Introduction to Swagger
Swagger is a set of open-source tools that help developers to generate interactive UI to document, test RESTful services.

Swagger is a set of tools to implement Open API.

1. Swasbuckle.AspNetCore

Framework that makes it easy to use swagger in asp.net core.

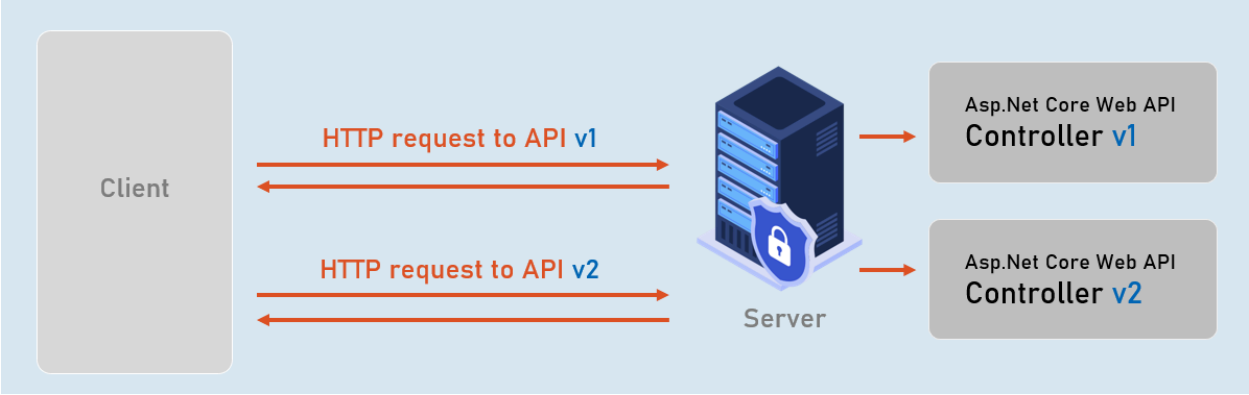
2. Swagger

Set of tools to generate UI to document & test RESTful services.

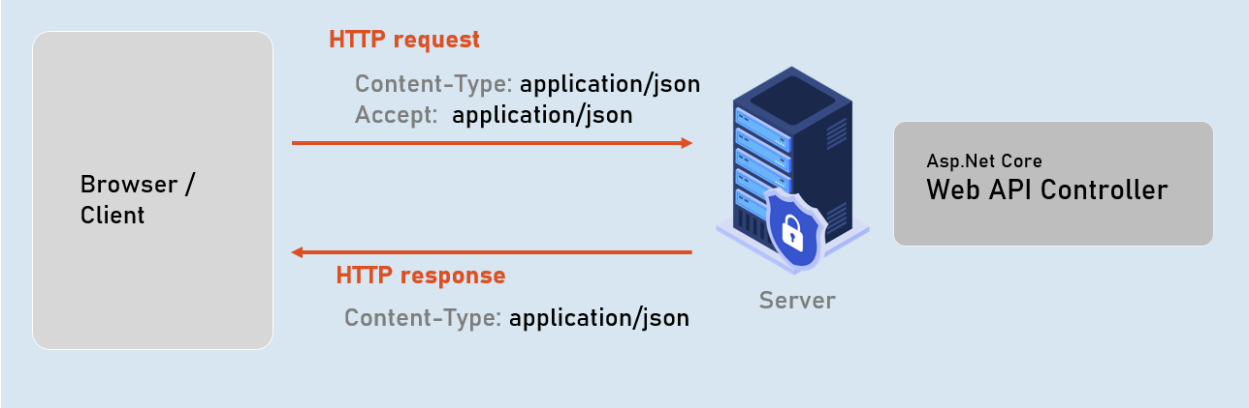
3. Open API

Specification that defines how to write API specifications in JSON).

API Versions
API Versioning is the practice of transparently managing changes to your API, where the client requests a specific version of API; and the server executes the same version of the API code.



Content Negotiation
Content negotiation is the process of selecting the appropriate format or language of the content to be exchanged between the client (browser) and Web API.



Swagger / OpenAPI
Swagger (now known as **OpenAPI**) is a framework for API documentation and specification. It allows you to describe your RESTful API in a machine-readable format, providing a way to generate interactive documentation, client SDKs, and server stubs. This makes APIs easier to understand, use, and test.

Overview of Swagger / OpenAPI
OpenAPI Specification (OAS) is a standard for defining RESTful APIs. It provides a way to describe the endpoints, request/response formats, parameters, authentication methods, and more.

Swagger tools are used to generate interactive documentation and client libraries based on the OpenAPI specification.

Setting Up Swagger in ASP.NET Core
1. Install Swagger NuGet Packages

To use Swagger in an ASP.NET Core application, you need to install the `Swashbuckle.AspNetCore` NuGet package, which provides the Swagger generator and UI.

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

2. Configure Swagger in `Program.cs`

In your `Program.cs` file, you need to add Swagger services and configure the Swagger middleware.

ConfigureServices Method:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllers();

   // Register Swagger services
   services.AddSwaggerGen(c =>
   {
       c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API",
Version = "v1" });
       // Optionally, include XML comments for richer
documentation
       // var xmlFile =
$"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
       // var xmlPath = Path.Combine(AppContext.BaseDirectory,
xmlFile);
       // c.IncludeXmlComments(xmlPath);
   });
}
```

Configure Method:

```
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
       app.UseDeveloperExceptionPage();
   }
   else
   {
       app.UseExceptionHandler("/Home/Error");
       app.UseHsts();
   }

   app.UseHttpsRedirection();
   app.UseStaticFiles();
   app.UseRouting();
   app.UseAuthorization();

   // Use Swagger
   app.UseSwagger();

   // Use Swagger UI
   app.UseSwaggerUI(c =>
   {
```

```
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API
V1");
        c.RoutePrefix = string.Empty; // Set Swagger UI at the
app's root (optional)
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

3. XML Comments for Enhanced Documentation

To enhance the documentation, you can include XML comments. This requires enabling XML documentation in your project file and configuring Swagger to include these comments.

Project File (`.csproj`):

```
<PropertyGroup>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
    <NoWarn>1591</NoWarn> <!-- Suppress missing XML comment
warnings -->
</PropertyGroup>
```

Update Swagger Configuration:

```
c.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
"MyApi.xml"));
```

4. Testing the API Documentation

Once configured, run your application and navigate to the Swagger UI (usually at `/swagger` or the root if configured) to see the interactive API documentation. Swagger UI allows you to explore and test your API endpoints directly from the browser.

Detailed Explanation of Code

- **`AddSwaggerGen` Method**: Registers the Swagger generator with default settings. You can provide additional options such as custom filters or document settings.
- **`SwaggerDoc` Method**: Defines a Swagger document with a title and version. You can create multiple versions if needed.
- **`UseSwagger` and `UseSwaggerUI` Methods**: Middleware components to serve the Swagger JSON endpoint and the interactive UI, respectively. `SwaggerEndpoint` specifies the path to the Swagger JSON file.

- **XML Comments**: Provides additional metadata for API methods and models, which is displayed in Swagger UI.

Content Negotiation

Content negotiation is a mechanism in HTTP that allows clients and servers to agree on the format of the response data. In ASP.NET Core, content negotiation determines how the response should be formatted based on the client's request headers and available formatters.

Overview of Content Negotiation

When a client sends a request, it may specify the desired response format through the `Accept` header. The server processes this header and selects the appropriate formatter to serialize the response data into the requested format (e.g., JSON, XML).

How Content Negotiation Works

1. **Client Request**: The client sends an HTTP request with the `Accept` header specifying the desired media type (e.g., `application/json`, `application/xml`).
2. **Server Response**: The server uses formatters to serialize the response data into the specified format. If the requested format is not supported or available, the server may return a default format or an error.

Configuring Formatters in ASP.NET Core

ASP.NET Core provides built-in formatters for JSON and XML. You can configure these formatters in the `Program.cs` file.

1. JSON Formatter

By default, ASP.NET Core includes the JSON formatter via `System.Text.Json`. You can also use `Newtonsoft.Json` if you prefer.

Example Configuration with `System.Text.Json`:

```
    services.AddControllers()
        .AddJsonOptions(options =>
        {
            options.JsonSerializerOptions.PropertyNamingPolicy =
null; // Disable camel casing
        });
```

Example Configuration with `Newtonsoft.Json`:

First, install the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` NuGet package:

```
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
```

Then configure it in `Program.cs`:

```
    services.AddControllers()
        .AddNewtonsoftJson(options =>
        {
            options.SerializerSettings.ContractResolver = new
CamelCasePropertyNamesContractResolver();
        });
```

2. XML Formatter

To enable XML formatting, you need to add the `AddXmlSerializerFormatters` method.

Example Configuration:

```
    services.AddControllers()
        .AddXmlSerializerFormatters(); // Add XML formatter
```

3. Custom Formatters

You can create custom formatters if you need to support additional formats.

Example of a Custom Formatter:

Create a custom `OutputFormatter`:

```
public class CustomXmlOutputFormatter : TextOutputFormatter
{
    public CustomXmlOutputFormatter()
    {

SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("application
/custom-xml"));
    }

    public override bool
CanWriteResult(OutputFormatterCanWriteContext context)
    {
        return
context.ContentType.Equals(MediaTypeHeaderValue.Parse("applicat
ion/custom-xml"));
    }

    public override Task
WriteResponseBodyAsync(OutputFormatterWriteContext context,
Encoding selectedEncoding)
```

```
        {
            // Implement custom XML serialization logic here
        }
    }
```

Register the custom formatter:

```
        services.AddControllers(options =>
        {
            options.OutputFormatters.Add(new
    CustomXmlOutputFormatter());
        });
```

Detailed Explanation of Code

- **`AddJsonOptions`**: Configures JSON serialization settings, such as property naming policies.
- **`AddNewtonsoftJson`**: Adds support for JSON serialization using Newtonsoft.Json, allowing for more advanced configuration.
- **`AddXmlSerializerFormatters`**: Enables XML serialization using the XML serializer.
- **Custom Formatters**: Allow you to create formatters for unsupported media types or customize serialization logic.

Testing Content Negotiation

You can test content negotiation by sending requests with different `Accept` headers using tools like Postman or `curl`.

Example Request with `curl`:

```
curl -H "Accept: application/json"
https://localhost:5001/api/products
curl -H "Accept: application/xml"
https://localhost:5001/api/products
```

Example Request with Postman:

- Set the `Accept` header to `application/json` or `application/xml` in Postman and observe the response format.

Key Points to Remember

1. **Content Negotiation**: Determines the format of the response based on the client's `Accept` header.
2. **Built-in Formatters**: ASP.NET Core provides JSON and XML formatters out of the box.
3. **Custom Formatters**: You can create custom formatters to support additional media types.
4. **Configuration**: Use `AddJsonOptions`, `AddNewtonsoftJson`, and `AddXmlSerializerFormatters` to configure formatters.

5. **Testing**: Use tools like Postman or `curl` to test different response formats.

API Versions

API versioning is crucial for managing changes in your API while keeping backward compatibility. With the deprecation of the `Microsoft.AspNetCore.Mvc.Versioning` package, you should use the new `Asp.Versioning.Mvc` package for implementing API versioning in ASP.NET Core.

Overview of API Versioning

API versioning allows you to introduce new features or changes in your API without breaking existing clients. It helps maintain multiple versions of an API simultaneously.

Implementing API Versioning with `Asp.Versioning.Mvc`

1. Install the New API Versioning NuGet Package

Install the `Asp.Versioning.Mvc` package to use the new API versioning library.

```
dotnet add package Asp.Versioning.Mvc
```

2. Configure API Versioning in `Program.cs`

Add and configure the API versioning services using the new package in the `ConfigureServices` method.

Example Configuration:

```
services.AddControllers();

// Add API versioning
services.AddApiVersioning(options =>
{
    options.ReportApiVersions = true; // Include API
versions in response headers
    options.AssumeDefaultVersionWhenUnspecified = true; //
Assume default version if none specified
    options.DefaultApiVersion = new ApiVersion(1, 0); // Set
default API version
    options.ApiVersionReader = new
HeaderApiVersionReader("api-version"); // Read version from
header
});
```

3. Define API Versions in Controllers

Use the `[ApiVersion]` attribute to specify which versions a controller or action method supports.

Example:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    [ApiVersion("1.0")]
    public IActionResult GetV1()
    {
        return Ok("API Version 1.0");
    }

    [HttpGet]
    [ApiVersion("2.0")]
    [Route("v2")]
    public IActionResult GetV2()
    {
        return Ok("API Version 2.0");
    }
}
```

4. Specify API Versions in Routes

You can include the API version in the route to differentiate between versions.

Example:

```
[ApiController]
[Route("api/v{version:apiVersion}/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        // Handle request for the specified API version
        return Ok("API Versioned");
    }
}
```

5. Use URL or Query String Versioning

Besides headers, you can use URL segments or query strings for versioning.

Example of URL Versioning:

```
[ApiController]
[Route("api/v{version:apiVersion}/products")]
public class ProductsController : ControllerBase
{
   [HttpGet]
   public IActionResult Get()
   {
       // Handle request for the specified API version
       return Ok("API Versioned via URL");
   }
}
```

Example of Query String Versioning:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddApiVersioning(options =>
   {
       options.ApiVersionReader = new
QueryStringApiVersionReader("api-version"); // Read version
from query string
   });
}

[ApiController]
[Route("api/products")]
public class ProductsController : ControllerBase
{
   [HttpGet]
   public IActionResult Get()
   {
       // Handle request for the specified API version
       return Ok("API Versioned via Query String");
   }
}
```

Detailed Explanation of Code

- `AddApiVersioning`: Configures API versioning services, including options for default version, version reporting, and version readers.
- `[ApiVersion]` **Attribute**: Specifies the supported versions for a controller or action method.
- **Routes**: Define versioned routes to access different API versions.

- **Version Readers**: Methods to extract version information from request headers, URLs, or query strings.

Example Request with Postman:

- Set the `api-version` header or use versioned URLs to test different API versions.

Key Points to Remember

1. **API Versioning**: Allows multiple versions of an API to coexist and ensures backward compatibility.
2. **New Package**: Use `Asp.Versioning.Mvc` instead of the deprecated `Microsoft.AspNetCore.Mvc.Versioning`.
3. **Configuration**: Set up API versioning in `Program.cs` using `AddApiVersioning`.
4. **Versioning Methods**: Use header, URL segment, or query string methods to handle API versions.
5. **Testing**: Validate versioning using tools like Postman or `curl` to ensure proper version handling.