Swashbuckle.AspNetCore Documentation

Release 5.0.0-beta

domaindrivendev

Mar 16, 2021

Contents

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 The OpenAPI Document . . . 3
- 1.3 Describing Parameters . . . 4
- 1.4 Describing a Request Body . . . 4
- 1.5 Describing Responses . . . 5
- 1.6 Media Types . . . 5
- 1.7 Data Models (Schemas) . . . 6
- 1.8 Providing General API Info . . . 6

2 Serializer Support **9**
- 2.1 System.Text.Json (STJ) . . . 9
- 2.2 Newtonsoft.Json . . . 9

3 Cookbook **11**
- 3.1 Including XML Comments . . . 11
- 3.2 Listing Response Types . . . 12
- 3.3 Forms and File Uploads . . . 13
- 3.4 XML Media Types . . . 14
- 3.5 Describing API Security . . . 14

Swagger/OpenAPI tools for documenting and testing API's build on ASP.NET Core.

CHAPTER 1

Getting Started

1.1 Installation

1. Install the `Swashbuckle.AspNetCore` metapackage into your ASP.NET Core application:

   ```
   > dotnet add package Swashbuckle.AspNetCore
   ```

   or via Package Manager ...

   ```
   > Install-Package Swashbuckle.AspNetCore
   ```

2. In the `ConfigureServices` method of `Startup.cs`, register the Swagger/OpenAPI generator

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

   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/OpenAPI document as a JSON endpoint

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

5. Optionally, if you want to expose interactive documentation, insert the Swagger UI middleware.

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

1.2 The OpenAPI Document

Swashbuckle brings the power of the OpenAPI 3.0 Specification (formerly Swagger Specification) and related tools (Swagger UI, Swagger Codegen, ReDoc etc.) to ASP.NET Core with minimal developer effort.

It does this by inspecting your application code (routes, controllers, models, attributes etc.) to generate an "OpenAPI" document that describes your API. It then exposes that document as a JSON or YAML endpoint that can be consumed by various tools in the Swagger ecosystem, including the Swagger UI and Swagger Codegen client generator.

The following example illustrates the basic structure of an OpenAPI document generated by Swashbuckle:

The `openapi` keyword specifies the *exact* version of the Open Specification that the document is based on - currently v3.0.1.

The `info` section contains general information about your API. With the default setup, Swashbuckle will set the `title` to the name of your Startup DLL and the `version` to "1.0". However, this data can be easily customized and enriched (e.g. adding a description, terms of service etc.) when configuring the Swagger generator. See *Providing General API Info* for more info.

The `paths` section defines the various routes exposed by your API, and the HTTP methods ("operations") supported for those routes. An operation definition includes parameters (if any), a request body (if any), possible response status codes (such as 200 OK or 404 Not Found) and response contents.

The `components` section defines various definitions (e.g. schemas, security schemes etc.) that can be referenced elsewhere in the document. For example, The "CreateUser" operation shown above defines a `requestBody` that references the "User" schema definition rather than defining it inline.

1.3 Describing Parameters

In OpenAPI 3.0, parameters are defined in the `parameters` section of an `operation`. The `in` keyword is used to indicate the type of parameter and can be set to `path`, `query`, `header` or `cookie`. To learn more about how request parameters are described by the OpenAPI Specification, checkout out the OpenAPI docs here.

When generating an `operation` for an action method, Swashbuckle will automatically generate parameter definitions for any parameters or model properties that are bound to the route, query string or headers collection.

For example, given the following action method:

Swashbuckle will generate the following parameters:

1.4 Describing a Request Body

Request bodies are typically used with "create" and "update" operations (POST, PUT, PATCH). For example, when creating a resource using POST or PUT, the request body usually contains the representation of the resource to be created. OpenAPI 3.0 provides the `requestBody` keyword to describe request bodies. To learn more about how request bodies are described by the OpenAPI Specification, checkout out the OpenAPI docs here.

When generating an `operation` for an action method, Swashbuckle will automatically include a `requestBody` if the action has a parameter or model property that is bound to the request body.

For example, given the following action method:

Swashbuckle will generate the following request body:

Note: In addition to the `[FromBody]` attribute, Swashbuckle also supports parameters that are bound to form data via the `[FromForm]` attribute and/or `IFormFile` and `IFormFileCollection` types. See *Forms and File Uploads* for more info.

1.5 Describing Responses

In an OpenAPI document, each operation must have at least one response defined, usually a successful response. A response is defined by its HTTP status code and the data returned in the response body and/or headers. To learn more about how responses are described by the OpenAPI Specification, checkout out the OpenAPI docs here.

By default, Swashbuckle will generate a "200" response for *all* operations. Additionally, if the action returns a response DTO (as a specific type or `ActionResult<T>`) then this will be used to generate a "schema" for the response body.

For example, given the following action method:

Swashbuckle will generate the following responses:

Note: 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 responses explicitly by annotating actions or controllers with the `[ProducesResponseType]` and/or `[ProducesDefaultResponseType]` attributes that ship with ASP.NET Core. See *Listing Response Types* for more info.

1.6 Media Types

For a given API, request and response contents may be transmitted in different *media types* - e.g. `appliction/json`, `application/xml` etc. OpenAPI 3.0 provides the `content` field on request body and response descriptions to list supported media types. To learn more about how media types are described by the OpenAPI Specification, checkout out the OpenAPI docs here.

When generating request and response definitions, Swashbuckle will list supported media types according to the input and output formatters configured for your application. For example, if you're using the `SystemTextJsonInputFormatter`, then Swashbuckle will include a definition for the following media types on request bodies, because these are the media types explicitly supported by that formatter:

- `application/json`
- `text/json`
- `application/*+json`

It's worth noting however, that ASP.NET Core does provide the `[Consumes]` and `[Produces]` attributes, which can be applied at the action or controller level, to further constrain the media types supported for a given operation or group of operations. In this case, Swashbuckle will honor the behavior and only list the explicitly supported media types.

For example, given the following controller:

Swashbuckle will only generate a single `application/json` media type for the relevant request body and response definitions:

Note: If you've configured your application to support XML media types (as described here), then Swashbuckle will automatically list the additional media type. However, support for honoring `XmlSerializer` behavior is currently limited and requires some workarounds to generate accurate schema definitions. See *XML Media Types* for more info.

1.7 Data Models (Schemas)

In OpenAPI 3.0, the data types exposed by an API are described using an extended subset of the JSON Schema Specification Wright Draft 00 (aka Draft 5).

Parameters, request body and response payloads can all be assigned a `schema` instance to describe their data structure. For a given schema instance, the `type` keyword indicates the data type that it represents and can be set to `string`, `number`, `integer`, `boolean`, `array` or `object`. Schemas can be defined inline or they can reference a shared definition from the `components.schemas` section of the OpenAPI document. To learn more about the use of JSON Schema's in the OpenAPI Specification, checkout out the OpenAPI docs here.

When generating `parameters`, a `requestBody`, or `responses` for an operation, Swashbuckle will automatically generate a corresponding schema according to the model type and your application's serialization settings. For simple types (e.g. `string`, `int` etc.), it will define the schema inline whereas for user-defined reference types (e.g. `User`) it will define the schema in the `components.schemas` section of the OpenAPI document and reference the definition via the `$ref` keyword.

For example, consider the following action that accepts a number of parameters from the request query string, and returns an object that will be serialized to the response payload.

For the parameters, Swashbuckle will generate the schemas according to ASP.NET Core's model binding behavior, and for the response it will generate the schema according to the JSON serializer behavior:

Note: By default, Swashbuckle will honor the behavior of the System.Text.Json (STJ) serializer that ships with ASP.NET Core. If you're using the Newtonsoft serializer, then you'll need to install an additional package and explicitly opt-in for Swashbuckle to honor it's behavior instead. See *Serializer Support* for more info.

1.8 Providing General API Info

In an OpenAPI document, the `info` section can be used to provide general information about an API. It includes a `title` and `version`, which are required, and a range of optional fields such as `description`, `termsOfService` etc. To learn more about this section of the OpenAPI document, checkout out the OpenAPI docs here.

With the default setup, Swashbuckle will generate a single OpenAPI document for your API, with the `title` set to to the name of your Startup DLL and the `version` to "1.0". To edit these values and/or provide additional info, you can register the document explicitly, and provide an `OpenApiInfo` instance:

Note: The first parameter to `SwaggerDoc` is a unique name for the document, and is significant because it corresponds to the `{documentName}` parameter in the URL for retrieving OpenAPI documents as JSON or YAML - e.g. `/swagger/{documentName}/swagger.json`. With the default setup, the SwaggerUI middleware assumes an OpenAPI document can be found at `/swagger/v1/swagger.json`. So, if you register the document here with a value other than `v1`, then you'll need to update the SwaggerUI middleware accordingly. See #TODO for more on this.

CHAPTER 2

Serializer Support

To generate accurate schemas for `requestBody` and `response` definitions, Swashbuckle needs to take serializer behavior into account. It currently supports the two most commonly used serializers - System.Text.Json and Newtonsoft.Json.

2.1 System.Text.Json (STJ)

By default, Swashbuckle will generate schemas based on the STJ serializer. This means it will honor the configured serializer behavior (i.e. `JsonSerializerOptions`) and the serialization attributes from the `System.Text.Json.Serialization` namespace. For example, given the following configuration and model classes:

Swashbuckle will generate the following schemas:

2.2 Newtonsoft.Json

If your application uses the Newtonsoft serializer, you should configure Swashbuckle to honor it's behavior instead of STJ. To do this, follow the steps below:

1. Install the `Swashbuckle.AspNetCore.Newtonsoft` package:

   ```
   > dotnet add package Swashbuckle.AspNetCore.Newtonsoft
   ```

   or via Package Manager ...

   ```
   > Install-Package Swashbuckle.AspNetCore.Newtonsoft
   ```

2. Explicitly opt-in by calling `AddSwaggerGenNewtonsoftSupport` in `Startup.cs`

Now, Swashbuckle will honor the configured Newtonsoft behavior (i.e. `JsonSerializerSettings`) and the serialization attributes from the `Newtonsoft.Json` namespace. For example, given the configuration above and the following model classes:

Swashbuckle will generate the following schemas:

CHAPTER 3

Cookbook

3.1 Including XML Comments

The various definitions (operations, parameters, responses, schemas etc.) in an OpenAPI document can include a `description` field, and in the case of operations - an additional `summary` field, to enrich the docs with human readable content.

With Swashbuckle, you can provide these values by annotating actions, classes and properties with a supported subset of XML Comments tags. To enable this feature, follow the steps below:

1. Configure your project to output an XML Comments file at buildtime:

   NOTE: Supressing the "1591" warning is not neccessary here but can be useful if you're using XML Comments for Swashbuckle only.

2. Configure Swashbuckle to incorporate one or more XML Comments file(s) into the generated OpenAPI document:

Now, you can start annotating your actions and properties to include additional descriptive text in the generated document.

3.1.1 Operations, Params & Responses

To enrich operation, parameter and response definitions with human readable descriptions, you can annotate action methods with the familiar `<summary>`, `<remarks>` and `<param>` tags, as well as the Swashbuckle-specific `<response>` tag. For example, given the following action method:

Swashbuckle will generate the following operation:

3.1.2 Schemas & Properties

To enrich schema and property definitions, you can annotate both classes and properties with the `<summary>` tag. For example, given the following class:

Swashbuckle will generate the following schema:

Note: By default, Swashbuckle does NOT include descriptions for "reference" schemas. This is because, as stated in the JSON Schema spec, the `$ref` field should not be accompanied by other fields. You can workaround this by using the `allOf` keyword to "extend" the reference schema. See #TODO for more info.

3.1.3 Global Tags

In OpenAPI, you can assign a list of `tags` to each operation for downstream tools and libraries to use as they see fit. For example, the Swagger UI uses `tags` to group the displayed operations. Additionally, you can specify a `description` for each tag by using the global `tags` section on the root document.

By default, Swashbuckle tags operations with the corresponding controller name but does not include global descriptions for those tags. However, you can add these to the generated document by passing the opt-in flag and decorating controllers with the `<summary>` tag:

Given the following controller:

Swashbuckle will generate the following tags section on the document root:

3.2 Listing Response Types

By default, Swashbuckle will generate a "200" response for every operation. Additionally, if an action returns a response DTO (as a specific type or `ActionResult<T>`), then a corresponding schema will be included with the "200" response definition. If you need to specify different status codes and/or additional responses, or you have actions that return `IActionResult`, you can annotate individual actions with explicit response attributes or you can apply application-wide *API Conventions*.

3.2.1 Explicit Responses

To specify explicit responses, you can decorate actions (or controllers) with the `[ProducesResponseType]` and `[ProducesDefaultResponseType]` attributes that come with ASP.NET Core. For example, given the following action method:

Swashbuckle will generate the following responses:

Note: With this approach, you have to include an attribute for *all* respones, including the "2xx" response. In other words, you can use the default responses generated by Swashbuckle or you can provide explicit responses, but you can't use a mixture of both.

3.2.2 API Conventions

Web API conventions, available in ASP.NET Core 2.2 and later, include a way to extract common documentation conventions and apply them to multiple actions, controllers, or all controllers within an assembly. They function as a substitute for decorating individual actions with explicit responses, as shown above. For example, if all actions with the word "Create" in their name ought to return a HTTP status of `201 Created` or `400 Bad Request`, you could codify that convention in a C# class and apply it for documentation purposes. In fact, this exact convention is captured in the `DefaultApiConventions` class provided with ASP.NET Core.

To apply the default conventions (or your own custom type), you can decorate controllers with the `[ApiConventionType]` attribute:

In this case, Swashbuckle will generate the following responses:

3.3 Forms and File Uploads

3.3.1 Form data

For example, given the following action method:

Swashbuckle will generate the following request body:

3.3.2 File Uploads

For example, given the following action method:

Swashbuckle will generate the following request body:

3.3.3 Multipart Requests

For example, given the following action method:

Swashbuckle will generate the following request body:

3.4 XML Media Types

3.5 Describing API Security

In OpenAPI 3.0, you can describe how an API is secured by defining one or more security schemes (e.g. basic, API key, OAuth2 etc.) and then specifying which of those schemes are required, either globally or for specific operations. To learn more about describing security in an OpenAPI document, checkout out the OpenAPI docs here.

In Swashbuckle, you can define schemes by invoking the `AddSecurityDefinition` method, providing a name and an instance of `OpenApiSecurityScheme` that describes the scheme. If the scheme is applicable to all operations, you can invoke the `AddSecurityRequirements` method, or alternatively you can wire up an operation filter that applies the scheme to specific operations based on the presence of `[Authorize]` attributes on controllers and action methods.

3.5.1 API Key Example

This example is for an API that requires a valid API key to be provided in the query string for *all* operations. You can describe this scheme with Swashbuckle as follows:

With this setup, Swashbuckle will generate the following scheme definition and (global) requirement:

Note: If you're using the Swagger UI, it will display authentication popups that can be used in conjuction with the "Try it out" functionality, based on the security schemes and requirements specified in the OpenAPI document.

3.5.2 OAuth2 Example

This example is for an API that accepts an OAuth2 access token, obtained via the Authorization Code flow , and authorizes operations based on scopes included with the token. To describe this with Swashbuckle, you can define an `OAuth2` scheme, and wire up an operation filter that applies the scheme to specific operations based on the presence of `[Authorize]` attributes:

The filter implementation will depend on how you've implemented authorization within your app. For example, if you're using named policies to encapsulate and apply scope requirements with the `[Authorize]` attribute, then you could implement the filter as follows:

With this setup, Swashbuckle will generate the following scheme definition:

And for any operations that have the custom policies applied . . .

It will generate the following operation metadata, including a `security` section that references the "oauth2" scheme and scopes required by the policy:

Note: While this setup is a bit more involved, it's extremely powerful, especially when combined with the Swagger UI, as it has built-in support for interactive OAuth2 flows.

If your OpenAPI document includes OAuth2 definitions and requirements, the interactive flow(s) will be enabled automatically. However, you can further customize OAuth2 support in the UI with the following settings. See the swagger-ui docs here for more info: