**An awesome guide on how to build RESTful APIs with ASP.NET Core**

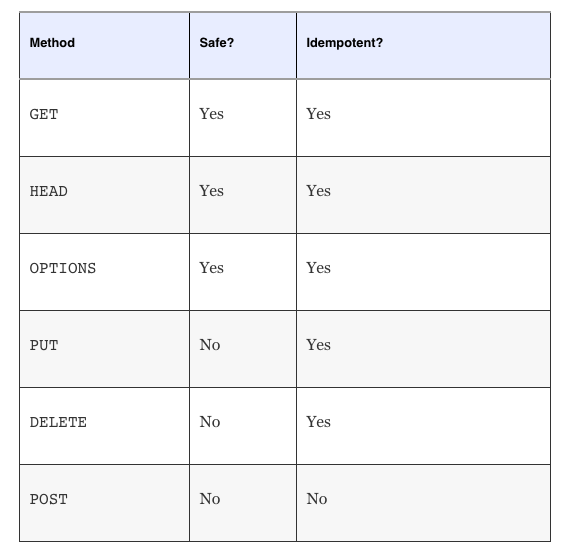
**A step by step guide on how to implement clean, maintainable RESTful APIs**

Full article:

<https://medium.com/free-code-camp/an-awesome-guide-on-how-to-build-restful-apis-with-asp-net-core-87b818123e28>

This article will show you how to integrate common frameworks and libraries, such as [Entity Framework Core](https://docs.microsoft.com/en-us/ef/core/) and [AutoMapper](https://automapper.org/), to deliver the necessary functionalities.



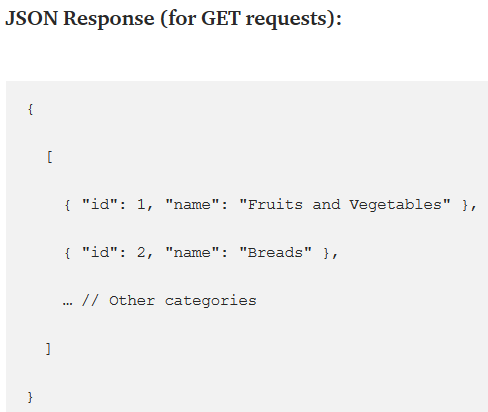


Objective – write a restful api for a supermarket.

* Create a RESTful service that allows client applications to manage the supermarket’s product catalog. It needs to expose endpoints to create, read, edit and delete **products categories**, such as dairy products and cosmetics, and also to manage products of these categories.
* For categories, we need to store their names. For products, we need to store their names, unit of measurement (for example, KG for products measured by weight), quantity in the package (for example, 10 if the product is a pack of biscuits) and their respective categories.

To develop this service, we basically need two API endpoints: one to manage categories and one to manage products. In terms of JSON communication, we can think of responses as follow:

**API endpoint:** /api/categories





**Structure Overview**

An ASP.NET Core application consists of a group of [middlewares](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.2) (small pieces of the application attached to the application pipeline, that handle requests and responses) configured in the Startup class.

When the application starts, the Mainmethod, from the Program class, is called. It creates a default web host using the startup configuration, exposing the application via HTTP through a specific port (by default, port 5000 for HTTP and 5001 for HTTPS).

Let’s start by writing the domain layer. This layer will have our models classes, the classes that will represent our products and categories, as well as repositories and services interfaces.

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.

Let’s start handling GET requests. 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.

Conceptually, a service is basically a class or interface that defines methods to handle some business logic. It is a common practice in many different programming languages to create services to handle business logic, such as [authentication and authorization](https://medium.com/@evandro.ggomes/json-web-token-authentication-with-asp-net-core-2-0-b074b0cfc870), payments, complex data flows, caching and tasks that require some interaction between other services or models.

The service we’re going to create initially will define a single behavior**,** or **method**: a listing method. We expect that this method returns all existing categories in the database.

We define an **interface**. An interface tells how something should work, but **does not implement the real logic for the behavior**. The logic is implemented in classes that implement the interface.

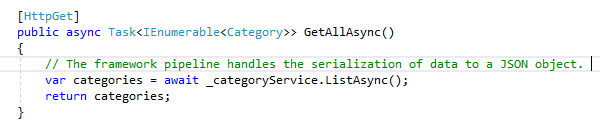


The implementations of the ListAsync method must **asynchronously** return an enumeration of categories.

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. Notice also the “async” suffix. It’s a convention that indicates that our method should be executed asynchronously.

Interfaces allow us to abstract the desired behavior from the real implementation. Using a mechanism known as [**dependency injection**](https://medium.freecodecamp.org/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f), we can implement these interfaces and isolate them from other components.

Basically, when you use dependency injection, you define some behaviors using an interface. Then, you create a class that implements the interface. Finally, you bind the references from the interface to the class you created.



The method uses our category service instance to list all categories and then returns the categories to the client. The framework pipeline handles the serialization of data to a JSON object. 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**. Finally, when we define an async method, we have to use the await keyword for tasks that can take a while.

A service class is not a class that should handle data access. There is a pattern called [**Repository Pattern**](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern) that is used to manage data from databases.

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://docs.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.

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.



The constructor we added to this class is responsible for passing the database configuration to the base class through dependency injection.

In the root folder of the application, open the Startup class. This class is responsible for configuring all kinds of configurations when the application starts.

The ConfigureServices and Configure methods are called at runtime by the framework pipeline to configure how the application should work and which components it must use.

We can use the ConfigureServices method, accessing the services parameter, to configure our dependency bindings.

Dependency injection

The configuration of these lines internally configures our database context for dependency injection using a [scoped lifetime](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2#service-lifetimes-and-registration-options).

The scoped lifetime tells the ASP.NET Core pipeline that every time it needs to resolve a class that receives an instance of AppDbContext as a constructor argument, it should use the same instance of the class. If there is no instance in memory, the pipeline will create a new instance, and reuse it throughout all classes that need it, during a given request. This way, you don’t need to manually create the class instance when you need to use it.

The dependency injection technique gives us many advantages, such as:

* Code reusability;
* Better productivity, since when we have to change implementation, we don’t need to bother to change a hundred places where you use that feature;
* You can easily test the application since we can isolate what we have to test using **mocks** (fake implementation of classes) where we have to pass interfaces as constructor arguments;
* When a class needs to receive more dependencies via a constructor, you don’t have to manually change all places where the instances are being created (**that’s awesome!**).

A [resource class is](https://restful-api-design.readthedocs.io/en/latest/resources.html) a class that contains only basic information that will be exchanged between client applications and API endpoints, generally in form of JSON data, to represent some particular information.

It is a bad practice to return the real model representation as the response since it can contain information that the client application does not need or that it doesn’t have permission to have (for example, a user model could return information of the user password, which would be a big security issue).

Notice that we easily changed the implementation without having to adapt the service class or repository, simply by injecting a new dependency (IMapper) to the constructor.

Dependency injection makes your application maintainable and easy to change since you don’t have to break all your code implementation to add or remove features.

You probably realized that not only the controller class but all classes that receive dependencies (including the dependencies themselves) were automatically resolved to receive the correct classes according to the binding configurations.

Notice the response type of this method, Task<IActionResult>. 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.

## ****Validating the Request Body Using the Model State****

ASP.NET Core controllers have a property called ModelState. This property is filled during request execution **before** reaching our action execution. It’s an instance of ModelStateDictionary, a class that contains information such as whether the request is valid and potential validation error messages.

The saving logic may fail due to problems when connecting to the database, or maybe because any internal business rule invalidates our data.

If something goes wrong, we can’t simply throw an error, because it could stop the API, and the client application wouldn’t know how to handle the problem. Also, we potentially would have some logging mechanism that would log the error.

The contract of the saving method, it means, the signature of the method and response type, needs to indicate us if the process was executed correctly. If the process goes ok, we’ll receive the category data. If not, we have to receive, at least, an error message telling why the process failed.

We can implement this feature by applying the **request-response pattern**. This enterprise design pattern encapsulates our request and response parameters into classes as a way to encapsulate information that our services will use to process some task and to return information to the class that is using the service.

This pattern gives us some advantages, such as:

* If we need to change our service to receive more parameters, we don’t have to break its signature;
* We can define a standard contract for our request and/or responses;
* 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.

**Tip:** it’s not a good practice to define base classes for everything, because [base classes couple your code](https://en.wikipedia.org/wiki/Fragile_base_class) and prevent you from easily modifying it. Prefer to use [composition over inheritance](https://medium.com/humans-create-software/composition-over-inheritance-cb6f88070205).

If you realize that a service or application will grow and change frequently, avoid using a base class.

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.

A common pattern to handle this issue is the [**Unit of Work Pattern**](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application). This pattern consists of a class that receives our AppDbContext instance as a dependency and exposes methods to start, complete or abort transactions.

Using the repository pattern, you can simply implement a new repository class and bind it using dependency injection.

In real world applications, you shouldn’t wrap everything inside a generic try-catch block, but instead you should handle all possible errors separately.

Simply adding a try-catch block won’t cover most of the possible failing scenarios. Be sure to correct implement error handling.