***http://www.mithunvp.com/user-secrets-asp-net-core/***

***What do we mean by User Secrets?***

This was the question which strike’d my mind when I first read about it. Is it really worth coming with something like. Yes, it’s really worth. Believe me at end of this article you will really feel its worth.

User Secrets never meant to be end user’s secrets, its all about developers secrets. Here are some scenario’s  for developer to have secrets.

* Any Social Media APP key which is used while development is secret. Twitter/ Facebook/ Google API keys are actually ones secret and why do you need to place them in source code.
* User specific passwords for accessing databases. Yes, many enterprise does give developers individual accounts for accessing databases.
* Any Token value for accessing some services.

One old school kind of dealing with this issue, be alert while working with source code repo’s. Place some dummy text there and have common understanding between developers to enter their respective secrets.

We will definitely mess up with these common understanding. I hope we have encountered these kinds of issues.

Here comes ***User Secrets of ASP.NET Core***, a very elegant way of keeping developers secrets up-to themselves.

Let’s explore more on this by creating ASP.NET Core web app, the tooling adds us necessary packages.

* Open ***project.json***, you will see on top “***userSecretsId***” containing unique identifier  for this projects for keeping user related secrets.
* We also see “***Microsoft.Extensions.SecretManager.Tools***“; this helps to get; set or view the secrets.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {  "userSecretsId": "aspnet-CoreDemoApp-7fdc0c49-5cef-407f-b51b-768f377fbee3",    //remaining code removed for clarity    "Microsoft.Extensions.Configuration.UserSecrets": "1.0.0-rc2-final",  "Microsoft.Extensions.SecretManager.Tools": {        "version": "1.0.0-preview1-final",        "imports": "portable-net45+win8+dnxcore50"      },  } |

* Open “Startup.cs”, the “Startup” method adds “*AddUserSecrets*()” to ConfigurationBuilder so that it keeps secrets



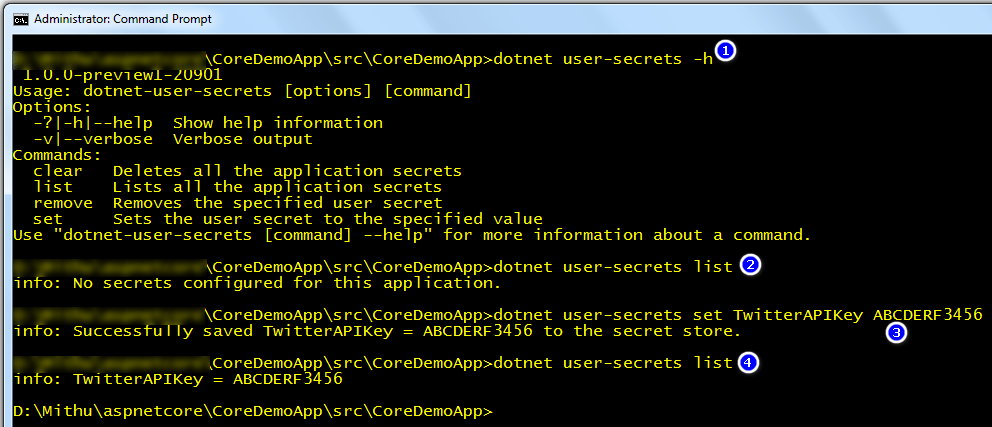
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public Startup(IHostingEnvironment env)          {              var builder = new ConfigurationBuilder()                  .SetBasePath(env.ContentRootPath)                  .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);                if (env.IsDevelopment())              {                  builder.AddUserSecrets();              }                builder.AddEnvironmentVariables();              Configuration = builder.Build();          } |

User Secrets should be used only during development by using env.IsDevelopment()

**How to add User secrets?**

In project.json we have added **SecretManager tool** extension to work with developer user secret. Using this we will be adding them in project. Open CMD from your project location.

Follow the commands as shown below

[](https://i0.wp.com/www.mithunvp.com/wp-content/uploads/2016/05/addkey.png)

Using Secret Manager Tool

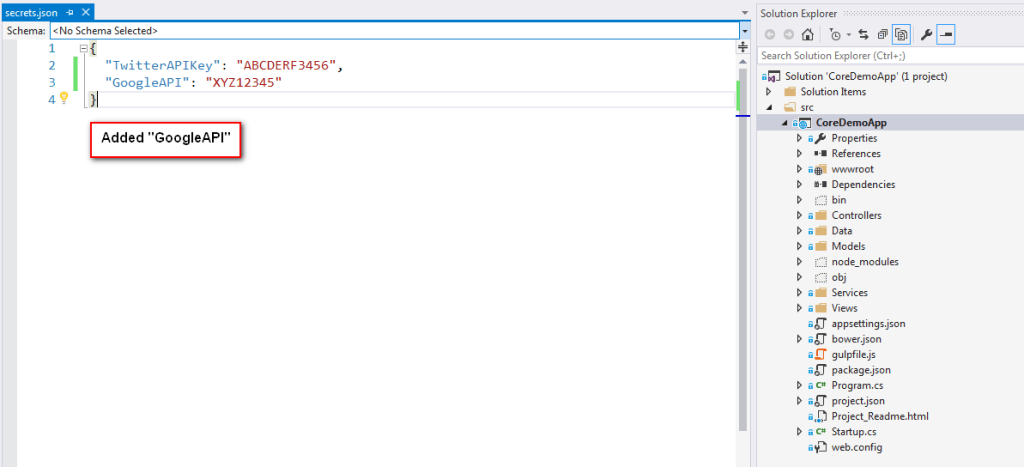
1. Shows the “user-secrets” commands *“dotnet user-secrets -h”*
2. Lists out the added secrets for the project “dotnet user-secrets list”
3. Setting “TwitterAPIKey” as secret *“dotnet user-secrets set TwitterAPIKey ABCDERF3456”*
4. Shows that we have added

This was Command Line based way of working with user secrets, lets now see how we can do with Visual Studio tooling.

Right Click the project name, navigate to “***Manage User Secrets***“, it opens up secret.json file containing above added “TwitterAPIKey”.

Suppose you are working with Google services, it provides account specific API key. We can add them using Visual Studio 2015 instead of command line.

In the below image, I clicked on “Show All Files” in Solution Explorer, right side i have “secrets.json” file which is not to seen in our explorer.

[](https://i0.wp.com/www.mithunvp.com/wp-content/uploads/2016/05/second.png)

Secrets.json not to be found in Solution Explorer

**Where is this secrets.json located?**

Right question at this point of time, User’s Secrets that get added using “Secret Manager Tool” are located in AppData of current logged in Windows users.

ASP.NET Core apps are cross platform, for NON windows machine they are located at *“~/.microsoft/usersecrets/<userSecretsId>/secrets.json”*

As secrets.json is already open, just mouse over it to see its location.You would see locations as

*“C:\Users\mithunvp\AppData\Roaming\Microsoft\UserSecrets\****aspnet-CoreDemoApp-7fdc0c49-5cef-407f-b51b-768f377fbee3****\secrets.json”*

If you see carefully the above highlighted blue text is nothing but the “**userSecretsId**” present in package.json.

User Secrets are stored as per USER per PROJECT. Every project has its own secrets.json

**Accessing these secrets in application**

In *Models* folder, create C# class file *AppKeyConfig.cs*. We will load those secrets in this class. *This C# class can be created any where.*



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | namespace CoreDemoApp.Models  {      public class AppKeyConfig      {          public string TwitterAPIKey { get; set; }          public string GoogleAPI { get; set; }      }  } |

We need to add configuration section called “AppKeys” in appsettings.json file.



|  |  |
| --- | --- |
| 1  2  3  4 | "AppKeys": {      "TwitterAPIKey": "",      "GoogleAPI": ""    } |

Right Click project name –> Click “Manage User Secrets” and modify it accordingly



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {    "AppKeys": {      "TwitterAPIKey": "ABCDERF3456",      "GoogleAPI": "XYZ12345"    }  } |

**Appsettings.json** and **secrets.json** structure should be same to use them in application.

Ensure that “**Microsoft.Extensions.Options.ConfigurationExtensions”: “1.0.0-rc2-final**” is added to project.json.

Open Startup.cs and add highlighted line.

C# class we created in Models folder will be loaded with values from secrets.json to accessed across application using DI.

Configure Services to include Appkeys

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void ConfigureServices(IServiceCollection services)          {              services.Configure<AppKeyConfig>(Configuration.GetSection("AppKeys"));              services.AddMvc();                  // Other code removed to have clarity.          } |

**Note**: The *appsettings.json* “AppKeys” section values will be overridden by values of secrets.json “*AppKeys*” because we have added “AddUserSecrets()” after appsettings.json is built.

Now open any file in MVC application to access these secret values. Since ASP.NET Core offers Dependency Injection by default, its easy to inject these secret values wherever needed.

I will open HomeController.cs, inject “AppKeysConfig” in constructor, read those values in About action method.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | using CoreDemoApp.Models;  using Microsoft.AspNetCore.Mvc;  using Microsoft.Extensions.Options;    namespace CoreDemoApp.Controllers  {      public class HomeController : Controller      {          public AppKeyConfig AppConfigs { get; }          public HomeController(IOptions<AppKeyConfig> appkeys)          {              AppConfigs = appkeys.Value;          }          public IActionResult Index()          {              return View();          }            public IActionResult About()          {              ViewData["Message"] = AppConfigs.TwitterAPIKey;                return View();          }            //Remaining code removed to have clarity      }  } |

When we run application, navigate to About() screen, we see the API key displayed on screen.

Since we see everything, we think that their no secret here, but secrets.json is in your machine, not on source code repo.

[How to secure an ASP.NET Web API](https://stackoverflow.com/questions/11775594/how-to-secure-an-asp-net-web-api/11782361#11782361) 4 years ago using HMAC.

Now, lots of things changed in security, esp JWT is getting popular. In here, I will try to explain how to use JWT in the simplest and basic way that I can, so we won't get lost from jungle of OWIN, Oauth2, ASP.NET Identity... :).

If you don't know JWT token, you need to take a look a little bit at:

<https://tools.ietf.org/html/rfc7519>

Basically, a JWT token look like:

<base64-encoded header>.<base64-encoded claims>.<base64-encoded signature>

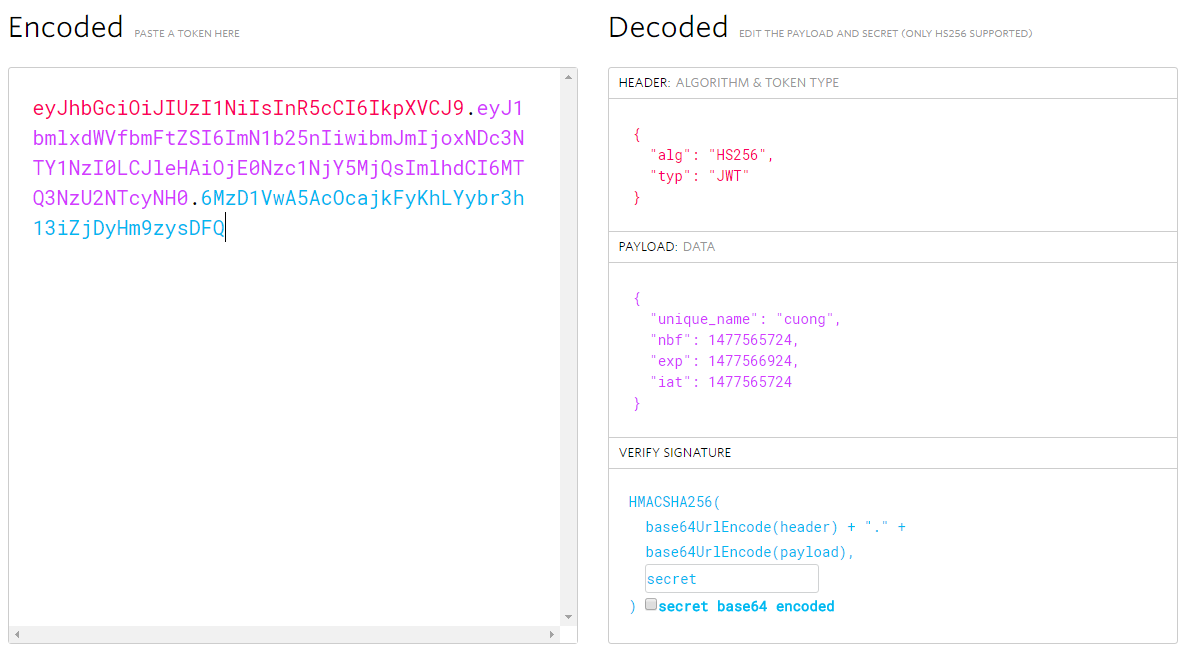
Example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVfbmFtZSI6ImN1b25nIiwibmJmIjoxNDc3NTY1NzI0LCJleHAiOjE0Nzc1NjY5MjQsImlhdCI6MTQ3NzU2NTcyNH0.6MzD1VwA5AcOcajkFyKhLYybr3h13iZjDyHm9zysDFQ

JWT token has three sections:

1. Header: JSON format which is encoded as a base64
2. Claims: JSON format which is encoded as a base64.
3. Signature: Created and signed based on Header and Claims which is encoded as a base64.

If you use the website [jwt.io](http://jwt.io) with token above, you can decode and see the token like below:

[](https://i.stack.imgur.com/WGVbv.png)

Technically, JWT uses signature which is signed from headers and claims with security algorithm specified in the headers (example: HMACSHA256). Therefore, JWT is required to be transferred over HTTPs if you store any sensitive information in claims.

Now, in order to use JWT authentication, you don't really need an OWIN middleware if you have legacy Web Api system. The simple concept is how to provide JWT token and how to validate token when the request comes. That's it.

Back to the demo, to keep JWT token lightweight, I only store username and expiration time in JWT. But this way, you have to re-build new local identity (principal) to add more information like: roles.. if you want to do role authorization. But, if you want to add more information into JWT, it's up to you, very flexible.

Instead of using OWIN middleware, you can simply provide JWT token endpoint by using action from controller:

public class TokenController : ApiController

{

// This is naive endpoint for demo, it should use Basic authentication to provide token or POST request

[AllowAnonymous]

public string Get(string username, string password)

{

if (CheckUser(username, password))

{

return JwtManager.GenerateToken(username);

}

throw new HttpResponseException(HttpStatusCode.Unauthorized);

}

public bool CheckUser(string username, string password)

{

// should check in the database

return true;

}

}

This is naive action, in production you should use POST request or Basic Authentication endpoint to provide JWT token.

How to generate the token based on username?

You can use the NuGet package called System.IdentityModel.Tokens.Jwt from MS to generate the token, or even another package if you like. In the demo, I use HMACSHA256 with SymmetricKey:

public const string Secret = "856FECBA3B06519C8DDDBC80BB080553"; // your symmetric

public static string GenerateToken(string username, int expireMinutes = 20)

{

var symmetricKey = Convert.FromBase64String(Secret);

var tokenHandler = new JwtSecurityTokenHandler();

var now = DateTime.UtcNow;

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new[]

{

new Claim(ClaimTypes.Name, username)

}),

Expires = now.AddMinutes(Convert.ToInt32(expireMinutes)),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(symmetricKey), SecurityAlgorithms.HmacSha256Signature)

};

var stoken = tokenHandler.CreateToken(tokenDescriptor);

var token = tokenHandler.WriteToken(stoken);

return token;

}

The endpoint to provide the JWT token is done, now, how to validate the JWT when the request comes, in the demo I have built JwtAuthenticationAttribute which inherits from IAuthenticationFilter, more detail about authentication filter in [here](https://www.asp.net/web-api/overview/security/authentication-filters).

With this attribute, you can authenticate any action, you just put this attribute on that action.

public class ValueController : ApiController

{

[JwtAuthentication]

public string Get()

{

return "value";

}

}

You also can use OWIN middleware or DelegateHander if you want to validate all incoming request for your WebApi (not specific on Controller or action)

Below is the core method from authentication filter:

private static bool ValidateToken(string token, out string username)

{

username = null;

var simplePrinciple = JwtManager.GetPrincipal(token);

var identity = simplePrinciple.Identity as ClaimsIdentity;

if (identity == null)

return false;

if (!identity.IsAuthenticated)

return false;

var usernameClaim = identity.FindFirst(ClaimTypes.Name);

username = usernameClaim?.Value;

if (string.IsNullOrEmpty(username))

return false;

// More validate to check whether username exists in system

return true;

}

protected Task<IPrincipal> AuthenticateJwtToken(string token)

{

string username;

if (ValidateToken(token, out username))

{

// based on username to get more information from database in order to build local identity

var claims = new List<Claim>

{

new Claim(ClaimTypes.Name, username)

// Add more claims if needed: Roles, ...

};

var identity = new ClaimsIdentity(claims, "Jwt");

IPrincipal user = new ClaimsPrincipal(identity);

return Task.FromResult(user);

}

return Task.FromResult<IPrincipal>(null);

}

The workflow is, using JWT library (NuGet package above) to validate JWT token and then return back ClaimsPrincipal. You can perform more validation like check whether user exists on your system and add other custom validations if you want. The code to validate JWT token and get principal back:

public static ClaimsPrincipal GetPrincipal(string token)

{

try

{

var tokenHandler = new JwtSecurityTokenHandler();

var jwtToken = tokenHandler.ReadToken(token) as JwtSecurityToken;

if (jwtToken == null)

return null;

var symmetricKey = Convert.FromBase64String(Secret);

var validationParameters = new TokenValidationParameters()

{

RequireExpirationTime = true,

ValidateIssuer = false,

ValidateAudience = false,

IssuerSigningKey = new SymmetricSecurityKey(symmetricKey)

};

SecurityToken securityToken;

var principal = tokenHandler.ValidateToken(token, validationParameters, out securityToken);

return principal;

}

catch (Exception)

{

//should write log

return null;

}

}

If the JWT token is validated and principal is return, you should build new local identity and put more information into it to check role authorization.

Remember to add config.Filters.Add(new AuthorizeAttribute()); (default authorization) at global scope in order to prevent any anonymous request to your resources.

You can use Postman to test the demo:

Request token (naive as I mentioned above, just for demo):

GET http://localhost:{port}/api/token?username=cuong&password=1

Put JWT token in the header for authorized request, example:

GET http://localhost:{port}/api/value

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVfbmFtZSI6ImN1b25nIiwibmJmIjoxNDc3NTY1MjU4LCJleHAiOjE0Nzc1NjY0NTgsImlhdCI6MTQ3NzU2NTI1OH0.dSwwufd4-gztkLpttZsZ1255oEzpWCJkayR\_4yvNL1s

I put my demo in here: <https://github.com/cuongle/WebApi.Jwt>

|  |
| --- |
| Well explained by @Cuong Le but i like to add more: If you are using OWIN check the UseJwtBearerAuthentication available in Microsoft.Owin.Security.Jwt you can use this owin middleware on the WebAPI to validate every incoming request automatically. use the owin startup class to register the middleware – |
| I've actually implemented in pretty much the same way after I found this: [asp.net/web-api/overview/security/authentication-filters](https://www.asp.net/web-api/overview/security/authentication-filters) Your answer is excellent. +1 + accept on it. I only have one open issue - Whats the best way to ping-pong (client-server) the bearer token? My client sends it - I go through the filter and either get a 401 or a valid authentication, do my stuff and want to set the token on the response. I can do it ugly with modifying the response header on the authentication filter, but is there a better way? – | |
| @AmirPopovich You don't need to set token on the response, token need to be stored somewhere else on the client side, for web, you can put in local storage, whenever you send HTTP request, put the token on the header. | |

https://stackoverflow.com/questions/40281050/jwt-authentication-for-asp-net-web-api

http://www.developerhandbook.com/c-sharp/create-restful-api-authentication-using-web-api-jwt/

https://api.codeproject.com/Samples/ClientCredCsDoc

This document shows how you can use the Secret Manager tool in development to keep secrets out of your code. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode. You can instead use the [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration) system to read these values from environment variables or from values stored using the Secret Manager tool. The Secret Manager tool helps prevent sensitive data from being checked into source control. The [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration) system can read secrets stored with the Secret Manager tool described in this article.

The Secret Manager tool is used only in development. You can safeguard Azure test and production secrets with the [Microsoft Azure Key Vault](https://azure.microsoft.com/services/key-vault/) configuration provider. See [Azure Key Vault configuration provider](https://docs.microsoft.com/aspnet/core/security/key-vault-configuration) for more information.

## Environment variables

To avoid storing app secrets in code or in local configuration files, you store secrets in environment variables. You can setup the [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration) framework to read values from environment variables by calling AddEnvironmentVariables. You can then use environment variables to override configuration values for all previously specified configuration sources.

For example, if you create a new ASP.NET Core web app with individual user accounts, it will add a default connection string to the appsettings.json file in the project with the key DefaultConnection. The default connection string is setup to use LocalDB, which runs in user mode and doesn't require a password. When you deploy your application to a test or production server, you can override the DefaultConnection key value with an environment variable setting that contains the connection string (potentially with sensitive credentials) for a test or production database server.

##### Warning

Environment variables are generally stored in plain text and are not encrypted. If the machine or process is compromised, then environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may still be required.

## Secret Manager

The Secret Manager tool stores sensitive data for development work outside of your project tree. The Secret Manager tool is a project tool that can be used to store secrets for a [.NET Core](https://microsoft.com/net/core) project during development. With the Secret Manager tool, you can associate app secrets with a specific project and share them across multiple projects.

##### Warning

The Secret Manager tool does not encrypt the stored secrets and should not be treated as a trusted store. It is for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

### Visual Studio 2017: Installing the Secret Manager tool

Right-click the project in Solution Explorer, and select **Edit <project\_name>.csproj** from the context menu. Add the highlighted line to the .csproj file, and save to restore the associated NuGet package:

XML

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp1.1</TargetFramework>

</PropertyGroup>

<PropertyGroup>

<UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>

</PropertyGroup>

<ItemGroup>

<Folder Include="wwwroot\" />

</ItemGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />

<PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets" Version="1.1.1" />

</ItemGroup>

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.1" />

</ItemGroup>

</Project>

Right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a new UserSecretsId node within a PropertyGroup of the .csproj file. It also opens a secrets.json file in the text editor.

Add the following to secrets.json:

JSON

{

"MySecret": "ValueOfMySecret"

}

### Visual Studio 2015: Installing the Secret Manager tool

Open the project's project.json file. Add a reference to Microsoft.Extensions.SecretManager.Tools within the tools property, and save to restore the associated NuGet package:

JSON

"tools": {

"Microsoft.Extensions.SecretManager.Tools": "1.0.0-preview2-final",

"Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"

},

Right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a new userSecretsId property to project.json. It also opens a secrets.json file in the text editor.

Add the following to secrets.json:

JSON

{

"MySecret": "ValueOfMySecret"

}

### Visual Studio Code or Command Line: Installing the Secret Manager tool

Add Microsoft.Extensions.SecretManager.Tools to the .csproj file and run dotnet restore.

XML

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp1.1</TargetFramework>

</PropertyGroup>

<PropertyGroup>

<UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>

</PropertyGroup>

<ItemGroup>

<Folder Include="wwwroot\" />

</ItemGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />

<PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets" Version="1.1.1" />

</ItemGroup>

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.1" />

</ItemGroup>

</Project>

Test the Secret Manager tool by running the following command:

console

dotnet user-secrets -h

The Secret Manager tool will display usage, options and command help.

##### Note

You must be in the same directory as the .csproj file to run tools defined in the .csproj file's DotNetCliToolReference nodes.

The Secret Manager tool operates on project-specific configuration settings that are stored in your user profile. To use user secrets, the project must specify a UserSecretsId value in its .csproj file. The value of UserSecretsId is arbitrary, but is generally unique to the project. Developers typically generate a GUID for the UserSecretsId.

Add a UserSecretsId for your project in the .csproj file:

XML

<PropertyGroup>

<UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>

</PropertyGroup>

Use the Secret Manager tool to set a secret. For example, in a command window from the project directory, enter the following:

console

dotnet user-secrets set MySecret ValueOfMySecret

You can run the Secret Manager tool from other directories, but you must use the --project option to pass in the path to the .csproj file:

console

dotnet user-secrets set MySecret ValueOfMySecret --project c:\work\WebApp1\src\webapp1

You can also use the Secret Manager tool to list, remove and clear app secrets.

## Accessing user secrets via configuration

You access Secret Manager secrets through the configuration system. Add the Microsoft.Extensions.Configuration.UserSecrets package and run dotnet restore.

Add the user secrets configuration source to the Startup method:

C#

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Http;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace UserSecrets

{

public class Startup

{

string \_testSecret = null;

public Startup(IHostingEnvironment env)

{

var builder = new ConfigurationBuilder();

if (env.IsDevelopment())

{

builder.AddUserSecrets<Startup>();

}

Configuration = builder.Build();

}

public IConfigurationRoot Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

\_testSecret = Configuration["MySecret"];

}

public void Configure(IApplicationBuilder app)

{

var result = string.IsNullOrEmpty(\_testSecret) ? "Null" : "Not Null";

app.Run(async (context) =>

{

await context.Response.WriteAsync($"Secret is {result}");

});

}

}

}

You can access user secrets via the configuration API:

C#

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Http;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace UserSecrets

{

public class Startup

{

string \_testSecret = null;

public Startup(IHostingEnvironment env)

{

var builder = new ConfigurationBuilder();

if (env.IsDevelopment())

{

builder.AddUserSecrets<Startup>();

}

Configuration = builder.Build();

}

public IConfigurationRoot Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

\_testSecret = Configuration["MySecret"];

}

public void Configure(IApplicationBuilder app)

{

var result = string.IsNullOrEmpty(\_testSecret) ? "Null" : "Not Null";

app.Run(async (context) =>

{

await context.Response.WriteAsync($"Secret is {result}");

});

}

}

}

## How the Secret Manager tool works

The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. In the current version, the values are stored in a [JSON](http://json.org/) configuration file in the user profile directory:

* Windows: %APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json
* Linux: ~/.microsoft/usersecrets/<userSecretsId>/secrets.json
* Mac: ~/.microsoft/usersecrets/<userSecretsId>/secrets.json

The value of userSecretsId comes from the value specified in .csproj file.

You should not write code that depends on the location or format of the data saved with the Secret Manager tool, as these implementation details might change. For example, the secret values are currently not encrypted today, but could be someday.

https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets

In your ASP.NET Core application you can load settings from a file named *secrets.json* that can store API ids and secrets. The whole point of using this secret storage is to avoid having your ClientId and ClientSecret exposed on source control. In production you can have it stored on environment settings.  
  
In your ASP.NET Core application you can load settings from a file named secrets.json that can store API ids and secrets. The default generated template includes,

1. if (env.IsDevelopment())
2. {
3. builder.AddUserSecrets();
4. }

That is going to add that file only on a development environment. So the whole point of using this secret storage is to avoid having your ClientId and ClientSecret exposed on source control. In production you can have it stored on environment settings, the generated template includes,

1. builder.AddEnvironmentVariables();

Which is going to add the environment variables on your application configuration.

In order to do that first test your environment by typing dnx in the command prompt. If it doesn’t find dnx then run the following,

1. cd %userprofile%\.dnx\runtimes\dnx-coreclr-win-x64.1.0.0-rc1-update1\bin
2. dnvm upgrade

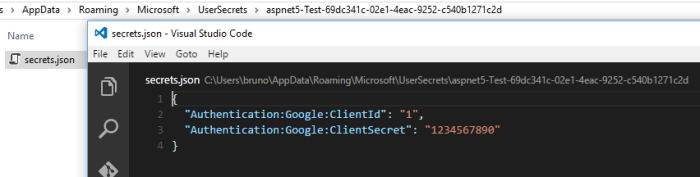
This is going to update the path and other things, after that you can run this other command to install the SecretManager,

1. dnu commands install Microsoft.Extensions.SecretManager

To finally store the application secrets you can run,

1. user-secret set Authentication:Google:ClientId <yourId>
2. user-secret set Authentication:Google:ClientSecret <yourSecret>

In the %APPDATA%\microsoft\UserSecrets folder there is going to be a folder for your project and then a secrets.json inside.

[](https://brunolm.files.wordpress.com/2016/02/secrets1.png)

Then finally using it on your application, install Google Authentication,

1. Install-Package Microsoft.AspNet.Authentication.Google -Pre

Note the -Pre option, as of the date of this post this package will not be found if you do not include this option.

1. app.UseGoogleAuthentication(options=>
2. {
3. options.ClientId = Configuration["Authentication:Google:ClientId"];
4. options.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
5. });

http://www.c-sharpcorner.com/blogs/installing-and-using-secret-manager-on-asp-net-core

This document shows how you can use the Secret Manager tool in development to keep secrets out of your code. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode. You can instead use the [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration) system to read these values from environment variables or from values stored using the Secret Manager tool. The Secret Manager tool helps prevent sensitive data from being checked into source control. The [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration) system can read secrets stored with the Secret Manager tool described in this article.

The Secret Manager tool is used only in development. You can safeguard Azure test and production secrets with the [Microsoft Azure Key Vault](https://azure.microsoft.com/services/key-vault/) configuration provider. See [Azure Key Vault configuration provider](https://docs.microsoft.com/aspnet/core/security/key-vault-configuration) for more information.

## Environment variables

To avoid storing app secrets in code or in local configuration files, you store secrets in environment variables. You can setup the [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration) framework to read values from environment variables by calling AddEnvironmentVariables. You can then use environment variables to override configuration values for all previously specified configuration sources.

For example, if you create a new ASP.NET Core web app with individual user accounts, it will add a default connection string to the appsettings.json file in the project with the key DefaultConnection. The default connection string is setup to use LocalDB, which runs in user mode and doesn't require a password. When you deploy your application to a test or production server, you can override the DefaultConnection key value with an environment variable setting that contains the connection string (potentially with sensitive credentials) for a test or production database server.

##### Warning

Environment variables are generally stored in plain text and are not encrypted. If the machine or process is compromised, then environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may still be required.

## Secret Manager

The Secret Manager tool stores sensitive data for development work outside of your project tree. The Secret Manager tool is a project tool that can be used to store secrets for a [.NET Core](https://microsoft.com/net/core) project during development. With the Secret Manager tool, you can associate app secrets with a specific project and share them across multiple projects.

##### Warning

The Secret Manager tool does not encrypt the stored secrets and should not be treated as a trusted store. It is for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

### Visual Studio 2017: Installing the Secret Manager tool

Right-click the project in Solution Explorer, and select **Edit <project\_name>.csproj** from the context menu. Add the highlighted line to the .csproj file, and save to restore the associated NuGet package:

XML

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp1.1</TargetFramework>

</PropertyGroup>

<PropertyGroup>

<UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>

</PropertyGroup>

<ItemGroup>

<Folder Include="wwwroot\" />

</ItemGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />

<PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets" Version="1.1.1" />

</ItemGroup>

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.1" />

</ItemGroup>

</Project>

Right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a new UserSecretsId node within a PropertyGroup of the .csproj file. It also opens a secrets.json file in the text editor.

Add the following to secrets.json:

JSON

{

"MySecret": "ValueOfMySecret"

}

### Visual Studio 2015: Installing the Secret Manager tool

Open the project's project.json file. Add a reference to Microsoft.Extensions.SecretManager.Tools within the tools property, and save to restore the associated NuGet package:

JSON

"tools": {

"Microsoft.Extensions.SecretManager.Tools": "1.0.0-preview2-final",

"Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"

},

Right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a new userSecretsId property to project.json. It also opens a secrets.json file in the text editor.

Add the following to secrets.json:

JSON

{

"MySecret": "ValueOfMySecret"

}

### Visual Studio Code or Command Line: Installing the Secret Manager tool

Add Microsoft.Extensions.SecretManager.Tools to the .csproj file and run dotnet restore.

XML

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp1.1</TargetFramework>

</PropertyGroup>

<PropertyGroup>

<UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>

</PropertyGroup>

<ItemGroup>

<Folder Include="wwwroot\" />

</ItemGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />

<PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets" Version="1.1.1" />

</ItemGroup>

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.1" />

</ItemGroup>

</Project>

Test the Secret Manager tool by running the following command:

console

dotnet user-secrets -h

The Secret Manager tool will display usage, options and command help.

##### Note

You must be in the same directory as the .csproj file to run tools defined in the .csproj file's DotNetCliToolReference nodes.

The Secret Manager tool operates on project-specific configuration settings that are stored in your user profile. To use user secrets, the project must specify a UserSecretsId value in its .csproj file. The value of UserSecretsId is arbitrary, but is generally unique to the project. Developers typically generate a GUID for the UserSecretsId.

Add a UserSecretsId for your project in the .csproj file:

XML

<PropertyGroup>

<UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>

</PropertyGroup>

Use the Secret Manager tool to set a secret. For example, in a command window from the project directory, enter the following:

console

dotnet user-secrets set MySecret ValueOfMySecret

You can run the Secret Manager tool from other directories, but you must use the --project option to pass in the path to the .csproj file:

console

dotnet user-secrets set MySecret ValueOfMySecret --project c:\work\WebApp1\src\webapp1

You can also use the Secret Manager tool to list, remove and clear app secrets.

## Accessing user secrets via configuration

You access Secret Manager secrets through the configuration system. Add the Microsoft.Extensions.Configuration.UserSecrets package and run dotnet restore.

Add the user secrets configuration source to the Startup method:

C#

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Http;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace UserSecrets

{

public class Startup

{

string \_testSecret = null;

public Startup(IHostingEnvironment env)

{

var builder = new ConfigurationBuilder();

if (env.IsDevelopment())

{

builder.AddUserSecrets<Startup>();

}

Configuration = builder.Build();

}

public IConfigurationRoot Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

\_testSecret = Configuration["MySecret"];

}

public void Configure(IApplicationBuilder app)

{

var result = string.IsNullOrEmpty(\_testSecret) ? "Null" : "Not Null";

app.Run(async (context) =>

{

await context.Response.WriteAsync($"Secret is {result}");

});

}

}

}

You can access user secrets via the configuration API:

C#

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Http;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace UserSecrets

{

public class Startup

{

string \_testSecret = null;

public Startup(IHostingEnvironment env)

{

var builder = new ConfigurationBuilder();

if (env.IsDevelopment())

{

builder.AddUserSecrets<Startup>();

}

Configuration = builder.Build();

}

public IConfigurationRoot Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

\_testSecret = Configuration["MySecret"];

}

public void Configure(IApplicationBuilder app)

{

var result = string.IsNullOrEmpty(\_testSecret) ? "Null" : "Not Null";

app.Run(async (context) =>

{

await context.Response.WriteAsync($"Secret is {result}");

});

}

}

}

## How the Secret Manager tool works

The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. In the current version, the values are stored in a [JSON](http://json.org/) configuration file in the user profile directory:

* Windows: %APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json
* Linux: ~/.microsoft/usersecrets/<userSecretsId>/secrets.json
* Mac: ~/.microsoft/usersecrets/<userSecretsId>/secrets.json

The value of userSecretsId comes from the value specified in .csproj file.

You should not write code that depends on the location or format of the data saved with the Secret Manager tool, as these implementation details might change. For example, the secret values are currently not encrypted today, but could be someday.

https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets

ou don’t want your passwords and other secrets stored in your source code. Why? A password should not be coupled to a specific version of your application because when a password or other secret needs to be changed, the application must be redeployed. And if your version control system gets hacked, your secrets will leak.

ASP.NET Core has a solution to store secrets outside the repository during development. It’s called user secrets’ and in this post, I’m going to show what they are and how to use them.

**Setting up**

Create a new ASP.NET Core Web Application for Windows, Linux and macOS.

Install the following NuGet package:  
Microsoft.Extensions.Configuration.UserSecrets

Add following code to the constructor in the startup class.



|  |  |
| --- | --- |
| 1  2  3  4 | if (env.IsDevelopment())  {      builder.AddUserSecrets();  } |

Create a new class called AppSecrets. You can choose a different name if you like but for this example I’ll use this name. This class contains all the properties, you want to put in your user secrets. You can also use objects as properties.



|  |  |
| --- | --- |
| 1  2  3  4 | public class AppSecrets  {      public string MySecret { get; set; }  } |

Add the following line of code to the ConfigureServices method in the startup class.



|  |  |
| --- | --- |
| 1 | services.Configure(Configuration); |

**Adding user secrets**

Now we’re done setting things up so it’s time to add a user secret. User secrets are defined in a file called secrets.json which is stored in:  
Windows: %APPDATA%\microsoft\UserSecrets\\secrets.json  
Linux: ~/.microsoft/usersecrets//secrets.json  
Mac: ~/.microsoft/usersecrets//secrets.json

As you can see, the secrets.json file is not stored in your repository. The file is NOT encrypted so user secrets should only be used for development purposes! The easiest way to open and edit the user secrets in Visual Studio is by right clicking your project and clicking ‘manage user secrets’. The UserSecretId, you see in the path is defined in the csproj file or in the project.json if you are using an older version of .NET Core. This id is unique to your app. Changing this id will generate a new, empty secrets.json file.

To add a user secret open the secrets.json file and paste the following code into it.



|  |  |
| --- | --- |
| 1  2  3 | {      "MySecret": "Password123!"  } |

**Retrieving user secrets**

Now that we have added a user secret, it’s time to retrieve it in our MVC Controller. The user secrets are retrieved the same way as the Configuration. For this example, I’m using the default HomeController.

Replace the default Index method in the HomeController by the following code:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public string MySecret { get; set; }  public HomeController(IOptions optionsAccessor)  {      MySecret = optionsAccessor.Value.MySecret;  }    public IActionResult Index()  {      ViewBag.MySecret = MySecret;      return View();  } |

Paste the following code somewhere in the Views/Home/Index.cshtml.



|  |  |
| --- | --- |
| 1 | @ViewBag.MySecret |

Now you can see your user secret on the homepage of your app. Off course, normally we wouldn’t show secrets on our webpages, but this is just for demo purposes so you can see it’s working.

**Command prompt**

The last part is about managing your user secrets from the command prompt. To do this, add the following line of XML to your csproj file between an Item Group element.



|  |  |
| --- | --- |
| 1 | <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.0-msbuild3-final" /> |

Now you can open a command prompt window and navigate to your project folder. There are a couple of commands and I’m going to show some of them.

* dotnet user-secrets –help  
  Executing this command will show you information about the user secrets command line tool.
* dotnet user-secrets set SecondSecret Password  
  Adds a new user secret with the key SecondSecret and value Password to your project.
* dotnet user-secrets list  
  lists all user secrets in your project.
* dotnet user-secrets remove SecondSecret  
  Removes SecondSecret from your project.
* dot net user-secret clear  
  Clears all secrets from your project.

https://arnhem.luminis.eu/user-secrets-in-asp-net-core/

https://andrewlock.net/how-to-use-the-ioptions-pattern-for-configuration-in-asp-net-core-rc2/

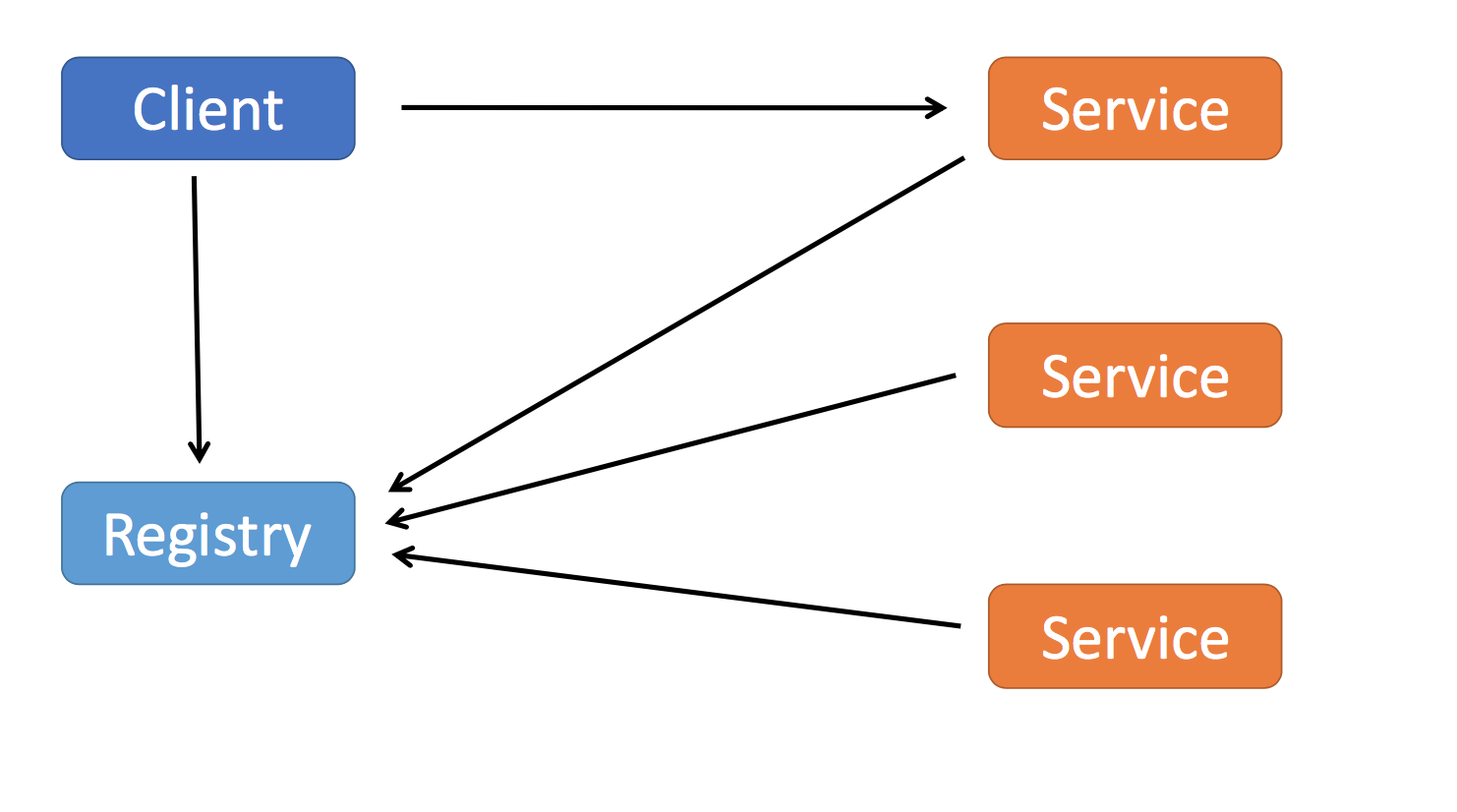
http://www.intstrings.com/ramivemula/articles/jumpstart-50-user-secrets-in-asp-net-core/

One of the benefits of adopting a microservice architectural style is the ability compose applications by bringing together smaller units of functionality (aka services). Not only does it become easier to swap out implementations of an individual service, but it also becomes easier to scale that service too. For example, imagine you are running an e-commerce website. The holidays are coming up and there is going to be a huge increase in orders. Instead of creating copies of the entire website to handle the load wouldn't be great if you could just scale up the ordering service or the payment processing service? Then after the holiday season is over those services can be scaled back down. Being able to quickly scale horizontally make microservices a very attractive option.

For an application that's built this way, those services are going to need to be able to talk to each other so that data can flow from one end of the process to the other. Going back to the e-commerce example above, an ordering service might need to talk to the shipping service which talks to the inventory service, and so on. With the way that we typically build software today, the locations of these services would be put in a configuration file somewhere. The configuration file gets loaded up and the application can select which services it wants to talk to. However, when you're dynamically creating and destroying instances of a services, it becomes difficult to keep configuration files updated with the latest information. One way we can solve this issue is by implementing some form of service discovery strategy.

#### Service Discovery

The idea of service discovery essentially is trying to find an answer to the question of what services are available and how do I get to them. Two approaches that you'll often hear about are Client Side and Server Side service discovery. In this post, we will just focus on Client Side.

With Client Side service discovery, the consumer of the service has to retrieve a listing of service information from given location. This would lead us to believe that there must be somewhere to retrieve that service information from. The medium where service information is stored and retrieved is referred to as a service registry.   


As services go live, they will register some information about themselves into the service registry; IP address, port numbers, service names, etc. When a service goes down gracefully, it can deregister itself from the registry. At some point later, the consumer would query the service registry to find out about services are available for it to use. It can then cycle through the service information and distribute requests across service instances as it sees fit.

This pattern is fairly straight forward to implement. However, the service registry does introduce an additional piece for you to manage. The flexibility you gain from centralizing this configuration is often more than worth it though.

There are a few options for implementing a service registry. I've seen implementations using data stores like Redis or document databases. In the Linux world, tools like [ZooKeeper](http://zookeeper.apache.org), [Consul](https://www.consul.io) and [etcd](https://coreos.com/etcd/) are very popular. Let's see how we can use consul as a service registry.

#### Setting up Consul

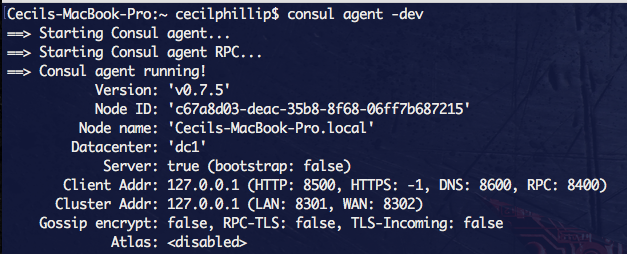
Consul is a tool created by [Hashicorp](https://www.hashicorp.com) that helps with the discovery and configuration of services in your infrastructure. It also has quite a few other interesting features such as heath checks, key/value storage and support for running in multiple data centers.

To get Consul on your machine, you can head over the [download page](https://www.consul.io/downloads.html) and grab a copy for the OS you are using. The zip file will contain the Consul command line executable that you can just run. It might also be available in your OS package manager. If you're using OSX for instance, you can use [homebrew](https://brew.sh) and brew install Consul in the terminal. I like the package manager route because then you'll have the Consul command on your system path.

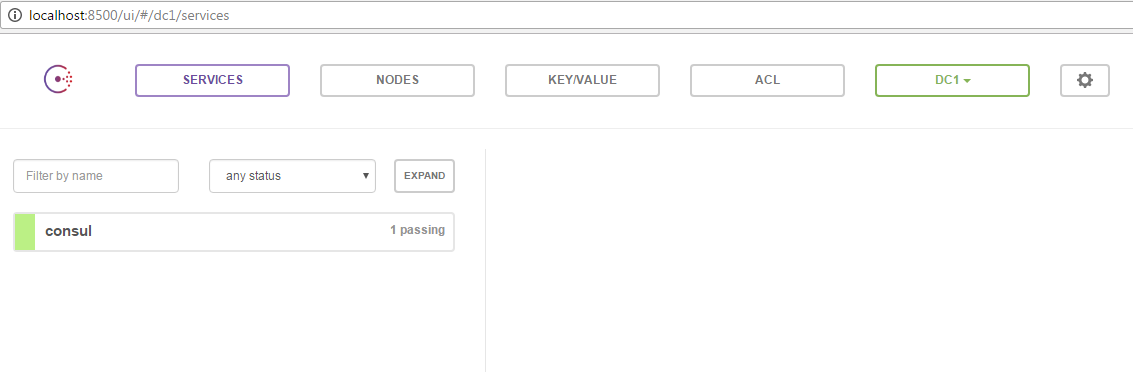
To quickly start Consul, enter the following into the command line:

consul agent -dev

Consul should now be running in dev mode. In this state, all the data will be stored in memory and not on disk. This is fine for development or demos but definitely not what you want to do on your production machines. If everything went well, you should be seeing something like this:



Open up your browser and head over to http://127.0.0.1:8500. You should now be seeing the Consul web UI. Here you can get some insight into what services are registered, their health status, and some other interesting information.



#### Registering a service

Now that the registry is up and running, let's put it to work. I have a Web API that I created with ASP.NET Core that I want to register. To get registration information into Consul, their HTTP API can be used directly, but instead I'm going to grab the [Consul NuGet package](https://www.nuget.org/packages/Consul) from PlayFab.

Here's what Startup.cs looks like:

public void ConfigureServices(IServiceCollection services)

{

services.Configure<ConsulConfig>(Configuration.GetSection("consulConfig"));

services.AddSingleton<IConsulClient, ConsulClient>(p => new ConsulClient(consulConfig =>

{

var address = Configuration["consulConfig:address"];

consulConfig.Address = new Uri(address);

}));

services.AddMvc();

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env,

ILoggerFactory loggerFactory, IApplicationLifetime lifetime)

{

loggerFactory.AddConsole();

app.UseMvc();

app.RegisterWithConsul(lifetime);

}

In ConfigureServices, I'm registering an instance of the ConsulClient and binding a section of my configuration file to an instance of ConsulConfig. Also if you take a look at the Configure method, you will see that I added an extension method called RegisterWithConsul that I want to be called once whenever an instance of my API gets created. I'm also injecting an instance of IApplicationLifetime. More on that later. Let's take a look at RegisterWithConsul.

public static IApplicationBuilder RegisterWithConsul(this IApplicationBuilder app,

IApplicationLifetime lifetime)

{

// Retrieve Consul client from DI

var consulClient = app.ApplicationServices

.GetRequiredService<IConsulClient>();

var consulConfig = app.ApplicationServices

.GetRequiredService<IOptions<ConsulConfig>>();

// Setup logger

var loggingFactory = app.ApplicationServices

.GetRequiredService<ILoggerFactory>();

var logger = loggingFactory.CreateLogger<IApplicationBuilder>();

// Get server IP address

var features = app.Properties["server.Features"] as FeatureCollection;

var addresses = features.Get<IServerAddressesFeature>();

var address = addresses.Addresses.First();

// Register service with consul

var uri = new Uri(address);

var registration = new AgentServiceRegistration()

{

ID = $"{consulConfig.Value.ServiceID}-{uri.Port}",

Name = consulConfig.Value.ServiceName,

Address = $"{uri.Scheme}://{uri.Host}",

Port = uri.Port,

Tags = new[] { "Students", "Courses", "School" }

};

logger.LogInformation("Registering with Consul");

consulClient.Agent.ServiceDeregister(registration.ID).Wait();

consulClient.Agent.ServiceRegister(registration).Wait();

lifetime.ApplicationStopping.Register(() => {

logger.LogInformation("Deregistering from Consul");

consulClient.Agent.ServiceDeregister(registration.ID).Wait();

});

}

The interesting part of this code is closer to the end where the registration is happening. Using an instance of AgentServiceRegistration (that's from the Consul NuGet package), I populate some metadata about the API and then register that information with Consul.

#### Deregistering the service

Whenever the service shuts down, it would be nice if it would tell our Consul service registry that it's not available anymore. To do that, we can leverage the ApplicationStopping event/trigger from IApplicationLifetime. At the end of RegisterWithConsul above, we make a call to ServiceDeregister and pass it the ID of the registration we want to remove.

When using these lifetime events on IApplicationLifetime, I'd recommend not doing too much work within your callbacks. Consider these events as an opportunity for quickly setting up and gracefully tearing down as needed. If any unhandled exceptions get thrown inside your callbacks, they will get swallowed and will never heard from again.

If you want to learn more about IApplicationLifetime, I'd recommend checking out [Khalid's blog post](http://www.khalidabuhakmeh.com/looking-at-asp-net-cores-iapplicationlifetime) on the subject.

#### Consuming the registrations

On the client that needs to consume the registration information, you can simply create an instance of ConsulClient and query the registry. In the code below, I'm using tags to filter out the service instances that I'm interested in. You can always use the service name too if you wish.

List<Uri> \_serverUrls = List<Uri>();

var consuleClient = new ConsulClient(c => c.Address = new Uri("http://127.0.0.1:8500"));

var services = consulClient.Agent.Services().Result.Response;

foreach (var service in services)

{

var isSchoolApi = service.Value.Tags.Any(t => t == "School") &&

service.Value.Tags.Any(t => t == "Students");

if (isSchoolApi)

{

var serviceUri = new Uri($"{service.Value.Address}:{service.Value.Port}");

serverUrls.Add(serviceUri);

}

}

The client can now manually load balance or failover its requests between the available service instances. One thing I like to do here is implement a retry policy with something like [Polly](http://www.thepollyproject.org). After a given number of retries, the client will switch over to the next service.

#### Conclusion

Regardless of the tool you use to register your services, implementing service discovery will make managing your containers and microservices much easier. We covered one implementation of Client side discovery here where the service registers/degregisters itself as the instance starts up and shuts down. There are some other options that are just as easy to implement but each with its own trade-offs. If you're interested in seeing more samples

http://cecilphillip.com/using-consul-for-service-discovery-with-asp-net-core/

https://msdn.microsoft.com/en-us/magazine/jj883957.aspx

https://docs.microsoft.com/en-us/azure/app-service-api/app-service-api-dotnet-get-started

ne of the useful configuration features of ASP.NET Web API is that it allows you to be explicit about the assemblies into which it will look in order to discover controller types.

This is especially useful if you have assemblies residing outside of the bin folder, or if you are doing self hosting, and the controllers assemblies are not automatically loaded into the current AppDomain.

There are several hooks in the pipeline that you can plug into to achieve this goal. Let’s explore them, discussing the pros and cons of using any of these.

### Custom IAssembliesResolver

The broadest reaching option, and first hook in the pipeline is IAssembliesResolver, with its default implementation, DefaultAssembliesResolver. Actually, we already discussed this option on the blog [in the past](http://www.strathweb.com/2012/06/using-controllers-from-an-external-assembly-in-asp-net-web-api/), when Web API was still in RC version

The interface let’s you specify a list of assemblies which Web API should use to discover controller types. The interface definition is very straight forward, and the default implementation it comes with, simply looks into the AppDomain.CurrentDomain.

C#



|  |  |
| --- | --- |
| 1  2  3  4 | public interface AssembliesResolver  {      ICollection<Assembly> GetAssemblies();  } |

Since all assemblies copied into the bin folder (when hosting on IIS) are loaded into the AppDomain, normally it’s enough to just copy your external assembly there.

However, in case you want to load DLL(s) from a different path (perhaps a shared network drive or a different pre-approved location) or you are self-hosting, and you don’t have the help of magical bin folder, you can easily add your assembly to the collection:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MyAssembliesResolver : DefaultAssembliesResolver  {     public override ICollection<Assembly> GetAssemblies()     {        ICollection<Assembly> baseAssemblies = base.GetAssemblies();        List<Assembly> assemblies = new List<Assembly>(baseAssemblies);        var controllersAssembly = Assembly.LoadFrom("c:/myAssymbly.dll");        baseAssemblies.Add(controllersAssembly);        return assemblies;     }  } |

Then, obviously, you need to register the resolver against your configuration:

C#



|  |  |
| --- | --- |
| 1 | config.Services.Replace(typeof(IAssembliesResolver), new MyAssemblyResolver()); |

In this particular case we didn’t even implement the interface directly, but rather extended its default implementation. Since we add our source to the assembly source provided by default, this resolver looks both in to the AppDomain.Current and into our DLL path.

Once this solution is in place, Web API, whenever it will try to obtain list of assemblies to resolve controllers (at any point in the pipeline), will resort to this custom implementation.

### Custom IHttpControllerTypeResolver

A bit deeper in the pipeline lies IHttpControllerTypeResolver which is responsible of taking in assemblies resolved by IAssembliesResolver and discover the types matching the predefined controller definition.

Due to such design, you can actually bypass implementing IAssembliesResolver altogether, and bunch up both assembly discovery and type discovery in a single place. This is useful if you wish to modify the rules which Web API uses to discover controller types.

The default rule set for a type to be discovered as valid API controller is as follows:

* – implements IHttpController (or inherits from ApiController)
* – is a public class
* – is a non-abstract class
* – has a “Controller” suffix

These rules are represented by the following method, found on DefaultHttpControllerTypeResolver.

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | internal static bool IsControllerType(Type t)  {     Contract.Assert(t != null);     return     t != null &&     t.IsClass &&     t.IsVisible &&     !t.IsAbstract &&     typeof(IHttpController).IsAssignableFrom(t) &&     HasValidControllerName(t);  } |

You could implement your own rules on top of that. An example implementation could extend the default DefaultHttpControllerTypeResolver as follows:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class CustomHttpControllerTypeResolver : DefaultHttpControllerTypeResolver  {      public CustomHttpControllerTypeResolver()              : base(IsHttpEndpoint)      {}        internal static bool IsHttpEndpoint(Type t)      {         if (t == null) throw new ArgumentNullException("t");           return          t.IsClass &&          t.IsVisible &&          !t.IsAbstract &&         typeof(MyBaseApiController).IsAssignableFrom(t);      }  } |

Notice that DefaultHttpControllerTypeResolver takes in a predicate defining rules to be used to discover controllers. In our example, we drop the HasValidControllerName(t) check – and require all controllers to derive from an imaginary *MyBaseApiController*. This is a neat way to force all the developers in our team to inherit from it when they develop their HTTP endpoints.

Another interesting thing worth mentioning here, is that later on, in an internal class responsible for caching controllers, Web API will perform another crucial check – whether the assembly from which controllers are being is discovered is dynamic or not – and if it is, it will not be processed.

This is very important if you wish to emit assemblies dynamically that would contain controller types. If you do that, this is not an extensibility point for you and you have to dig a step deeper.

### Custom IHttpControllerSelector

Finally, even deeper into the pipeline, you can find & implement your own IHttpControllerSelector.

There are many reasons of doing that, mainly if, as the name suggests, you want to introduce a custom mechanism of action selection. This is quite useful if you wish to override the default action selection mechanism or introduce some versioning mechanism into your API, and dispatch correct version of the action/controller based on the incoming request.

However, there are some specific cases where using IHttpControllerSelector for controller type discovery makes sense. IHttpControllerSelector runs after the cache of controllers has been established (in IHttpControllerTypeResolver). Therefore, implementing your own type discovery logic at this level, allows you to bypass any caching mechanism present at the earlier stages.

While this has little production-environment value (as it’s very inefficient), it can be sometimes useful during development.

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class BypassCacheSelector : DefaultHttpControllerSelector  {      private readonly HttpConfiguration \_configuration;        public BypassCacheSelector(HttpConfiguration configuration)          : base(configuration)      {          \_configuration = configuration;      }        public override HttpControllerDescriptor SelectController(HttpRequestMessage request)      {          var assembly = Assembly.LoadFile("c:/myAssembly.dll");          var types = assembly.GetTypes(); //GetExportedTypes doesn't work with dynamic assemblies          var matchedTypes = types.Where(i => typeof (IHttpController).IsAssignableFrom(i)).ToList();            var controllerName = base.GetControllerName(request);          var matchedController =              matchedTypes.FirstOrDefault(i => i.Name.ToLower() == controllerName.ToLower() + "controller");            return new HttpControllerDescriptor(\_configuration, controllerName, matchedController);      }  } |

It obviously needs to be registered against the HttpConfiguration too:

C#



|  |  |
| --- | --- |
| 1 | config.Services.Replace(typeof(IHttpControllerSelector), new BypassCacheSelector(config)); |

In this implementation, we load the assembly manually, and then resolve the controller types by looking through all ExportedTypes. Doing it at this level, will force the assembly to be reloaded and rescanned at every request, which now allows us to freely recompile that assembly and see the changes reflected upon next request, as there is no controller caching in place anymore.

As briefly mentioned before, if you use dynamic assemblies (generated i.e. using CodeDom or Roslyn) this is the route you have to take – because otherwise Web API will ignore your dynamic assemblies.

For more information on the topic of dynamic compilation I encourage you read one of the older posts about [Roslyn and Web API.](http://www.strathweb.com/2013/03/leveraging-roslyn-to-author-asp-net-web-api-without-recompiling/)

https://www.strathweb.com/2013/08/customizing-controller-discovery-in-asp-net-web-api/

https://dannyvanderkraan.wordpress.com/2016/09/09/asp-net-core-1-0-web-api-automatic-documentation-with-swagger-and-swashbuckle/

**Synopsis:** In this article I will help you with getting automatically generated and interactive documentation going for your ASP.NET Core Web API by using Swagger and Swashbuckle. **As of December 23rd upgraded to latest version**. We’ll touch the following subjects:

* Getting started
* Multiple versions
* Complex models
* Adding XML notations
* Multiple Responses

Want to read this in Dutch instead of English then click [here](https://www.kpit-recruitment.nl/blog/blogasp-net-core-1-0-web-api-automatic-documentation-with-swagger-and-swashbuckle/). Small caveat, that article is not upgraded to the latest version!

# Intro

The problem with any REST API is that you need to maintain the documentation describing the API manually (or even if you have it generated from code by some tool it’s still semi-automatic). So it will often be out of date, inaccurate, prone to mistakes and not interactive. To the rescue comes [Swagger](http://swagger.io/), which provides a powerful representation of your RESTful API, thus providing automatic documentation. Open sourced Swagger is platform-agnostic and this article is specifically about adding the power of Swagger to an ASP.NET Core Web API. In order to be able to do that I’d need an implementation specifically for this platform, which luckily is provided by the open source implementation called [Swashbuckle 6.0.0](https://github.com/domaindrivendev/Ahoy).

# Getting started

Swashbuckle seamlessly adds Swagger by combining the built in Api Explorer from ASP.NET Core MVC and Swagger’s swagger-ui to enable discovery and generate interactive documentation for your API’s users. There is only one NuGet package which contains the Swagger generator and Swagger-ui embedded. Add the package Swashbuckle 6.0.0-beta902 to the dependencies section of the project.json file:

|  |  |
| --- | --- |
| 1 | "Swashbuckle": "6.0.0-beta902" |

Once this is done we’ll need to configure Swagger in the Startup class. First we’ll add the services needed in the ConfigureServices method of the StartUp class to generate the files Swagger needs, like this:

|  |  |
| --- | --- |
| 1 | services.AddSwaggerGen(); |

This will register an implementation of ISwaggerProvider with the default settings. Then in the same class in the Configure method you need to add the middleware to the HTTP request pipeline like this:

|  |  |
| --- | --- |
| 1 | app.UseSwagger(); |

This will serve the generated swagger as a JSON endpoint. If you would start up the web application now and you’d navigate to the standard route which is: “/swagger/v1/swagger.json”, you’d see the following JSON:

|  |  |
| --- | --- |
| 1 | {"swagger":"2.0","info":{"version":"v1","title":"API V1"},"basePath":"/","paths":{},"definitions":{},"securityDefinitions":{}} |

**Side note:** You can change this standard route as an option of the UseSwagger extension method.

Well, let’s put Swagger to work by adding an MVC controller. Add the following code to the ConfigureServices method of the StartUp class:

|  |  |
| --- | --- |
| 1 | services.AddMvc(); |

Add the following code to the Configure method of the StartUp class:

|  |  |
| --- | --- |
| 1 | app.UseMvcWithDefaultRoute(); |

Add a folder named “Controllers”.

Add a class to this folder named “HomeController” and have it inherit from Controller. Add the following action method to the Home controller:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [HttpGet("About")]  public ContentResult About()  {      return Content("An API to sample Swagger with Swashbuckle in ASP.NET Core.");  } |

Now run the web application again and navigate to “/swagger/v1/swagger.json” and you should see the following JSON:

|  |  |
| --- | --- |
| 1 | {"swagger":"2.0","info":{"version":"v1","title":"API V1"},"basePath":"/","paths":{"/About":{"get":{"tags":["Home"],"operationId":"AboutGet","produces":[],"responses":{"200":{"description":"OK"}},"deprecated":false}}},"definitions":{},"securityDefinitions":{}} |

Notice how the key “paths” has an entry now, due to standard API discovery which Swashbuckle utilizes to generate the swagger.json to fuel the Swagger engine. To utilize swagger-ui and get interactive documentation you add the following line of code to the Configure method in the Startup class:

|  |  |
| --- | --- |
| 1 | app.UseSwaggerUi(); |

This will add middleware to the HTTP request pipeline which will generate the interactive documentation based off of the swagger.json. So if you navigate to “swagger/ui” it will look like this:

[](https://dannyvanderkraan.files.wordpress.com/2016/03/figure1_swaggerui.png)

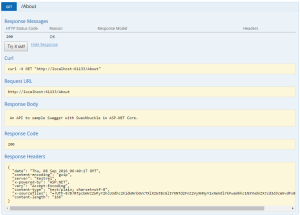
**Side note:** UseSwaggerUI takes two parameters, “baseRoute” (to change the “swagger/ui” default route) and “swaggerURL” (to change the default location “/swagger/v1/swagger.json” where it expects the swagger.json file to be).

The interactive documentation is built up in a RESTful manner. So you first see the link “Home”, because the ‘resource’ “About” is beneath “Home”. If you’d click on it, you’d see the following:

[figure2_swaggerui](https://dannyvanderkraan.files.wordpress.com/2016/09/figure2_swaggerui.png)We can see it’s a “Get” method and we see the relative path “/About”. If you click on “/About” you’d see:

[figure3_swaggerui](https://dannyvanderkraan.files.wordpress.com/2016/09/figure3_swaggerui.png)

As you see it can list the response messages and you can also “Try it out!”. And if you’d click on that you’d see:

[](https://dannyvanderkraan.files.wordpress.com/2016/09/figure4_swaggerui.png)

You can see the “CURL”, the “Request URL”, “Response Body”, “Response Code” and even the “Response Headers” with only three lines of code! Well, I think that is pretty sweet!

# Multiple Versions

A common problem with API’s for external use is that you don’t know how many people are depending on your API and when you make a breaking change you don’t want to break everyone else’s code and you start versioning your API. Luckily Swagger has support for multiple versions and together with Swashbuckle you can easily configure it. Let’s start with making a second version of our “About” action method on the “Home” controller. Change the action methods as the code below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [HttpGet("api/v1/About")]  public ContentResult About()  {      return Content("An API to sample Swagger with Swashbuckle in ASP.NET Core.");  }    [HttpGet("api/v2/About")]  public ContentResult About2()  {      return Content("An API (v2) to sample Swagger with Swashbuckle in ASP.NET Core.");  } |

In the ConfigureServices method of the StartUp class add the following code:

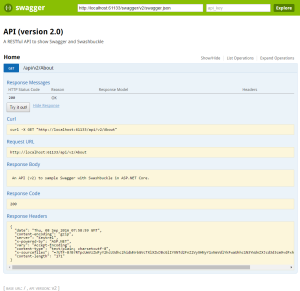
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | services.AddSwaggerGen(options =>      {          options.MultipleApiVersions(new Swashbuckle.Swagger.Model.Info[]          {              new Swashbuckle.Swagger.Model.Info              {                  Version = "v2",                  Title = "API (version 2.0)",                  Description = "A RESTful API to show Swagger and Swashbuckle"              },              new Swashbuckle.Swagger.Model.Info              {                  Version = "v1",                  Title = "API",                  Description = "A RESTful API to show Swagger and Swashbuckle"              }          }, (description, version) =>          {              return description.RelativePath.Contains($"api/{version}");          });        }); |

**Side note:** If your Web API is hosted in IIS, you should avoid using full-stops in the version name (e.g. “1.0”). The full-stop at the tail of the URL will cause IIS to treat it as a static file (i.e. with an extension) and bypass the URL Routing Module and therefore, Web API.

The “AddSwaggerGen” method is the extension method with which you’ll be able to configure your Swagger document. It takes an Action of SwaggerGenOptions, which has a “MultipleApiVersions” method which takes a collection of Info objects with which you can describe your versions. These are used purely for descriptive purposes.

The real workhorse is the second parameter which is a “Function” which takes an instance of “ApiDescription” and a “version” as a string and is expected to return a boolean, true if the specific version is hit, false if not. Because the solutions for this is not homogenised, Swashbuckle solved it like this and you can resolve the version anyway you like really, because of this clever mechanism. In this example I expect that in the route the version (e.g.: “v1” or “v2”) comes after “api/”.

If everything has been done correctly, you should be able to navigate to “swagger/ui” again and change “v2” in the URL to the swagger.json and press “Explore” to be able to try out version 2 of the About action method:

[](https://dannyvanderkraan.files.wordpress.com/2016/09/figure5_swaggerui.png)

# Complex Models

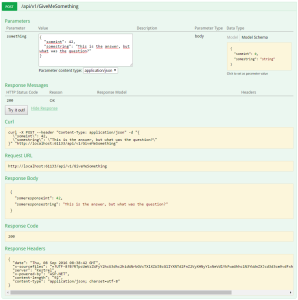
So that’s all fine and dandy, but what about more complex use cases where the action method expects an object and perhaps returns an object of some kind. Let me show you what that looks like with Swagger, by adding the following two class first:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Something  {      [JsonProperty("someint")]      public int SomeInt { get; set; }      [JsonProperty("somestring")]      public string SomeString { get; set; }  }    public class SomeResponse  {      [JsonProperty("someresponseint")]      public int SomeResponseInt { get; set; }      [JsonProperty("someresponsestring")]      public string SomeResponseString { get; set; }  } |

Then add the following action method to HomeController:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | [HttpPost("api/v1/GiveMeSomething")]  public IActionResult GiveMeSomething([FromBody] Something something)  {      return Ok(new SomeResponse()      {          SomeResponseInt = something.SomeInt,          SomeResponseString = something.SomeString      });  } |

Navigate to “swagger/ui” again. Notice how “GiveMeSomething” is added. Notice how it discovered that it needs a parameter called “something”. See the “Model Schema” on the right, click on it to add it as a value of parameter “something”. Change the individual values and click “Try it out!” and it should look like:

[](https://dannyvanderkraan.files.wordpress.com/2016/09/figure6_swaggerui2.png)

**Side note:** Notice how Swagger uses the JSON names that are declared above the properties.

# Adding XML notations

So far the generated documentation hasn’t been very descriptive. Luckily Swashbuckle can utilize XML comments to add documentation to Swagger. First we’ll need to check the “XML documentation file” (VS2015 update 3) checkbox on the “Build” tab of the project properties (or set xmlDoc to true in the buildOptions section in the project.json file):

[figure7_produceoutputsonbuild](https://dannyvanderkraan.files.wordpress.com/2016/09/figure7_produceoutputsonbuild.png)

This will produce the file with the XML comments in it, which Swashbuckle needs to be able to utilize them. Now adjust your code as follows:

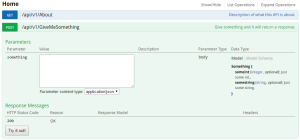
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | services.AddSwaggerGen(options =>      {          options.MultipleApiVersions(new Swashbuckle.Swagger.Model.Info[]          {              new Swashbuckle.Swagger.Model.Info              {                  Version = "v2",                  Title = "API (version 2.0)",                  Description = "A RESTful API to show Swagger and Swashbuckle"              },              new Swashbuckle.Swagger.Model.Info              {                  Version = "v1",                  Title = "API",                  Description = "A RESTful API to show Swagger and Swashbuckle"              }          }, (description, version) =>          {              return description.RelativePath.Contains($"api/{version}");          });          options.IncludeXmlComments(pathToDoc);      }); |

**Side note:** The variable pathToDoc is the path to the XML documentation file. Adjust this to your own personal needs.

Now we can add some XML comments to the controller and models:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | /// <summary>  /// Default entrypoint of the API.  /// </summary>  public class HomeController: Controller  {      /// <summary>      /// Description of what this API is about.      /// </summary>      /// <returns></returns>      [HttpGet("api/v1/About")]      public ContentResult About()      {          return Content("An API to sample Swagger with Swashbuckle in ASP.NET Core.");      }        /// <summary>      /// Give something and it will return a response.      /// </summary>      /// <param name="something"></param>      /// <returns></returns>      [HttpPost("api/v1/GiveMeSomething")]      public IActionResult GiveMeSomething([FromBody] Something something)      {          return Ok(new SomeResponse()          {              SomeResponseInt = something.SomeInt,              SomeResponseString = something.SomeString          });      }        /// <summary>      /// Description of what API v2 is about.      /// </summary>      /// <returns></returns>      [HttpGet("api/v2/About")]      public ContentResult About2()      {          return Content("An API (v2) to sample Swagger with Swashbuckle in ASP.NET Core.");      }  }    /// <summary>  /// Just something to put in the request.  /// </summary>  public class Something  {      /// <summary>      /// Just some int.      /// </summary>      [JsonProperty("someint")]      public int SomeInt { get; set; }      /// <summary>      /// Just some string.      /// </summary>      [JsonProperty("somestring")]      public string SomeString { get; set; }  }    /// <summary>  /// Just some response to give back.  /// </summary>  public class SomeResponse  {      /// <summary>      /// Some int for the response.      /// </summary>      [JsonProperty("someresponseint")]      public int SomeResponseInt { get; set; }      /// <summary>      /// Some string for the response.      /// </summary>      [JsonProperty("someresponsestring")]      public string SomeResponseString { get; set; }  } |

And if you’d navigate again to “swagger/ui” you should see something like:

[](https://dannyvanderkraan.files.wordpress.com/2016/09/figure8_xmlcomments.png)

Please note that “Model” is clicked instead of “Model Schema” beneath “Data Type” to make the XML comments visible on the model. Furthermore, note the XML comments visible behind the relative URL’s.

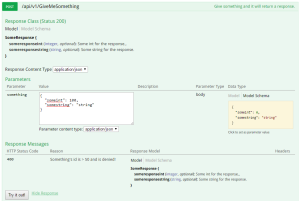
We are almost done with the basics, but before I end this article we should really look at the response, which is still at its bare minimum right now.

# Multiple Responses

What if we’d adjust the “GiveMeSomething” method to check if “SomeInt” is below 50 and return a BadRequest if it isn’t, as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | /// <summary>  /// Give something and it will return a response.  /// </summary>  /// <param name="something"></param>  /// <returns></returns>  [ProducesResponseType(typeof(SomeResponse), 200)]  [ProducesResponseType(typeof(BadRequestResultObject), 400)]  [HttpPost("api/v1/GiveMeSomething")]  public IActionResult GiveMeSomething([FromBody] Something something)  {      if (something.SomeInt <= 50)      {          return Ok(new SomeResponse()          {              SomeResponseInt = something.SomeInt,              SomeResponseString = something.SomeString          });      }      else      {          return new BadRequestObjectResult(new SomeResponse()          {              SomeResponseInt = 0,              SomeResponseString = string.Empty          });      }  } |

Now it returns two responses, an OK status and a BadRequest. For Swashbuckle to be able to utilize this in the documentation you need to add the ProducesResponseType attribute. It’ll take a status code it’s triggered on, and a type of the response. If you provide a type it’ll enable Swashbuckle to utilize the XML comments on the model again. So if you’d navigate to “swagger/ui” again it should look like:

[](https://dannyvanderkraan.files.wordpress.com/2016/09/figure9_multipleresponses.png)  
**Side note**Image may not be entirely accurate anymore.

You see the main response, in this case “Response Class (Status 200), right beneath the relative URL “/api/v1/GiveMeSomething”. As you can see the XML comments are visible for “someresponseint” and “someresponsestring”. And beneath the “Parameters” section you see the “Response Messages” section which will list all the other responses. In this case it shows us we can expect a HTTP Status Code of 400 and the XML comments are shown again at “Response Model”.

https://developers.google.com/api-client-library/dotnet/get\_started

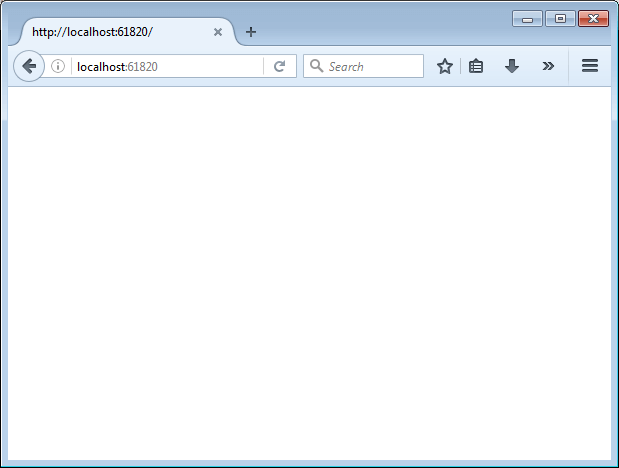
https://www.mytectra.com/interview-question/top-asp-net-web-api-essentials-using-c-interview-questions-2017/

https://github.com/domaindrivendev/Swashbuckle

[Swagger](http://swagger.io/) is a simple yet powerful representation of your RESTful API. Once integrated with WEB API, it becomes easy to test the API without using any third-party tool. In this post, we will see how to add Swagger to ASP.NET Core Web API.

## Add Swagger to ASP.NET Core Web API

Let’s create an [ASP.NET Core](http://www.talkingdotnet.com/asp-net-core-1-0/) Web API project. The default Web API template comes with a controller named “Values” with some GET, POST, PUT and DELETE method. Once the WEB API project is created, just run the app. You will see the following. There is no UI. There isn’t a list of all API methods and any other detail about them.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/WebAPI-without-swagger.png)  
Here, Swagger can be great help. To configure Swagger, we need to add the required nuget package for it. So open [project.json](http://www.talkingdotnet.com/what-is-project-json-file-in-asp-net-5-vnext/) and add the following line under dependencies section and save it. Visual studio will restore the package.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | dependencies": {      "Microsoft.NETCore.App": {        "version": "1.0.0",        "type": "platform"      },      "Swashbuckle": "6.0.0-beta901"    }, |

Now open Startup.cs file to add swagger service to middleware.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public void ConfigureServices(IServiceCollection services)  {      // Add framework services.      services.AddApplicationInsightsTelemetry(Configuration);      services.AddMvc();      services.AddSwaggerGen();  } |

And also add the highlighted lines code in Configure method.

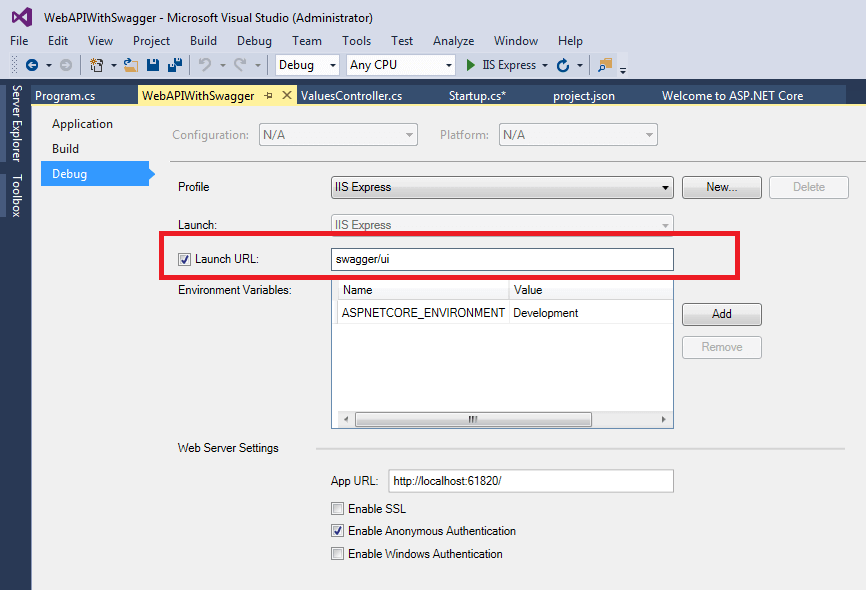
[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)  {      loggerFactory.AddConsole(Configuration.GetSection("Logging"));      loggerFactory.AddDebug();        app.UseApplicationInsightsRequestTelemetry();      app.UseApplicationInsightsExceptionTelemetry();        app.UseMvc();      app.UseSwagger();      app.UseSwaggerUi();  } |

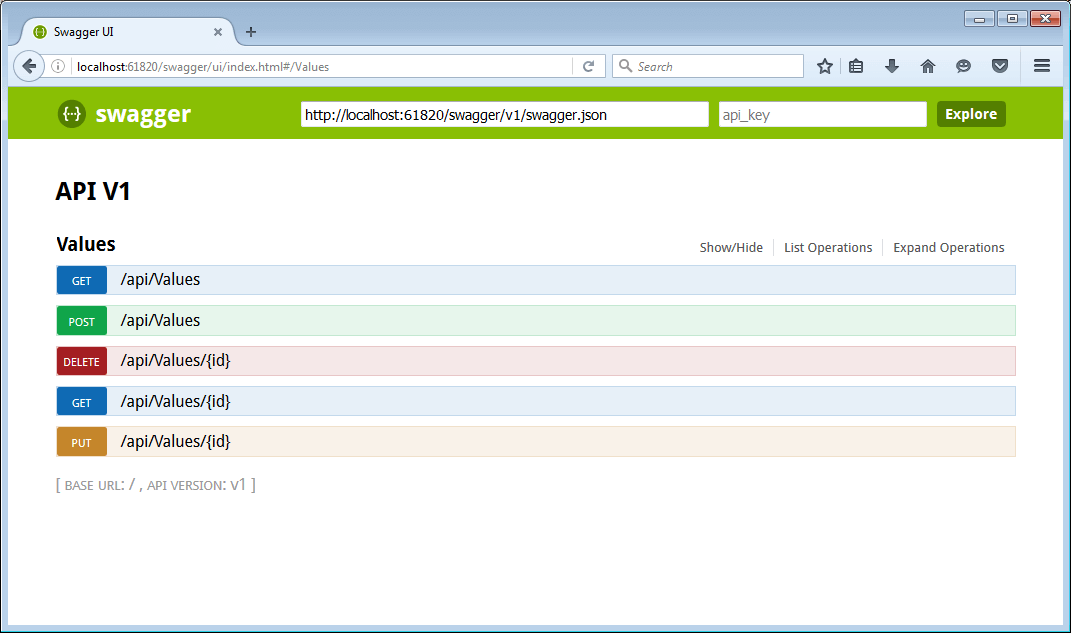
Now run the app and guess what still no swagger UI . Well to see swagger UI, append **/swagger/ui** to the Web API URL. For example, if you are running on localhost, then URL should be,  
**http://localhost:61820/swagger/ui/**

### Set Swagger URL as launch URL

But every time, appending **/swagger/ui** to URL is pain. This can be fixed and swagger URL can be set as application’s **launch URL**. To set it, right-click on Project -> select properties -> navigate to debug tab. On debug tab, change Launch URL value to “swagger/ui”.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/Setting-Swagger-URL-as-Launch-URL.png)

When you run the app with Swagger URL, you should see following.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/WebAPI-with-swagger.png)  
Here you can see Values controller with all the API methods along with HTTP verb settings. And different colors are used for different verbs to easily identify actions. Clicking on any method will give you details about accepted parameters, return type and allows you to test the method.

We achieved this with minimal configuration and without any customization. Swagger can be customized to include your own API description, show method’s XML comments, show enum values as string and many more. Let’s do some customization.

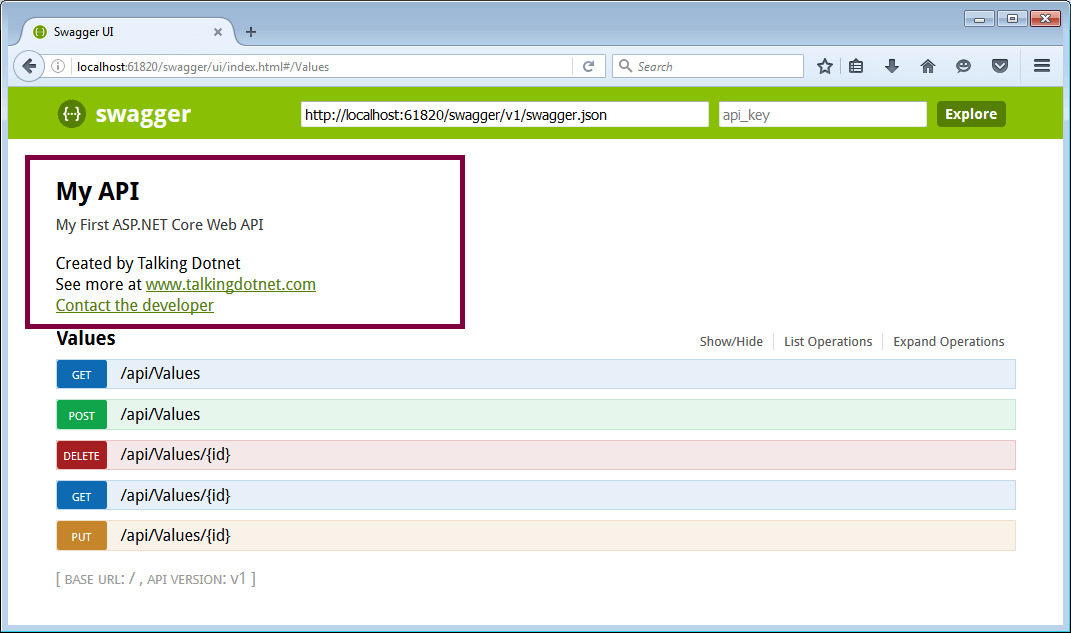
### Add API Description

To add your own description, add the following highlight code to ConfigureServices method of Startup.cs. You can add version number, title, description, author details and any terms to use the API.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public void ConfigureServices(IServiceCollection services)  {      // Add framework services.      services.AddApplicationInsightsTelemetry(Configuration);      services.AddMvc();      services.AddSwaggerGen();      services.ConfigureSwaggerGen(options =>      {          options.SingleApiVersion(new Info          {              Version = "v1",              Title = "My API",              Description = "My First ASP.NET Core Web API",              TermsOfService = "None",              Contact = new Contact() { Name = "Talking Dotnet", Email = "contact@talkingdotnet.com", Url = "www.talkingdotnet.com" }          });      });  } |

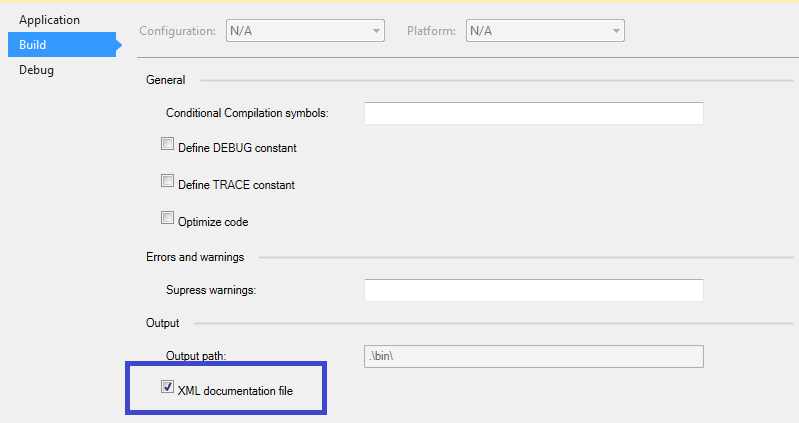
Run the app and you should see the description on top as highlighted in below image.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/WebAPI-with-customized-swagger-description.png)

### Add action’s XML Comments

By default, swagger does not use XML comments which we put on top of actions. But there is an option to display them with Swagger UI. First, we need to enable a setting in our project so that when the project is built, all the XML comments get saved in a XML file and then swagger can use it to display the comments.

To enable it, right-click on project -> select properties option then on Build tab. And on build tab check “XML documentation file” option.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/Enable-XML-documentation-for-Swagger.png)  
By enabling this option, all xml comments of your project are saved in a XML file with the name [your assembly].xml. And this file is saved in bin\[Debug/Release]\newcoreapp1.0 folder. We need to supply location of this file to Swashbuckle’s IncludeXMLComments method.

Add a method in Startup.cs to get the path of generated XML. This code to get XML path will work in your local environment as well as in production environment.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | private string GetXmlCommentsPath()  {      var app = PlatformServices.Default.Application;      return System.IO.Path.Combine(app.ApplicationBasePath, "WebAPIWithSwagger.xml");  } |

And now add code to include XML comments in ConfigureService method.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

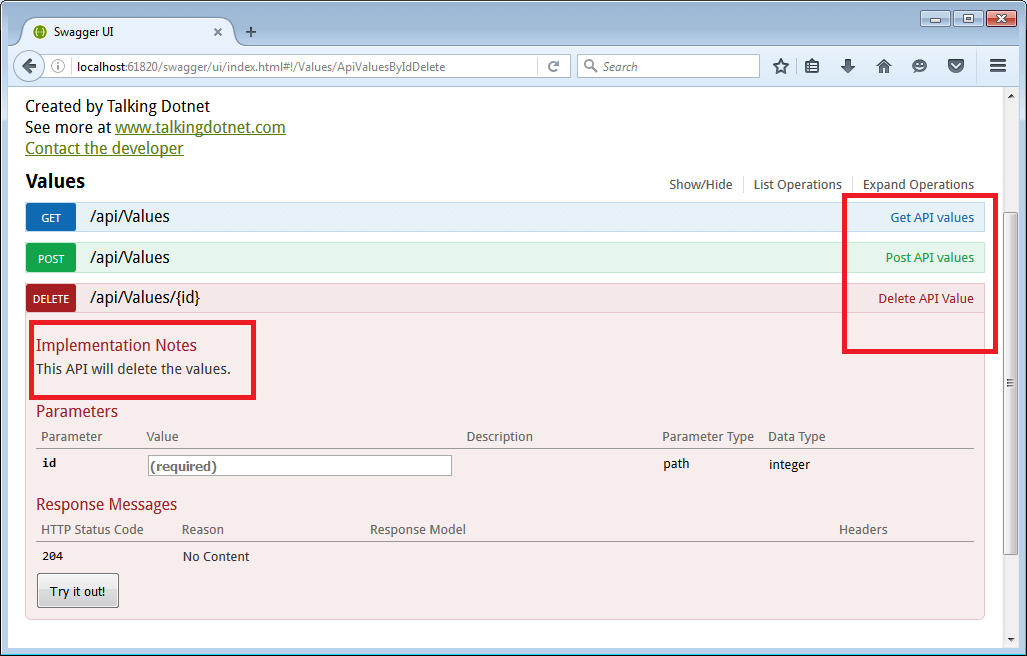
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public void ConfigureServices(IServiceCollection services)  {      // Add framework services.      services.AddApplicationInsightsTelemetry(Configuration);      var xmlPath = GetXmlCommentsPath();      services.AddMvc();      services.AddSwaggerGen();      services.ConfigureSwaggerGen(options =>      {          options.SingleApiVersion(new Info          {              Version = "v1",              Title = "My API",              Description = "My First ASP.NET Core Web API",              TermsOfService = "None",              Contact = new Contact() { Name = "Talking Dotnet", Email = "contact@talkingdotnet.com", Url = "www.talkingdotnet.com" }          });          options.IncludeXmlComments(xmlPath);      });  } |

Now let’s add some XML comments to our actions. I added XML comments to all the actions. Here is XML comment written for Delete method.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // DELETE api/values/5  /// <summary>  /// Delete API Value  /// </summary>  /// <remarks>This API will delete the values.</remarks>  /// <param name="id"></param>  [HttpDelete("{id}")]  public void Delete(int id)  {  } |

Now run the app, you should see XML comments.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/Swagger-showing-XML-comments.png)

It’s good to see that XML comments converted into beautiful documentation. Isn’t it?

### Showing Enum values as string

Another customization, we can make it to display enum values as strings. By default, it will be displayed as numbers. Let’s add an enum and modify Get method to accept enum type parameter.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

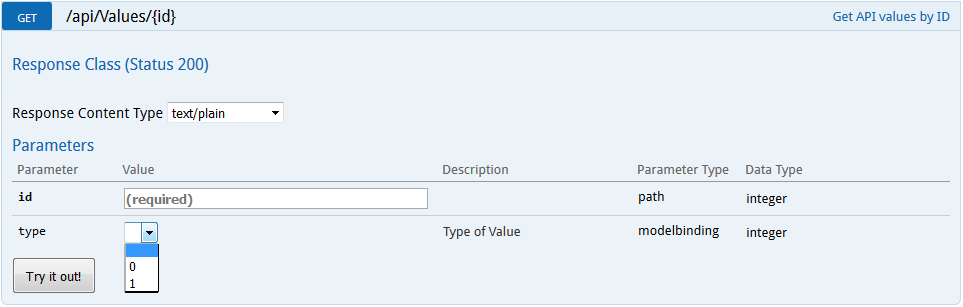
|  |  |
| --- | --- |
| 1  2  3  4  5 | public enum eValueType  {      Number,      Text  } |

And updated Get method is,

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // GET api/values/5  /// <summary>  /// Get API values by ID  /// </summary>  /// <param name="id"></param>  /// <param name="type">Type of Value</param>  /// <returns></returns>  [HttpGet("{id}")]  public string Get(int id, eValueType type)  {      return "value";  } |

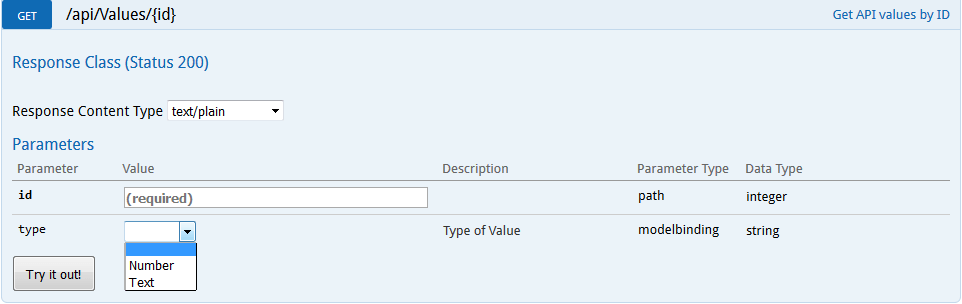
Run the app and see the swagger UI for this method. The enum values are displayed in the dropdown as numbers. Based on number, it’s difficult to identify the string value for enum.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/Swagger-Without-DescribeAllEnumsAsStrings.png)  
Swashbuckles has method DescribeAllEnumsAsStrings() which needs to be added to code to display enum values as string.

[?](http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | services.ConfigureSwaggerGen(options =>  {      options.SingleApiVersion(new Info      {          Version = "v1",          Title = "My API",          Description = "My First Core Web API",          TermsOfService = "None",          Contact = new Contact() { Name = "Talking Dotnet", Email = "contact@talkingdotnet.com", Url = "www.talkingdotnet.com" }      });      options.IncludeXmlComments(xmlPath);      options.DescribeAllEnumsAsStrings();  }); |

And now run the app. You should see actual enum string values in drop down instead of numbers. Quite useful.

[](http://www.talkingdotnet.com/wp-content/uploads/2016/08/Swagger-With-DescribeAllEnumsAsStrings.png)

There is also a method named DescribeStringEnumsInCamelCase to covert enum string values in camel case.

### Summary

Swagger is really useful. It provides an UI to your WEB API which helps in testing the API’s easily. And integrating this with ASP.NET Core WEB API is also quite simple and straight forward.

Thank you for reading. Keep visiting this blog and share this in your network. Please put your thoughts and feedback in comments section.

http://www.talkingdotnet.com/add-swagger-to-asp-net-core-web-api/

https://swagger.io/specification/

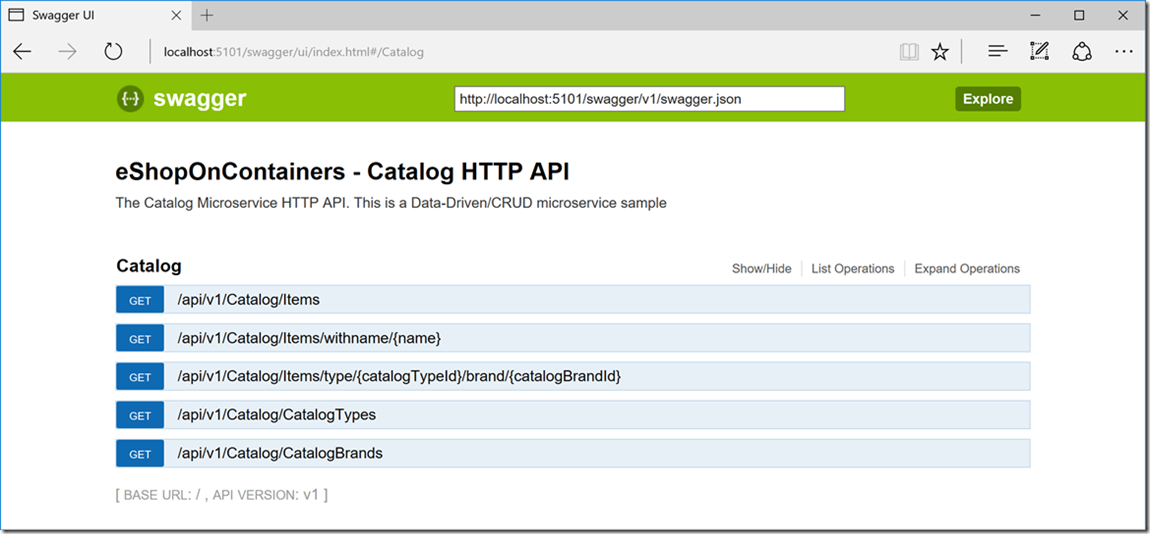
Generating Swagger metadata manually (JSON or YAML file) can be a tedious work if you have to write it manually. However, you can automate API discovery of ASP.NET Web API services by using the [Swashbuckle NuGet package](http://aka.ms/swashbuckledotnetcore) to dynamically generate Swagger API metadata.

Swashbuckle is seamlessly and automatically adds Swagger metadata to ASP.NET Web Api projects. Depending on the package version, it supports ASP.NET Core Web API projects and the traditional ASP.NET Web API and any other “flavor” like Azure API App, Azure Mobile App, Azure Service Fabric microservices based on ASP.NET or plain Web API on containers, as in this case.

Swashbuckle combines ApiExplorer and Swagger/swagger-ui to provide a rich discovery and documentation to your API consumers.

In addition to its Swagger metadata generator engine, Swashbuckle also contains an embedded version of swagger-ui which it will automatically serve up once Swashbuckle is installed.

This means you can complement your API with a slick discovery UI to assist developers with their integration efforts. Best of all, it requires minimal coding and maintenance because it is automatically generated, allowing you to focus on building your API. The final result for the API explorer will look as the image below.

[](https://msdnshared.blob.core.windows.net/media/2016/12/image137.png)

But that UI explorer is not the most important thing here, as mentioned, once you have a Web API that can describe itself in Swagger metadata, your API can seamlessly be used from Swagger-based tools including client proxy classes code generator that can target many platforms, like using [swagger-codegen](https://github.com/swagger-api/swagger-codegen), for example, which allows code generation of API client libraries, server stubs and documentation automatically.

Currently, Swashbuckle consists of two NuGet packages – *Swashbuckle.SwaggerGen* and *Swashbuckle.SwaggerUi*. The former provides functionality to generate one or more Swagger documents directly from your API implementation and expose them as JSON endpoints. The latter provides an embedded version of the swagger-ui tool that can be served by your application and powered by the generated Swagger documents to describe your API.

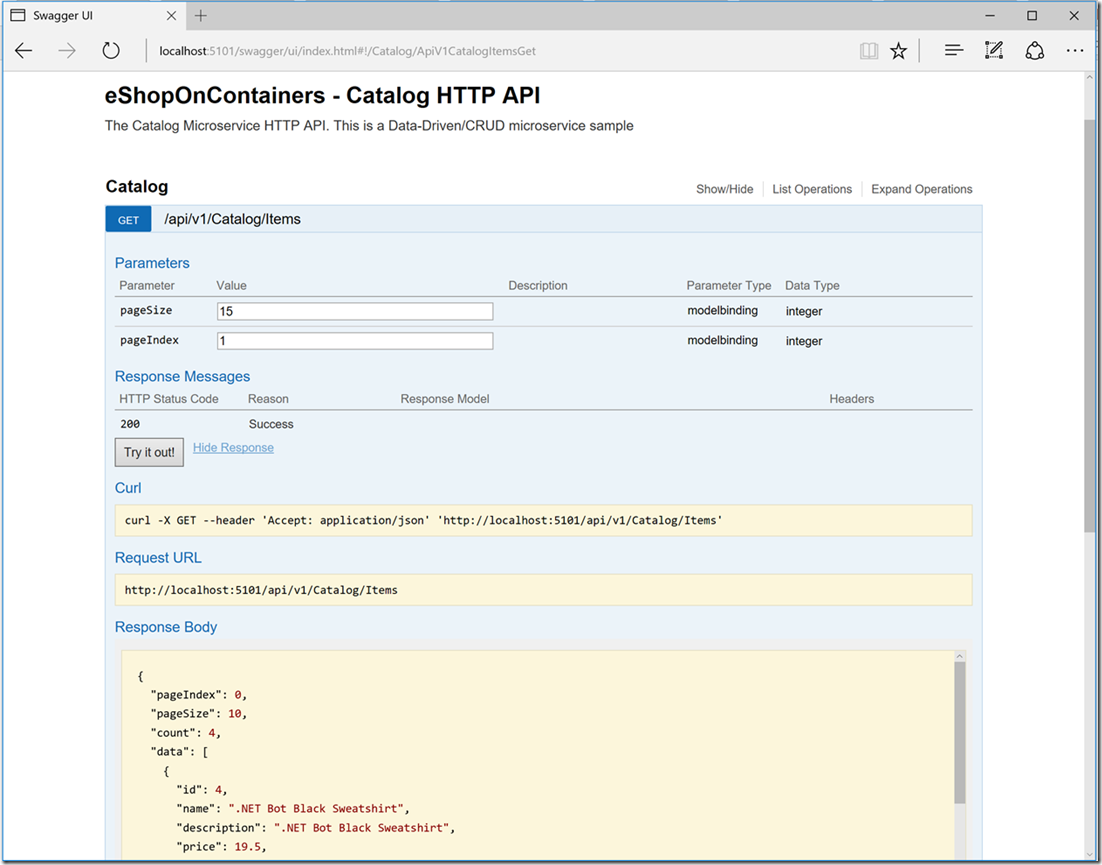
Once you have installed those Nuget packages on your Web API project, you will need to configure Swagger in your Startup.cs class, as in the following code in **your ASP.NET Core Web API project:**

|  |
| --- |
| **public class Startup**  {  public IConfigurationRoot Configuration { get; }  //Other Startup code…  public void ConfigureServices(IServiceCollection services)  {  //Other ConfigureServices() code…  **services.AddSwaggerGen();**  **services.ConfigureSwaggerGen(options =>**  {  options.DescribeAllEnumsAsStrings();  **options.SingleApiVersion(new Swashbuckle.Swagger.Model.Info()**  **{**  **Title = “MyCatalog HTTP API”,**  **Version = “v1”,**  **Description = “My Catalog Microservice HTTP API”,**  **TermsOfService = “My terms Of Service”**  **});**  });  //Other ConfigureServices() code…  }  public void Configure(IApplicationBuilder app,  IHostingEnvironment env,  ILoggerFactory loggerFactory)  {  //Other Configure() code…  // …  app.UseSwagger()  .UseSwaggerUi();  }  } |

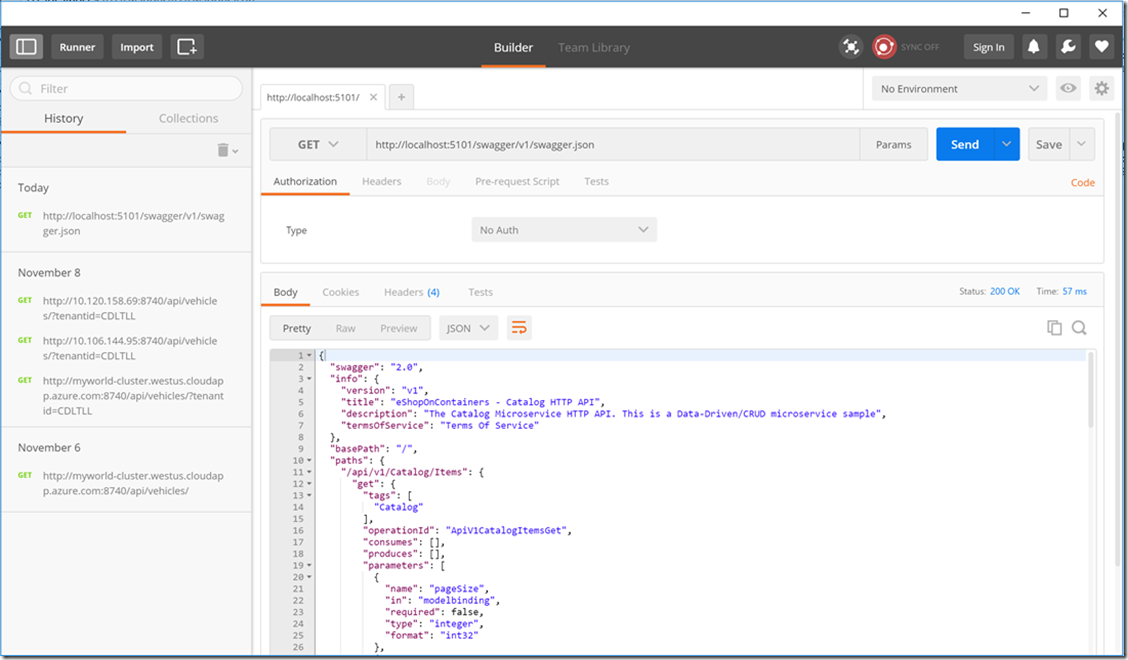
Once this is done, you should be able to spin up your app and browse the following Swagger JSON and UI endpoints respectively.

|  |
| --- |
| **<your-root-url>/swagger/v1/swagger.json**  **<your-root-url>/swagger/ui** |

You previously showed the generated UI created by Swashbuckle with the URL “<your-root-url>/swagger/ui”. Below you can also see how you can test any specific API method.

[](https://msdnshared.blob.core.windows.net/media/2016/12/image138.png)

Now, the following image is the Swagger JSON metadata generated from the eShopOnContainer microservice (which is really what the tools use underneath) when you test it and request that <your-root-url>/swagger/v1/swagger.json URL using the convenient [Postman](https://www.getpostman.com/) tool.

[](https://msdnshared.blob.core.windows.net/media/2016/12/image139.png)

It is that simple, and because it is automatically generated, the Swagger metadata will grow while you add more functionality to your API.

NOTE: Currently, [Swashbuckle 6.0.0](http://aka.ms/swashbuckledotnetcore) version is what you need to use for ASP.NET Core Web API projects which is by the way the most common case when building Docker containers with .NET Core (as in the code I’m testing using Docker), but you can also use plain ASP.NET Core on Linux or Windows, with no Docker, of course.

If using the traditional .NET Framework for Windows, you need to use a different NuGet package version.

https://blogs.msdn.microsoft.com/cesardelatorre/2016/12/05/generating-swagger-description-metadata-from-your-asp-net-core-web-apis-with-swashbuckle/

https://github.com/domaindrivendev/Swashbuckle.AspNetCore

https://blogs.msdn.microsoft.com/cesardelatorre/2016/12/05/generating-swagger-description-metadata-from-your-asp-net-core-web-apis-with-swashbuckle/

# What is Swagger

[Swagger](http://swagger.io/) is a very much used open source framework backed by a large ecosystem of tools that helps you design, build, document, and consume your RESTful APIs. It is probably becoming as the main standard for this domain (APIs description metadata).

The heart of Swagger is the Swagger Specification (API description metadata which is a JSON or YAML file). The specification creates the RESTful contract for your API, detailing all of its resources and operations in a human and machine readable format for easy development, discovery, and integration.

The specification is the basis of the OpenAPI Specification (OAS) and is developed in an open, transparent, and collaborative community to standardize the way RESTful interfaces are defined.

This specification defines the structure for how a service can be discovered and its capabilities understood. More information, a Web Editor, and examples of Swaggers from companies like Spotify, Uber, Slack, Microsoft and many more can be found at <http://swagger.io>

# Why to use Swagger

The main reasons why you would want to generate Swagger metadata about your APIs are basically the following:

– **Ability to automatically consume and integrate your APIs with tens of products** and [commercial tools supporting Swagger](http://swagger.io/commercial-tools/) plus many [libraries and frameworks](http://swagger.io/open-source-integrations/) serving the Swagger ecosystem. Microsoft has high level products and tools that can automatically consume Swagger based APIs, like the following:

o [**Microsoft Flow**](https://flow.microsoft.com/en-us/) – Ability to automatically [use and integrate your API](https://flow.microsoft.com/en-us/blog/integrating-custom-api/) into a high-level Microsoft Flow workflow, with no programming skills required.

o [**Microsoft PowerApps**](https://powerapps.microsoft.com/en-us/) – Ability to automatically consume your API from [PowerApps mobile apps](https://powerapps.microsoft.com/en-us/blog/register-and-use-custom-apis-in-powerapps/) built with [PowerApps Studio](https://powerapps.microsoft.com/en-us/guided-learning/learning-powerapps-parts/), with no programming skills required.

o [**Azure App Service Logic Apps**](https://docs.microsoft.com/en-us/azure/app-service-logic/app-service-logic-what-are-logic-apps) – Ability to automatically [use and integrate your API into an Azure App Service Logic App](https://docs.microsoft.com/en-us/azure/app-service-logic/app-service-logic-custom-hosted-api), with no programming skills required.

– **APIs documentation automatically generated** – When creating large scale RESTful APIs, like when building complex microservice based applications, you will need to handle a large number of endpoints with different data models used in the request/response payloads. Proper documentation and having a solid API explorer is a crucial pillar for your API success and likability by developers.

Swagger’s metadata is basically what Microsoft Flow, PowerApps and Azure Logic Apps use to understand how to use services/APIs and connect to it.

# How to automate API Swagger metadata generation with Swashbuckle NuGet package from your ASP.NET Core Web APIs

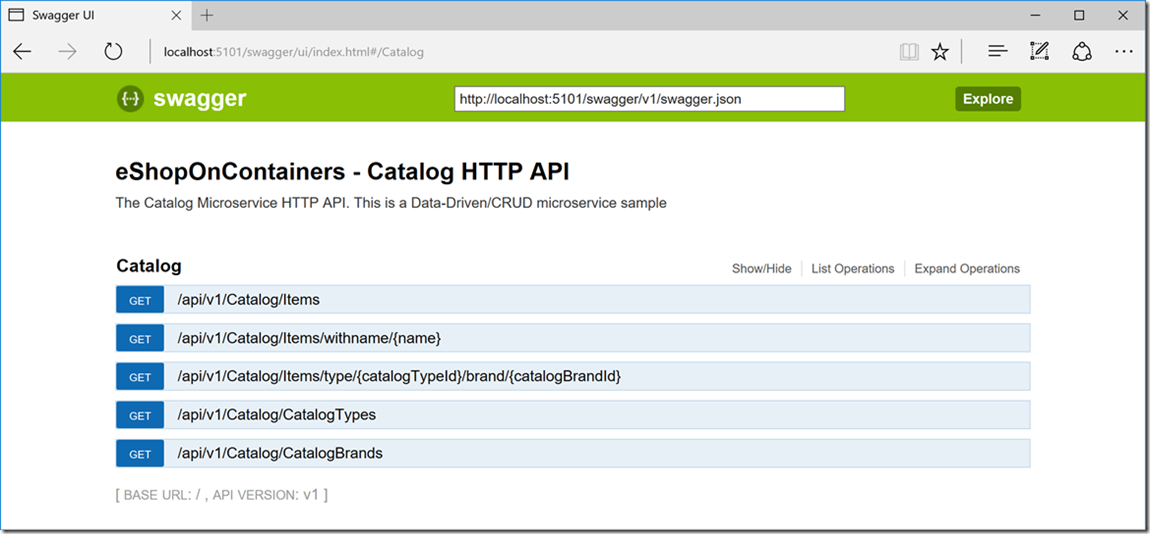
Generating Swagger metadata manually (JSON or YAML file) can be a tedious work if you have to write it manually. However, you can automate API discovery of ASP.NET Web API services by using the [Swashbuckle NuGet package](http://aka.ms/swashbuckledotnetcore) to dynamically generate Swagger API metadata.

Swashbuckle is seamlessly and automatically adds Swagger metadata to ASP.NET Web Api projects. Depending on the package version, it supports ASP.NET Core Web API projects and the traditional ASP.NET Web API and any other “flavor” like Azure API App, Azure Mobile App, Azure Service Fabric microservices based on ASP.NET or plain Web API on containers, as in this case.

Swashbuckle combines ApiExplorer and Swagger/swagger-ui to provide a rich discovery and documentation to your API consumers.

In addition to its Swagger metadata generator engine, Swashbuckle also contains an embedded version of swagger-ui which it will automatically serve up once Swashbuckle is installed.

This means you can complement your API with a slick discovery UI to assist developers with their integration efforts. Best of all, it requires minimal coding and maintenance because it is automatically generated, allowing you to focus on building your API. The final result for the API explorer will look as the image below.

[](https://msdnshared.blob.core.windows.net/media/2016/12/image137.png)

But that UI explorer is not the most important thing here, as mentioned, once you have a Web API that can describe itself in Swagger metadata, your API can seamlessly be used from Swagger-based tools including client proxy classes code generator that can target many platforms, like using [swagger-codegen](https://github.com/swagger-api/swagger-codegen), for example, which allows code generation of API client libraries, server stubs and documentation automatically.

Currently, Swashbuckle consists of two NuGet packages – *Swashbuckle.SwaggerGen* and *Swashbuckle.SwaggerUi*. The former provides functionality to generate one or more Swagger documents directly from your API implementation and expose them as JSON endpoints. The latter provides an embedded version of the swagger-ui tool that can be served by your application and powered by the generated Swagger documents to describe your API.

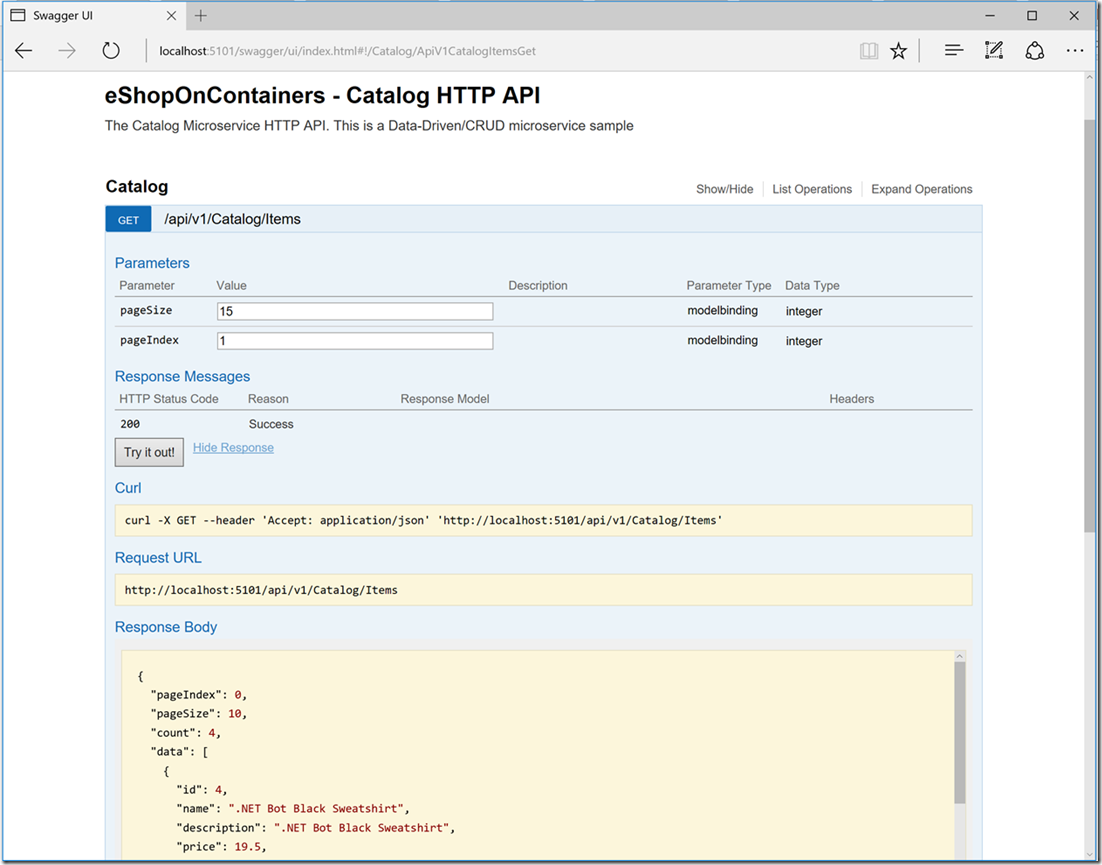
Once you have installed those Nuget packages on your Web API project, you will need to configure Swagger in your Startup.cs class, as in the following code in **your ASP.NET Core Web API project:**

|  |
| --- |
| **public class Startup**  {  public IConfigurationRoot Configuration { get; }  //Other Startup code…  public void ConfigureServices(IServiceCollection services)  {  //Other ConfigureServices() code…  **services.AddSwaggerGen();**  **services.ConfigureSwaggerGen(options =>**  {  options.DescribeAllEnumsAsStrings();  **options.SingleApiVersion(new Swashbuckle.Swagger.Model.Info()**  **{**  **Title = “MyCatalog HTTP API”,**  **Version = “v1”,**  **Description = “My Catalog Microservice HTTP API”,**  **TermsOfService = “My terms Of Service”**  **});**  });  //Other ConfigureServices() code…  }  public void Configure(IApplicationBuilder app,  IHostingEnvironment env,  ILoggerFactory loggerFactory)  {  //Other Configure() code…  // …  app.UseSwagger()  .UseSwaggerUi();  }  } |

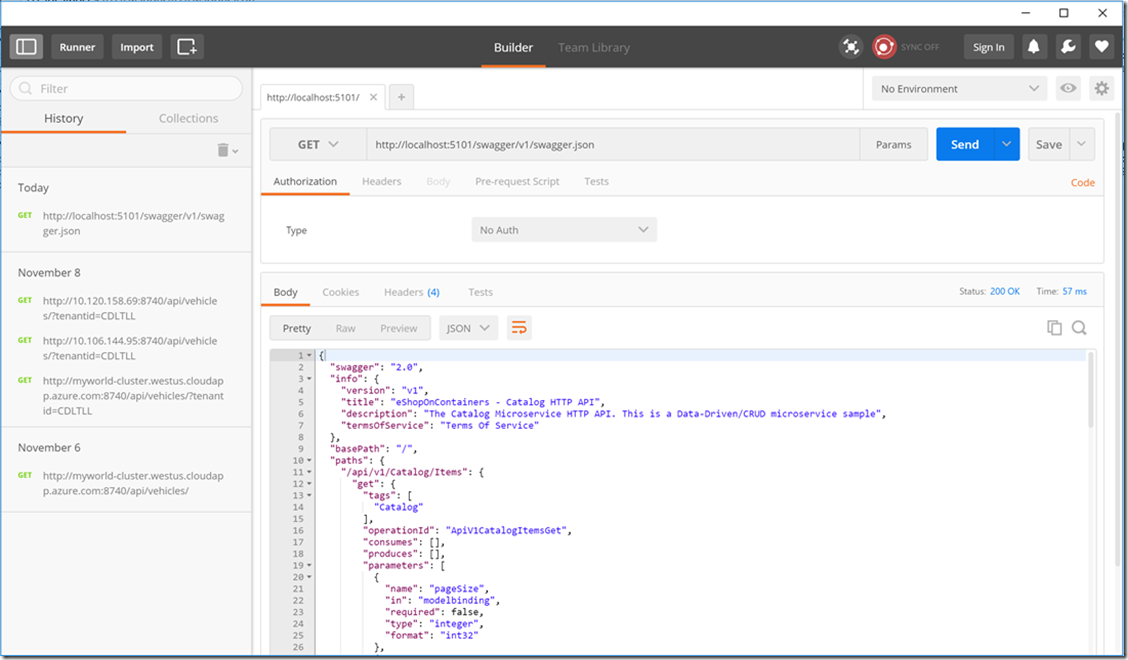
Once this is done, you should be able to spin up your app and browse the following Swagger JSON and UI endpoints respectively.

|  |
| --- |
| **<your-root-url>/swagger/v1/swagger.json**  **<your-root-url>/swagger/ui** |

You previously showed the generated UI created by Swashbuckle with the URL “<your-root-url>/swagger/ui”. Below you can also see how you can test any specific API method.

[](https://msdnshared.blob.core.windows.net/media/2016/12/image138.png)

Now, the following image is the Swagger JSON metadata generated from the eShopOnContainer microservice (which is really what the tools use underneath) when you test it and request that <your-root-url>/swagger/v1/swagger.json URL using the convenient [Postman](https://www.getpostman.com/) tool.

[](https://msdnshared.blob.core.windows.net/media/2016/12/image139.png)

It is that simple, and because it is automatically generated, the Swagger metadata will grow while you add more functionality to your API.

NOTE: Currently, [Swashbuckle 6.0.0](http://aka.ms/swashbuckledotnetcore) version is what you need to use for ASP.NET Core Web API projects which is by the way the most common case when building Docker containers with .NET Core (as in the code I’m testing using Docker), but you can also use plain ASP.NET Core on Linux or Windows, with no Docker, of course.

If using the traditional .NET Framework for Windows, y

https://github.com/swagger-api/swagger-ui

https://github.com/domaindrivendev/Swashbuckle.AspNetCore

https://www.codeproject.com/Articles/1078249/RESTful-Web-API-Help-Documentation-using-Swagger-U