API DEVELOPMENT PATH

WEB API DOCUMENTATION CHEAT SHEET

v1.0

A downloadable resource of the *Hands-on .NET Web API Documentation with Swagger/OpenAPI* course

---

WWW.CODEWITHPRAVEEN.COM

.NET Web API Documentation Cheat Sheet

(This is a downloadable resource of **Hands-on .NET Web API Documentation with Swagger/OpenAPI** course)

www.CodeWithPraveen.com

Preface

Hi there!

I hope you are doing well.

I've created this ASP.NET Core Web API Documentation cheat sheet to help you with your everyday programming in C#. You can **use this as a reference document** to check the list of steps to be followed to document a Web API from scratch. You can find the syntax for commands as well as code snippets in this guide.

ASP.NET Core Web API is one of the hot topics in the ASP.NET world. **This downloadable resource is part of the *Hands-on .NET Web API Documentation with Swagger/OpenAPI* course**, that covers the essential steps to document a Web API using various approaches. Check out my website for more information.

View the companion course videos by clicking the link below:
Hands-on .NET Web API Documentation with Swagger/OpenAPI

See you in the course video!
Praveen.

Table of Contents

Creating OpenAPI using Swashbuckle

Step 1: Install Swashbuckle

1. **Install the latest Swashbuckle.AspNetCore NuGet package** to your Web API project. **Click Restore** when the IDE prompts.

   ```
   <PackageReference Include="Swashbuckle.AspNetCore" Version="6.1.4"/>
   ```

   This package includes the following 3 packages:

   a. Swashbuckle.AspNetCore.Swagger - Middleware to EXPOSE the API's built on AspNetCore
   b. Swashbuckle.AspNetCore.SwaggerGen - OpenAPI GENERATOR for API's built on AspNetCore
   c. Swashbuckle.AspNetCore.SwaggerUI - Middleware to expose an embedded version of SwaggerUI for AspNetCore application.

Step 2: Configure Swashbuckle

1. First, **add services.AddSwaggerGen() to the service collection** to register the swagger generator with the service collection.

   ```
   services.AddSwaggerGen();
   ```

2. Second, **add app.UseSwagger() to the request pipeline to include swagger middleware.** This is done to serve generated Swagger as a JSON endpoint.

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

Step 3: Add SwaggerUI

SwaggerUI shows the OpenAPI specification in a neat readable format in the web page.

1. **Add app.UseSwaggerUI() middleware to the request pipeline** to add SwaggerUI to your Web API project. **Pass the Swagger Endpoint** specifying the end point of the swagger documentation that you had created before.

   ```
   app.UseSwaggerUI(c =>
   {
         c.SwaggerEndpoint("/swagger/v1/swagger.json", "v1");
   ```

```
});
```

2. **Start the application and navigate to "../swagger/index.html"** to see the Swagger UI loaded. For example,

   https://localhost:5001/swagger/index.html

3. **Set the RoutePrefix to an empty string** to remove the swagger prefix in the URL. Eg. https://localhost:5001

   ```
   c.RoutePrefix = string.Empty;
   ```

Adding Documentation using Swashbuckle

Step 4: Add OpenAPI Metadata

It is recommended to add API metadata to provide additional details for the user.

1. **Include more details about API** such as title, description, version, contact, terms, etc in the Startup.cs file. For this, **use SwaggerDoc inside AddSwaggerGen()** to pass the API details.

```
services.AddSwaggerGen(s =>
{
    // Define and customize a Swagger document.
    s.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "CMS OpenAPI",
        Description = "OpenAPI Specification for CMS",
        Contact = new OpenApiContact
        {
            Name = "Praveenkumar Bouna",
            Email = "hello@codewithpraveen.com"
        },
        License = new OpenApiLicense
        {
            Name = "MIT",
            Url = new Uri("https://opensource.org/licenses/MIT")
        }
```

```
    });
});
```

2. **Add the Microsoft.OpenApi.Models namespace**.

```
using Microsoft.OpenApi.Models;
```

Step 5: Enable Documentation using XML Comments

Include more details about the API such as its purpose, parameter types, response, etc using XML comments.

1. First, **add the GenerateDocumentationFile setting to generate the XML document** in the .csproj file

```
<PropertyGroup>
       <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

2. Second, **add IncludeXmlComments() inside AddSwaggerGen() to include the XML file** to the service collection. You need to pass the XML Document path.

```
services.AddSwaggerGen(s =>
{
    s.SwaggerDoc("v1", new OpenApiInfo
    {
      ...
      // Add XML comments to documentation.
      var xmlPath = Path.Combine(System.AppContext.BaseDirectory,
      "Cms.WebApi.xml");
      s.IncludeXmlComments(xmlPath);
    });
});
```

Step 6: Add Documentation using XML Comments

1. **Add Summary for action methods using <summary> tag**.

   For eg.,

```
  /// <summary>
```

```
/// Get courses from the CMS system
/// </summary>
[HttpGet]
public ActionResult<IEnumerable<CourseDto>> GetAllCourses()
{
    // ...
}
```

2. **Add additional details for action methods using <remarks> tag.**

   For eg.,

```
 /// <remarks>
/// This API should be used to fetch all courses from the system.
/// </remarks>
/// <returns></returns>
[HttpGet]
public ActionResult<IEnumerable<CourseDto>> GetAllCourses()
{
    // ...
}
```

3. **Add response types using the [Produces] attribute.** It can be added at an individual action level as well at the controller class level.

   For eg.,

```
[Produces("application/json")]
public class CoursesController : ControllerBase
{
  // ...
}
```

4. **Add response codes using <response> tag or [ProducesResponseType] attribute.**

   For eg.,

```
 /// <response code="200">Success</response>
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
[HttpGet]
```

```
public ActionResult<IEnumerable<CourseDto>> GetAllCourses()
{
    // ...
}
```

Step 7: Add Data Annotations

1. **Add summary for model class and its members.** Now, the UI document will include these comments for models in both API as well as Models section.

   For eg.,

   ```
   /// <summary>
   /// Course type
   /// </summary>
   public class CourseDto
   {
       /// <summary>
       /// Unique ID of the system.
       /// </summary>
       public int CourseId { get; set; }
   }
   ```

2. **Add System.ComponentModel.DataAnnotations namespace** reference to add annotations.
3. **Add data annotations for class members of the model class,** if required.

   For eg.,

   ```
   [Required]
   [MaxLength(50)]
   public string CourseName { get; set; }

   [Required]
   [Range(1, 5)]
   public int CourseDuration { get; set; }
   ```

Handle Compiler Warnings

You can follow any of the below approaches to handle the compiler warnings that are shown when you enable XML documentation.

1. First, you can **add the missing XML comments** for each action and model.
2. Second, you can **suppress the warning at the file level using #pramga directives**.

```
#pragma warning disable CS1591
// ...
#pragma warning restore CS1591
```

3. Third, you can **suppress warnings at the entire assembly level by setting the NoWarn flag** in the .csproj file.

```
<PropertyGroup>
  ...
  <NoWarn>1591</NoWarn>
  ...
</PropertyGroup>
```

Creating OpenAPI using NSwag

Install NSwag

1. **Install the package NSwag.AspNetCore to add NSwag to your Web API project**. I recommend using the *NuGet Package Manager* extension to easily install the packages.
2. Check the package entry in the .csproj file. Note that if you are going to use Swashbuckle later, you need to **comment out Swashbuckle package entry to avoid ambiguity**.

```
<ItemGroup>
  <!-- <PackageReference Include="Swashbuckle.AspNetCore"
Version="6.1.4"/> →
  ...
  <PackageReference Include="NSwag.AspNetCore" Version="13.12.1"/>
</ItemGroup>
```

Configure NSwag

1. **Register NSwag to the service collection** using AddOpenApiDocument() command.

   ```
   services.AddOpenApiDocument();
   ```

2. Second, **include the details of the swagger document** such as its name, title, version, and so on.

   Use DocumentName to mention the name of the document.

   Use PostProcess inside AddOpenApiDocument() to pass the API details.

   ```
   services.AddOpenApiDocument(c =>
         {
             c.DocumentName = "v1";
             c.PostProcess = doc =>
             {
                 doc.Info.Version = "v1";
                 doc.Info.Title = "CMS OpenAPI";
                 doc.Info.Description = "OpenAPI Specification for
                         CMS";
                 doc.Info.License = new NSwag.OpenApiLicense()
                 {
                     Name = "MIT",
                 };
                 doc.Info.Contact = new NSwag.OpenApiContact()
                 {
                     Name = "Praveenkumar Bouna",
                     Email = "hello@codewithpraveen.com"
                 };
             };
         });
   ```

3. Third, **add swagger middleware to the request pipeline by calling UseOpenApi()** in the Configure method. This is done to serve generated Swagger as a JSON endpoint.

   ```
   app.UseOpenApi();
   ```

Add SwaggerUI

1. **Add SwaggerUI to the request pipeline using UseSwaggerUI3** command. Note that the **usage of UseSwaggerUI is deprecated**.

   ```
   app.UseSwaggerUi3();
   ```

2. **Access the Swagger UI** at "../swagger/index.html".

   Eg. https://localhost:5001/swagger/index.html

Using Web API Analyzers

What are Web API Analyzers?

- Web API Analyzers provides a mechanism to validate the action response types to work with API documentation.
- Applicable only for Web API projects.
- It checks all controllers that are marked with the ApiController attribute.
- It validates the following for all actions within each controller class:
  - Returns an undeclared status code.
  - Returns an undeclared success result.
  - Documents a status code that isn't returned.
  - Includes an explicit model validation check.

Step 8: Enable Web API Analyzers

1. ASP.NET Core SDK already includes the required assemblies to support Web API Analyzers. Hence, **no external package is needed** to be included in your project.
2. **Set the IncludeOpenAPIAnalyzers property to true** in the project file to enable Web API Analyzers.

   ```
   <PropertyGroup>
     ...
     <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
     ...
   </PropertyGroup>
   ```

Step 9: Add Annotations as per Warnings

1. **Add the respective ProducesResponseType for each of the actions** as per the warnings.

```
[ProducesResponseType(200)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
[HttpGet]
public ActionResult<IEnumerable<CourseDto>> GetCourses() {}
```

2. If needed, **add annotations for all actions within a controller at the individual controller level**. This is useful only for those that are common for all. Eg. 500 Internal Server Error.

```
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public class CoursesController : ControllerBase {}
```

3. If needed, **add annotations globally for all controllers and actions methods using Filters property in Startup.cs.**

```
services.AddControllers(c =>
{
    c.Filters.Add(new ProducesResponseTypeAttribute(500));
});
```

Using Web API Conventions

What are Web API Conventions?

- API Conventions:
  - A mechanism provided by ASP.NET to extract the common API documentations to actions or controllers.
  - It is automated to apply [ProducesResponseType] attributes to actions and controllers.
  - They are defined in the Microsoft.AspNetCore.Mvc.DefaultApiConventions namespace.
  - Each action may be associated with exactly one convention.
  - More specific conventions take precedence over less specific conventions.

- Key Benefits of API Conventions:
  - Define the most common return types and status codes returned from a specific type of action.
  - Identify actions that deviate from the defined standard.
- You can apply conventions in 3 different ways:
  - Action level
  - Controller level
  - Assembly level

Step 10: Apply Web API Conventions

Method 1: Applying Conventions to Actions

Use [ApiConventionMethod] attribute to add a convention to an action. It provides the convention type and the convention method to be used to apply the conventions.

```
[HttpGet("{courseId}")]
[ApiConventionMethod(typeof(DefaultApiConventions),
      nameof(DefaultApiConventions.Get))]
public ActionResult<CourseDto> GetCourse(int courseId)
{}
```

Method 2: Applying Conventions to Controllers

When a convention is applied to a controller, it is applied on all actions within that controller. The convention class has rules defined within it to determine which response type should be applicable for which actions.

Apply convention to controllers using [ApiConventionType] attribute.

```
[ApiConventionType(typeof(DefaultApiConventions))]
public class CoursesController : ControllerBase
{}
```

Method 3: Applying Conventions to Assembly

When a convention is applied at the assembly level, it is applied to all actions for all controllers present within that assembly.

Apply conventions at the assembly level using [ApiConventionType] attribute in Startup.cs.

```
[assembly: ApiConventionType(typeof(DefaultApiConventions))]
namespace Cms.WebApi
{
    public class Startup
    {}
}
```

Step 11: Update Action Names as per Conventions

API conventions will work only if our code follows the naming convention. Note: If you explicitly apply convention to an action method, then it will be applicable irrespective of the naming convention at the class level.

1. **Update the name of each action method as expected by the default conventions.**

Action	Method Name Match	Attributes Applied	Parameter Name	Parameter Name Match	Parameter Type Match
Get Find	Prefix	[ProducesDefaultResponseType] [ProducesResponseType(200)] [ProducesResponseType(404)]	id	Suffix	Any
Post Create	Prefix	[ProducesDefaultResponseType] [ProducesResponseType(201)] [ProducesResponseType(400)]	model	Any	Any
Put Update Edit	Prefix	[ProducesDefaultResponseType] [ProducesResponseType(204)] [ProducesResponseType(404)] [ProducesResponseType(400)]	id, model	Suffix, Any	Any, Any
Delete	Prefix	[ProducesDefaultResponseType] [ProducesResponseType(200)] [ProducesResponseType(404)] [ProducesResponseType(400)]	id	Suffix	Any

Step 12: Create Custom API Conventions

If needed, **define custom API conventions** using static class and static method.

1. **Add a static class with the required set of static methods**.

```
public static class CmsApiConventions
{
     [ApiConventionNameMatch(ApiConventionNameMatchBehavior.Prefix)]
     [ProducesResponseType(200)]
     [ProducesResponseType(500)]
     public static void GetAll()
     {}
}
```

2. **Add the list of response types and naming requirements on actions**.
3. **Apply these custom API conventions to your methods** using the same approach as DefaultApiConventions.

```
[HttpGet]
[ApiConventionMethod(typeof(CmsApiConventions),
      nameof(CmsApiConventions.GetAll))]
public ActionResult<IEnumerable<CourseDto>> GetAllCourses()
{}
```

4. **Tweak the custom api convention definitions to include matching patterns** for action name, parameter type, and parameter name.

```
public static class CmsApiConventions
{
     [ApiConventionNameMatch(ApiConventionNameMatchBehavior.Prefix)]
     [ProducesResponseType(200)]
     [ProducesResponseType(500)]
     public static void GetAll()
     {}
}
```

TIP: The simplest way to create custom API conventions is to duplicate the DefaultApiConventions class and update only the required part.

Documenting API Versions

Step 13: Enable API Versioning

1. Versioning web API is optional. However, it is **strongly recommended for the Web API project to support versioning** for easier maintenance in future.
2. **Enable versioning to the web api project by adding AddApiVersioning() method** to the service collection.

```
services.AddApiVersioning(setupAction =>
{
    setupAction.AssumeDefaultVersionWhenUnspecified = true;
    setupAction.DefaultApiVersion = new ApiVersion(1, 0);
});
```

3. **Add [ApiVersion] attribute** to the controller.
4. **Update the [Route] attribute to refer to the version in the path.**

```
[ApiController]
[Route("v{version:apiVersion}/[controller]")]
[ApiVersion("1.0")]
public class CoursesController : ControllerBase
{}
```

5. **Create the additional versioned controllers for your resource** and make the necessary changes.

```
[ApiController]
[Route("v{version:apiVersion}/courses")]
[ApiVersion("2.0")]
public class Courses2Controller : ControllerBase
{}
```

Step 14: Add Support for Multiple Versions

Swashbuckle uses ApiExplorer to extract the API endpoints details required for documentation.

1. **Add "Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer" NuGet package** to your Web API project.

2. **Remove AddApiExplorer and add AddVersionedApiExplorer() to the service collection** to add support for versioned controllers.

```
services.AddVersionedApiExplorer(option =>
{
   option.SubstituteApiVersionInUrl = true;
});
```

3. **Set GroupName in [ApiExplorerSettings] attribute in each controller to identify them uniquely**.

```
[ApiExplorerSettings(GroupName = "v1")]
public class CoursesController : ControllerBase
{}
```

Step 15: Add Document for Each Version

1. **Add one instance of Swagger Document in configure services for each version.**

```
services.AddSwaggerGen(s =>
      {
          s.SwaggerDoc("v1", new OpenApiInfo
          {
              Title = "CMS OpenAPI",
              Version = "v1",
              Description = "OpenAPI Specification for CMS",
              License = new OpenApiLicense()
              {
                  Name = "MIT",
              },
              Contact = new OpenApiContact()
              {
                  Name = "Praveenkumar Bouna",
                  Email = "hello@codewithpraveen.com"
              },
          });

          s.SwaggerDoc("v2", new OpenApiInfo
          {
```

```
            Title = "CMS OpenAPI",
            Version = "v2",
            Description = "OpenAPI Specification for CMS",
            License = new OpenApiLicense()
            {
                Name = "MIT",
            },
            Contact = new OpenApiContact()
            {
                Name = "Praveenkumar Bouna",
                Email = "hello@codewithpraveen.com"
            },
        });

        var xmlPath =
               Path.Combine(System.AppContext.BaseDirectory,
               "Cms.WebAPI.xml");
        s.IncludeXmlComments(xmlPath);
    });
```

2. **Add one instance of Swagger UI in the Configure method for each version.**

```
app.UseSwaggerUI(s =>
{
    s.SwaggerEndpoint("/swagger/v1/swagger.json", "v1");
    s.SwaggerEndpoint("/swagger/v2/swagger.json", "v2");
});
```

3. **Check the output in the UI to view multiple documents.**

Using Generic Version Handler

Creating individual documents is fine if you have few controllers. However, if the version increases, then it leads to unnecessary duplication of code. Also, you can keep the code generic so that future changes don't need code updates. ApiExplorer provides description of API versions added to each controller. You can use this to fetch the version details to make the document and ui creation generic.

1. **Fetch the IApiVersionDescriptionProvider service and enumerate the API version descriptions present in them**.

```
var apiVersionDesc = services
      .BuildServiceProvider()
      .GetRequiredService<IApiVersionDescriptionProvider>();

foreach(var apiVersion in apiVersionDesc.ApiVersionDescriptions)
{
       services.AddSwaggerGen(s =>
       {
           s.SwaggerDoc(apiVersion.GroupName, new OpenApiInfo
           {
               Title = "CMS OpenAPI",
               Version = apiVersion.GroupName,
               …
           };
       };
}
```

2. **AddMvcCore should be added to the service collection before using IApiVersionDescriptionProvider.**
3. **Set the GroupNameFormat to the required format.**

```
services.AddVersionedApiExplorer(option =>
{
     option.GroupNameFormat = "'v'V";
     option.SubstituteApiVersionInUrl = true;
});
```

4. **Remove individual document logics**, if present.
5. **Update Swagger UI logic to enumerate the API version descriptions in IApiVersionDescriptionProvider**.

```
app.UseSwaggerUI(s =>
{
  foreach (var item in
           apiVersionDescriptionProvider.ApiVersionDescriptions)
  {
       s.SwaggerEndpoint($"/swagger/{item.GroupName}/swagger.json",
```

```
                item.GroupName);
        }
    });
```

Reference: API Versioning Formats

Thank you!

I hope this resource was helpful to you.