This repository Search Pull requests Issues Gist

domaindrivendev / **Swashbuckle.AspNetCore**

Watch 86 Star 413 Fork 153

Code Issues 37 Pull requests 6 Projects 0 Wiki Pulse Graphs

Swagger tools for documenting API's built on ASP.NET Core

302 commits 8 branches 9 releases 23 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

domaindrivendev cleanup from previous two commits Latest commit 383ee28 3 hours ago

src	cleanup from previous two commits	3 hours ago
test	serve swagger-ui at friendly URL - i.e. /{RoutePrefix}/	3 hours ago
tools/psake	Adds a psake (PowerShell) based build script that restores, builds, t…	7 months ago
.editorconfig	Added .editorconfig to help enforce indentation/style.	7 months ago
.gitignore	Ignoring .pdb so downloaded symbols during test debugging don't get c…	2 months ago
LICENSE	Add MIT License	a year ago
NuGet.config	Fixed tests to use the latest XUnit and work in Visual Studio Test Ex…	7 months ago
README.md	Use MS Versioning lib in MultipleVersions sample site	a day ago
Swashbuckle.AspNetCore.sln	Some additional post-rename cleanup	a month ago
appveyor.yml	Rename to Swashbuckle.AspNetCore	a month ago
build-definition.ps1	Further build script tweaks	a month ago
build.cmd	AppVoyer CI build and push preview packages to MyGet	3 months ago
build.ps1	AppVoyer CI build and push preview packages to MyGet	3 months ago
global.json	Updated project.json for dotnet RTM and easier comparison (sort/tidy).	7 months ago

README.md

Attention early adopters - please note the package rename to Swashbuckle.AspNetCore

If you've been using Swashbuckle for ASP.NET Core since it's inception, you may be referencing "Swashbuckle.6.0.0-beta*" packages. These are no longer valid. The original intention had been to move the Swashbuckle project from ASP.NET WebApi onto the ASP.NET Core stack in a major version bump. However, now that Microsoft plans to support WebApi for at least 4 more years, I felt it was more appropriate to split into separate projects, allowing both to live on with their respective frameworks. Although, I personally will be devoting most of my time to this version.

Swashbuckle.AspNetCore

build passing

Swagger tooling for API's built with ASP.NET Core. Generate beautiful API documentation, including a UI to explore and test operations, directly from your routes, controllers and models.

In addition to its Swagger generator, Swashbuckle also provides an embedded version of the awesome swagger-ui that's powered by the generated Swagger JSON. This means you can complement your API with living documentation that's always in sync with the latest code. Best of all, it requires minimal coding and maintenance, allowing you to focus on building an awesome API.

And that's not all ...

Once you have an API that can describe itself in Swagger, you've opened the treasure chest of Swagger-based tools including a client generator that can be targeted to a wide range of popular platforms. See swagger-codegen for more details.

Getting Started

1. Install the standard Nuget package (currently pre-release) into your ASP.NET Core application.

   ```
   Install-Package Swashbuckle.AspNetCore -Pre
   ```

2. In the *ConfigureServices* method of *Startup.cs*, register the Swagger generator, defining one or more Swagger documents.

   ```
   services.AddSwaggerGen(c =>
   {
       c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
   });
   ```

3. Ensure your API actions and non-route parameters are decorated with explicit "Http" and "From" bindings.

   ```
   [HttpPost]
   public void Create([FromBody]Product product)
   ...

   [HttpGet]
   public IEnumerable<Product> Search([FromQuery]string keywords)
   ...
   ```

   NOTE: If you omit the explicit parameter bindings, the generator will describe them as "query" params by default.

4. In the *Configure* method, insert middleware to expose the generated Swagger as JSON endpoint(s)

   ```
   app.UseSwagger();
   ```

   At this point, you can spin up your application and view the generated Swagger JSON at "/swagger/v1/swagger.json."

5. Optionally insert the swagger-ui middleware if you want to expose interactive documentation, specifying the Swagger JSON endpoint(s) to power it from.

   ```
   app.UseSwaggerUi(c =>
   {
       c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
   })
   ```

   Now you can restart your application and check out the auto-generated, interactive docs at "/swagger".

Components

Swashbuckle consists of three packages - a Swagger generator, middleware to expose the generated Swagger as JSON endpoints and middleware to expose a swagger-ui that's powered by those endpoints. They can be installed together, via the "Swashbuckle.AspNetCore" meta-package, or independently according to your specific requirements. See the table below for more details.

Package	Description
Swashbuckle.AspNetCore.Swagger	Exposes *SwaggerDocument* objects as a JSON API. It expects an implementation of *ISwaggerProvider* to be registered which it queries to retrieve Swagger document(s) before returning as serialized JSON
Swashbuckle.AspNetCore.SwaggerGen	Injects an implementation of *ISwaggerProvider* that can be used by the above component. This particular implementation automatically generates *SwaggerDocument*(s) from your routes, controllers and models

Package	Description
Swashbuckle.AspNetCore.SwaggerUi	Exposes an embedded version of the swagger-ui. You specify the API endpoints where it can obtain Swagger JSON and it uses them to power interactive docs for your API

Configuration & Customization

The steps described above will get you up and running with minimal setup. However, Swashbuckle offers a lot of flexibility to customize as you see fit. Check out the table below for the full list of options:

- Swashbuckle.AspNetCore.Swagger
  - Change the Path for Swagger JSON Endpoints
  - Modify Swagger with Request Context
- Swashbuckle.AspNetCore.SwaggerGen
  - List Operations Responses
  - Include Descriptions from XML Comments
  - Provide Global API Metadata
  - Generate Multiple Swagger Documents
  - Omit Obsolete Operations and/or Schema Properties
  - Omit Arbitrary Operations
  - Customize Operation Tags (e.g. for UI Grouping)
  - Change Operation Sort Order (e.g. for UI Sorting)
  - Customize Schema Id's
  - Customize Schema for Enum Types
  - Override Schema for Specific Types
  - Extend Generator with Operation, Schema & Document Filters
  - Add Security Definitions and Requirements
- Swashbuckle.AspNetCore.SwaggerUi
  - Change Releative Path to the UI
  - List Multiple Swagger Documents
  - Apply swagger-ui Parameters
  - Inject Custom CSS
  - Enable OAuth2.0 Flows

Swashbuckle.AspNetCore.Swagger

Change the Path for Swagger JSON Endpoints

By default, Swagger JSON will be exposed at the following route - "/swagger/{documentName}/swagger.json". If neccessary, you can change this when enabling the Swagger middleware. Custom routes MUST include the {documentName} parameter.

```
app.UseSwagger(c =>
{
    c.RouteTemplate = "api-docs/{documentName}/swagger.json";
});
```

NOTE: If you're using the SwaggerUi middleware, you'll also need to update it's configuration to reflect the new endpoints:

```
app.UseSwaggerUi(c =>
{
    c.SwaggerEndpoint("/api-docs/v1/swagger.json", "My API V1");
})
```

Modify Swagger with Request Context

If you need to set some Swagger metadata based on the current request, you can configure a filter that's executed prior to serializing the document.

```
app.UseSwagger(c =>
{
    c.PreSerializeFilters.Add((swaggerDoc, httpReq) => swaggerDoc.Host = httpReq.Host.Value);
});
```

The *SwaggerDocument* and the current *HttpRequest* are passed to the filter. This provides a lot of flexibilty. For example, you can assign the "host" property (as shown) or you could inspect session information or an Authoriation header and remove operations int the document based on user permissions.

Swashbuckle.AspNetCore.SwaggerGen

List Operation Responses

By default, Swashbuckle will generate a "200" response for each operation. If the action returns a response DTO, then this will be used to generate a "schema" for the response body. For example ...

```
[HttpPost("{id}")]
public Product GetById(int id)
```

Will produce the following response metadata:

```
responses: {
  200: {
    description: "Success",
    schema: {
      $ref: "#/definitions/Product"
    }
  }
}
```

Explicit Responses

If you need to specify a different status code and/or additional responses, or your actions return *IActionResult* instead of a response DTO, you can describe explicit responses with the *ProducesResponseTypeAttribute* that ships with ASP.NET Core. For example ...

```
[HttpPost("{id}")]
[ProducesResponseType(typeof(Product), 200)]
[ProducesResponseType(typeof(IDictionary<string, string>), 400)]
[ProducesResponseType(typeof(void), 500)]
public IActionResult GetById(int id)
```

Will produce the following response metadata:

```
responses: {
  200: {
    description: "Success",
    schema: {
      $ref: "#/definitions/Product"
    }
  },
  400: {
    description: "Bad Request",
    schema: {
      type: "object",
      additionalProperties: {
        type: "string"
      }
    }
  },
  500: {
        description: "Server Error"
      }
    }
```

Include Descriptions from XML Comments

To enhance the generated docs with human-friendly descriptions, you can annotate controllers and models with Xml Comments and configure Swashbuckle to incorporate those comments into the outputted Swagger JSON:

1. Open the Properties dialog for your project, click the "Build" tab and ensure that "XML documentation file" is checked. This will produce a file containing all XML comments at build-time.

   At this point, any classes or methods that are NOT annotated with XML comments will trigger a build warning. To supress this, enter the warning code "1591" into the "Supress warnings" field in the properties dialog.

2. Configure Swashbuckle to incorporate the XML comments on file into the generated Swagger JSON:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1",
        new Info
        {
            Title = "My API - V1",
            Version = "v1"
        }
     );

     var filePath = Path.Combine(PlatformServices.Default.Application.ApplicationBasePath, "MyApi.xml");
     c.IncludeXmlComments(filePath);
}
```

3. Annotate your actions with summary, remarks and response tags

```
/// <summary>
/// Retrieves a specific product by unique id
/// </summary>
/// <remarks>Awesomeness!</remarks>
/// <response code="200">Product created</response>
/// <response code="400">Product has missing/invalid values</response>
/// <response code="500">Oops! Can't create your product right now</response>
[HttpGet("{id}")]
[ProducesResponseType(typeof(Product), 200)]
[ProducesResponseType(typeof(IDictionary<string, string>), 400)]
[ProducesResponseType(typeof(void), 500)]
public Product GetById(int id)
```

4. Rebuild your project to update the XML Comments file and navigate to the Swagger JSON endpoint. Note how the descriptions are mapped onto corresponding Swagger fields.

NOTE: You can also provide Swagger Schema descriptions by annotating your API models and their properties with summary tags. If you have multiple XML comments files (e.g. separate libraries for controllers and models), you can invoke the IncludeXmlComments method multiple times and they will all be merged into the outputted Swagger JSON.

Provide Global API Metadata

In addition to *Paths*, *Operations* and *Responses*, which Swashbuckle generates for you, Swagger also supports global metadata (see http://swagger.io/specification/#swaggerObject). For example, you can provide a full description for your API, terms of service or even contact and licensing information:

```
c.SwaggerDoc("v1",
    new Info
    {
        Title = "My API - V1",
        Version = "v1",
        Description = "A sample API to demo Swashbuckle",
        TermsOfService = "Knock yourself out",
        Contact = new Contact
        {
            Name = "Joe Developer",
                Email = "joe.developer@tempuri.org"
            },
            License = new License
            {
                Name = "Apache 2.0",
                Url = "http://www.apache.org/licenses/LICENSE-2.0.html"
            }
        }
    )
```

Use IntelliSense to see what other fields are available.

Generate Multiple Swagger Documents

With the setup described above, the generator will include all API operations in a single Swagger document. However, you can create multiple documents if necessary. For example, you may want a separate document for each version of your API. To do this, start by defining multiple Swagger docs in *Startup.cs*:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info { Title = "My API - V1", Version = "v1" });
    c.SwaggerDoc("v2", new Info { Title = "My API - V2", Version = "v2" });
})
```

Take note of the first argument to SwaggerDoc. It MUST be a URI-friendly name that uniquely identifies the document. It's subsequently used to make up the path for requesting the corresponding Swagger JSON. For example, with the default routing, the above documents will be available at "/swagger/v1/swagger.json" and "/swagger/v2/swagger.json".

Next, you'll need to inform Swashbuckle which actions to include in each document. Although this can be customized (see below), by default, the generator will use the *ApiDescription.GroupName* property, part of the built-in metadata layer that ships with ASP.NET Core, to make this distinction. You can set this by decorating individual actions OR by applying an application wide convention.

Decorate Individual Actions

To include an action in a specific Swagger document, decorate it with the *ApiExplorerSettingsAttribute* and set *GroupName* to the corresponding document name (case sensitive):

```
[HttpPost]
[ApiExplorerSettings(GroupName = "v2")]
public void Post([FromBody]Product product)
```

Assign Actions to Documents by Convention

To group by convention instead of decorating every action, you can apply a custom controller or action convention. For example, you could wire up the following convention to assign actions to documents based on the controller namespace.

```
// ApiExplorerGroupPerVersionConvention.cs
public class ApiExplorerGroupPerVersionConvention : IControllerModelConvention
{
    public void Apply(ControllerModel controller)
    {
        var controllerNamespace = controller.ControllerType.Namespace; // e.g. "Controllers.V1"
        var apiVersion = controllerNamespace.Split('.').Last().ToLower();

        controller.ApiExplorer.GroupName = apiVersion;
    }
}

// Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(c =>
        c.Conventions.Add(new ApiExplorerGroupPerVersionConvention())
    );

    ...
}
```

Customize the Action Selection Process

When selecting actions for a given Swagger document, the generator invokes a *DocInclusionPredicate* against every *ApiDescription* that's surfaced by the framework. The default implementation inspects *ApiDescription.GroupName* and returns true if the value is either null OR equal to the requested document name. However, you can also provide a custom inclusion predicate. For example, if you're using an attribute-based approach to implement API versioning (e.g. Microsoft.AspNetCore.Mvc.Versioning), you could configure a custom predicate that leverages this instead:

```
c.DocInclusionPredicate((docName, apiDesc) =>
{
    var versions = apiDesc.ControllerAttributes()
        .OfType<ApiVersionAttribute>()
        .SelectMany(attr => attr.Versions);

    return versions.Any(v => $"v{v.ToString()}" == docName);
});
```

Exposing Multiple Documents through the UI

If you're using the *SwaggerUi* middleware, you'll need to specify any additional Swagger endpoints you want to expose. See List Multiple Swagger Documents for more.

Omit Obsolete Operations and/or Schema Properties

The Swagger spec includes a "deprecated" flag for indicating that an operation is deprecated and should be refrained from use. The Swagger generator will automatically set this flag if the corresponding action is decorated with the *ObsoleteAttribute*. However, instead of setting a flag, you can configure the generator to ignore obsolete actions altogether:

```
services.AddSwaggerGen(c =>
{
    ...
    c.IgnoreObsoleteActions();
};
```

A similar approach can also be used to omit obsolete properties from Schema's in the Swagger output. That is, you can decorate model properties with the *ObsoleteAttribute* and configure Swashbuckle to omit those properties when generating JSON Schemas:

```
services.AddSwaggerGen(c =>
{
    ...
    c.IgnoreObsoleteProperties();
};
```

Omit Arbitrary Operations

You can omit operations from the Swagger output by decorating individual actions OR by applying an application wide convention.

Decorate Individual Actions

To omit a specific action, decorate it with the *ApiExplorerSettingsAttribute* and set the *IgnoreApi* flag:

```
[HttpGet("{id}")]
[ApiExplorerSettings(IgnoreApi = true)]
public Product GetById(int id)
```

Omit Actions by Convention

To omit actions by convention instead of decorating them individually, you can apply a custom action convention. For example, you could wire up the following convention to only document GET operations:

```
// ApiExplorerGetsOnlyConvention.cs
public class ApiExplorerGetsOnlyConvention : IActionModelConvention
{
        public void Apply(ActionModel action)
        {
            action.ApiExplorer.IsVisible = action.Attributes.OfType<HttpGetAttribute>().Any();
        }
    }

    // Startup.cs
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc(c =>
            c.Conventions.Add(new ApiExplorerGetsOnlyConvention())
        );

        ...
    }
```

Customize Operation Tags (e.g. for UI Grouping)

The Swagger spec allows one or more "tags" to be assigned to an operation. The Swagger generator will assign the controller name as the default tag. This is particularly interesting if you're using the *SwaggerUi* middleware as it uses this value to group operations.

You can override the default tag by providing a function that applies tags by convention. For example, the following configuration will tag, and therefore group operations in the UI, by HTTP method:

```
services.AddSwaggerGen(c =>
{
    ...
    c.TagActionsBy(api => api.HttpMethod);
};
```

Change Operation Sort Order (e.g. for UI Sorting)

By default, actions are ordered by assigned tag (see above) before they're grouped into the path-based, hierarchical structure imposed by the Swagger spec. You can change this behavior with a custom sorting strategy:

```
services.AddSwaggerGen(c =>
{
    ...
    c.OrderActionsBy((apiDesc) => $"{apiDesc.ControllerName()}_{apiDesc.HttpMethod}");
};
```

NOTE: This dictates the sort order BEFORE actions are grouped and transformed into the Swagger format. So, it affects the ordering of groups (i.e. Swagger PathItems), AND the ordering of operations within a group, in the Swagger output.

Customize Schema Id's

If the generator encounters complex parameter or response types, it will generate a corresponding JSON Schema, add it to the global "definitions" dictionary, and reference it from the operation description by unique Id. For example, if you have an action that returns a "Product" type, the generated schema will be referenced as follows:

```
responses: {
  200: {
    description: "Success",
    schema: {
      $ref: "#/definitions/Product"
    }
  }
}
```

However, if it encounters multiple "Product" classes under different namespaces (e.g. "RequestModels.Product" & "ResponseModels.Product"), then Swashbuckle will raise an exception due to "Conflicting schemaIds". In this case, you'll need to provide a custom Id strategy that further qualifies the name:

```
services.AddSwaggerGen(c =>
{
    ...
    c.CustomSchemaIds((type) => type.FullName);
};
```

Customize Schema for Enum Types

When describing parameters and responses, Swashbuckle does its best to reflect the application's serialization settings. For example, if the *CamelCaseContractResolver* is enabled, Schema property names will be camelCased in the generated Swagger.

Similarly for enum types, if the *StringEnumConverter* is enabled, then the corresponding Schema's will list enum names rather than integer values.

For most cases this should be sufficient. However, if you need more control, Swashbuckle exposes the following options to override the default behavior:

```
services.AddSwaggerGen(c =>
{
    ...
    c.DescribeAllEnumsAsStrings();
    c.DescribeStringEnumsInCamelCase();
};
```

Override Schema for Specific Types

Out-of-the-box, Swashbuckle does a decent job at generating JSON Schema's that accurately describe your request and response payloads. However, if you're customizing serialization behavior for certain types in your API, you may need to help it out.

For example, you might have a class with muliple properties that you want to represent in JSON as a comma-separated string. To do this you would probably implement a custom *JsonConverter*. In this case, Swashbuckle doesn't know how the converter is implemented and so you would need to provide it with a Schema that accurately describes the type:

```
// PhoneNumber.cs
public class PhoneNumber
{
    public string CountryCode { get; set; }

    public string AreaCode { get; set; }

    public string SubscriberId { get; set; }
}

// Startup.cs
services.AddSwaggerGen(c =>
{
    ...
    c.MapType<PhoneNumber>(() => new Schema { Type = "string" });
};
```

Extend Generator with Operation, Schema & Document Filters

Swashbuckle exposes a filter pipeline that hooks into the generation process. Once generated, individual metadata objects are passed into the pipeline where they can be modified further. You can wire up one or more custom filters for *Operation*, *Schema* and *Document* objects:

Operation Filters

Swashbuckle retrieves an *ApiDescription*, part of ASP.NET Core, for every action and uses it to generate a corresponding *Swagger Operation*. Once generated, it passes the *Operation* and the *ApiDescription* through the list of configured Operation Filters.

In a typical filter implementation, you inspect the *ApiDescription* for relevant information (e.g. route info, action attributes etc.) and then update the Swagger *Operation* accordingly. For example, the following filter lists an additional "401" response for all actions that are decorated with the *AuthorizeAttribute*:

```
// AuthResponsesOperationFilter.cs
public class AuthResponsesOperationFilter : IOperationFilter
{
    public void Apply(Operation operation, OperationFilterContext context)
    {
        var authAttributes = context.ApiDescription
            .ControllerAttributes()
            .Union(context.ApiDescription.ActionAttributes())
            .OfType<AuthorizeAttribute>();

        if (authAttributes.Any())
            operation.Responses.Add("401", new Response { Description = "Unauthorized" });
    }
}

// Startup.cs
services.AddSwaggerGen(c =>
{
    ...
    c.OperationFilter<AuthResponsesOperationFilter>();
};
```

NOTE: Filter pipelines are DI-aware. That is, you can create filters with constructor parameters and if the parameter types are registered with the DI framework, they'll be automatically injected when the filters are instantiated

Schema Filters

Swashbuckle generates a Swagger-flavored *JSONSchema* for every parameter, response and property type that's exposed by your controller actions. Once generated, it passes the *Schema* and *Type* through the list of configured Schema Filters.

The example below adds an AutoRest vendor extension (see https://github.com/Azure/autorest/blob/master/docs/extensions/readme.md#x-ms-enum) to inform the AutoRest tool how enums should be modelled when it generates the API client.

```
// AutoRestSchemaFilter.cs
public class AutoRestSchemaFilter : ISchemaFilter
{
    public void Apply(Schema schema, SchemaFilterContext context)
    {
        var typeInfo = context.SystemType.GetTypeInfo();

        if (typeInfo.IsEnum)
        {
            schema.Extensions.Add(
                "x-ms-enum",
                new { name = typeInfo.Name,  modelAsString = true }
            );
        };
    }
}

// Startup.cs
services.AddSwaggerGen(c =>
{
    ...
    c.SchemaFilter<AutoRestSchemaFilter>();
};
```

Document Filters

Once a *Swagger Document* has been generated, it too can be passed through a set of pre-configured *Document* Filters. This gives full control to modify the document however you see fit. To ensure you're still returning valid Swagger JSON, you should have a read through the specification before using this filter type.

The example below provides a description for any tags that are assigned to operations in the document:

```
public class TagDescriptionsDocumentFilter : IDocumentFilter
{
    public void Apply(SwaggerDocument swaggerDoc, DocumentFilterContext context)
    {
        swaggerDoc.Tags = new[] {
            new Tag { Name = "Products", Description = "Browse/manage the product catalog" },
            new Tag { Name = "Orders", Description = "Submit orders" }
        };
        }
    }
```

NOTE: If you're using the SwaggerUi middleware, this filter can be used to display additional descriptions beside each group of Operations.

Add Security Definitions and Requirements

In Swagger, you can describe how your API is secured by defining one or more *Security Scheme's* (e.g basic, api key, oauth etc.) and declaring which of those schemes are applicable globally OR for specific operations. For more details, take a look at the "securityDefinitions" and "security" fields in the Swagger spec.

You can use some of the options described above to include security metadata in the generated *Swagger Document*. The example below adds an OAuth 2.0 definition to the global metadata and a corresponding *Operation Filter* that uses the presence of an *AuthorizeAttribute* to determine which operations the scheme applies to.

```
// Startup.cs
services.AddSwaggerGen(c =>
{
    ...
    // Define the OAuth2.0 scheme that's in use (i.e. Implicit Flow)
    c.AddSecurityDefinition("oauth2", new OAuth2Scheme
    {
        Type = "oauth2",
        Flow = "implicit",
        AuthorizationUrl = "http://petstore.swagger.io/oauth/dialog"
        Scopes = new Dictionary<string, string>
        {
            { "readAccess", "Access read operations" },
            { "writeAccess", "Access write operations" }
        }
    });
    // Assign scope requirements to operations based on AuthorizeAttribute
    c.OperationFilter<SecurityRequirementsOperationFilter>();
};

// SecurityRequirementsOperationFilter.cs
public class SecurityRequirementsOperationFilter : IOperationFilter
{
    public void Apply(Operation operation, OperationFilterContext context)
    {
        // Policy names map to scopes
        var controllerScopes = context.ApiDescription.ControllerAttributes()
            .OfType<AuthorizeAttribute>()
            .Select(attr => attr.Policy);

        var actionScopes = context.ApiDescription.ActionAttributes()
            .OfType<AuthorizeAttribute>()
            .Select(attr => attr.Policy);

        var requiredScopes = controllerScopes.Union(actionScopes).Distinct();

        if (requiredScopes.Any())
        {
            operation.Responses.Add("401", new Response { Description = "Unauthorized" });
            operation.Responses.Add("403", new Response { Description = "Forbidden" });

            operation.Security = new List<IDictionary<string, IEnumerable<string>>>();
            operation.Security.Add(new Dictionary<string, IEnumerable<string>>
            {
                { "oauth2", requiredScopes }
            });
        }
    }
}
```

NOTE: If you're using the SwaggerUi middleware, you can enable interactive OAuth2.0 flows that are powered by the emitted security metadata. See Enabling OAuth2.0 Flows for more details.

Swashbuckle.AspNetCore.SwaggerUi

Change Relative Path to the UI

By default, the Swagger UI will be exposed at "/swagger". If neccessary, you can alter this when enabling the SwaggerUi middleware:

```
app.UseSwaggerUi(c =>
{
    c.RoutePrefix = "api-docs"
    ...
}
```

List Multiple Swagger Documents

When enabling the middleware, you're required to specify one or more Swagger endpoints (fully qualified or relative to the current host) to power the UI. If you provide multiple endpoints, they'll be listed in the top right corner of the page, allowing users to toggle between the different documents. For example, the following configuration could be used to document different versions of an API.

```
app.UseSwaggerUi(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "V1 Docs");
    c.SwaggerEndpoint("/swagger/v2/swagger.json", "V2 Docs");
}
```

Apply swagger-ui Parameters

The swagger-ui ships with it's own set of configuration parameters, all described here https://github.com/swagger-api/swagger-ui#swaggerui. In Swashbuckle, most of these are surfaced through the SwaggerUi middleware options:

```
app.UseSwaggerUi(c =>
{
    c.EnabledValidator();
    c.BooleanValues(new object[] { 0, 1 });
    c.DocExpansion("full");
    c.InjectOnCompleteJavaScript("/swagger-ui/on-complete.js");
    c.InjectOnFailureJavaScript("/swagger-ui/on-failure.js");
    c.SupportedSubmitMethods(new[] { "get", "post", "put", "patch" });
    c.ShowRequestHeaders();
    c.ShowJsonEditor();
});
```

Most of them are self explanatory, mapping back to the corresponding swagger-ui docs. To inject custom JavaScript (i.e. *InjectOnCompleteJavaScript* and *InjectOnFailureJavaScript*), you'll need to add the scripts to your application and provide the relative paths as shown above. In ASP.NET Core, this is easily done by placing your script files in the *wwwroot* folder.

Inject Custom CSS

To tweak the look and feel, you can inject additional CSS stylesheets by adding them to your *wwwroot* folder and specifying the relative paths in the middleware options:

```
app.UseSwaggerUi(c =>
{
    ...
    c.InjectStylesheet("/swagger-ui/custom.css");
}
```

Enable OAuth2.0 Flows

The swagger-ui has built-in support to participate in OAuth2.0 authorization flows. It interacts with authorization and/or token endpoints, as specified in the Swagger JSON, to obtain access tokens for subsequent API calls. See Adding Security Definitions and Requirements for an example of adding OAuth2.0 metadata to the generated Swagger.

If you're Swagger endpoint includes the appropriate security metadata, you can enable the UI interaction as follows:

```
    app.UseSwaggerUi(c =>
    {
        ...
        // Provide client ID, client ID, realm and application name
        c.ConfigureOAuth2("swagger-ui", "swagger-ui-secret", "swagger-ui-realm", "Swagger UI");
    }
```

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About