# Kestrel web server:

This project was designed to be a starter project that is run in a docker container, and run behind a reverse proxy.

Find info at:

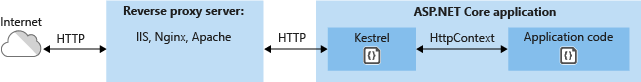
<https://stackify.com/what-is-kestrel-web-server/>

<https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/>

Kestrel was built for speed, and is a production server. It’s six times faster than node.js. It will not do:

* SSL termination.
* URL re-writes
* GZip compression
* Limited ability to serve static files (graphics files/icons/documents/static html pages).

It was meant to be run behind IIS/Apache/NGINX.



## Reverse proxy:

Info at:

https://en.wikipedia.org/wiki/Reverse\_proxy

A reverse proxy is a proxy server that takes requests from the web and forwards them to a series of web servers inside a DMV. It functions as:

* Firewall (DMZ)
* Takes requests via HTTPS, decrypts them, and forwards them as
* HTTP
* Load balancing.
* GZip compression.
* URL re-writes

## Configure production environments

https://stackify.com/kestrel-web-server-asp-net-core-kestrel-vs-iis/

https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/index?view=aspnetcore-2.1

https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/proxy-load-balancer?view=aspnetcore-2.1

# Design guidelines

## Models and Entities are not the same thing

There is a strong tendency to re-use your POCO(s) throughout your back end. This is DRY, right? Don’t repeat yourself?

So you want to use ONE set of POCOs to read from the database, and return those same objects to the clients making web requests.

The problem is that the POCO that you use in the EF layer is different from the POCO that you return when servicing a client request (or accept as a POST/PUT request). Namely:

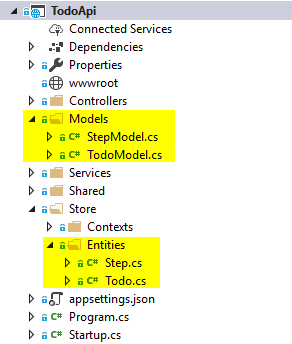
* At the Entity Framework layer: you have all kinds of attributes and properties that EF wants to see, but which mean nothing to WebAPI.
* At the WebAPI layer: vice versa—you don’t need any of that EF stuff, and you’ll want to attach all kinds of additional attributes for validation and hinting for WebAPI.

So, don’t do it!

You have two sets of POCOs:

1. Entities: these are EF classes that represent records in the data store.
2. Models: These are for the client. They can be used for requests and for responses\

Here is how I’ve arranged things in the project:



# API Controller

There are a lot of similarities, but a few important differences.

Just read through TodoController to see how this works. It’s kind of easy to understand.

## PATCH

Yet another reason why I hate REST. I wish REST wasn’t so popular.

Some notes I took: 😊

1. RFC 6902. Read up on this.
2. Take a parameter [FromBody] JsonPatchDocument<SomeType.Dto> patchDoc
3. The JsonPatchDocument class will iterate through all the PATCH instructions, and update the … thing.
4. Man, PATCH is still a major PITA.

# Validating input

Ok, to reiterate, our models and our entities are not the same set of POCOs, and here’s a good example of why.

Your API controller inherits an object from the ControllerBase base class, called ModelState, which is a collection of errors that you can accumulate and then return whenever there is a bad request.

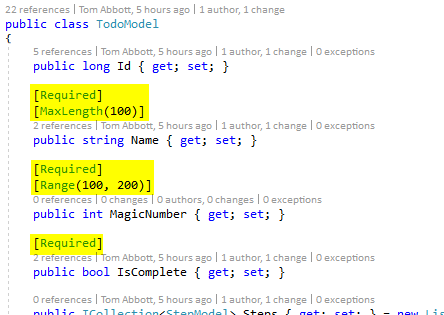
ModelState offers the following properties / methods:

* ModelState.IsValid: a property that tells whether or not any errors have been added.
* ModelState.AddModelError: A function to add an error.

You can do quite a bit of custom validation logic just from this.

## WebAPI’s out of the box validation using DataAnnotations (BOOOOO!)

This looks like it could be so promising, but it’s really kind of lame… It doesn’t actually work as advertised. It’s kind of hit and miss.



I tried this out, and only got a few of these to work. Not very good.

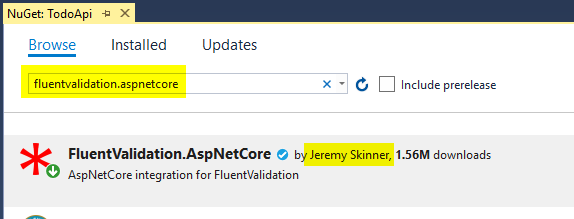
## FluentValidation

Ok, this really worked—for real.

I followed their quick-start documentation here: <https://fluentvalidation.net/aspnet>

I still need to try it out on PUT requests, but on POST it worked like a charm.

First you install it via NuGet… It’s called FluentValidation.AspNetCore:

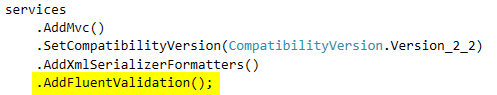


Now, go into Startup.cs, and look for your ConfigureServices() function. Add some using statements:

using FluentValidation;

using FluentValidation.AspNetCore;

Now add the following call, after calling AddMvc():



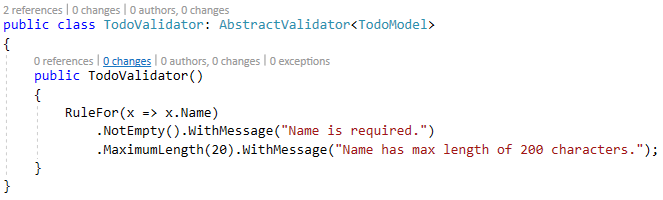
The documentation says to call AddFluentValidation() right after calling AddMvc(), but I found that it still worked. Whatever.

Now, in the same ConfigureServices() function, look for where you’re registering all your services and stuff, set up DI for all your validators. This looks pretty straightforward.



It seems like you ought to be able to chain as many of these as you need, and have one for all your POCOs.

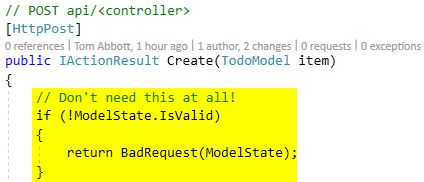
Now you need to actually write a validator. That’s pretty easy, too.



And that’s pretty much it. Here are the steps again, in summary:

1. in ConfigureServices
   1. you turn it on by calling AddFluentValidation()
   2. and you link a validator to a POCO using ASP’s.
2. And then you write some validator classes to do their thing.

NOTE: If validation fails, ASP will never call your controller. So, you don’t need any checks like this:



Thoughts:

The only real problem I foresee here is that I don’t know how to make your validator tell the difference between an HTTP put and an HTTP post. I feel like that could be a problem, but I’ll have to play around with it a little bit more.

Either way, it seems like I can use FV to do some basic common validation

# Building a data Service layer / custom services

The reason you build a data service layer (or sometimes called “the repository pattern”) isn’t so that you can swap out the underlying DB someday (if you feel the need). It is so that

1. **Single responsibility.** (most important) You can add higher-level query logic that doesn’t belong with the entities or the db context
2. **Object mapping.** You need logic that maps between your Database entities (ORM) and the DTOs (models) that get sent/received to/from the client in each web request. Your data layer should handle this conversion
3. **You don’t have to be stuck with EF** (not that that’s a bad thing), but now you have a little flexibility. For instance, you can switch to ADO.NET and call an sproc.
4. **More testable.** You can unit test all this business logic without having to rely on an actual database, which needs to be set up, initialized with seed data, etc.
5. **Less code duplication.** Without a data service, you end up duplicating code previously described.

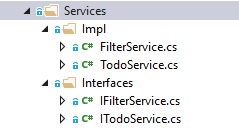
So, there’s two types of services you can add

1. Business logic that does stuff that is not related to a database.
2. Business logic that abstracts the database.

Declaring a service will be the same for both approaches.

## Folder structure

Put it into the Services folder:



This folder has one sub-folder for interfaces, and another folder for concrete implementations.

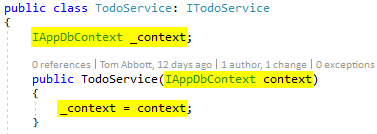
## Writing a service layer class

Now just write a class, like you normally would:



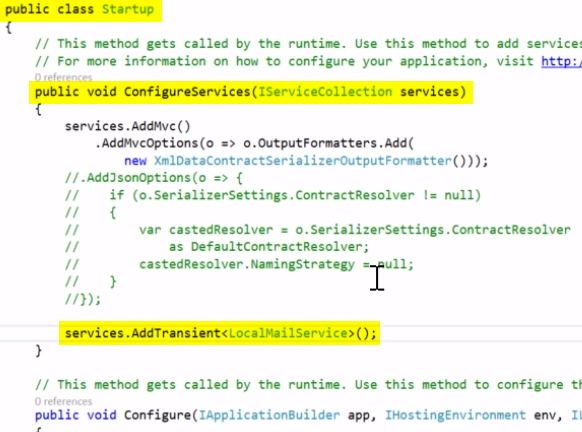
## Class constructor

So, you’ll need the DI to supply you with other registered types. .net core’s DI is pretty magical. Really all you need to do is add an IWhatever to your constructor, and the DI will build you one and pass it in when your ctor gets called.



## Registering your class

Now you need to tell ASP about it. You do this by registering with the DI container, in your Startup class, in the ConfigureServices method (important thing to remember).

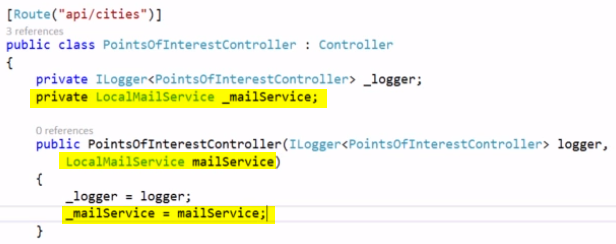


There are three lifetime properties:

1. Transient: ASP will make a new one of these every time someone asks for one.
2. Scoped: Lives for the life of a single request. So, per request, everyone gets the same object.
3. Singleton: one and only one, which gets created when the server starts up.

NOTE: data services will want to use a scoped lifetime. Services that are stateless or otherwise don’t care how many instances there are will be transient.

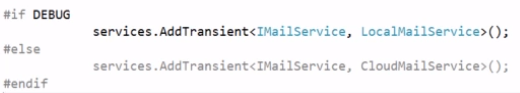
Now we can use it via ctor injection:



Now, typically you won’t want to inject a class directly. Instead, you’ll want to inject an interface, and have your class inherit that interface. When you register your type (in Startup.ConfigureServices()), you’ll give an interface name and a concrete class name.



You can do all kinds of fancy kinds of things with this. Here is an example of using a production service for prod, and a dev service for your local environment:



## Querying the DI framework for a service.

There are two ways you can get a service

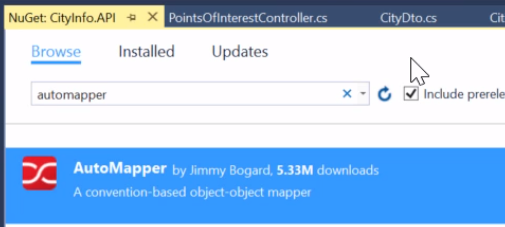
1. Recommended way is to use ctor injection, and let ASP’s DI framework conjure one up for you.
2. request an instance manually in your controller, you just call HttpContext.RequestServices.GetService()

Advised that you use ctor injection, instead.

## Mapping logic between the Entity layer and the Data service layer

So, one of the primary jobs of your data service layer is to translate between the entities in your database layer and the DTO/POCO classes that are sent/received to/from the client.

There’s a framework you can use called AutoMapper.



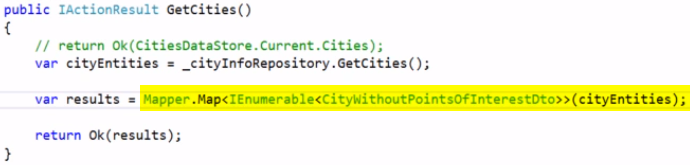
You need to configure the AutoMapper in your Startup.cs Configure() method (in the Pluralsight video, he put it before the app.UseMvc() call.



AutoMapper is convention-based

* if two props have the same name, it will try to convert between them
* If a prop is missing from the target or the source objects, it will ignore them.

Using AutoMapper is straightforward.



Note that in the Pluralsight video, he was doing all his conversion in the controller. I don’t like this approach. I don’t want the controller knowing anything about the lower layers. The controller’s currency should be DTOs/POCOs.

# Entity Framework Core

Ok, so EF Core is a whole, brand-new thang.

1. This is not EF5 or EF6. It’s its own thing.
2. It was designed from EF6, but it’s **lighter-weight**, and it’s meant for .NET Core.
3. If you’re using ASP Core 1, then use EF Core 1. If you’re using ASP Core 2, then use EF Core 2.
4. It’s cross-platform, and cross-database. You can use it with anything that has a Provider class:
   1. SQL Server
   2. Postgres
   3. SQL lite
   4. MYSql
   5. Sql
   6. And a really cool in-memory provider for testing.
5. You can do code-first, or you can do DB first.

## Philosophy behind EF

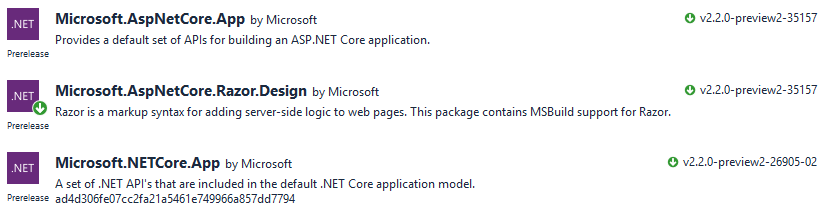
With EF (especially with code-first, which is the most popular), you really have to take a leap of faith.

You’re abstracted from the database by quite a lot. You can control details like constraints and default values and indexes, but for the most part you step away from that stuff whenever you can. All that logic should be in your data-service layer.

The database will now become a dumb thing that stores data.

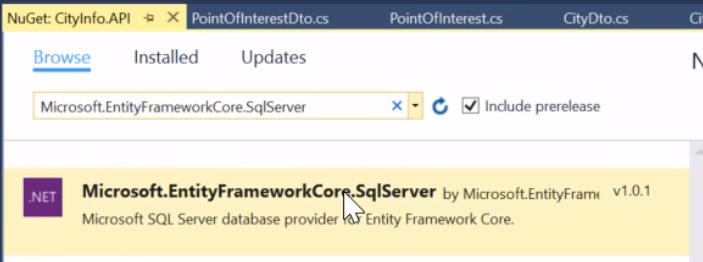
## Required NuGet packages

I think you only need to get these packages:



I’m pretty sure you don’t need Razor. I can’t remember how this got pulled in.

If you want to connect to SQL server, you might need more than this.

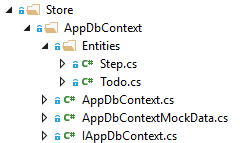


## Directory structure

Here’s what I recommend

* Have a project-level folder, called Store.
* Underneath that, have one folder for each context
* Beneath each context, have a third folder called entities.

Here’s a screenshot of what I did:



In the folder for the db context, you’ll need three files:

1. A class for your actual DB context
2. An interface to wrap the DB context, so you can unit test any logic that uses the DB context.
3. A class to add some mock seed data to the context

## Creating a DB context

A DB context (with a provider) allows us to connect to a database. Generally you’ll need one context per DB, but if you have a large DB with lots of tables for different feature areas then you can have multiple contexts to kind of split things up.

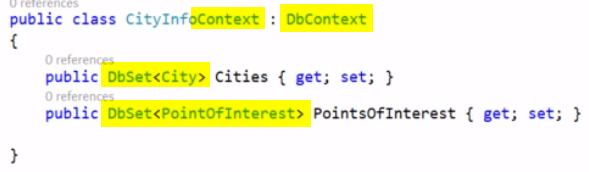
### Creating a DB Context

To create a DB context,

1. make a class that ends with the word Context,
2. have it inherit Microsoft.EntityFrameworkCore.DbContext.
3. Add one or more properties of type DbSet<T>, one for each entity you want in your context

EF will figure out all the PK/FK mappings, and just make it all magically work.

Like, literally. There really isn’t much more to it than that.



Now we need to tell ASP about our new DB context. You do this in Startup.cs, in the ConfigureServices() method.



NOTE: By default this will be registered with a scope lifetime (not sure what that means).

What I have noticed is that a new context gets created on every request.

### Connecting to a database

So, there are some considerations when keeping your connection string in an app-settings file:

1. Your connection string needs to change based on whether your microservice is running in a local dev environment or a test environment, or a production environment.
2. You DO NOT want to keep your production username / password in a local dev app-settings file, and you DO NOT want that file getting checked into source control.

For your production connection string, you want these pieces of info kept secret:

1. Db hostname.
2. DB instance name
3. Username / password
4. Anything else that would point a hacker to your database.

In the Pluralsight video the guy talked about using an environment variable to keep the production connection string (he put the whole thing as an env var).

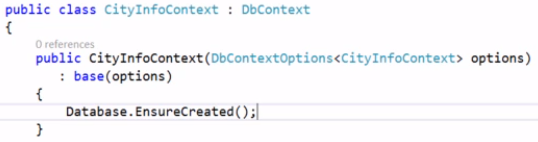
It turns out that the

### Using SQL Server LocalDB

Here is an example that uses SQL Server localdb:

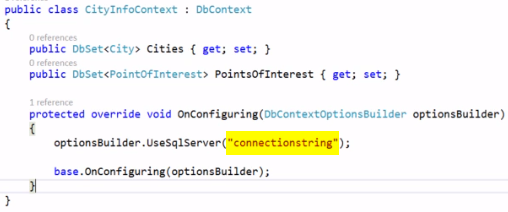


If you’re using localDB then you’ll want to have this call in the ctor for your context:



That’s only for development, though. I feel like you shouldn’t be using localDB unless your microservice owns the database.

Another way to set your connection string is by overriding the OnConfiguring() method, which you inherit from the DbContext base class.

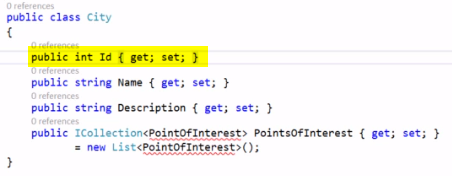


## Defining Entities

An entity is a class that holds a single record from a table. Usually the entity is singular, and the table name is plural.

### Primary keys

EF uses your Id field to be the PK. It will also look for <table-name>Id (for example CityId):

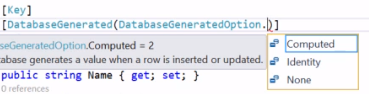


If you don’t like the convention-based approach, you can tag whatever field you like, using the [Key] attribute. I like this approach because it means I can forget about this project and then pick it up a good deal of time later and still understand it:

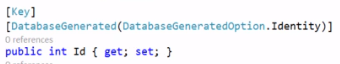


PKs are generated for you automatically, when you add. EF will assume that anything tagged with a [Key], or anything called Id or TableNameId will be a PK, and will make it an identity.

Again, if you’d rather override this, or if you DON’T want an identity then you can use the [DatabaseGenerated] attribute, like so:

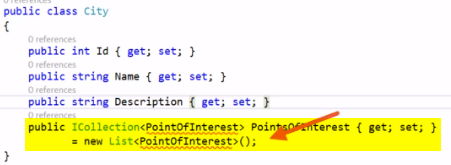


And here you go:



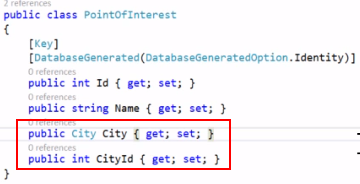
### Defining one to many relationships

In the entity for the parent table, you want to declare a list of items for the dependent collection:

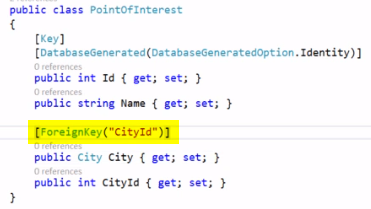


And on the dependent entity, you want to declare a “navigation property”. With all things, there is a convention-based approach, and an explicit approach.

By convention, EF will recognize a navigation property based on the name and type. As long as you have a property with a non-scalar type (a class name) that is also an entity, EF will set up the relationship for you. It is also recommended that we have an <table-name>Id property (not a requirement).

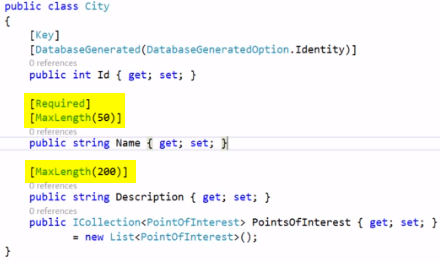


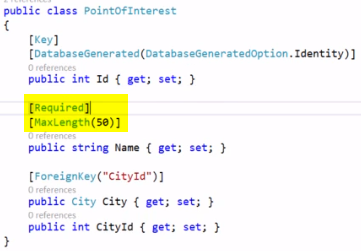
If you need to be explicit then you can use some data annotations:



### Setting constraints and other properties:

You’ll also want to add some constraints on your fields. You can set some as required, and set max lengths on strings:





## Migrations

EF has a tooling that will generate C# code which will take your database to the next version. This allows you to have logic that will handle DB upgrades

I am not sold on this idea.

For one thing, my lessons at inContact & DealerSocket have taught me what happens when databases get really, really large.

Using C# to create your database means you’ll eventually have a ton of C# that has to be compiled and merged, making a lengthy build process take even longer.

Maybe this should really be a cautionary tale against letting your databases get too freaking huge.

Working at DealerSocket has taught me that seed data for a development environment is best managed by a separate migration tool. Your migration tool should have these three components:

1. **Migration code:** The code to upgrade your DB from one version to the next.
   1. For most web-apps, you only need to maintain backward compatibility with whatever is currently in production. This means you only need the current version, some hot-fix versions, and the vNext version.
   2. If your DB is public then you’ll need an upgrade path for every version that you’ve supported. I don’t know why you’d ever run into this scenario, unless you were supporting some kind of premise-based / COTS product.
2. **Production seed data:** Data to provision a new customer, or a new production database
3. **Developer seed data:** This is , with some kind of comprehensive versioning

### Seed data

You have two kinds of seed data:

1. **Production seed data:** for static tables, or to initialize a customer with some initial settings.
2. **Developer seed data:** this gives developers something to test their app with so they don’t have to continually rebuild their development environments by hand.

In my opinion, all seed data should go into a migration tool, which builds your DB from the ground up.

The proper way to unit test is

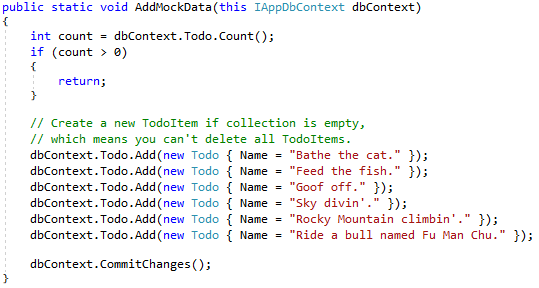
1. Mock the DB context, and have it return the data you need.
2. Each group of unit tests should be creating their own local seed data.
3. You can keep large collections of seed data in C# files or JSON / XML files which you import and export through your mock contexts.

**You should not be using seed data for unit testing**. There are a number of problems with this approach:

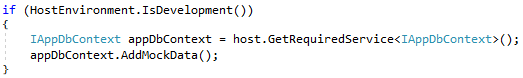
1. Database-based tests do not scale. The seed data becomes brittle, and you end up with a bunch of bad/stale data in your DB that no one knows how to fix. Often, this data is in an incorrect state because of changes to your business logic.
2. There is a tendency to re-use data across tests (DRY, right???), and this just makes tests brittle in the end.
3. and it does not allow your tests to be executed in parallel (which can really kill your build times)
4. It takes time to build/tear down your database.

That said, when you are still in the prototyping phase, you’ll need some seed data to get you off the ground. The best way to add seed data is to make an extension class that lives alongside your DB context. This extension class should get called in Startup.cs, at the end of your Configure method.

Here is an example seed data class:



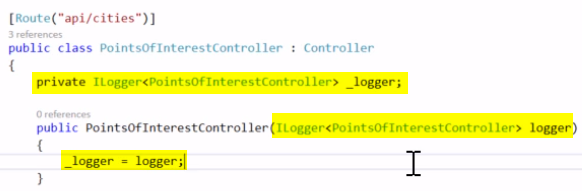
And you can call it in Startup.cs, at the end of your Configure() method.

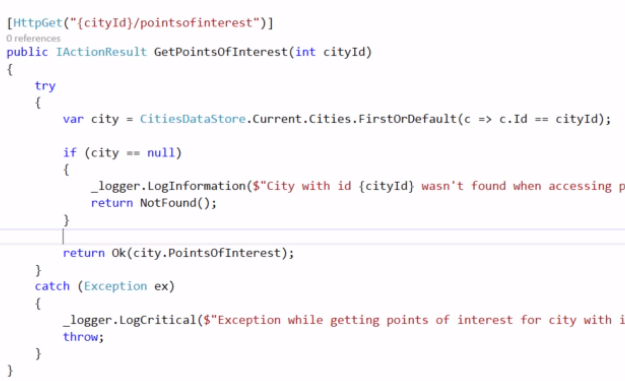


I could have had Configure() take another parameter for IAppDbContext, but this should be a temporary thing. This won’t scale well, at all, and I’m not convinced it should be the job of the microservice to initialize the database unless the DB is wholly an integral part of the microservice, itself.

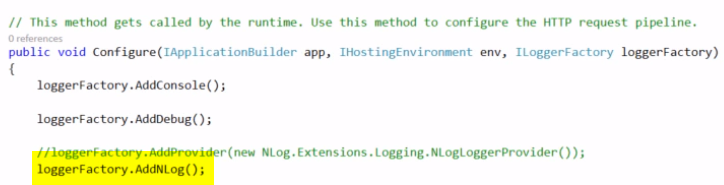
# Logging

Note: ASP already configures console + debug window logging in CreateDefaultBuilder(), which you call in Program.cs. To see exactly what it does, you can go to the actual source in GitHub.





You can also add third-party loggers. Here is an example using NLog (in Startup.cs):



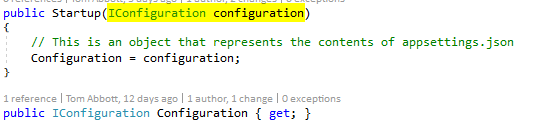
# App Settings (system configuration)

So, just like that SnapFactory that I built a long time ago, Net Core 2 has this whiz-bang configuration builder thing that you can just query for what you need.

Cool!

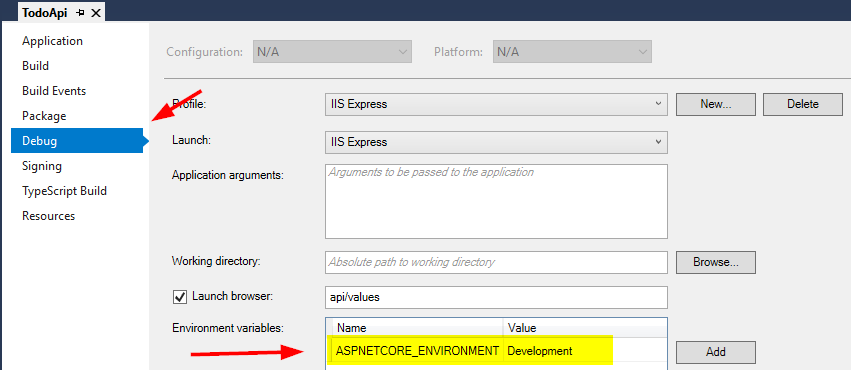
In Program.cs, in your CreateWebHostBuilder method, this calls CreateDefaultBuilder(). CreateDefaultBuilder builds a whole crap-ton of stuff for you. It’s amazing. One of the things it does is configure a chain of environment / app-settings / command-line variables for you, which you can then query through ONE easy-to-use interface!

You can ask for an interface to this object in Startup.cs, in your Startup() method, and then store it as a public property on your startup class:

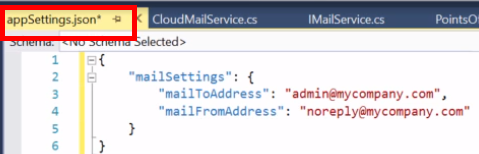


.net core no longer uses app.config / web.config files (thank heaven---ugh!)

Ok, so you can now store settings in a JSON file, called appSettings.json. You can also have appSettings.Production.json, and appSettings.Staging.json. These will be chosen based on the ASPNETCORE\_ENVIRONMENT environment variable:



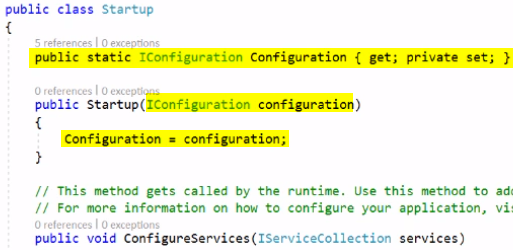
Here is a .json file that has some email settings:



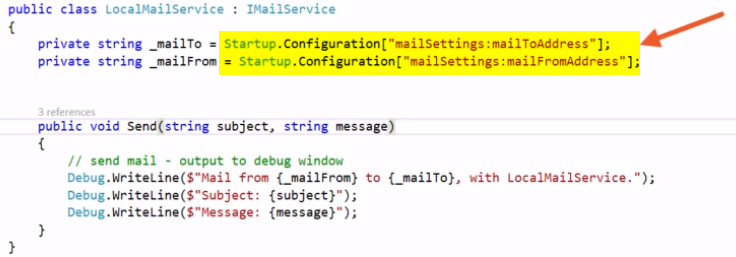
Now we need to configure this in the startup class.

1. There’s a configuration interface that ASP provides, called IConfiguration. We need to store an instance of this in a static field somewhere. A good place is our startup class
2. To build an IConfiguration, we can specify it as a ctor parameter, and ASP will give us one. We then cache this interface, and we’re ready to go.

Here is the code to do this (I think this was from ASP 2.2):

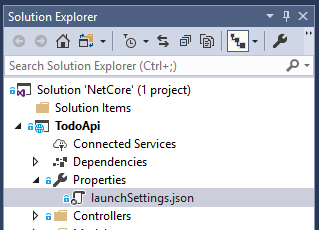


Now we can call it anywhere we need to pull in config data. This is SOOOOOO much more modular than app.config!



## LaunchSettings.json

I’m not sure what this file does, but it can end up having some sensitive information in it, so you need to be aware of it, and make sure that sensitive info doesn’t get checked in



If you have to set an environment variable, do not set it in VS. Set it in your system properties dialog (Windows-key + Pause-key)