# Integrating Swagger UI

We can use the Swashbuckle package to easily integrate Swagger into our .NET Core Web API projects. It will generate the Swagger specification for our project. Additionally, the Swagger UI is also contained within Swashbuckle.

There are three main components in the Swashbuckle package:

**Swashbuckle.AspNetCore.Swagger:** This contains the Swagger object model and the middleware to expose SwaggerDocument objects as JSON.

**Swashbuckle.AspNetCore.SwaggerGen:** A Swagger generator that builds SwaggerDocument objects directly from our routes, controllers, and models.

**Swashbuckle.AspNetCore.SwaggerUI**: An embedded version of the Swagger UI tool. It interprets Swagger JSON to build a rich, customizable experience for describing the web API functionality.

The next step is to configure the Swagger Middleware.

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

**{**

// Register the Swagger generator, defining 1 or more Swagger documents

services.AddSwaggerGen**(**c =**>**

**{**

c.SwaggerDoc**(**"v1", new OpenApiInfo **{** Title = "My API", Version = "v1" **})**;

**})**;

services.AddControllers**()**;

**}**

 let’s enable the middleware for serving the generated JSON document and the Swagger UI:

public void Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

// Enable middleware to serve generated Swagger as a JSON endpoint.

app.UseSwagger**()**;

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),

// specifying the Swagger JSON endpoint.

app.UseSwaggerUI**(**c =**>**

**{**

c.SwaggerEndpoint**(**"/swagger/v1/swagger.json", "My API V1"**)**;

**})**;

**}**

By executing these steps, the Swagger is configured and ready for use in our project.

**Exploring the Swagger UI**

First, we are going to create an **Employee** class:

public class Employee

**{**

public int Id **{** get; set; **}**

public string FirstName **{** get; set; **}**

public string LastName **{** get; set; **}**

public string EmailId **{** get; set; **}**

**}**

Next, let’s create an API controller with action methods. We are going to use some mock data and keep the focus on understanding Swagger’s capabilities:

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

**[**ApiController**]**

public class EmployeeController : ControllerBase

**{**

// GET: api/Employee

**[**HttpGet**]**

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

**{**

return GetEmployees**()**;

**}**

// GET: api/Employee/5

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

public Employee Get**(**int id**)**

**{**

return GetEmployees**()**.Find**(**e =**>** e.Id == id**)**;

**}**

// POST: api/Employee

**[**HttpPost**]**

**[**Produces**(**"application/json"**)]**

public Employee Post**([**FromBody**]** Employee employee**)**

**{**

// Logic to create new Employee

return new Employee**()**;

**}**

// PUT: api/Employee/5

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

public void Put**(**int id, **[**FromBody**]** Employee employee**)**

**{**

// Logic to update an Employee

**}**

// DELETE: api/Employee/5

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

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

**{**

**}**

private List**<**Employee**>** GetEmployees**()**

**{**

return new List**<**Employee**>()**

**{**

new Employee**()**

**{**

Id = 1,

FirstName= "John",

LastName = "Smith",

EmailId ="John.Smith@gmail.com"

**}**,

new Employee**()**

**{**

Id = 2,

FirstName= "Jane",

LastName = "Doe",

EmailId ="Jane.Doe@gmail.com"

**}**

**}**;

**}**

**}**

Now, let’s run the app and navigate to https://localhost:<port>/swagger/v1/swagger.json. We can see that a document describing the endpoints is generated:

**Extending the documentation**

Let’s look at the various options to extend the documentation.

**API Info & Description**

First, let’s see how we can specify the API info and description.

The configuration action passed to the AddSwaggerGen() method adds information such as Contact, License, and Description. Let’s provide some values for those:

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

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

**{**

// Register the Swagger generator, defining 1 or more Swagger documents

services.AddSwaggerGen**(**c =**>**

**{**

c.SwaggerDoc**(**"v1", new OpenApiInfo

**{**

Title = "Employee API",

Version = "v1",

Description = "An API to perform Employee operations",

TermsOfService = new Uri**(**"https://example.com/terms"**)**,

Contact = new OpenApiContact

**{**

Name = "John Walkner",

Email = "John.Walkner@gmail.com",

Url = new Uri**(**"https://twitter.com/jwalkner"**)**,

**}**,

License = new OpenApiLicense

**{**

Name = "Employee API LICX",

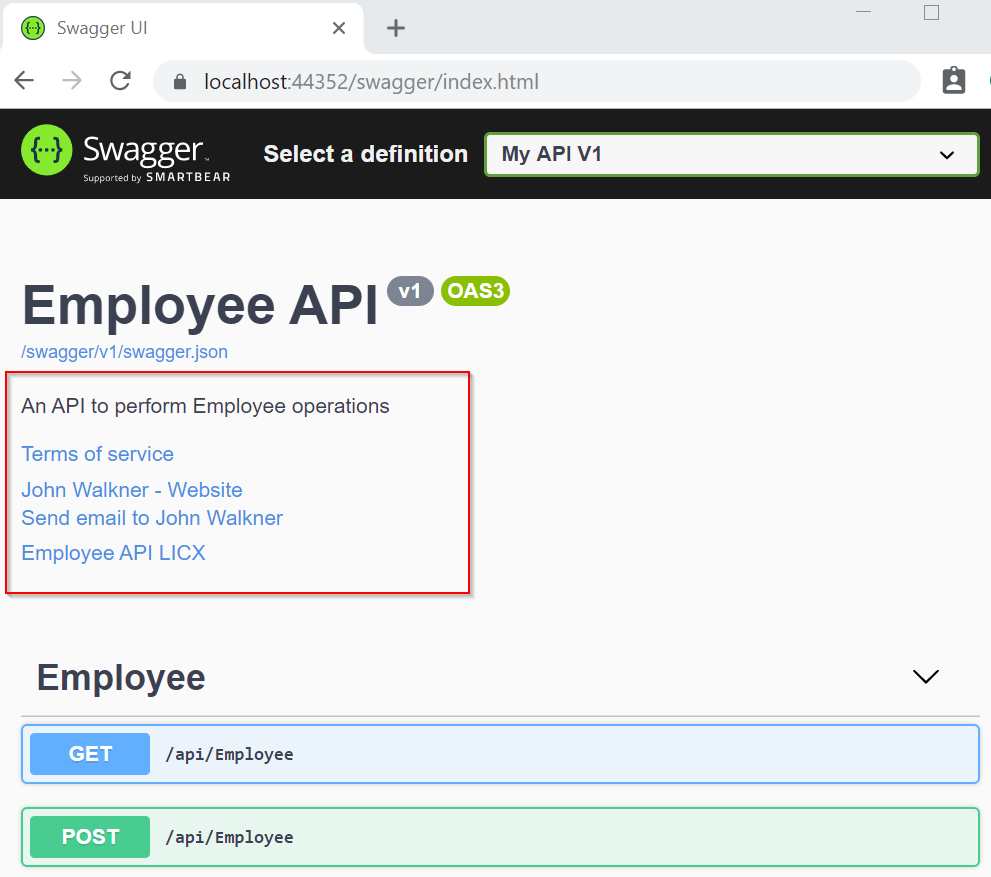
Url = new Uri**(**"https://example.com/license"**)**,

**}**

**})**;

**})**;

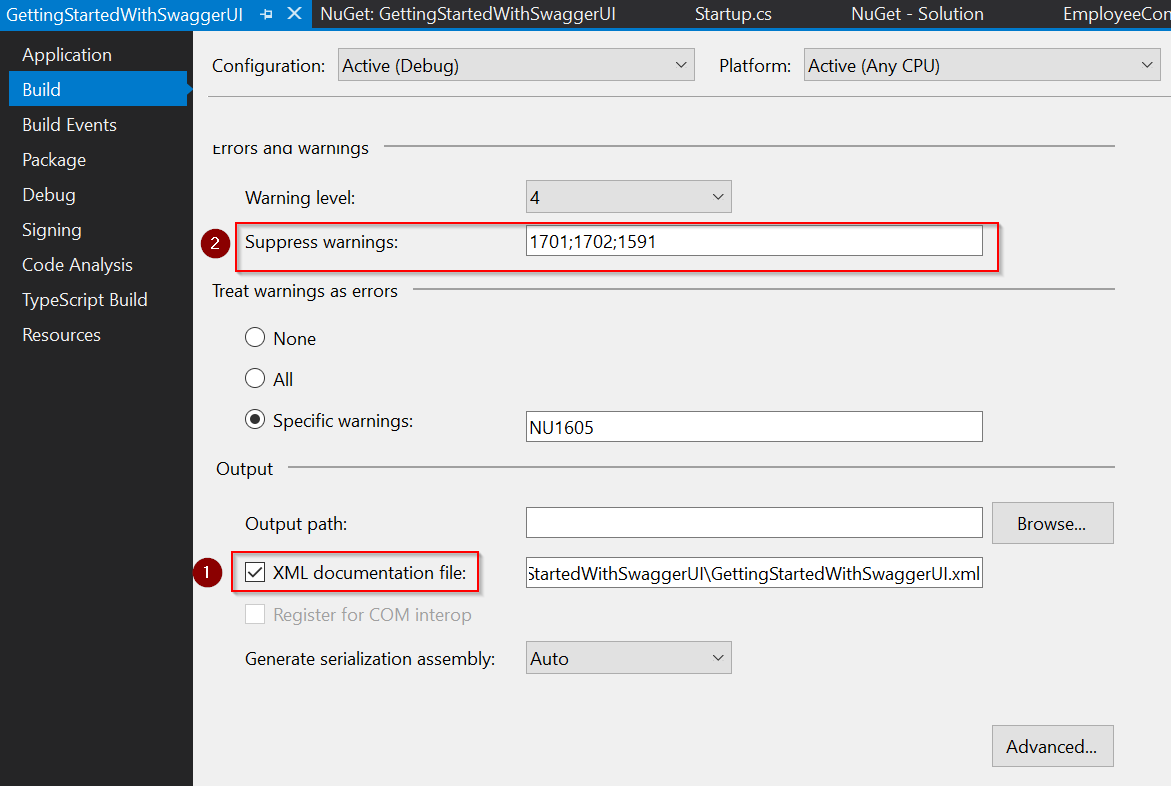
**}**



**XML Comments**

For enabling XML comments, we need to do the following steps:

1. In the Build tab of the project properties, check the box labeled XML documentation file. Let’s keep the auto-generated file path.
2. Suppress warning 1591, which will now give warnings about any method, class, or field that doesn’t have triple-slash comments.



In the ConfigureServices() method, configure Swagger to use the XML file that’s generated in the above step:

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

**{**

// Register the Swagger generator, defining 1 or more Swagger documents

services.AddSwaggerGen**(**c =**>**

**{**

c.SwaggerDoc**(**"v1", new OpenApiInfo

**{**

Title = "Employee API",

Version = "v1",

Description = "An API to perform Employee operations",

TermsOfService = new Uri**(**"https://example.com/terms"**)**,

Contact = new OpenApiContact

**{**

Name = "John Walkner",

Email = "John.Walkner@gmail.com",

Url = new Uri**(**"https://twitter.com/jwalkner"**)**,

**}**,

License = new OpenApiLicense

**{**

Name = "Employee API LICX",

Url = new Uri**(**"https://example.com/license"**)**,

**}**

**})**;

// Set the comments path for the Swagger JSON and UI.

var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";

var xmlPath = Path.Combine**(**AppContext.BaseDirectory, xmlFile**)**;

c.IncludeXmlComments**(**xmlPath**)**;

**})**;

services.AddControllers**()**;

**}**

Now, adding triple-slash comments to the action method enhances the Swagger UI by adding a description to the section header.

Let’s add a summary:

/// <summary>

/// Gets the list of all Employees.

/// </summary>

/// <returns>The list of Employees.</returns>

// GET: api/Employee

**[**HttpGet**]**

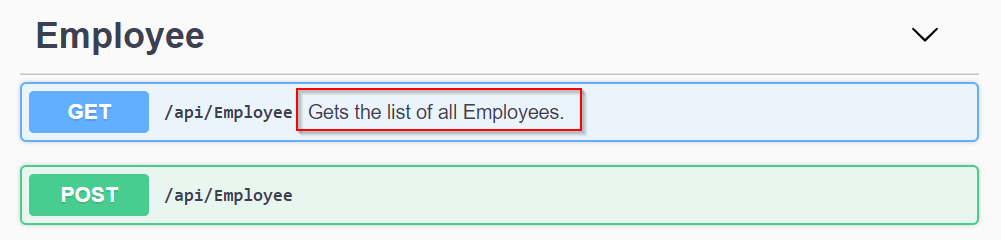
public IEnumerable**<**Employee**>** Get**()**

**{**

return GetEmployees**()**;

**}**

This displays the summary against the action method:



We can additionally add <remarks> element to the documentation. It supplements information specified in the <summary> element and provides a more robust Swagger UI. The <remarks> element content can consist of text, JSON, or XML:

/// <summary>

/// Creates an Employee.

/// </summary>

/// <remarks>

/// Sample request:

///

/// POST api/Employee

/// {

/// "firstName": "Mike",

/// "lastName": "Andrew",

/// "emailId": "Mike.Andrew@gmail.com"

/// }

/// </remarks>

/// <param name="employee"></param>

**[**HttpPost**]**

**[**Produces**(**"application/json"**)]**

public Employee Post**([**FromBody**]** Employee employee**)**

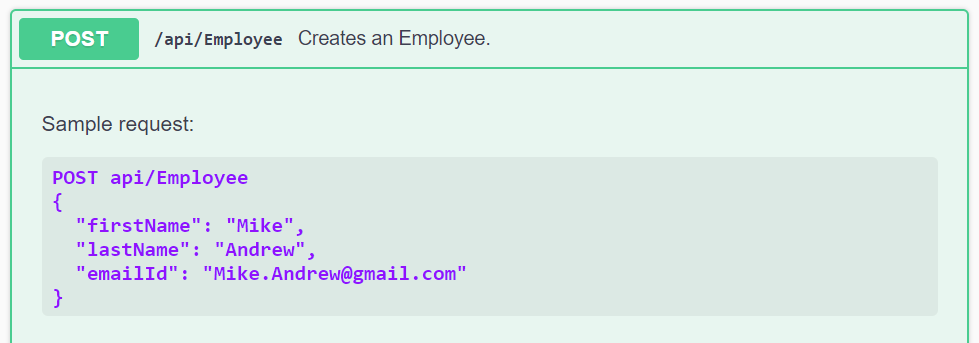
**{**

// Logic to create new Employee

return new Employee**()**;

**}**

This will enhance the UI with additional info:



**Using Data Annotations**

We can decorate a model with attributes to enhance the documentation.

Let’s add a [Required] attribute to the EmailId field of the Employee model:

public class Employee

**{**

public int Id **{** get; set; **}**

public string FirstName **{** get; set; **}**

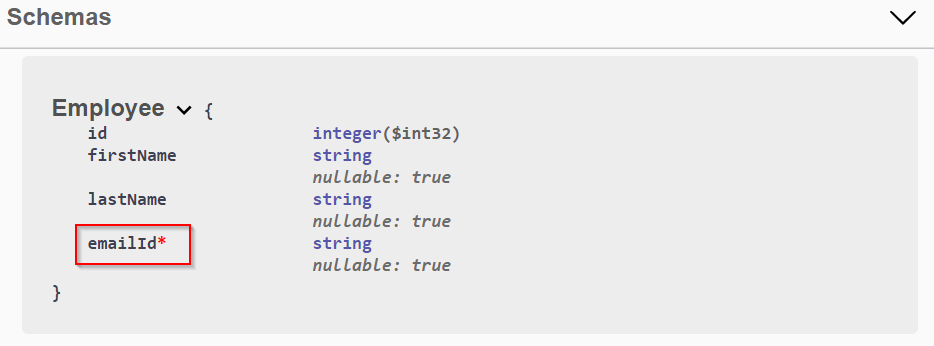
public string LastName **{** get; set; **}**

**[**Required**]**

public string EmailId **{** get; set; **}**

**}**

As a result, the Swagger UI will accept this change:



**Describing Response Types**

The developers who consume our APIs are usually more interested in what it returns- specifically the response types and error codes. Hence it is very important to describe our response types.  These are denoted using XML comments & data annotations.

/// <summary>

/// Creates an Employee.

/// </summary>

/// <remarks>

/// Sample request:

///

/// POST api/Employee

/// {

/// "firstName": "Mike",

/// "lastName": "Andrew",

/// "emailId": "Mike.Andrew@gmail.com"

/// }

/// </remarks>

/// <param name="employee"></param>

/// <returns>A newly created employee</returns>

/// <response code="201">Returns the newly created item</response>

/// <response code="400">If the item is null</response>

**[**HttpPost**]**

**[**ProducesResponseType**(**201**)]**

**[**ProducesResponseType**(**400**)]**

**[**Produces**(**"application/json"**)]**

public Employee Post**([**FromBody**]** Employee employee**)**

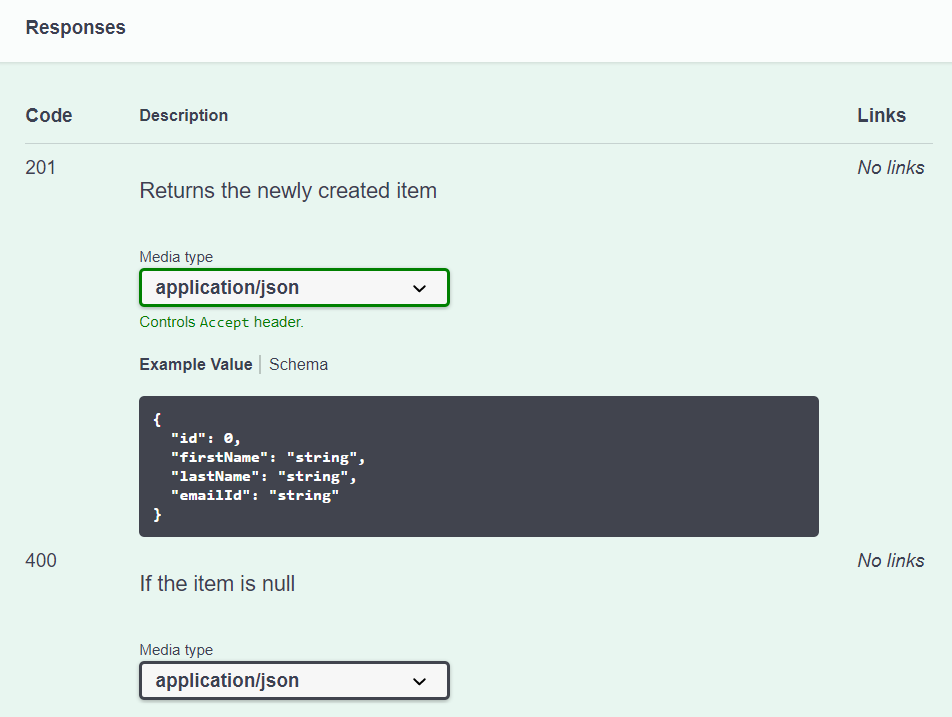
**{**

// Logic to create new Employee

return new Employee**()**;

**}**

This will reflect in the Responses section:



**Customizing the UI**

The default UI of Swagger is pretty good. But we can customize it If we wish to do so. We may change the documentation pages to represent our brand or theme. Branding the Swagger components requires adding the resources to serve static files and building the folder structure to host those files.

First of all, we are going to enable static file middleware in the Configure() method in the Startup.cs file:

**app.UseStaticFiles();**

**All the static assets must be stored within the wwwroot folder**. It does not exist by default, so you have to create it manually. Click on the API project, add a new folder, and name it “wwwroot”. Since it’s a special folder, by default Visual Studio will show it with a special icon (it’s a sort of blue world, similar to 🌐).

I’ve created a single CSS file under */wwwroot/assets/css/xmas-style.css*. Of course, name it as you wish - as long as it is within the wwwroot folder, it’s fine.

My CSS file is quite minimal:

|  |
| --- |
| body {  background-image: url("../images/snowflakes.webp");  }  div.topbar {  background-color: #34a65f !important;  }  h2,  h3 {  color: #f5624d !important;  }  .opblock-summary-get > button > span.opblock-summary-method {  background-color: #235e6f !important;  }  .opblock-summary-post > button > span.opblock-summary-method {  background-color: #0f8a5f !important;  }  .opblock-summary-delete > button > span.opblock-summary-method {  background-color: #cc231e !important;  } |

There are 3 main things to notice:

1. the element selectors are taken directly from the Swagger UI - you’ll need a bit of reverse-engineering skills: just open the Browser Console and find the elements you want to update;
2. unless the element does not already have the rule you want to apply, you have to add the !important CSS operator. Otherwise, your code won’t affect the UI;
3. you can add assets from other folders: I’ve added background-image: url("../images/snowflakes.webp"); to the body style. That image is, as you can imagine, under the wwwroot folder we created before.

you have to update the UseSwaggerUI command within the Main method:

|  |
| --- |
| app.UseSwaggerUI(c =>  c.InjectStylesheet("/assets/css/xmas-style.css")  ); |