        "owner": {
            "id": "5c2a0d4e-2b55-4d90-a7ab-a70af8987825",
            "username": "bigSmoke"
        },
        "type": "Normal",
        "postedOn": "2015-08-13T14:51:40.353",
        "categories": []
    }
]
```