# Kestrel web server:

Find info at:

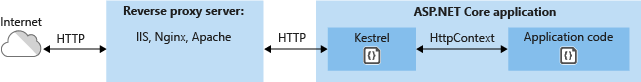
<https://stackify.com/what-is-kestrel-web-server/>

<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

# Pluralsight: Building Your First API with ASP.NET Core

## Module 4: Manipulating Resources

1. Strong recommendation to separate the POCO used for the database from the POCO used to serialize and de-serialize request/response payloads.
2. Demo: Validating input
   1. The new WebAPI has a validation object, called ModelState. You can do ModelState.IsValid, and ModelState.AddModelError.
3. FluentValidation
   1. This is even more cool. You can create validation objects in a separate library, and keep complete separation of concerns.
   2. The important thing is that it keeps you from having to repeat validation code.
4. PATCH: 😊
   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.

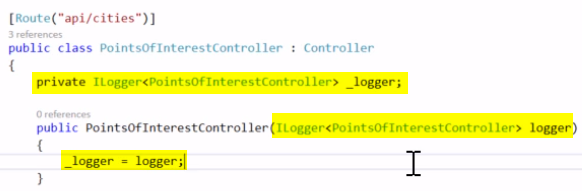
## Module 5: Services, and MVC’s DI framework.

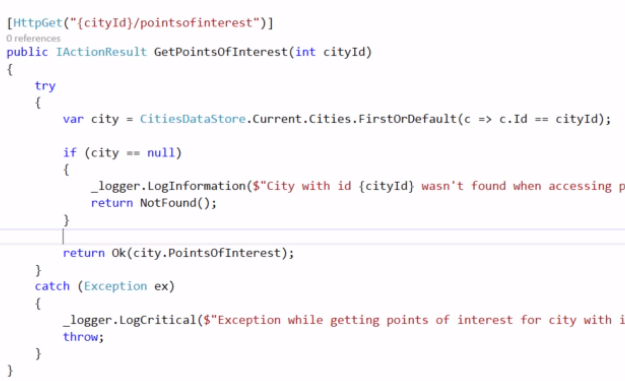
Author goes into detail on MS’s DI framework.

1. It looks like all controllers and the Startup class get DI-ed, out of the box. You don’t even need attributes, or anything. Magical, mystical reflection!
2. You can request something from the service layer using the HttpContext object, using: HttpContext.RequestServices.GetService()
   1. Advised that you use ctor injection, instead.

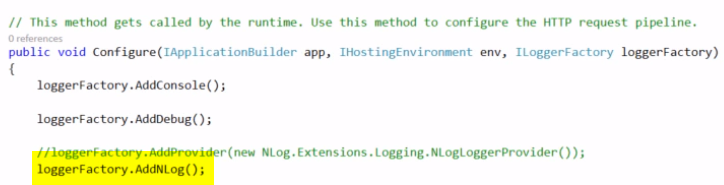
### Logging is a good example of DI in ASP.

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):



### You can add your own custom services

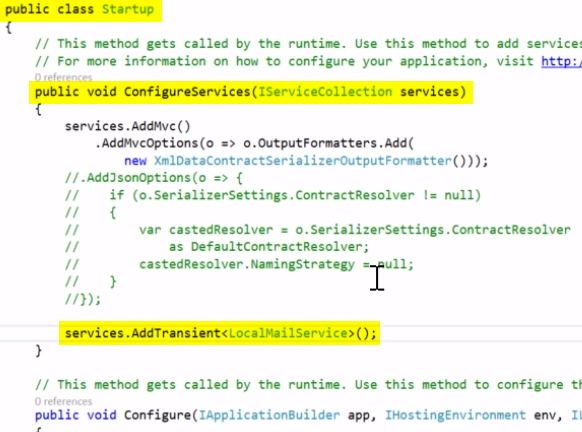
Put it into the Services folder:



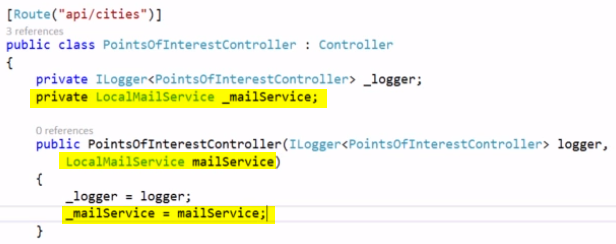
Now just write a class, like you normally would:



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).



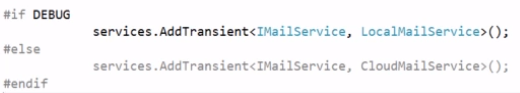
Now we can use it via ctor injection:



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.

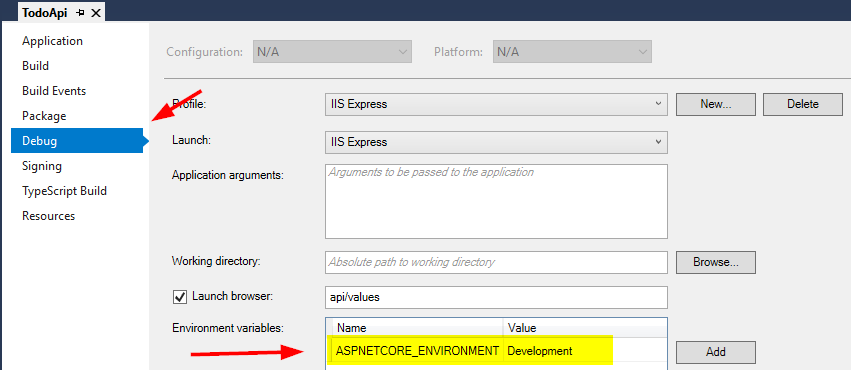


You can do all kinds of fancy kinds of things with this. Here is an example of using a production service for prod, and a dev service for your local environment:

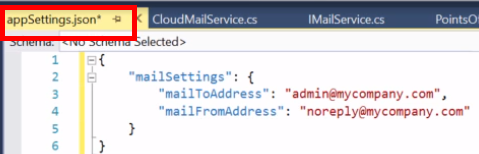


### Settings using JSON

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:



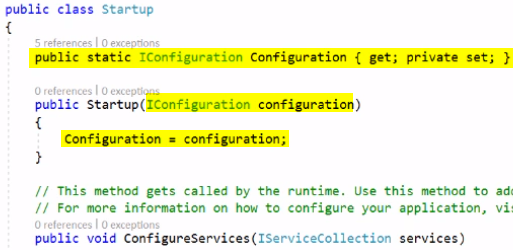
Here is a .json file that has some email settings:



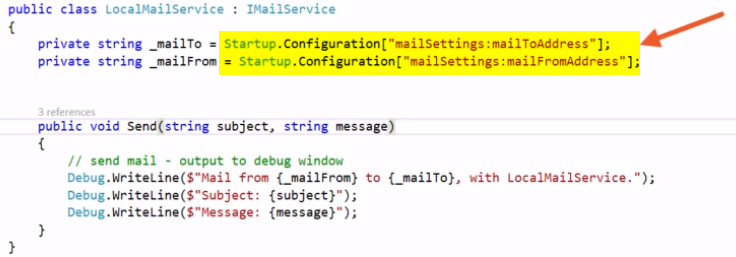
Now we need to configure this in the startup class.

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!



### Module 6: Getting Acquainted with 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. 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.