<https://www.apriorit.com/dev-blog/788-web-generate-client-code-for-angular-services-with-swagger#6>

Visual Studio 22

Install .Net 6.

New Project> Choose .Net Core with Angular

Create Project

Let’s start with installing the NSwag library to get access to the functionality of the Swagger framework.

**1. Install the NSwag library**

Before you start using Swagger to generate client code, you need to install the following packages:

* NSwag.AspNetCore, NSwag.MSBuild,
* Microsoft.AspNetCore.Mvc.NewtonsoftJson
* NSwag.AspNetCore for documentation and UI representation
* NSwag.MSBuild for code generation
* Microsoft.AspNetCore.Mvc.NewtonsoftJson for JSON serialization

The .csproj file should contain values that look like the following:

|  |
| --- |
| <ItemGroup>  <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="6.0.5" />  <PackageReference Include="NSwag.AspNetCore" Version="13.16.1" />  <PackageReference Include="NSwag.MSBuild" Version="13.16.1">  <PrivateAssets>all</PrivateAssets>  <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>  </PackageReference>  </ItemGroup> |

**2. Modify the Startup.cs file**

To make NSwag work after installation, you need to add a few lines of code to the Startup.cs file. Here’s an example of the configuration:

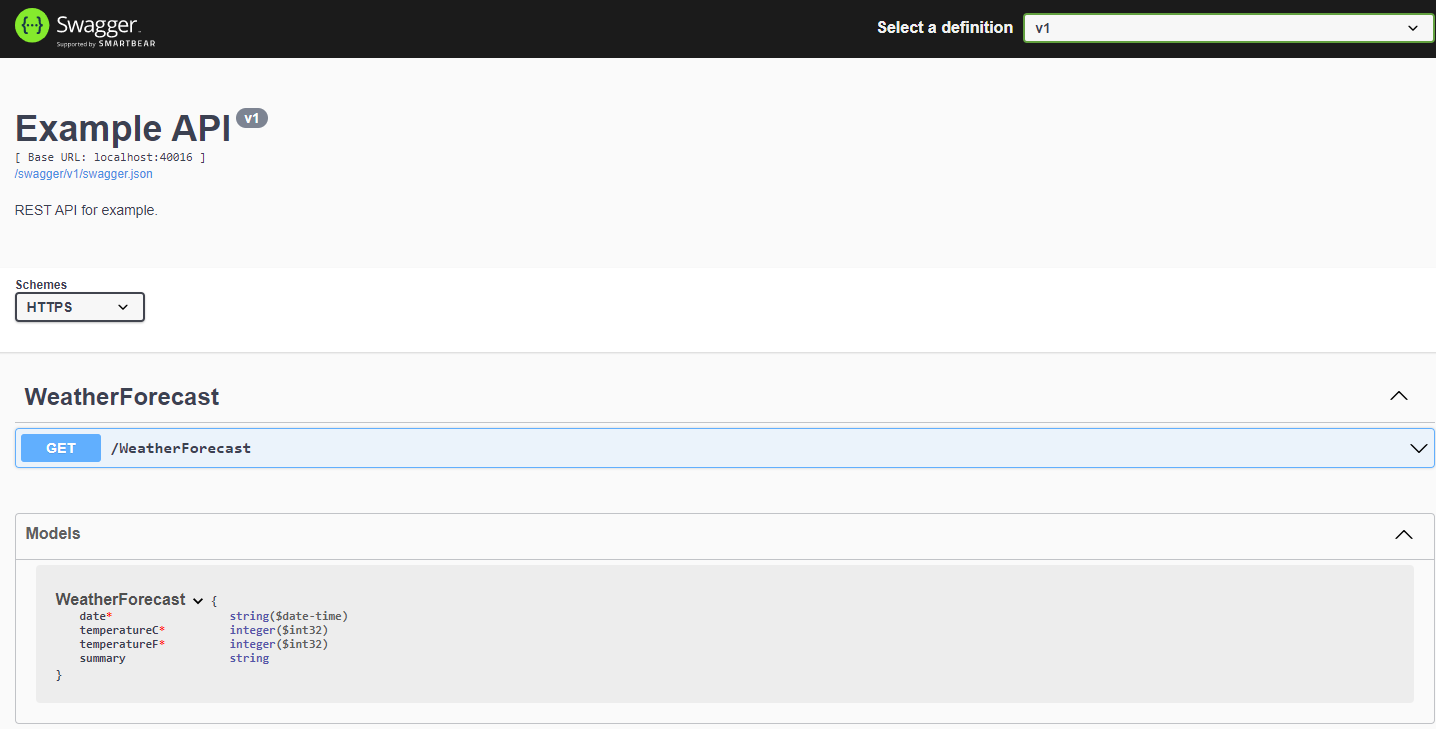
|  |
| --- |
| builder.Services.AddControllersWithViews().AddNewtonsoftJson();  builder.Services.AddSwaggerDocument(settings =>  {  settings.PostProcess = document =>  {  document.Info.Version = "v1";  document.Info.Title = "Example API";  document.Info.Description = "REST API for example.";  };  }); |

And here’s the middleware:

|  |
| --- |
| app.UseOpenApi();  app.UseSwaggerUi3(); |

Now you can see all the project’s endpoints, parameters, models, etc. You also can test requests using the UI by sending requests and retrieving responses.

Now you can see all the project’s endpoints, parameters, models, etc. You also can test requests using the UI by sending requests and retrieving responses.



**3. Create a configuration file**

The next step for generating Angular service code using Swagger is creating the nswag.json file, which is responsible for configuring the code generation process.

The configuration file contains settings that define how exactly code will be generated. For example, it specifies:

* What language to generate code in
* Which language version to use
* What data types to use
* Whether to generate classes, interfaces, etc.

You can find the full template for creating the nswag.json file on the [NSwag Configuration Document](https://github.com/RicoSuter/NSwag/wiki/NSwag-Configuration-Document) GitHub page.

Here’s part of our file:

|  |
| --- |
| {    "runtime": "Net60",  "defaultVariables": null,  "documentGenerator": {  "aspNetCoreToOpenApi": {    ...  }  },  "codeGenerators": {  "openApiToTypeScriptClient": {  ...  "template": "Angular",  "promiseType": "Promise",  "httpClass": "HttpClient",  ...  "injectionTokenType": "InjectionToken",  "rxJsVersion": 6.0,  "dateTimeType": "Luxon",    ...  "output": "./src/app/services/Core-api-v1.ts"  }  }  } |

Instead of creating the file manually, you can also do it automatically using [NSwagStudio](https://github.com/RicoSuter/NSwag/wiki/NSwagStudio). This tool helps to correctly configure all code generation rules in the nswag.json file. Here’s what the program’s interface looks like:

Graphical user interface, application

Description automatically generated  
Screenshot 2. NSwagStudio interface

To create a configuration file in NSwagStudio, you need to perform the following steps:

1. Add the project assembly to the *Input* section in the .NET Assembly tab.
2. In the *Outputs* section, choose the clients you want to generate and configure the settings you need.
3. Click *Generate File* to generate a client file.
4. Click *File → Save As*, type nswag.json as the file name, and choose a path.

Then, NSwagStudio will create a configuration for you.

**4. Add the build task**

By now, you’ve almost configured automated code generation. The last step is to add the Build task to the AfterTargets attribute. To do that, modify your .csproj file by adding a new task to it, as shown below:

|  |
| --- |
| <Target Name="NSwag" AfterTargets="Build">      <Exec Command="$(NSwagExe\_Net60) run ./ClientApp/nswag.json"/>  </Target> |

That’s all. If you’ve done all the steps correctly, you’ll get code that you can use in an Angular service with all application endpoints saved in a new file stored at the path selected in the output option of the nswag.json file.

After that, everything works automatically. Whenever you add a new endpoint to your API, you only have to add it to the back end and build the project. After that, Swagger will automatically generate endpoint specifications, so the code will be ready for you to use.

## Conclusion

The more complex a web application is, the more APIs it requires. And ensuring proper documentation for each new endpoint can be extremely time-consuming. Swagger is a helpful framework that allows you to accelerate client code generation, saving resources for focusing on the bigger picture.

At Apriorit, we’ve built a strong team of [experienced web developers](https://www.apriorit.com/competences/web-app-development) who are passionate about creating reliable and secure solutions. We know how to efficiently organize solutions development and optimize processes to deliver final products in an agreed timeline, with their quality and security in mind.