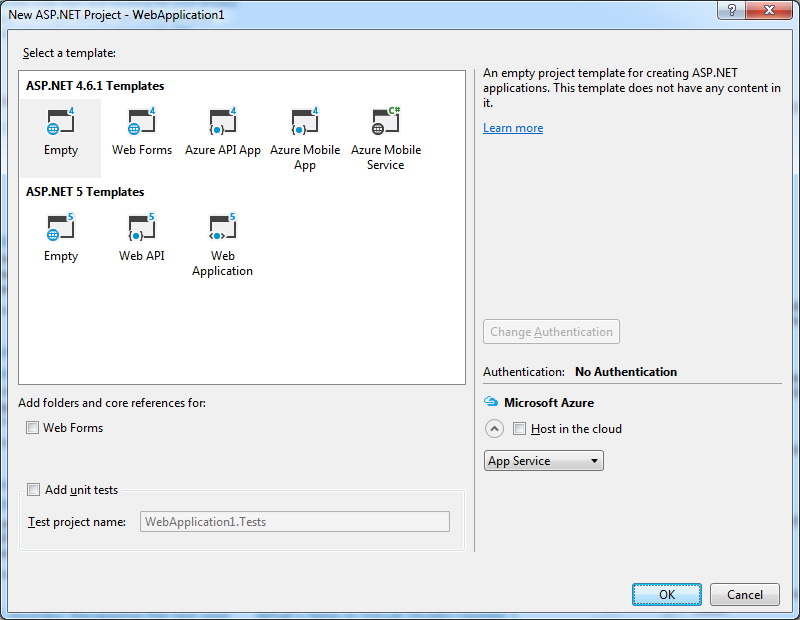
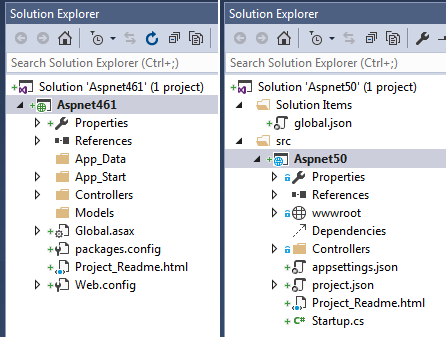
**API project template**

Selecting *ASP.NET Web Application* from the *New Project* dialog results in the following dialog:



No prizes for guessing that I’m going to choose ***Web API*** from the *ASP.NET 5* templates.

For comparison, I’m also going to create another new web application using the ***Azure API App*** template (I’m not interested in Azure for this article but there is no plain old Web API option under ASP.NET 4.6.1 - don’t worry, there is actually nothing Azure specific in the screenshot anyway). The results are shown below:



So there are a few obvious differences:

* Startup.cs is included
* App\_Data, App\_Start and Global.asax have been removed
* Three new \*.json files have been added
* (There are two fancy new nodes: wwwroot and Dependencies)

I’m not going to talk about wwwroot or Dependencies because they are covered in [our previous post](http://blog.scottlogic.com/2015/05/14/aspnet50-intro.html#client-side-libraries) and they are not relevant to writing a RESTful API. I will talk about the other changes in the rest of the post.

**Startup.cs**

The [Startup](https://docs.asp.net/en/latest/fundamentals/startup.html#the-startup-class) class seems to have been inspired by [OWIN](http://owin.org/). In recent years OWIN has become a hugely popular .NET tool for decoupling web server and web application. Microsoft have taken notice so Startup is now *the one and only place* to start up your web application. It is important to note that this is not an OWIN class, just an OWIN-like class.

By convention, the framework finds the Startup class and runs its Main method (which uses a new [C#6 “lambda method”](https://github.com/dotnet/roslyn/wiki/New-Language-Features-in-C%23-6#user-content-expression-bodied-function-members)) to bootstrap the application using the Startup class. This is what the class looks like out of the box:

**public** **class** **Startup**

{

**public** **Startup**(IHostingEnvironment env)

{

*// Set up configuration sources.*

**var** builder = **new** **ConfigurationBuilder**()

.**AddJsonFile**("appsettings.json")

.**AddEnvironmentVariables**();

Configuration = builder.**Build**();

}

**public** IConfigurationRoot Configuration { **get**; **set**; }

*// This method gets called by the runtime. Use this method to add services to the container.*

**public** **void** **ConfigureServices**(IServiceCollection services)

{

*// Add framework services.*

services.**AddMvc**();

}

*// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.*

**public** **void** **Configure**(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

loggerFactory.**AddConsole**(Configuration.**GetSection**("Logging"));

loggerFactory.**AddDebug**();

app.**UseIISPlatformHandler**();

app.**UseStaticFiles**();

app.**UseMvc**();

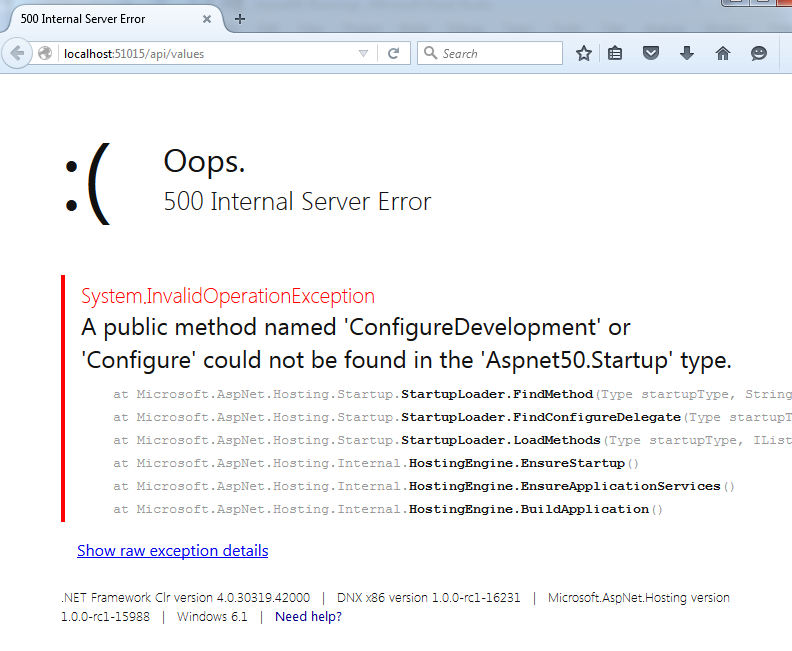
}

*// Entry point for the application.*

**public** **static** **void** **Main**(**string**[] args) => WebApplication.Run<Startup>(args);

}

The Configure method is required by convention. Let’s see what happens if you take it out:



This error message reveals that the framework will look for a method named either Configure or ConfigureDevelopment. The framework will actually look for a method named Configure{EnvironmentName} on the Startup class, where {EnvironmentName} is read from the **ASPNET\_ENV** or **Hosting:Environment** environment variable which you can set in the environment itself (**GOTCHA:** I had to restart Visual Studio after changing the Windows 7 system environment variable before the value would be picked up) or using **launchSettings.json**:

{

...

"profiles": {

"IIS Express": {

...

"environmentVariables": {

"Hosting:Environment": "Development"

}

},

"web": {

...

"environmentVariables": {

"Hosting:Environment": "Development"

}

}

}

}

We also have the option of creating a set of Startup{EnvironmentName} classes, one of which will be selected based on the current environment name. If no environment name is configured then Production will be used - a safe and sensible default!

So ASP.NET 5 gives us a neat new option for writing environment specific code just by applying a naming convention to Startup classes and methods, but that is just one of your options. You can also inject the new [IHostingEnvironment](https://github.com/aspnet/Hosting/blob/cad9ea1df7e78c52c99346de17b6fc3c2bc2c67e/src/Microsoft.AspNet.Hosting.Abstractions/IHostingEnvironment.cs) interface anywhere that you would like to have environment specific code (including the Startup constructor!). See [Working with Multiple Environments](http://docs.asp.net/en/latest/fundamentals/environments.html) for more information.

**No Global.asax**

If you have ever written a Web API 2 application in the past you will be familiar with the **Global.asax** file, which contains a WebApiApplication class by default:

**public** **class** **WebApiApplication** : System.Web.HttpApplication

{

**protected** **void** **Application\_Start**()

{

GlobalConfiguration.**Configure**(WebApiConfig.Register);

}

}

This means the static WebApiConfig.Register method will be called when the application is started:

**public** **static** **class** **WebApiConfig**

{

**public** **static** **void** **Register**(HttpConfiguration config)

{

*// Web API configuration and services*

*// Web API routes*

config.**MapHttpAttributeRoutes**();

config.Routes.**MapHttpRoute**(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: **new** { id = RouteParameter.Optional }

);

}

}

This code switches on attribute based routing and sets up Web API routing conventions. In the new world the same is achieved in the Startup class by calling services.AddMvc() from ConfigureServices and app.UseMvc() from Configure. So we don’t need any smelly GlobalConfiguration class any more.

**Controllers**

Here is the automatically created ValuesController:

[Route("api/[controller]")]

**public** **class** **ValuesController** : Controller

{

*// GET: api/values*

[HttpGet]

**public** IEnumerable<**string**> **Get**()

{

**return** **new** **string**[] { "value1", "value2" };

}

*// GET api/values/5*

[**HttpGet**("{id}")]

**public** **string** **Get**(**int** id)

{

**return** "value";

}

*// POST api/values*

[HttpPost]

**public** **void** **Post**([FromBody]**string** **value**)

{

}

*// PUT api/values/5*

[**HttpPut**("{id}")]

**public** **void** **Put**(**int** id, [FromBody]**string** **value**)

{

}

*// DELETE api/values/5*

[**HttpDelete**("{id}")]

**public** **void** **Delete**(**int** id)

{

}

}

My first observation was the base class. Pretty much all Web API 2 controllers inherit from ApiController but [Controller](https://github.com/aspnet/Mvc/blob/dev/src/Microsoft.AspNet.Mvc.ViewFeatures/Controller.cs) is the base class in the new world. In MVC 6 the same Controller base class can be used whether you are writing a RESTful API or an MVC website. The framework also supports [POCO](https://en.wikipedia.org/wiki/Plain_Old_CLR_Object) controllers so you can remove the base class completely and a request to ‘api/values’ will still work. This is made possible by [convention based controller discovery](http://www.strathweb.com/2015/04/asp-net-mvc-6-discovers-controllers/), which is completely pluggable.

**ControllerBase**

On 10th December 2015 Microsoft introduced [a new ControllerBase class](https://github.com/aspnet/Mvc/commit/a2393f21be55c5c02ea67df55407aa7585121173) which helps us implement a RESTful API controller without bringing in unnecessary properties/methods related to MVC websites such as ViewBag, ViewData and ViewResult. At the time of writing this is still on the **dev** branch so it will be interesting to see where Microsoft end up going with this new class…

**Routing**

You can specify a base route at the controller level using the new ‘[controller]’ placeholder instead of hard-coding the controller name (‘[action]’ is also supported but this would produce unRESTful URIs like ‘api/values/get’). Routing has also been nicely combined with the various [Http{Verb}] attributes so you can succinctly state that a PUT request with and ‘id’ will be handled by the Put method:

[HttpPut("{id}")]

**public** **void** **Put**(**int** id, [FromBody]**string** **value**)

{

}

If you have used Web API 2 then you might expect the route to ‘api/values’ to work with no attribute routing at all. See the following controller:

**public** **class** **ValuesController** : ApiController

{

**public** IEnumerable<**string**> **Get**()

{

**return** **new** **string**[] { "value1", "value2" };

}

}

The above Web API 2 controller would handle a GET request to ‘api/values’ for two reasons.

Firstly, the correct **controller** was selected because convention based routing is configured by default:

**public** **static** **class** **WebApiConfig**

{

**public** **static** **void** **Register**(HttpConfiguration config)

{

...

config.Routes.**MapHttpRoute**(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: **new** { id = RouteParameter.Optional }

);

}

}

Secondly, the correct **action** was selected based on naming convention so a GET request will resolve to the Get method.

MVC 6 still allows actions to be selected based on the name matching the HTTP verb of the request, but since attribute based routing is now the recommended approach, you will no longer get a default routing configuration for selecting controllers by name.

Books API

I’ve written a basic API which is a tweaked version of Microsoft’s tutorial: [Building Your First Web API with MVC 6](https://docs.asp.net/projects/mvc/en/latest/getting-started/first-web-api.html) based on books instead of to-do items. I’ll describe my thoughts having followed through the tutorial so you can either do the tutorial first or get [my code from Github](https://github.com/niksoper/aspnet5-books/tree/blog-dotnet-rc1).

**Dependency injection**

In order to implement a books API I’ll need somewhere to store my books so I’ve created a basic repository which will store a collection of books in memory. The repository code is not of interest for this post, you just need to understand that there is an IBookRepository which is implemented by BookRepository.

**Property injection**

For some reason Microsoft decided to use property injection to introduce a repository to their TodoController:

**public** **class** **TodoController** : Controller

{

[FromServices]

**public** ITodoRepository TodoItems { **get**; **set**; }

}

I can only assume that property injection was chosen by the author in order to show off the [FromServices] attribute because I think constructor injection should be preferred since it advertises the dependencies of a class. I guess it is useful to know that property injection is also supported out of the box but the tutorial may push a less experienced developer towards using property injection as the main way to inject dependencies.

End of minor rant.

**Configuring dependencies**

I’m injecting an IBookRepository into the BooksController constructor:

**private** **readonly** IBookRepository books;

**public** **BooksController**(IBookRepository books)

{

**this**.books = books;

}

Registering the concrete BookRepository is a one-liner in Startup.ConfigureServices():

**public** **void** **ConfigureServices**(IServiceCollection services)

{

services.**AddMvc**();

services.AddSingleton<IBookRepository, BookRepository>();

}

That’s it.

The framework will now instantiate the controller with the same BookRepository instance on every request.

Compare this with the steps that I would need to take to introduce dependency injection into a Web API 2 project using [Autofac](http://docs.autofac.org/en/stable/integration/webapi.html):

1. Add ‘Autofac.WebApi2’ via NuGet
2. Create a ContainerBuilder
3. Register the API controller classes
4. Configure dependencies
5. Set the HttpConfiguration.DependencyResolver

So you may end up with some code like this:

**public** **class** **IocConfig**

{

**public** **static** **void** **Register**(HttpConfiguration config)

{

**var** builder = **new** **ContainerBuilder**();

builder.**RegisterApiControllers**(Assembly.**GetExecutingAssembly**());

builder.RegisterType<BookRepository>().As<IBookRepository>().**SingleInstance**();

**var** container = builder.**Build**();

config.DependencyResolver = **new** **AutofacWebApiDependencyResolver**(container);

}

}

This was fairly straightforward and you could have this up and running in no time at all if you’ve done this before but you are still likely to be slowed down when starting up a project by the need to bring in a 3rd party solution like Autofac. I really like the way you can get up and running with DI support straight away with ASP.NET 5.

**Model binding**

I want to add a book using the following HTTP request:

POST http://localhost:5000/api/books HTTP/1.1

Host: localhost:5000

Content-Type: application/json

{

"title": "my title",

"author": "my author"

}

Here is the BooksController focussed on the signature for my Create method:

[Route("api/[controller]")]

**public** **class** **BooksController** : Controller

{

...

[HttpPost]

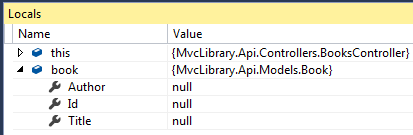
**public** IActionResult **Create**([FromBody]Book book)

...

}

The [FromBody] attribute looks familiar enough to Web API 2 developers, but in Web API 2 it is used to tell the framework to get a primitive value from the body of a request. In Web API 2 a complex type like Book would automatically be read from the request body, so what is it doing here?

Let’s try removing the [FromBody] attribute:



The above screen shot shows that neither the Author nor the Title property was set, even though the request contained these values as JSON.

[FromBody] is required if you want to bind an action parameter the body of an ‘application/json’ request. This is pretty annoying if you want to write a RESTful API since you are likely to want to do this on all POST and PUT requests. It is perfectly possible to apply a convention to your controller actions as describe in [this post](http://www.dotnetcurry.com/aspnet-mvc/1149/convert-aspnet-webapi2-aspnet5-mvc6), but this fairly major change in the rules between Web API 2 and MVC 6 will be frustrating for many experienced Web API 2 developers.

It’s worth noting that **as long as there is no [FromBody] (or similar) on a complex action parameter** then the framework will also allow properties to be bound via the query string like this:

POST http://localhost:5000/api/books?title=query+title&author=query+author HTTP/1.1

Host: localhost:5000

Content-Type: application/json

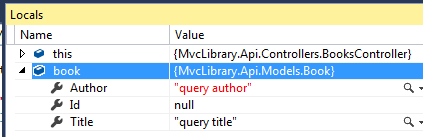
{

"title": "body title",

"author": "body author"

}

In this case the body is ignored and the title and author are bound from the query string. Note the Author and Title properties shown in the Locals window below:



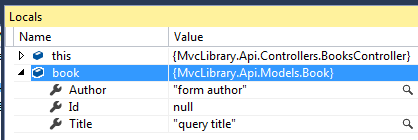
The fun doesn’t stop there. If you use content-type ‘application/x-www-form-urlencoded’ you can also mix and match between query string and body like this:

POST http://localhost:5000/api/books?title=query%20title HTTP/1.1

Host: localhost:5000

Content-Type: application/x-www-form-urlencoded

title=form+title&author=form+author



This time the body contains the author and title, and the query string also contains the title. This means the Title property is bound from the query string, not the body.

Check out [this post](https://lbadri.wordpress.com/2014/11/23/web-api-model-binding-in-asp-net-mvc-6-asp-net-5/) for a nice summary of model binding in the new world.

**Returning from actions**

The IActionResult class is the flexible choice and this will feel familiar to ASP.NET MVC developers because previous versions of ASP.NET MVC provided an abstract ActionResult class and all action methods needed to return a type that derived from ActionResult.

One benefit of using the new Controller base class is that you get a set of helpful methods for returning various flavours of IActionResult. I’ve used four of these helper methods in my BooksController:

[HttpGet("{id}", Name = "GetBook")]

**public** IActionResult **GetById**(**string** id)

{

**var** book = **this**.books.**Find**(id);

**if** (book == **null**)

{

**return** **this**.**HttpNotFound**();

}

**return** **this**.**Ok**(book);

}

[HttpPost]

**public** IActionResult **Create**([FromBody]Book book)

{

**this**.books.**Add**(book);

**return** **this**.**CreatedAtRoute**("GetBook", **new** { controller = "Books", id = book.Id }, book);

}

[HttpPut("{id}")]

**public** IActionResult **Update**(**string** id, [FromBody]Book book)

{

**if** (book.Id != id)

{

**return** **this**.**HttpBadRequest**();

}

**if** (**this**.books.**Find**(id) == **null**)

{

**return** **this**.**HttpNotFound**();

}

**this**.books.**Update**(book);

**return** **new** **NoContentResult**();

}

So I’ve used:

* HttpNotFound(): **404** response with no body
* Ok(book): **200** response with a JSON formatted book in the body
* HttpBadRequest(): **400** response with no body
* CreatedAtRoute("GetBook", new { id = book.Id }, book): **201** response with a Location header containing the URI of the new resource (based on the named route: "GetBook") and a JSON formatted book in the body

Note that we always have the option of creating a concrete implementation of IActionResult without the use of a helper method. The happy path of the Update method above just returns a new NoContentResult() without needing to call a helper. This will produce a **204** response.

Incidentally, RC1 does not provide a helper method for creating a NoContentResult but [Microsoft have since added one](https://github.com/aspnet/Mvc/blame/2063356f2442cee3c538bd2abc47b6cbbef7cf3a/src/Microsoft.AspNet.Mvc.Core/ControllerBase.cs#L286). The usefulness of this helper is questionable, but it looks like it will be part of ControllerBase in RC2.

These should all feel familiar enough to Web API 2 developers (although I admit that CreatedAtRoute() was new to me). There are plenty of other helpers that will be familiar to MVC and Web API 2 developers alike so it would be worth checking out the source code for [Controller](https://github.com/aspnet/Mvc/blob/master/src/Microsoft.AspNet.Mvc.ViewFeatures/Controller.cs) or [ControllerBase](https://github.com/aspnet/Mvc/blob/dev/src/Microsoft.AspNet.Mvc.Core/ControllerBase.cs) to see what is available.

**Other return types**

In my opinion IActionResult is very often the correct return type since it offers the most flexibility but returning void, string or [POCO](https://en.wikipedia.org/wiki/Plain_Old_CLR_Object) is also supported if you don’t need the flexibility of IActionResult and want a simple 200 response.

A couple of noteworthy points:

* void gives you a 200 response instead of the (surely, more appropriate) 204 response that Web API 2 produces.
* string gives you content-type ‘text/plain’ instead of ‘application/json’, which seems like a more sensible change.

**Logging**

Any production system needs logging and that usually means writing some basic infrastructure based around ILoggerFactory and ILogger services.

ASP.NET 5 introduces a new [logging framework](https://github.com/aspnet/Logging) which provides these services for us. When you create a new ASP.NET 5 application you get the ILoggerFactory injected into the default Startup.Configure() method:

**public** **void** **Configure**(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

loggerFactory.**AddConsole**(Configuration.**GetSection**("Logging"));

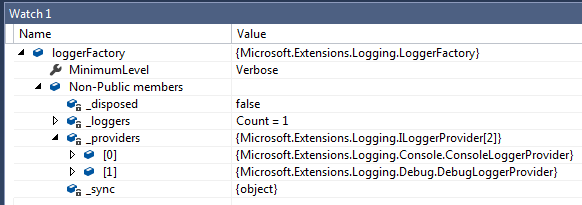
loggerFactory.**AddDebug**();

app.**UseIISPlatformHandler**();

app.**UseStaticFiles**();

}

The AddConsole() and AddDebug() methods that you see here are extension methods that call ILoggerFactory.AddProvider(). I’ve set a breakpoint and added the loggerFactory to the watch window:



So we automatically get console logging and debug logging (Visual Studio output window) in our application. This is all very nice but these loggers aren’t going to be of much use in production so I’m going to add another provider that will log to a text file (I don’t want to be distracted by setting up a database).

I’m going to use [serilog](http://serilog.net/). [This post](http://www.davidhayden.me/blog/asp-net-5-logging-using-serilog-and-rolling-log-file) describes how to configure serilog as a provider. It’s simply a case of adding a reference to the ‘Serilog.Framework.Logging’ package in **project.json** and adding some configuration:

**var** log = **new** Serilog.**LoggerConfiguration**()

.MinimumLevel.**Debug**()

.WriteTo.**RollingFile**(

pathFormat: env.**MapPath**("MvcLibrary-{Date}.log"),

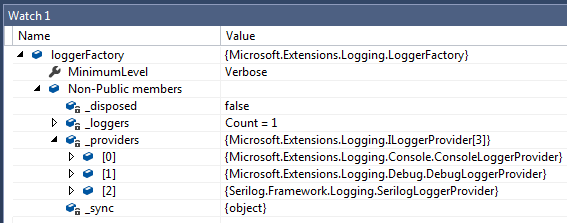
outputTemplate: "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} {SourceContext} [{Level}] {Message}{NewLine}{Exception}")

.**CreateLogger**();

loggerFactory.**AddSerilog**(log);

I’ve chosen to override the default serilog outputTemplate because I want to include the ‘{SourceContext}’ in the template.

The AddSerilog() method adds a provider using the given serilog configuration:



We now have three providers configured so we’ll get logs to the console, debug output and a local file.

**Running from the console**

You will only be able to see console logs if you launch the application from the console. You can do this by running **dnx web** from the project directory (the one containing **project.json**), or you can use Visual Studio to do the same thing:



What you are actually doing is running the web command from **project.json**:

{

...

"commands": {

"web": "Microsoft.AspNet.Server.Kestrel"

},

...

}

**Resolving a logger**

Having configured the ILoggerFactory you could now inject an ILoggerFactory into your controller and call the [CreateLogger<T> extension method](https://github.com/aspnet/Logging/blob/d679c78ab257375d26343207877b3d483c093d29/src/Microsoft.Extensions.Logging.Abstractions/LoggerFactoryExtensions.cs#L18) to create a logger instance named after a given type. However, I’m going to inject an ILogger<BooksController> directly:

[Route("api/[controller]")]

**public** **class** **BooksController** : Controller

{

**private** **readonly** IBookRepository books;

**private** **readonly** ILogger logger;

**public** **BooksController**(IBookRepository books, ILogger<BooksController> logger)

{

**this**.books = books;

**this**.logger = logger;

}

...

}

This is neat because the framework authors have realised that most consumers would just use the ILoggerFactory to create an ILogger with a name relevant to the context from which it is being called, so when the framework sees that we want an ILogger<BooksController> it uses the configured ILoggerFactory to provide one so we don’t have to pollute the constructor code with a call to CreateLogger.

The [default LoggerFactory](https://github.com/aspnet/Logging/blob/1308245d2c470fcf437299331b8175e2e417af04/src/Microsoft.Extensions.Logging/LoggerFactory.cs) maintains an in-memory dictionary of Logger instances keyed by name so injecting an ILoggerFactory<ConcreteType> anywhere in your application will resolve to the same logger instance as long as the type is the same.

**Using a logger**

I’ll log a message whenever a book is created or edited using the ILogger.LogVerbose() extension method:

[HttpPost]

**public** IActionResult **Create**([FromBody]Book book)

{

...

**this**.logger.**LogVerbose**("Added {0} by {1}", book.Title, book.Author);

...

}

[HttpPut("{id}")]

**public** IActionResult **Update**(**string** id, [FromBody]Book book)

{

...

**this**.logger.**LogVerbose**(

"Updated {0} by {1} to {2} by {3}",

existingBook.Title,

existingBook.Author,

book.Title,

book.Author);

...

}

**Log levels**

Each provider can have a minimum required LogLevel. The ConsoleLoggerProvider is configured from a configuration file; in this case the file is **appsettings.json** because this path is given in the Startup constructor:

**public** **Startup**(IHostingEnvironment env)

{

*// Set up configuration sources.*

**var** builder = **new** **ConfigurationBuilder**()

.**AddJsonFile**("appsettings.json");

**this**.Configuration = builder.**Build**();

}

This is what **appsettings.json** looks like:

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Verbose",

"System": "Information",

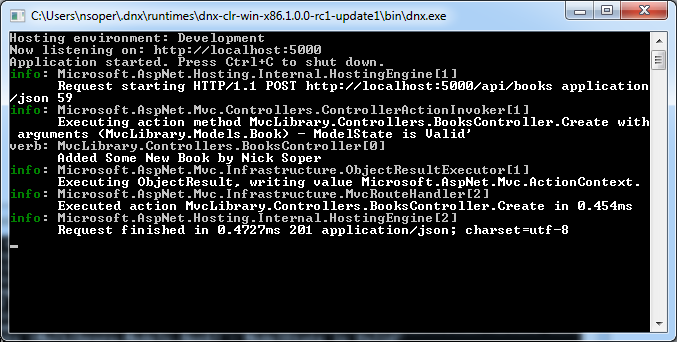
"Microsoft": "Information"

}

}

}

The ability to set the minimum log level using a JSON configuration file is a [relatively new addition](https://github.com/aspnet/Logging/pull/276) to the framework so it is currently only supported by the ConsoleLoggerProvider and the use of the ‘Default’, ‘System’ and ‘Microsoft’ properties above is not very well documented. Having played around a little, I can confirm that the ASP.NET framework will log as "Microsoft", so with the default value of ‘Information’ I get the following log messages when POSTing a new book:



If I now change the minimum logging level for ‘Microsoft’ to ‘Error’ then the ASP.NET log messages are not included and I only see my message:

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Verbose",

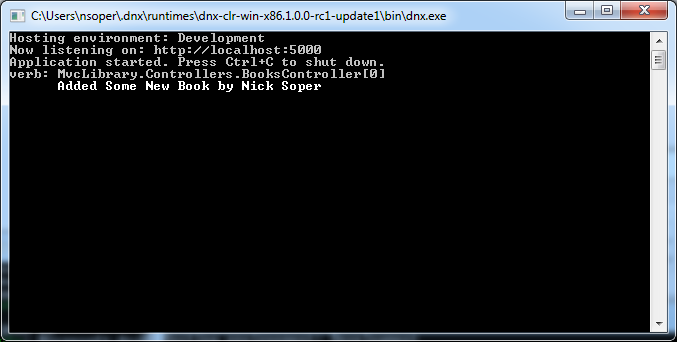
"System": "Information",

"Microsoft": "Error"

}

}

}



**A note on serilog**

Serilog will map LogLevel.Verbose to its own Debug level. So you will see the following message logged in the wwwroot/MvcLibrary-YYYYMMYY.log file (note the [Debug] label):

2016-01-18 17:18:05.650 +00:00 MvcLibrary.Controllers.BooksController [Debug] Added "Some New Book" by "Nick Soper"

**Verbose -> Trace**

Since RC1, Microsoft have [renamed Verbose to Trace](https://github.com/aspnet/Logging/pull/314). The following comment was made on the pull request:

Renaming LogLevels in logging so that Verbose is renamed to Trace and is considered less severe than Debug.

So the new default set in **appsettings.json** will be ‘Debug’ which is considered the second most detailed level of logging - ‘Trace’ being the most detailed.

For a more detailed overview of logging in ASP.NET 5, see [Microsoft’s documentation](http://docs.asp.net/en/latest/fundamentals/logging.html).

**Summary**

With ASP.NET 5 and MVC 6, Microsoft have really modernized the platform. The new **Startup** approach with configurable middleware is more in line with the [expressjs](http://expressjs.com/) way of doing things and with no static GlobalConfiguration class the design just feels cleaner. The support for **environment based configuration** simplifies builds by allowing making configuration an environment concern. Both **dependency injection** and **logging** are far better supported out of the box but we still have the option of plugging our own solutions in these areas too.

The only real gripe I have with the experience of creating a RESTful API is the lack of support for JSON requests without the [FromBody] attribute.

So on balance I really like what Microsoft are doing with the new platform. It will be interesting to see what it looks like come v1.0.0 since they are already making fairly major changes since RC1, and remember, if you want to see something that isn’t supported then you can always contribute by adding a pull request.