# Startup

add services to the container

configure the HTTP request pipeline.

responsible for configuring all kinds of configurations when the application starts.

My understand of the following picture is: add relationships between domain’s interfaces and real class.



# Controller

 all classes that end with the suffix *“Controller”* will become controllers of application. It means they are going to **handle requests and responses**. You have to **inherit this class from the Controller class**, defined in the [namespace](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/namespace) Microsoft.AspNetCore.Mvc.

# Services

First of all, when someone requests data from /api/categories via GET verb, the API needs to return all categories. We can create a **category service** for this purpose.

A service is basically a **class or interface** that **defines methods** to handle some business **logic.**

We need to access the database and return all categories, then we need to return this data to the client.

**A service class is not a class that should handle data access.** There is a pattern called [Repository Pattern](#_Repository_Pattern)that is used to manage data from databases. Our service needs to **talk to a category repository**, to get the list of objects.

Conceptually, a service can “talk” to one or more repositories or other services to perform operations.

Using services, we can isolate the request and response handling from the real logic needed to complete tasks.

# [Repository Pattern](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design" \l "the-repository-pattern" \t "_blank)

When using the Repository Pattern, we define **repository classes**, that basically encapsulate all logic to handle data access. These repositories **expose methods to list, create, edit and delete objects of a given model**, the same way you can manipulate [collections](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections). Internally, these methods talk to the database to perform **CRUD operations**, isolating the database access from the rest of the application.

* We can handle business logic and potential fails without stopping the application process, and we won’t need to use tons of try-catch blocks.

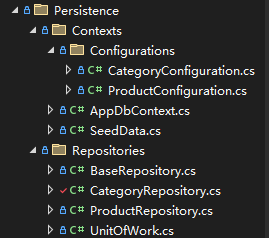
# Persistence →Database and Repository

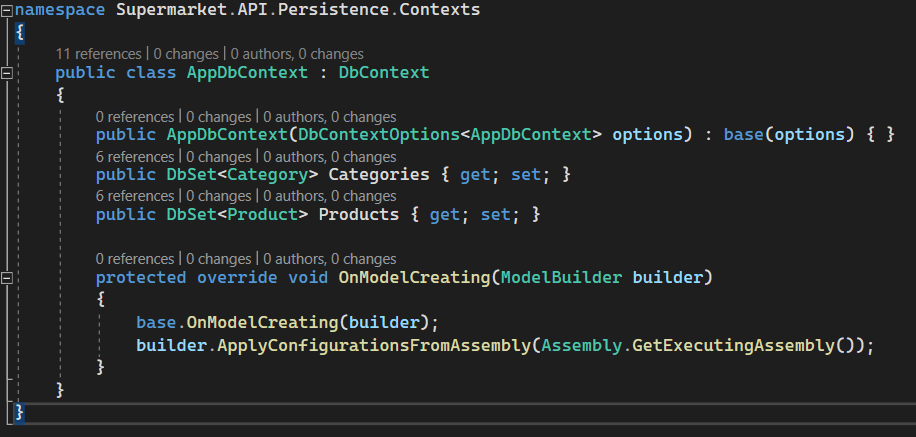
This directory is going to have everything we need **to access the database**, such as repositories implementations.

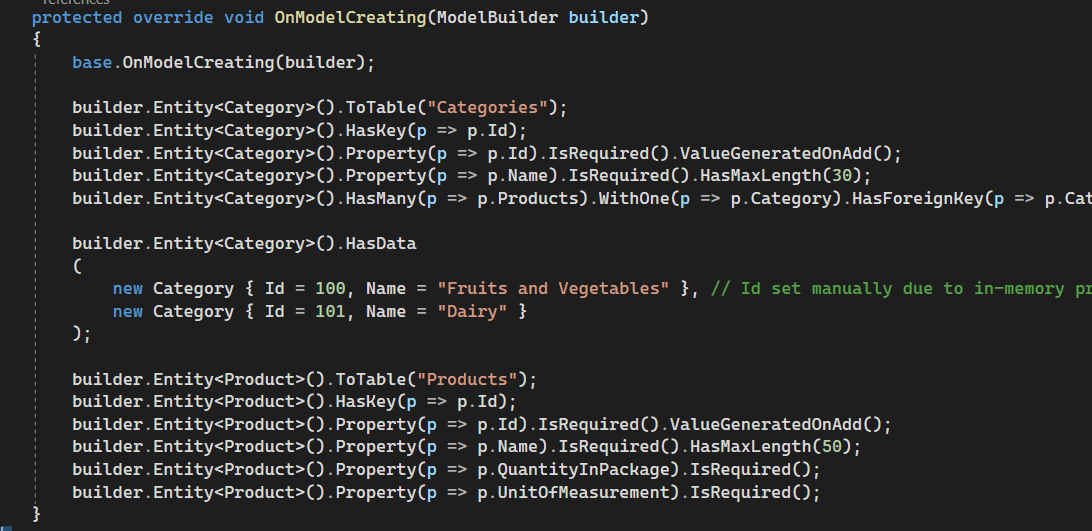
We’ll use the Entity Framework Core (**EF Core**for simplicity) as our database ORM. This framework comes with ASP.NET Core as its default ORM and exposes a friendly API that allows us to map classes of our applications to database tables.

The EF Core also allows us to design our application first, and then generate a database according to what we defined in our code. This technique is called **code first**. We’ll use the code first approach to generate a database (in this example going to use an in-memory database, but it will be able to easily change it to a SQL Server or MySQL server instance, for example).

## database







The constructor we added to this class is responsible for passing the database configuration to the base class through dependency injection. 我们添加到此类的构造函数负责通过依赖注入将数据库配置传递给基类。

## Repository

# [AutoMapper](https://automapper.org/)

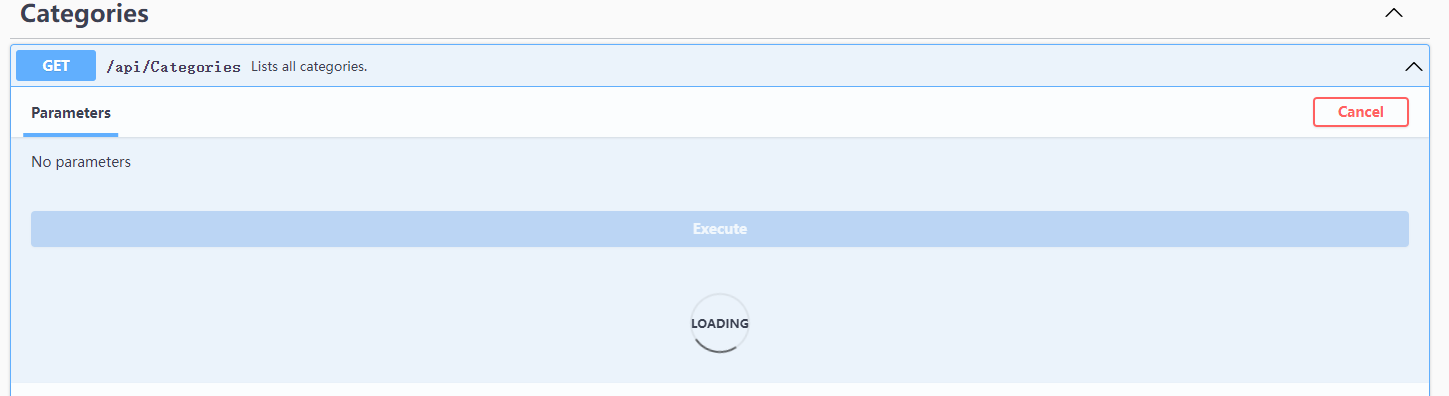
a library called [**AutoMapper**](https://automapper.org/) to handle mapping between objects. AutoMapper is a very popular library in the .NET world, and it is used in many commercial and open source projects.

To use AutoMapper, we have to do two things:

* Register it for dependency injection;
* Create a class that will tell AutoMapper how to handle classes mapping.

# GET request

when click the „GET“ button to get “List all categories”.



**1. Controller**

The request first handled by Controller.

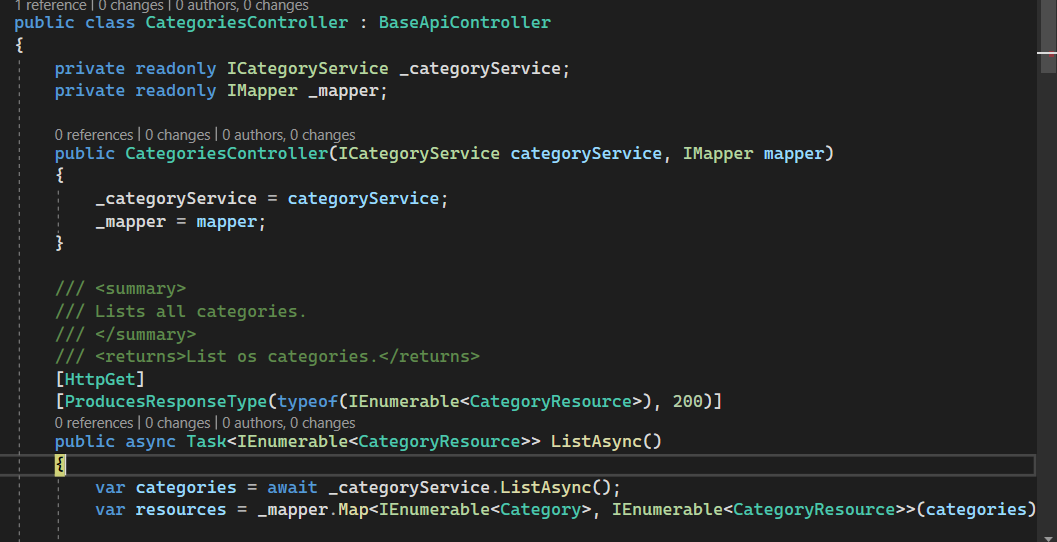
Therefore, controller is the part receive and output the request.

In CategoriesController has always a ICategoryService implement called “\_categoryService”, and a IMapper implement, one is used to invoke Service and another is to do a mapping operation between real-resource and need-resource and give only need-resource out.

(The difference between two resources is, real-resource have many information we don’t need, so we need to filter it.)

1.1. Use function “ListAsync ();” , the result will storage in “categories” variable, it’s **a list of all categories**.

1.2. Mapping Category class and CategoryResource class, storage finally resource in “resources” variable.



1.2.

1.1

1.1

Turn to **CategoryService**,

**Services contain the logic** (steps) handle request.

* However, solution will take charge by implement of ICategoryRespoitory called **“\_categoryRepository**”.
* Implement of IUnitOfWork “**\_unitOfWork**”:
* Implement of IMemoryCache “**\_cache**”:

#### ListAsync:

The implementations of the ListAsync method must **asynchronously** return an enumeration of categories. (返回类别枚举).

#### Task class:

The Task class, encapsulating the return, indicates asynchrony. We need to think in an asynchronous method due to the fact that we have to **wait for the database** to complete some operation to return the data, and this process can take a while.

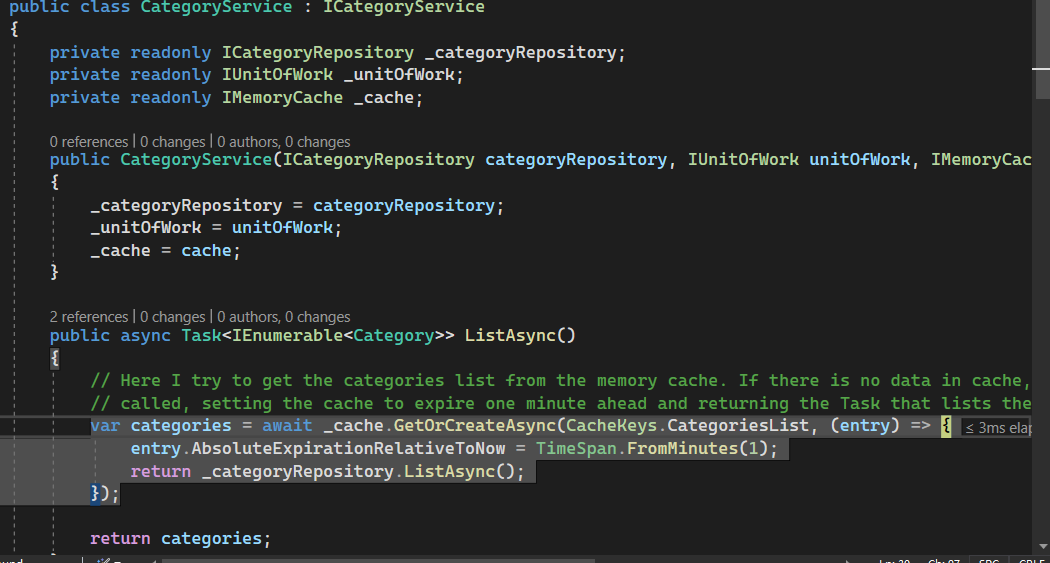
We need the attribute “categories” finally.

#### IEnumerable<Category>

The IEnumerable<Category>type tells the framework that we want to return an enumeration of categories, and the Task type, preceded by the async keyword, tells the pipeline that this method should be executed **asynchronously**.

#### await keyword

we have to use the await keyword for tasks that can take a while.



1.1.2

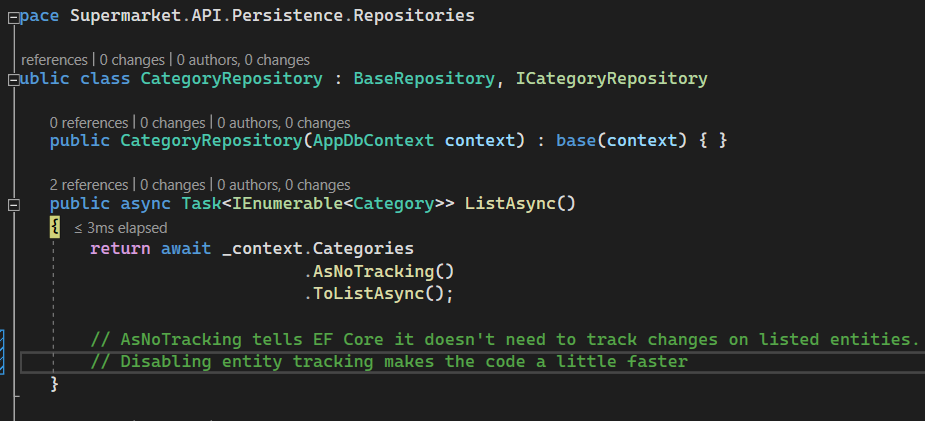
#### // Here I try to get the categories list from the memory cache(内存缓存). If there is no data in cache, the anonymous method will be called, setting the cache to expire(过期) one minute ahead and returning the Task that lists the categories from the repository.

(entry 入口)

1.1.2

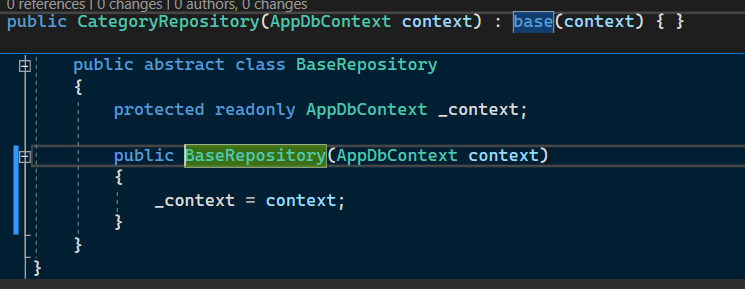
turn to **CategoryRepository**

service has used the from method ” \_categoryRepository.ListAsync();”



listed entities 列表实体

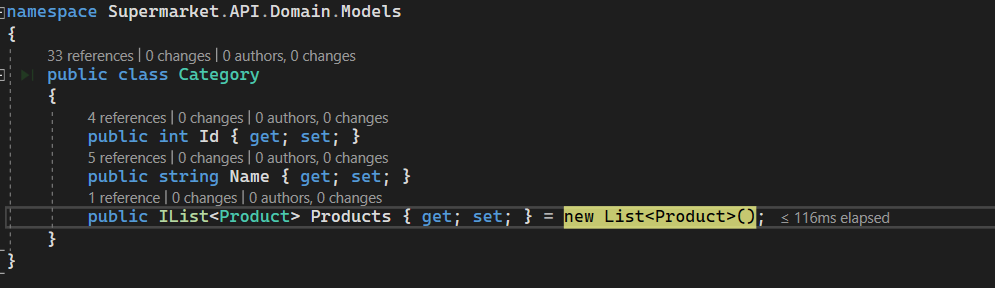
It’s constructor of CategoryRepository class, create an implement of AppDbContext.



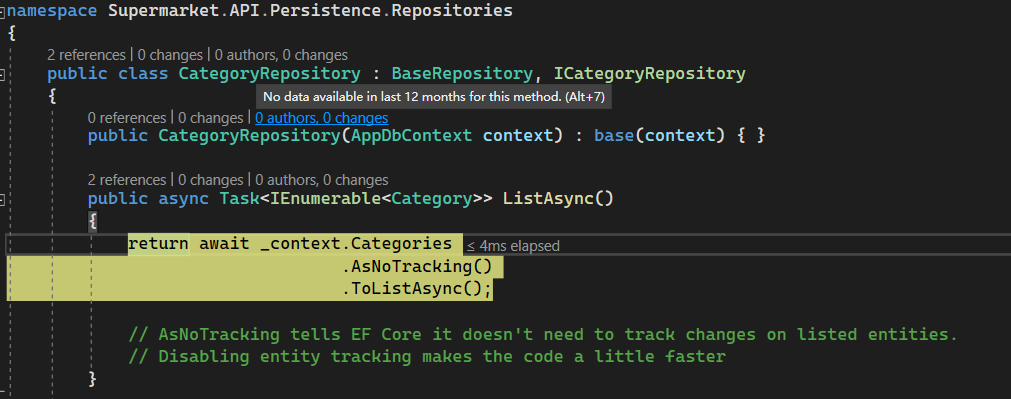
father

#### ?↑what contains in a AppDbContext? which function it has?

**.ToListAsync**→turn to Domain.Models **Category** class, **get a List<T> contains elements from the input sequence.**

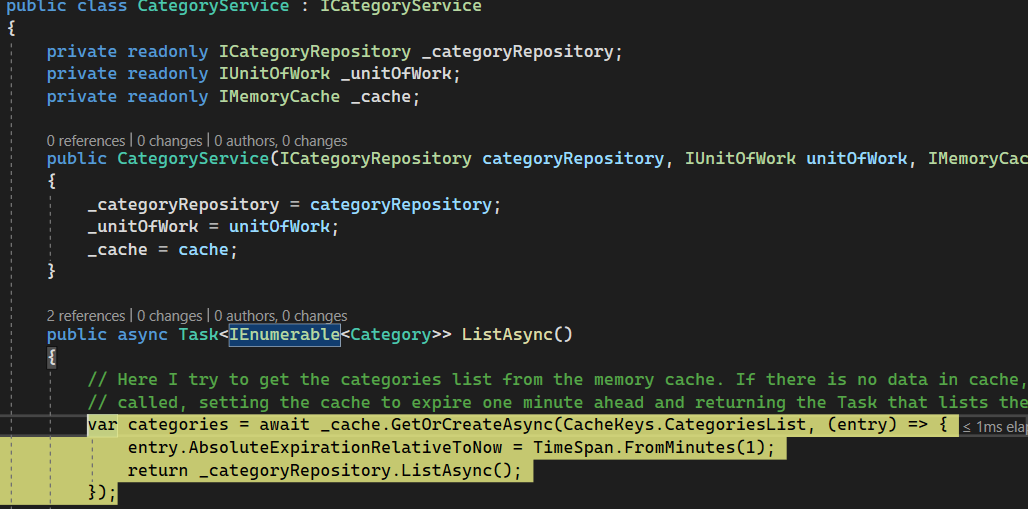


return id, name, and products list to **repository**.

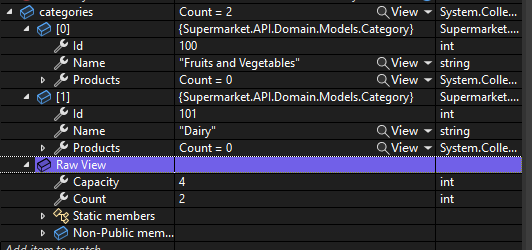


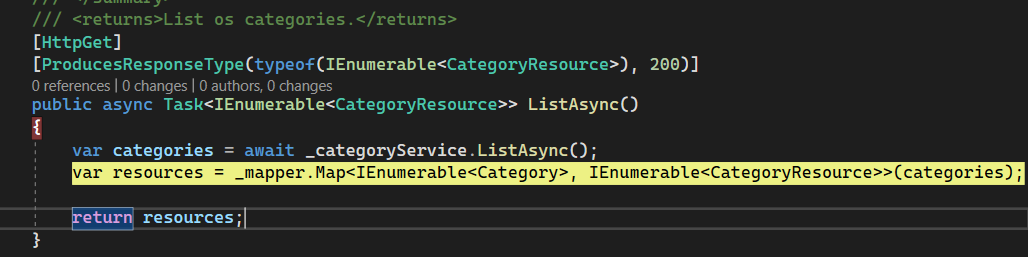
return

* 所以只要创建一个接口类就可以? -------是,且必须创建接口类,否则所有功能无法实现.
* 因为他的功能是靠继承类实现的, 所以每次调用接口类都会自动跳到继承类中? ----是,因为接口不能实现功能,必定会自动调用,创建能实现的继承者.
* 跳到继承类是会自动创建继承类对象? ----------是,见上



a list of category is returned↓

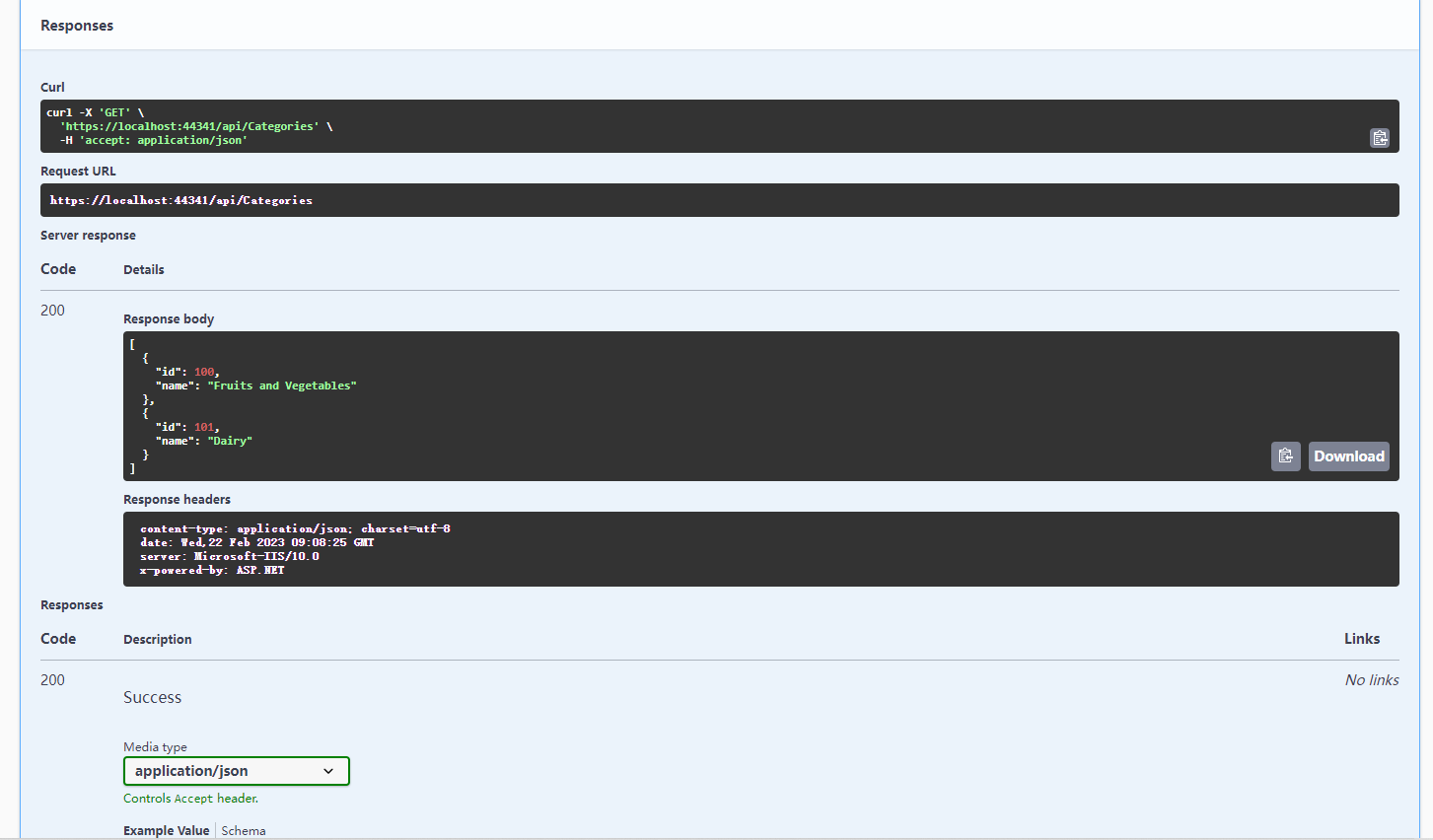


* 1. finish
  2. use “**Map**” to mapping all the information of category to finally resource.

Finally, we use only the resources which after mapping. The advantage of map is to simplify the response (if not we will get lots of information we don’t need.) and ensure the security of data.

#### ?but how works Map?

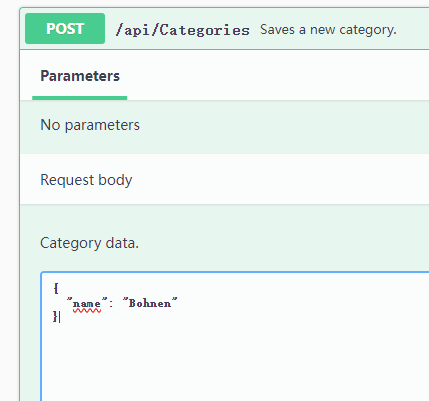
## response

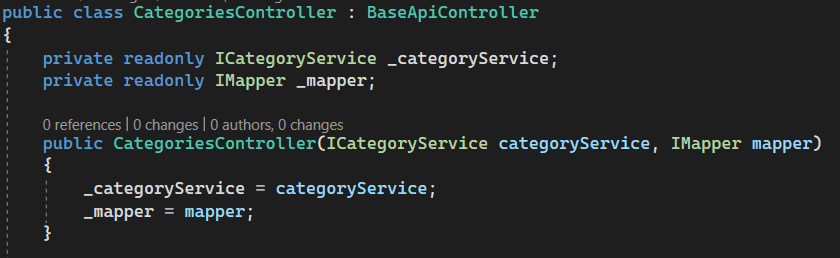


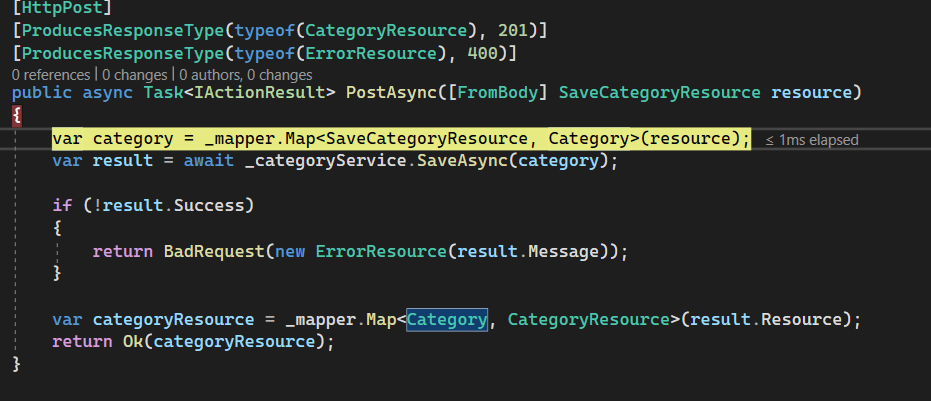
# POST request

to save a new category

**request↓**







1.

2.

3.

#### ?[ProducesResponseType(typeof(CategoryResource),201)]

#### <IActionResult>

Summary:

Defines a contract that represents the result of an action method.

Methods present in controller classes are called **actions**, and they have this signature because we can return **more than one possible result** after the application executes the action.

In this case, if the category name is invalid, or if something goes wrong, we have to return a **400 code (bad request)** response, containing generally an error message that client apps can use to treat the problem, or we can have a **200 response (success)**with data if everything goes ok.

#### [FromBody]

Specifies that a parameter or property should be bound using the request body.

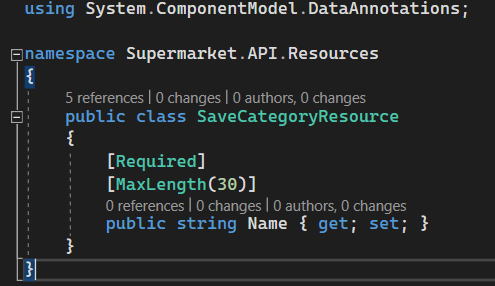
The FromBody attribute tells ASP.NET Core to parse(解析) the request body data into our new resource class. It means that when a JSON containing the category name is sent to our application, the framework will automatically parse it to our new class.

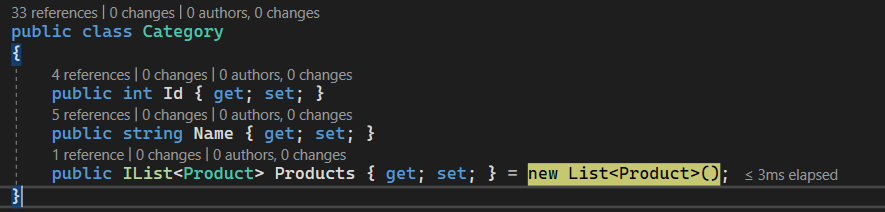
#### ? \_mapper.Map<SaveCategoryResource,Category>

1.

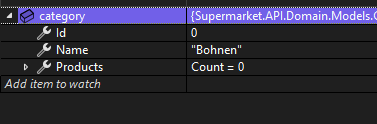
#### ?turn to **Category klass** to create a empty category

why use map??

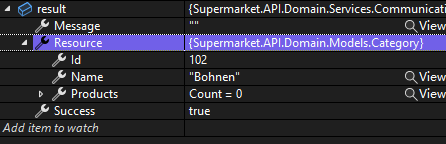




the category↓

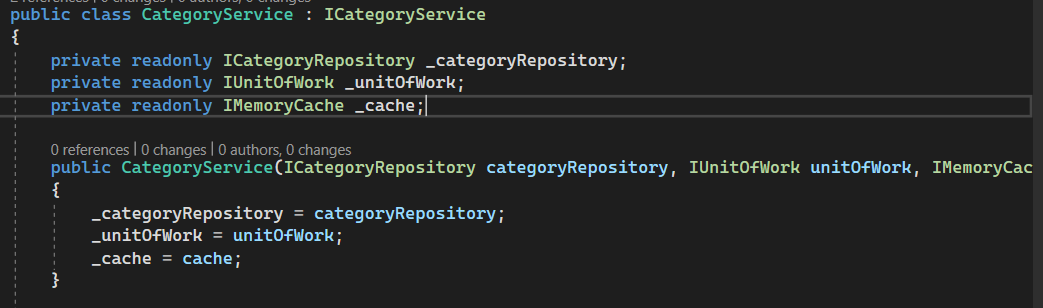


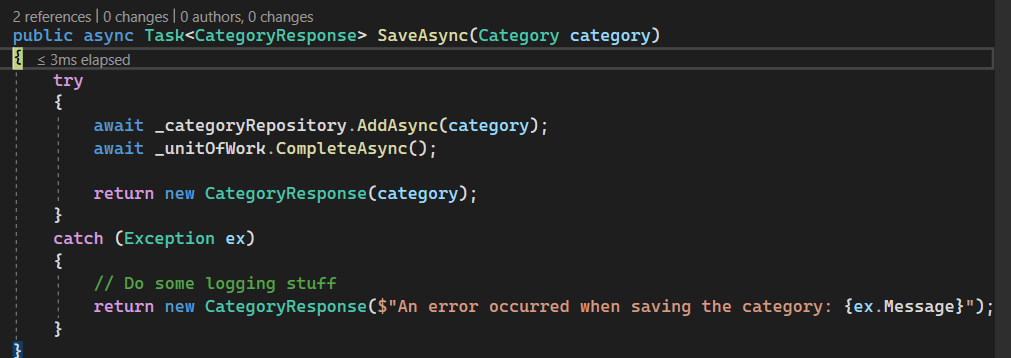
2.save the new category in database and return the result↓



turn to **Service** (the same as “GET”) → **CategoryService**

here is all the permanent “category” the new-created category





2.2

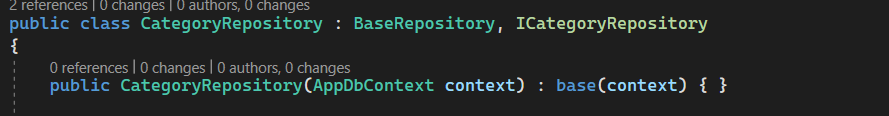
2.3

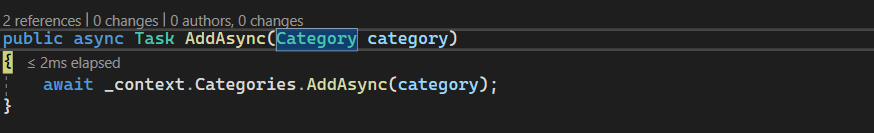
2.1

2.1

turn to **Repository** → **CategoryRepository** ,it will create a database which contains “AddAsync” method,

and add the new category **to database**.(did a real job)





#### SaveChanges()

There is a method called SaveChanges present at the context class that we have to call to really execute the queries into the database. I didn’t call it here because [**a repository shouldn’t persist data**](https://programmingwithmosh.com/entity-framework/common-mistakes-with-the-repository-pattern/)(data persistent数据在创建结束后仍然存在), it’s **just an in-memory collection of objects**.

2.2

#### √UnitOfWork

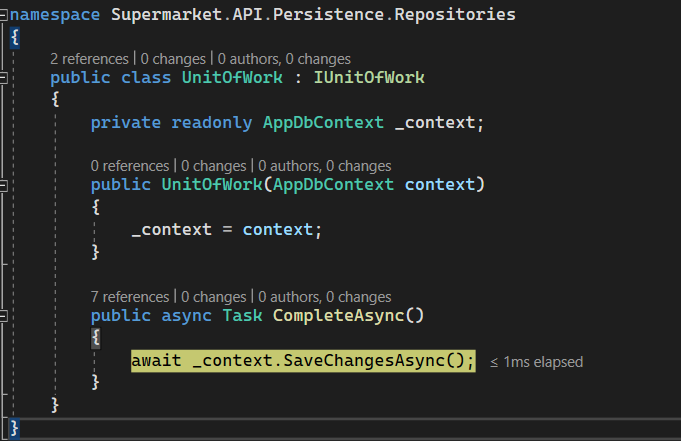
save the change current

if don’t use, current change will not be save.

**add and save is separate function**

This pattern consists of a class that receives our instance as a AppDbContextdependency and exposes methods to start, complete or abort transactions.

Because: We should save our changes into the database only after everything finishes. To do this, we have to use a [**transaction**](https://en.wikipedia.org/wiki/Database_transaction), that is basically a feature most databases implement to save data only after a complex operation finishes.



真正的数据库

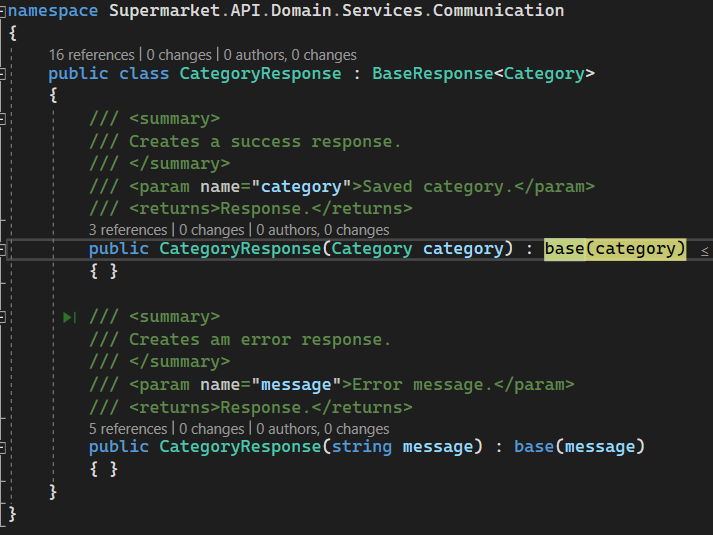
用于改动的数据库,短期

2.3

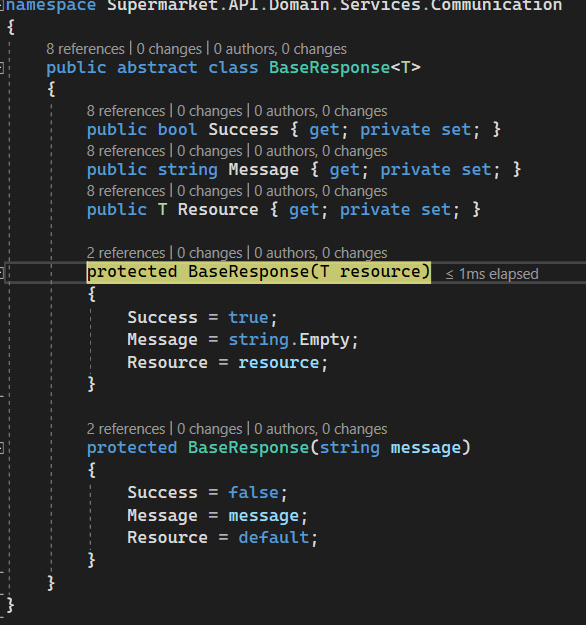
**give response**

go to Services.**Communication**, it’s the place where give a “ture or false” message.

create a new response-pipe for the new category



just use the vather method↓

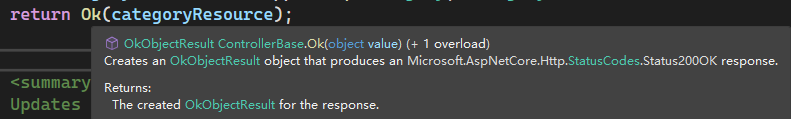


3.

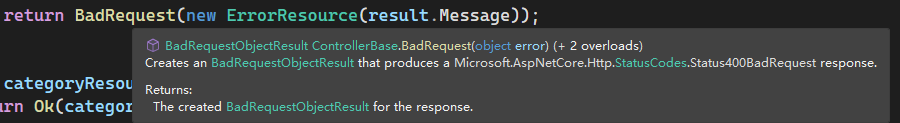
result



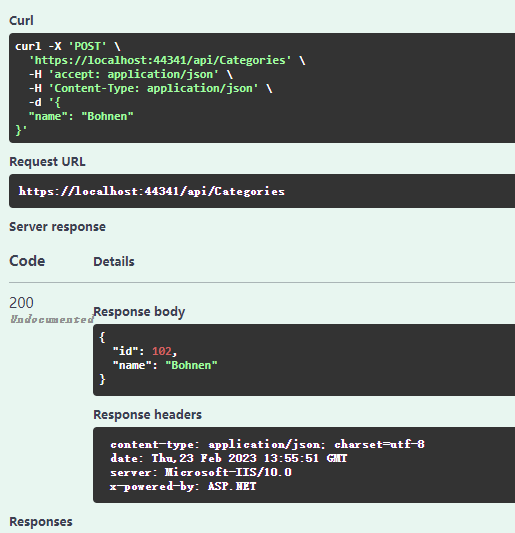
Ok(categoryResource)



BadRequest()

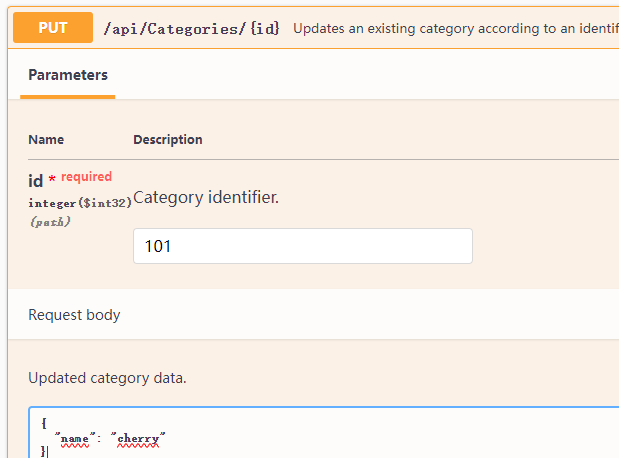


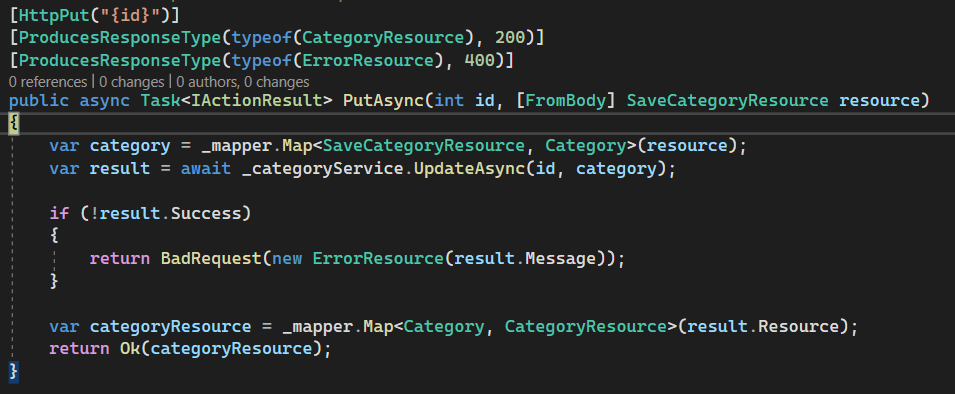
## response



# PUT request

update a existing category according to an identifier.





[1.](#_1._1)

[3.](#_3._1)

[2.](#_2.)

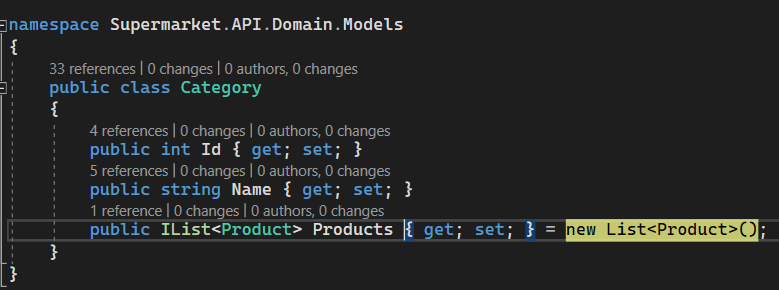
value of “resource”, it’s the new name of category

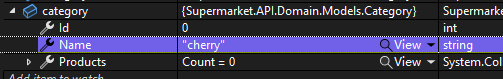


1.

turn to **Domain Category**

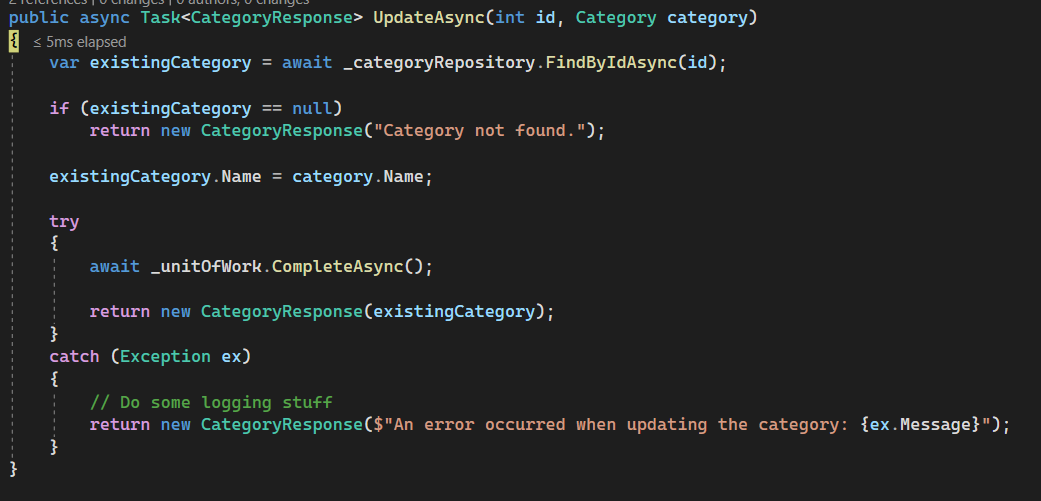
and create a new category which id is “0” ,name as the identifier gived.





2.

go **Service**, call the update function ”UpdateAsync”



2.3

2.5

2.4

2.2

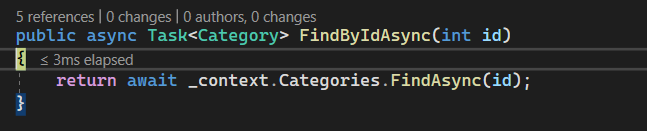
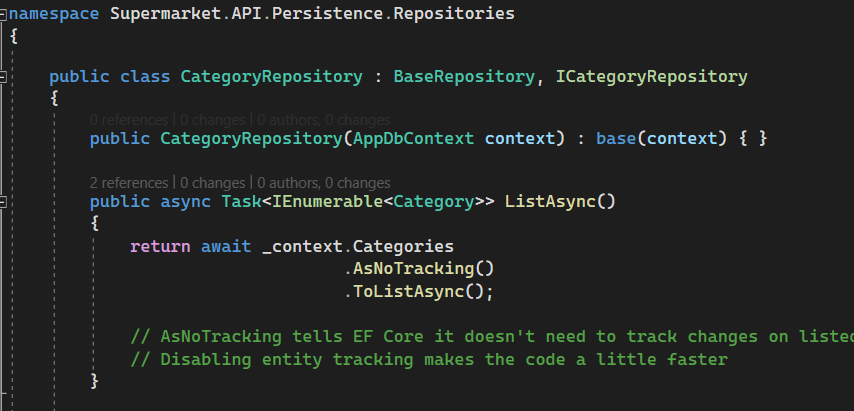
2.1

2.1

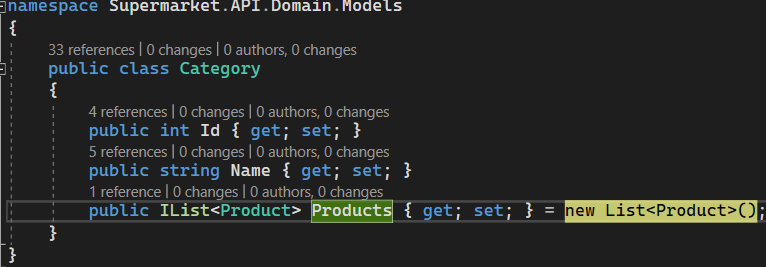
first use “id” search the category

when Service want to do response, it doesn’t really do this job.

It must call **Respositroy** to do , so we turn ro respository.

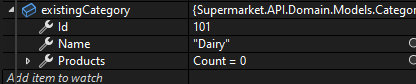


find from here?



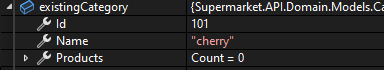
2.2

the result from last step is a category be found.

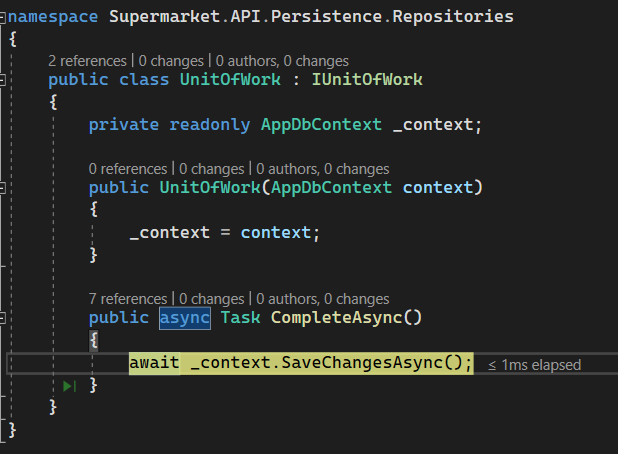


2.3

name has been changed .

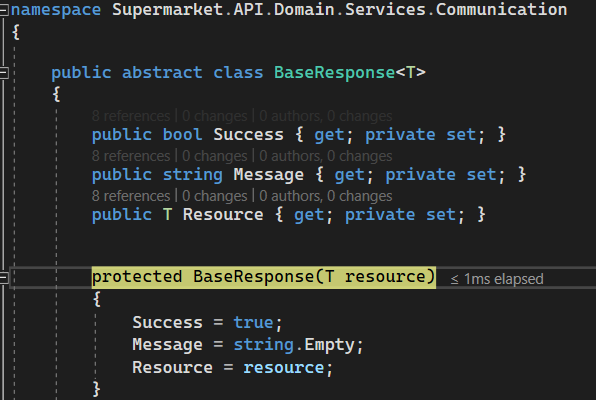
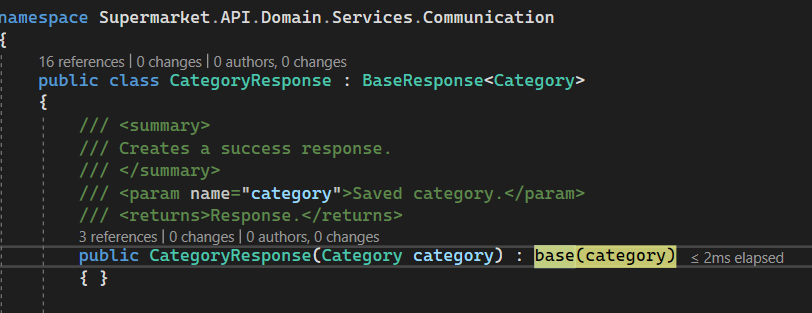


2.4



2.5

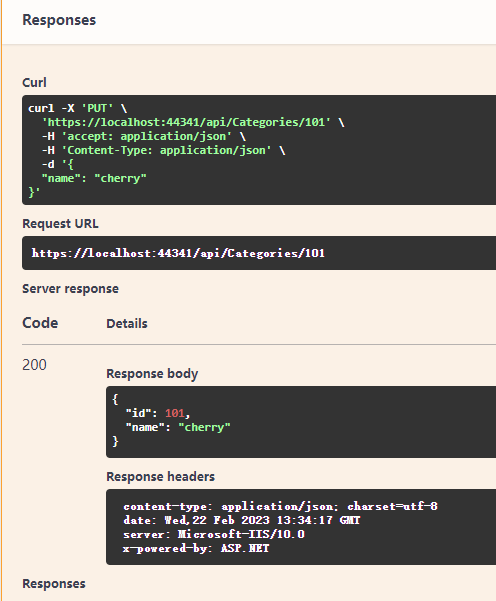
create a new response to new category and return this category as “result”.



3.

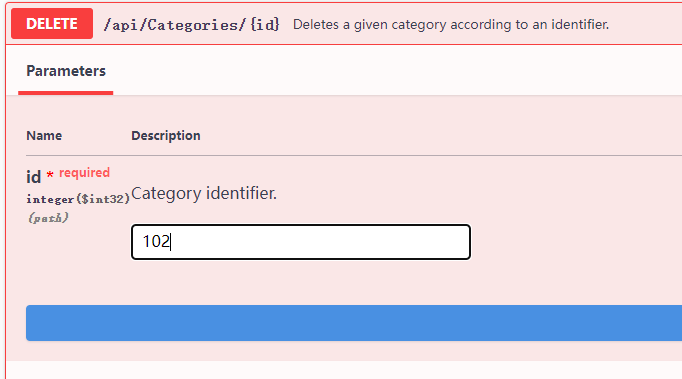
mapping this response.

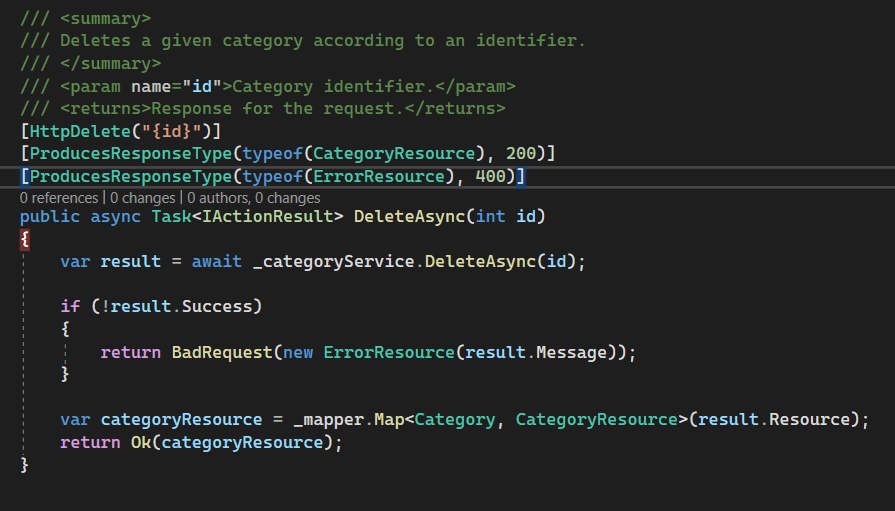
## response



# DELETE request

deletes a given category according to an identifier”102”(not exist yet)





3.

4.

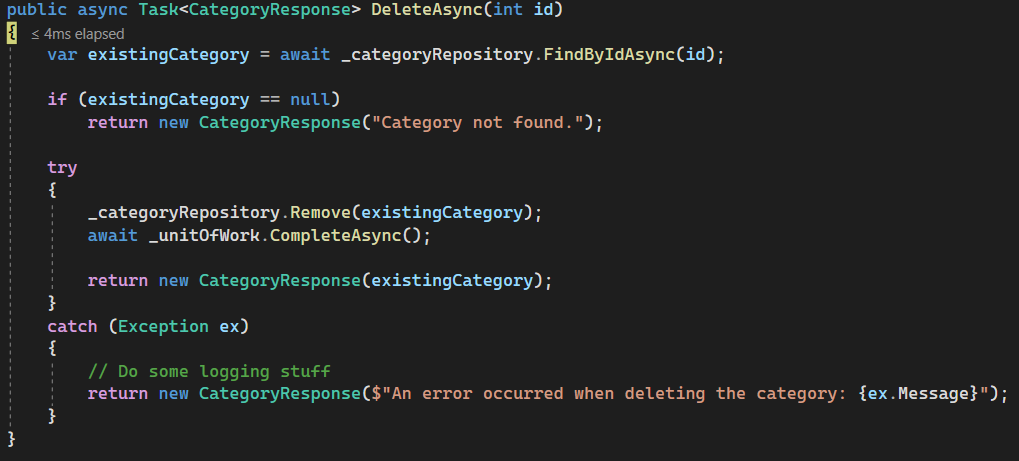
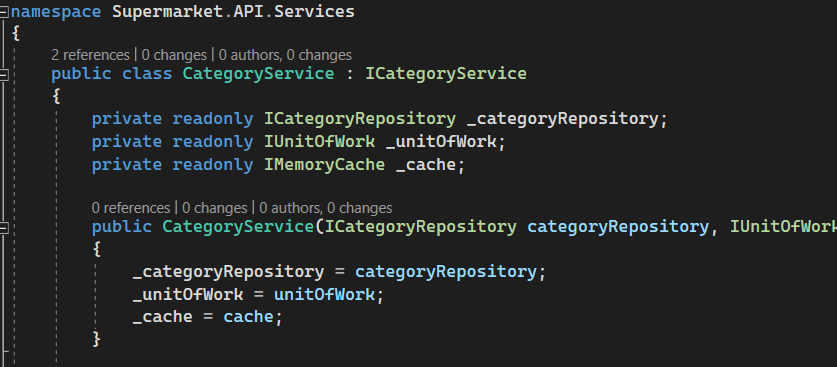
2.

1.

## for bad response

1.

turn to Service (as nomal….)



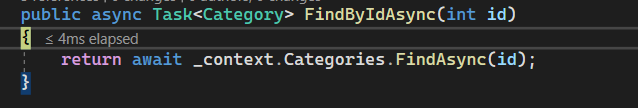
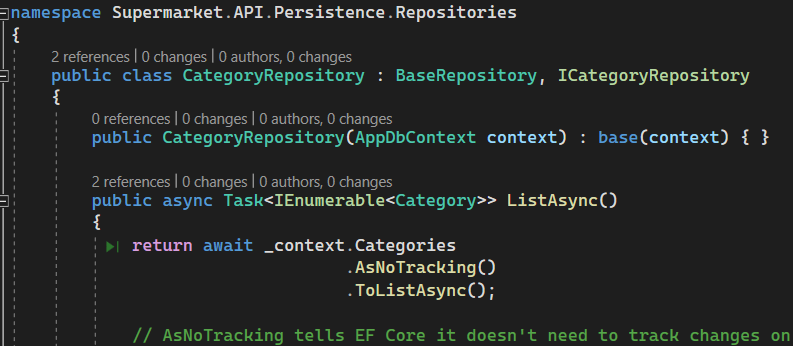
1.2

1.1

1.1

find the category according to identifier.

as usual, go to repository.

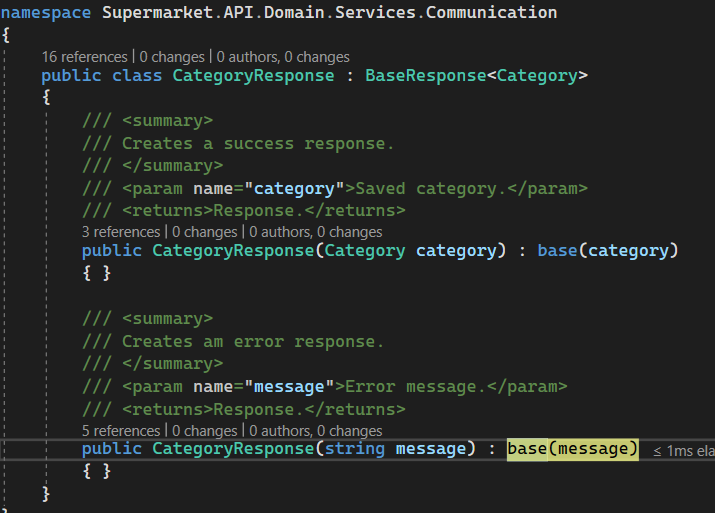


1.2

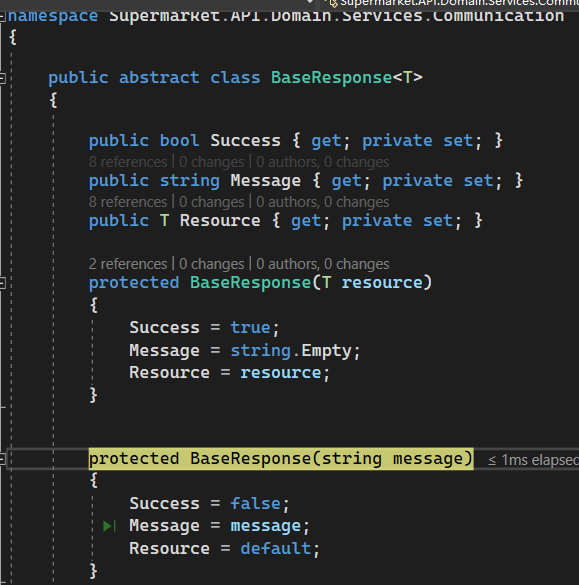
its found nothing.



use **Communication** to return a bad request.

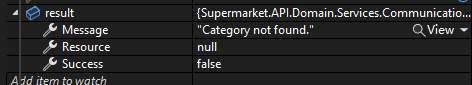


just use the vater function, perament “Category not found.”↓



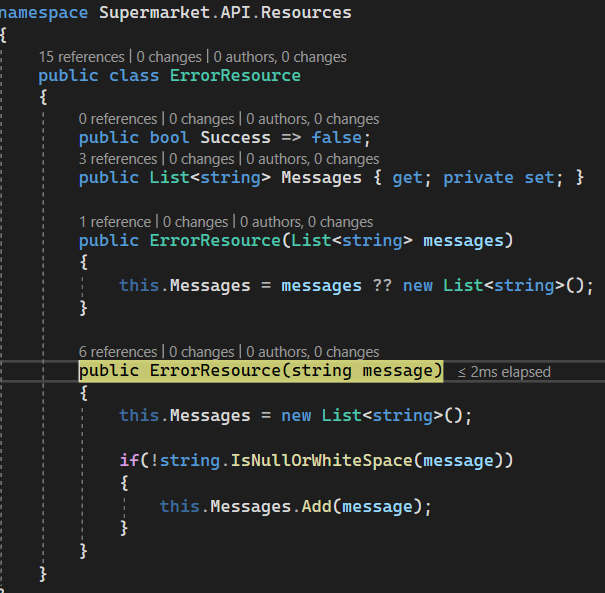
2.

the value of the priority “result.Success” is false.

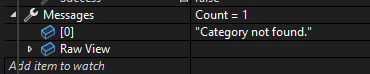


go to **Resources** →**ErrorResource**

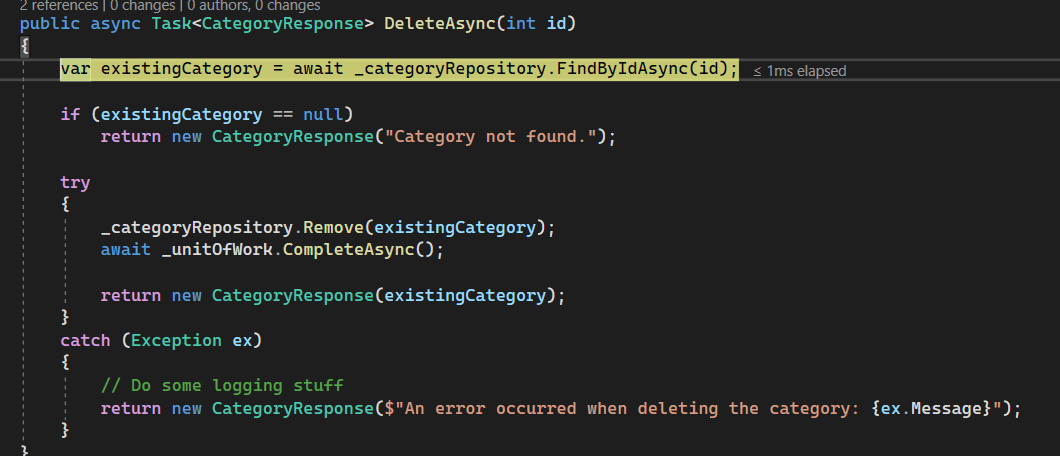
add a error message to ErrorMessage.(it looks slightly unintuitive, but through this way makes the structure more reusable )



a new message is been added.



## for good response



1.4

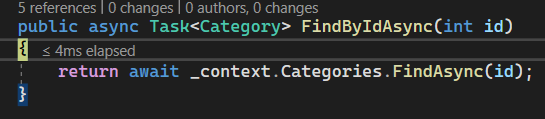
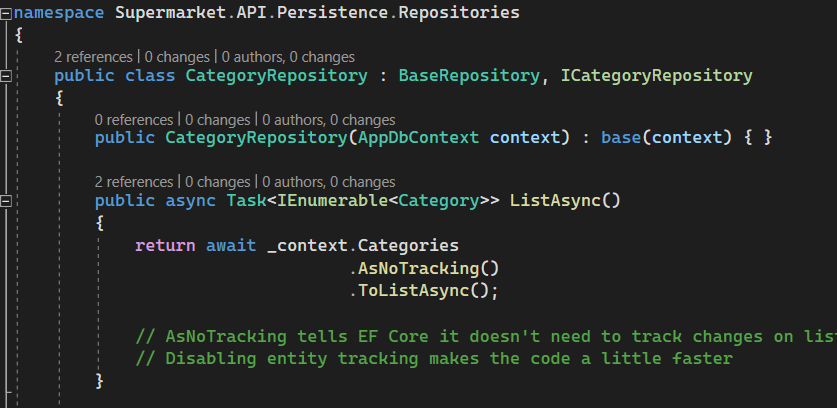
1.3

1.2

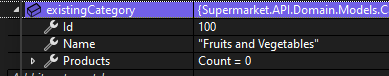
1.1

1.1

use the given id ”100” to find the category.

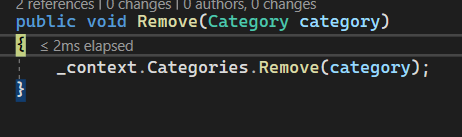
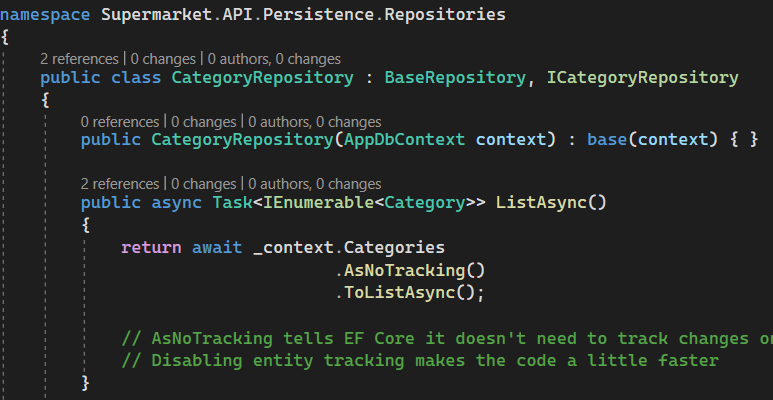


the category has been founded.



1.2

turn to **Repository**, call **Remove** function.

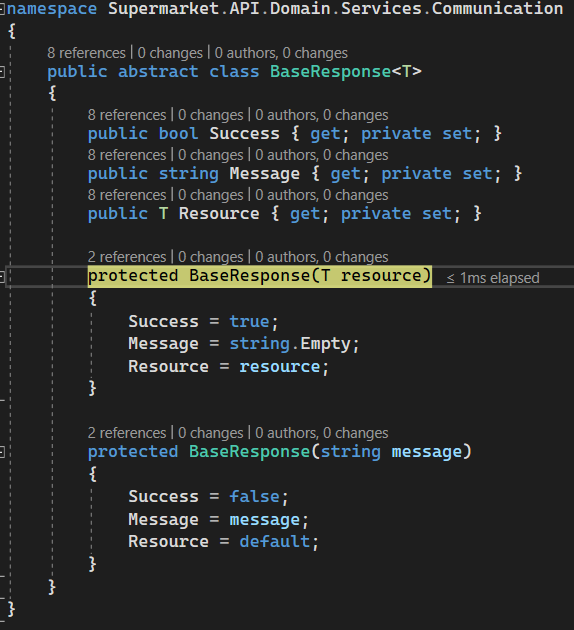


1.3

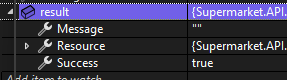


1.4

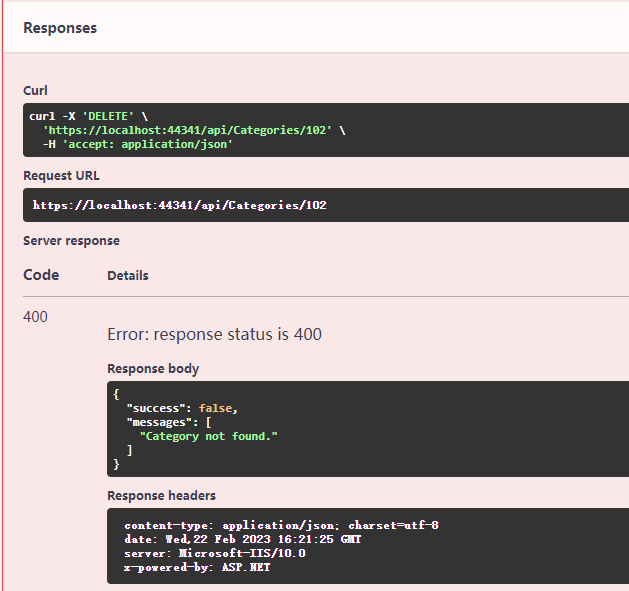
just create the response for the result



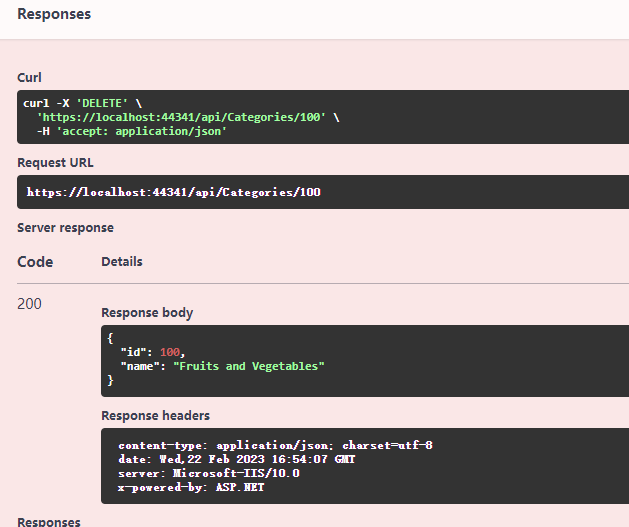
the returned result↓



## bad response



## good response



# Main classes

## Task<TResult>

Summary:

Represents an asynchronous operation that can return a value.Type parameters:

TResult:

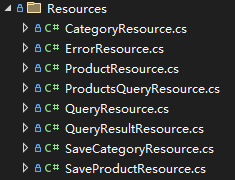
The type of the result produced by this System.Threading.Tasks.Task`1.

## Resources

Contains: attribute type we need.

Generally been used with original modals by Mapper.

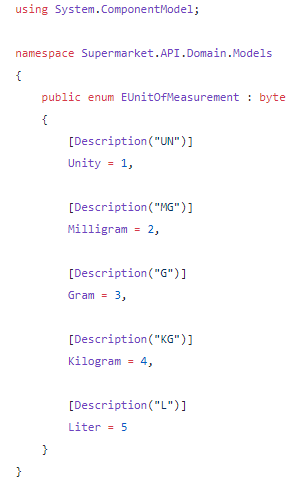




### CategoryResource.cs

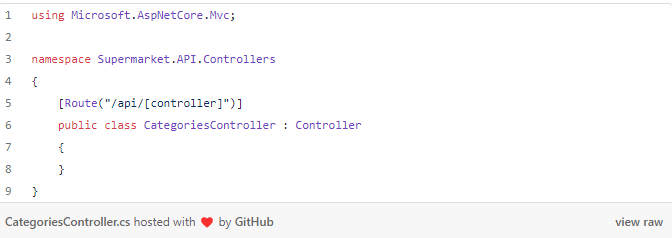
# Attributes

## Description attribute



An attribute is a way to **define metadata**(元数据) over classes, interfaces, properties and other components of the C# language. In this case, we’ll use it to simplify the responses of the products API endpoint.

## Route

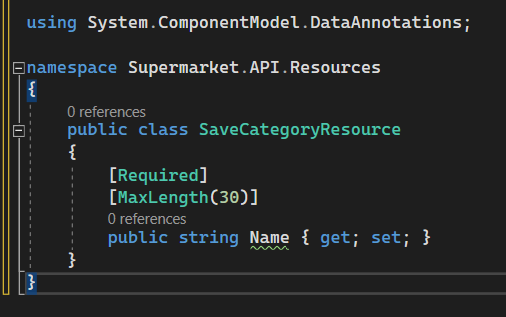


The new controller should **respond through the route /api/categories**. We achieve this by adding the Routeattribute above the class name, specifying a placeholder that **indicates that the route ??should use the class name without the controller suffix**, by convention.

## HttpGetattribute

The HttpGetattribute tells the ASP.NET Core pipeline to use it to handle GET requests (this attribute can be omitted, but it’s better to write it for easier legibility).

## Required and MaxLength attributes



Notice the Required and MaxLength attributes applied over the Name property. These attributes are called [data annotations](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=netframework-4.7.2). The ASP.NET Core pipeline uses this metadata to validate requests and responses. As the names suggest, the category name is required and has a max length of 30 characters.