Integrating Swagger and Generating API Clients

Objective: in this lab, you'll integrate Swagger with an ASP.NET Core application and generate API clients. You'll practice configuring Swagger, generating client code, and customizing the generated code.

Step 1: Set Up a New Console Application

In this step, you'll create a new ASP.NET Core console application, setting up the folders and initial structure you'll need for this lab.

1. Open Visual Studio Code and select a new terminal.
2. Run the command to create a new console application: dotnet new console -o SwaggerApiClientLab
3. Navigate into the project folder: cd SwaggerApiClientLab
4. Add the necessary packages for working with Swagger and NSwag by running: dotnet add package Swashbuckle.AspNetCore dotnet add package NSwag.Core dotnet add package NSwag.CodeGeneration.CSharp dotnet add package NSwag.ApiDescription.Client
5. Inside your project folder, create a folder named Controllers to hold your API controller.
6. Confirm that your project structure looks like this:

   SwaggerApiClientLab ├── Controllers ├── Program.cs └── SwaggerApiClientLab.csproj

Step 2: Configure Swagger in the Application

Next, you'll add Swagger configuration to the application. This step involves setting up Swagger middleware so the API documentation is available and ready for client generation.

1. Open the Program.cs file and locate the appropriate section to add Swagger configuration.
2. Configure Swagger so it serves the API documentation in JSON format. Use the Swagger UI path /swagger/v1/swagger.json.
3. Start the server. Use app.Run();
4. Run the application using dotnet run, and navigate to the Swagger UI endpoint in your browser to verify the Swagger setup.

Step 3: Create the AP1 Specification

Now, you'll define an API endpoint to provide the functionality for client code generation. This specification will document the API routes, parameters, and response types.

1. Inside the Controllers folder, create a new file named UserController.cs.
2. Create a basic User class with that has Id and Name properties.
3. Set up a simple GetUser endpoint that will accept a user ID and return sample JSON User detail.
4. Run the application again and view the Swagger documentation in your browser to confirm the API endpoint appears correctly in the documentation.

Step 4: Generate Client Code with NSwag

In this step, you'll use NSwag to generate client code from the Swagger documentation, which will allow client-side code to interact with the server endpoints automatically.

1. Create a new file named ClientGenerator.cs in the project root.

2. Place the following code in ClientGenerator.cs to fetch the Swagger JSON and generate client code. This code uses the NSwag library and sets a custom namespace and class name for your generated client.

```
public class ClientGenerator
{
    public async Task GenerateClient()
    {
        using var httpClient = new HttpClient();
        var swaggerJson = await
httpClient.GetStringAsync("http://localhost:<port>/swagger/v1/swagger.js
on");
        var document = await OpenApiDocument.FromJsonAsync(swaggerJson);

        var settings = new CSharpClientGeneratorSettings
        {
            ClassName = "GeneratedApiClient",
            CSharpGeneratorSettings = { Namespace =
"MyApiClientNamespace" }
        };

        var generator = new CSharpClientGenerator(document, settings);
        var code = generator.GenerateFile();

        await File.WriteAllTextAsync("GeneratedApiClient.cs", code);
    }
}
```

3. In Program.cs Run the client generator to produce a file containing the generated client code, verifying that it's created in the project directory. To create the client code, follow these steps:

- Replace app.Run(); with Task.Run(() => app.RunAsync());. This will run the server asynchronously.
- Below the code that starts the server, add an awaited delay of 3 seconds or more. This will let the server start up before you generate the client code.
- Below the delay, add an awaited call to the GenerateClient method in the ClientGenerator class.

4. Run the application again to generate the client code.

Step 5: Customize the Generated Client Code

In this step, you'll make adjustments to the generated client settings, like customizing the namespace and class names.

1. Open ClientGenerator.cs and locate the configuration settings for class name and namespace.
2. Change the settings to match custom preferences, such as setting the class name to CustomApiClient and the namespace to CustomNamespace.
3. Run the client generator again to verify the generated client code reflects your custom settings.
4. In Program.cs, comment out or remove the call to GenerateClient

Step 6: Integrate the Client Code into the Application

In this final step, you'll use the generated client to make API requests from your application.

1. Open Program.cs and instantiate the generated client class, passing in the API base URL and HttpClient instance.
2. Use the client's methods to call the GetUser endpoint, retrieving and displaying the user data.
3. Run the application to confirm the API client integration is functioning as expected.

UserController.cs:

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class UserController : ControllerBase
{
    [HttpGet("{id}")]
    [ProducesResponseType(typeof(User), 200)]
    public ActionResult<User> GetUser(int id)
    {
        return Ok(new User { Id = id, Name = $"User{id}" });
    }
}

public class User
{
    public int Id { get; set; }
    public required string Name { get; set; }
}
```

ClientGenerator.cs:

```
using NSwag;
using NSwag.CodeGeneration.CSharp;

public class ClientGenerator
{
    private readonly string _baseUrl;

    public ClientGenerator(string baseUrl)
    {
        _baseUrl = baseUrl.TrimEnd('/');
    }

    public async Task GenerateClient()
    {
        using var httpClient = new HttpClient();
        var swaggerUrl = $"{_baseUrl}/swagger/v1/swagger.json";

        var swaggerJson = await httpClient.GetStringAsync(swaggerUrl);
        var document = await OpenApiDocument.FromJsonAsync(swaggerJson);

        var settings = new CSharpClientGeneratorSettings
        {
            ClassName = "CustomApiClient",
            CSharpGeneratorSettings = { Namespace = "CustomNamespace" }
        };

        var generator = new CSharpClientGenerator(document, settings);
        var code = generator.GenerateFile();

        await File.WriteAllTextAsync("GeneratedApiClient.cs", code);
        Console.WriteLine($"Client code generated from {swaggerUrl}");
    }
}
```

Program.cs:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using System.Reflection;

public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        var app = builder.Build();

        app.UseSwagger();
        app.UseSwaggerUI();
        app.MapControllers();

        _ = Task.Run(async () => await app.RunAsync());
        await Task.Delay(3000);

        if (!File.Exists("GeneratedApiClient.cs"))
        {
            var url = app.Urls.FirstOrDefault() ??
"http://localhost:5000";
            var generator = new ClientGenerator(url);
            await generator.GenerateClient();
            Console.WriteLine("GeneratedApiClient.cs created. Please
restart the application.");
            return;
        }

        await RunClient(app);
    }

    private static async Task RunClient(WebApplication app)
    {
        var url = app.Urls.FirstOrDefault() ?? "http://localhost:5000";
        var assembly = Assembly.GetExecutingAssembly();
        var clientType =
assembly.GetType("CustomNamespace.CustomApiClient");

        if (clientType == null)
        {
            Console.WriteLine("Client type not found. Please rebuild the
project.");
            return;
        }

        using var http = new HttpClient();
        var client = Activator.CreateInstance(clientType, url, http);

        if (client == null)
        {
            Console.WriteLine("Failed to create client instance.");
```

```
            return;
        }

        var method = clientType.GetMethod("GetUserAsync", new[] {
typeof(int), typeof(CancellationToken) })
                     ?? clientType.GetMethod("UserAsync", new[] {
typeof(int) });

        if (method == null)
        {
            Console.WriteLine("API method not found in generated
client.");
            return;
        }

        var parameters = method.GetParameters().Length == 2
            ? new object[] { 1, CancellationToken.None }
            : new object[] { 1 };

        var task = method.Invoke(client, parameters) as Task;

        if (task == null)
        {
            Console.WriteLine("Failed to invoke client method.");
            return;
        }

        await task.ConfigureAwait(false);

        var resultProperty = task.GetType().GetProperty("Result");
        if (resultProperty != null)
        {
            var user = resultProperty.GetValue(task);
            var idProp = user?.GetType().GetProperty("Id");
            var nameProp = user?.GetType().GetProperty("Name");

            Console.WriteLine($"User ID: {idProp?.GetValue(user)}, User
Name: {nameProp?.GetValue(user)}");
        }
        else
        {
            Console.WriteLine("Request completed successfully (200
OK).");
        }
    }
}
```