# 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:

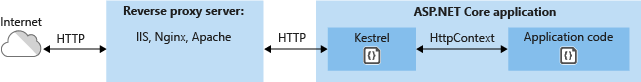
<https://stackify.com/what-is-kestrel-web-server/>

<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:

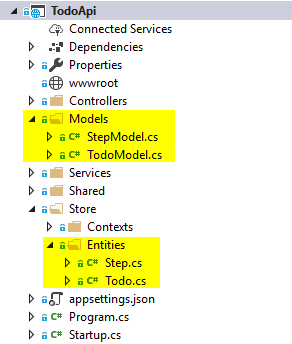
* At the Entity Framework layer: you have all kinds of attributes and properties that EF wants to see, but which mean nothing to WebAPI.
* At the WebAPI layer: vice versa—you don’t need any of that EF stuff, and you’ll want to attach all kinds of additional attributes for validation and hinting for WebAPI.

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.

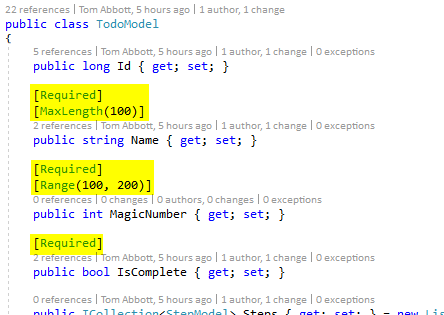
ModelState offers the following properties / methods:

* 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.

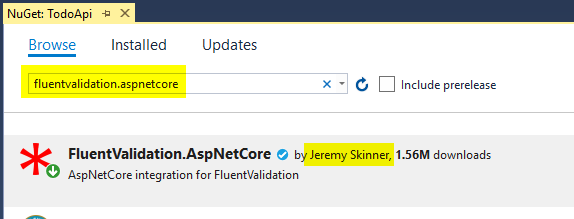
## FluentValidation

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:

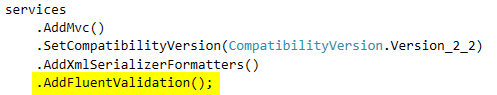


Now, go into Startup.cs, and look for your ConfigureServices() function. Add some using statements:

using FluentValidation;

using FluentValidation.AspNetCore;

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



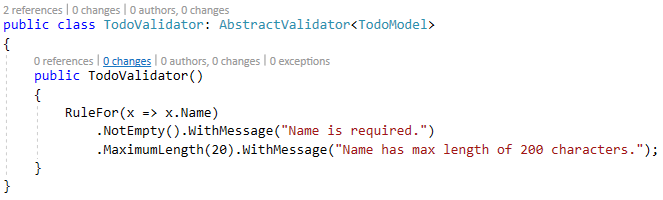
The documentation says to call AddFluentValidation() right after calling AddMvc(), but I found that it still worked. Whatever.

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.

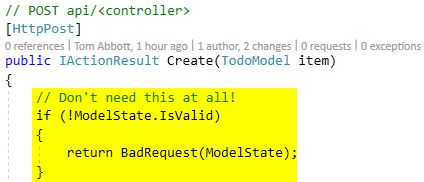
Now you need to actually write a validator. That’s pretty easy, too.



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 Service layer / custom services

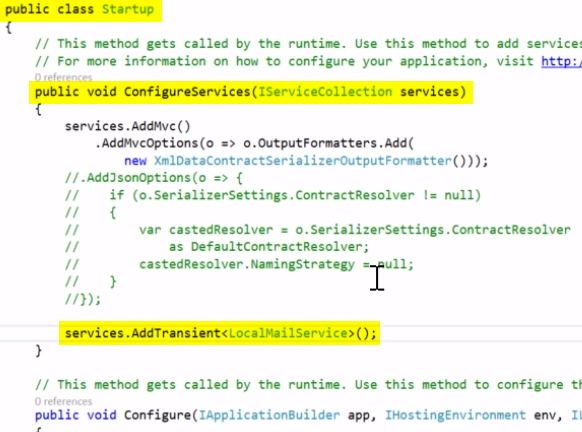
Put it into the Services folder:



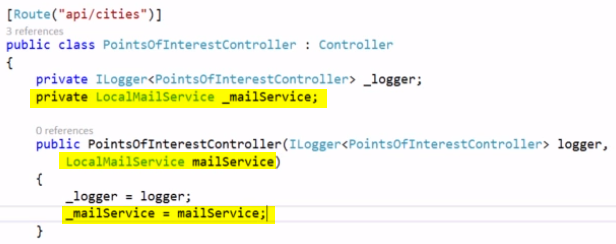
Now just write a class, like you normally would:



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).



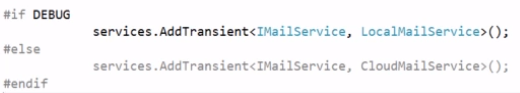
Now we can use it via ctor injection:



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.

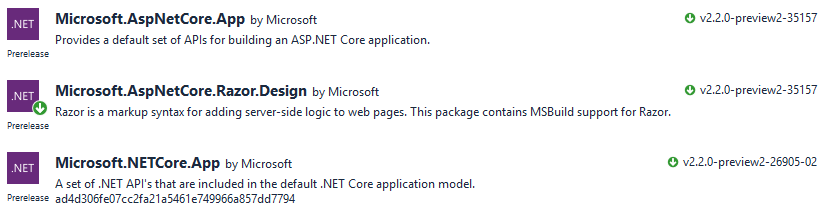
# Entity Framework Core

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.

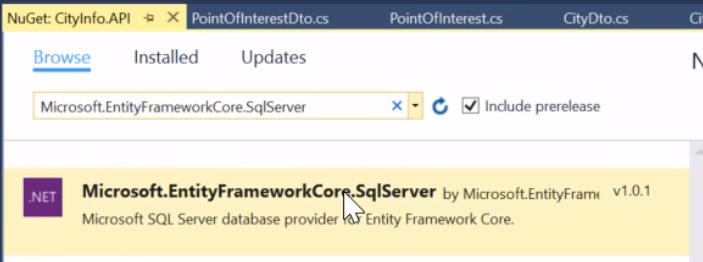
## Required NuGet packages

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

I created a directory called Store. Underneath that, I have two other directories:

1. **Contexts**: This is where all our DB contexts live. We have one of these per DB. Since it is common for web-apps to talk to multiple databases, it’s smart to have this in its own folder.
2. **Entities**: This is where all the entities live
   1. NOTE: if we’re really going to have multiple databases, we might want to have one subdirectory under the Entities folder, for each 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.

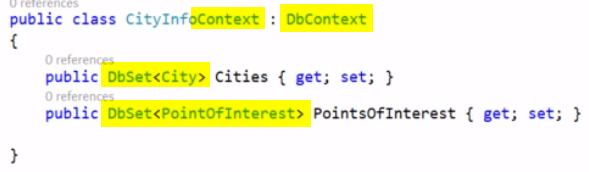
### Creating a DB Context

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

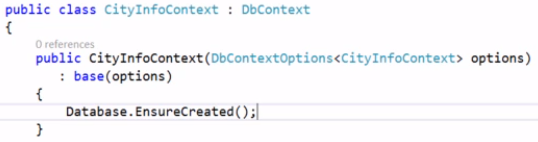
You’ll also need to set up a connection string. There are two ways to do this.

The best way is to set it in Startup.cs, in your ConfigureServices method when you call AddDbContext().

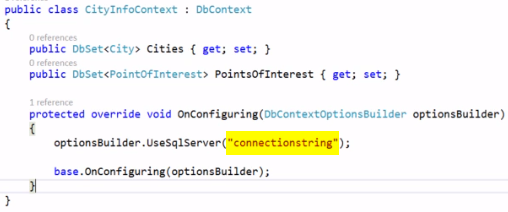
Here is an example that uses SQL Server localdb:



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



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

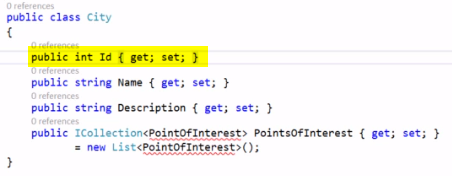


## Defining Entities

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):

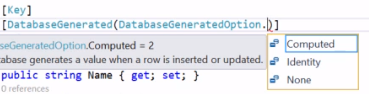


If you don’t like the convention-based approach, you can tag whatever field you like, using the [Key] attribute. I like this approach because it means I can forget about this project and then pick it up a good deal of time later and still understand it:

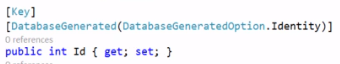


PKs are generated for you automatically, when you add. EF will assume that anything tagged with a [Key], or anything called Id or TableNameId will be a PK, and will make it an identity.

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

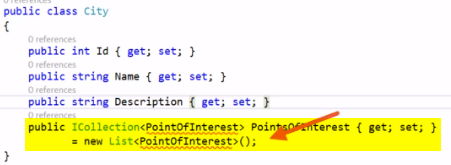


And here you go:



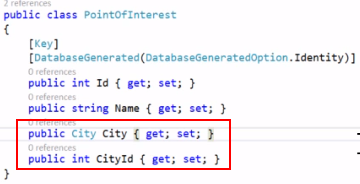
### Defining one to many relationships

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

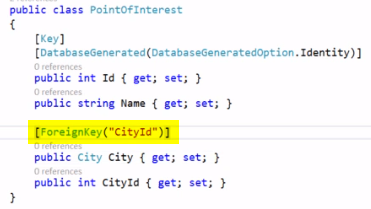


And on the dependent entity, you want to declare a “navigation property”. With all things, there is a convention-based approach, and an explicit approach.

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).

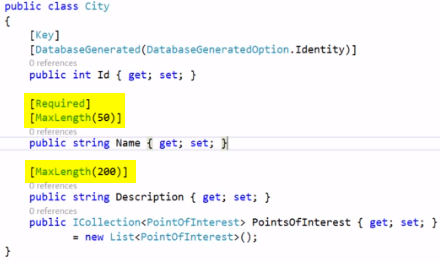


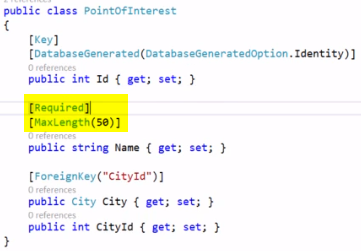
If you need to be explicit then you can use some data annotations:



### 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:



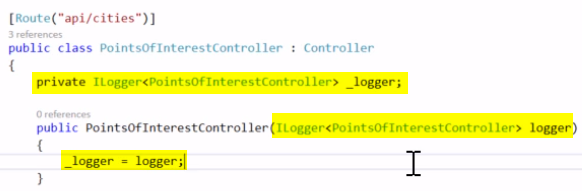


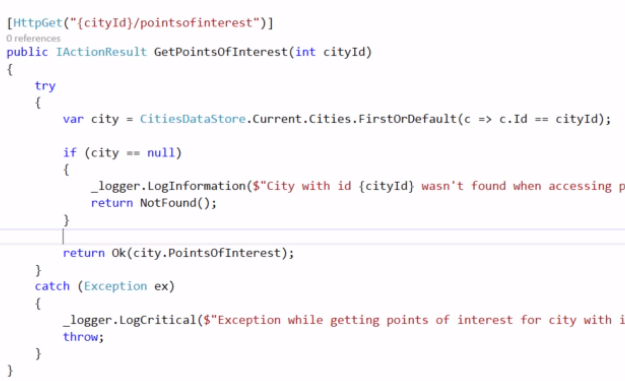
### Seed data

You’ll want to have seed data for your database. Probably the best place to keep it is in your store along with your DbContext, but have it called in the startup code (I.e., Startup.cs) for your app. I think that keeps a clean separation of concerns.

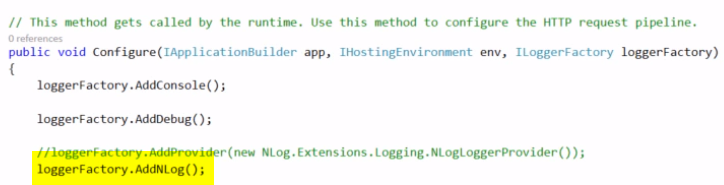
# Logging

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.



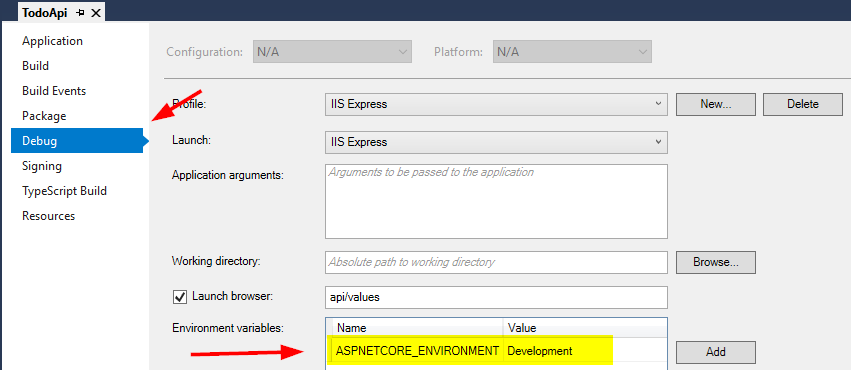


You can also add third-party loggers. Here is an example using NLog (in Startup.cs):

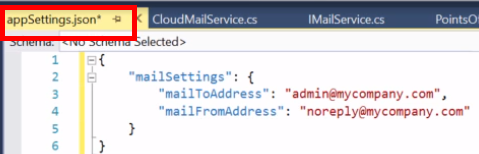


# Settings using JSON

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:



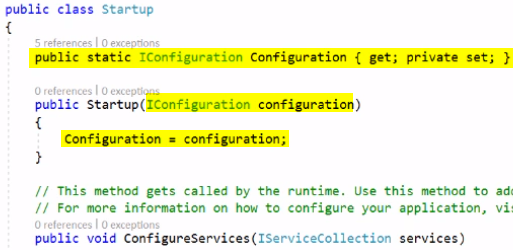
Here is a .json file that has some email settings:



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:



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

