Creating and Using HTTP Client SDKs

Oleksii Nikiforov

Software Engineer at **EPAM**

TABLE OF CONTENTS.

01

WHY?

<p> To provide a consistent and manageable way of integrating with a service in the form of a distributable package </p>

WHYs

Demand

Distributed systems are quite popular

Meaningful abstraction

Distributable

Pack it as NuGet package

Speed-up itegration process

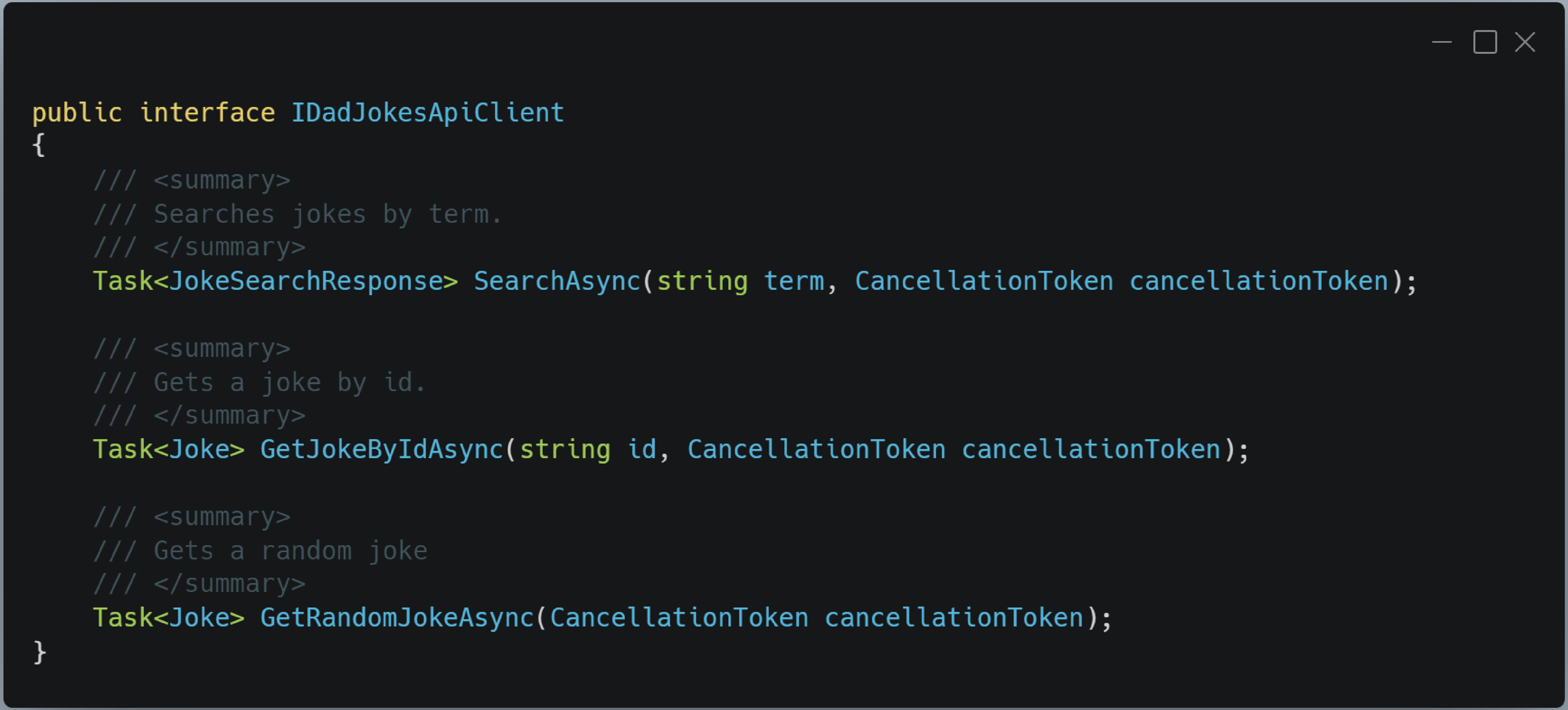
Consistent

A unified approach for all consumers

Versioning

Easy to release and version

02 How to write client SDKs manually

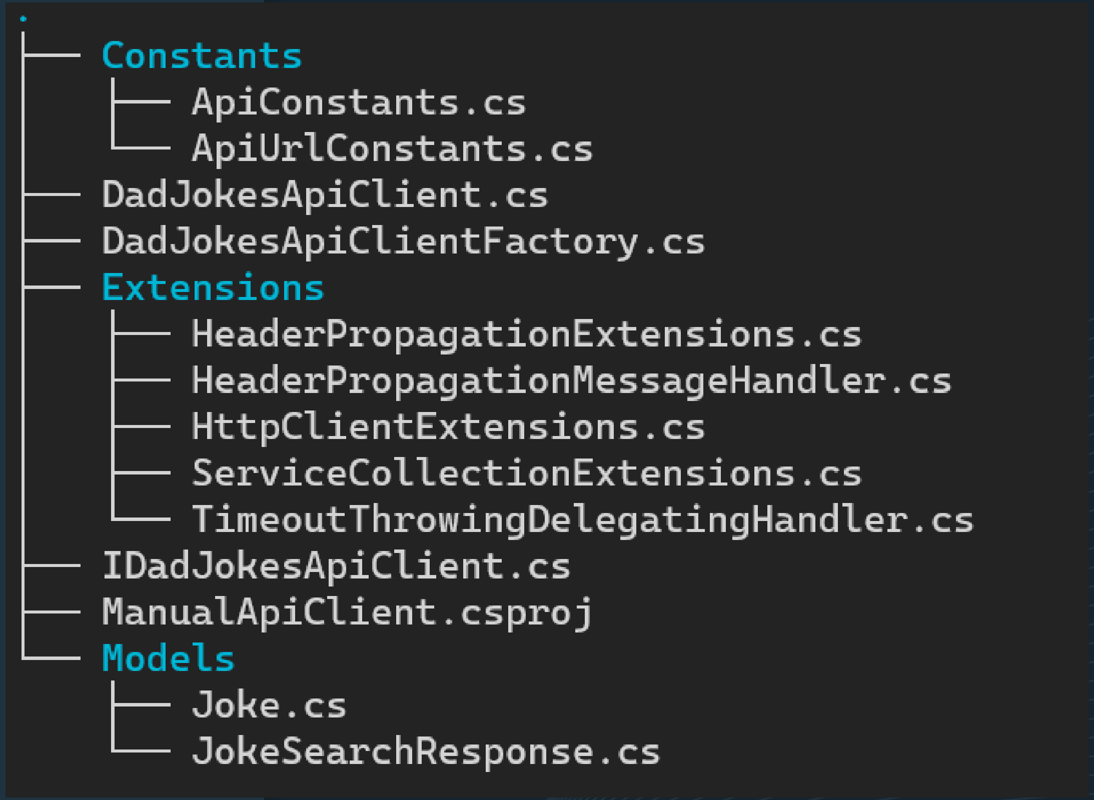


```csharp
public interface IDadJokesApiClient
{
    /// <summary>
    /// Searches jokes by term.
    /// </summary>
    Task<JokeSearchResponse> SearchAsync(string term, CancellationToken cancellationToken);

    /// <summary>
    /// Gets a joke by id.
    /// </summary>
    Task<Joke> GetJokeByIdAsync(string id, CancellationToken cancellationToken);

    /// <summary>
    /// Gets a random joke
    /// </summary>
    Task<Joke> GetRandomJokeAsync(CancellationToken cancellationToken);
}
```

It is a good idea to start from the contract

DEMO

The bread and butter of HTTP-based integrations is the HttpClient. It contains everything you need to work with HTTP abstractions successfully



```
.
├── Constants
│   ├── ApiConstants.cs
│   └── ApiUrlConstants.cs
├── DadJokesApiClient.cs
├── DadJokesApiClientFactory.cs
├── Extensions
│   ├── HeaderPropagationExtensions.cs
│   ├── HeaderPropagationMessageHandler.cs
│   ├── HttpClientExtensions.cs
│   ├── ServiceCollectionExtensions.cs
│   └── TimeoutThrowingDelegatingHandler.cs
├── IDadJokesApiClient.cs
├── ManualApiClient.csproj
└── Models
    ├── Joke.cs
    └── JokeSearchResponse.cs
```

Pros and Cons

✅ Full control over behavior and data contracts. Throw custom exceptions, transform payload, etc

✅ Easy to debug and troubleshoot. Simple stack trace

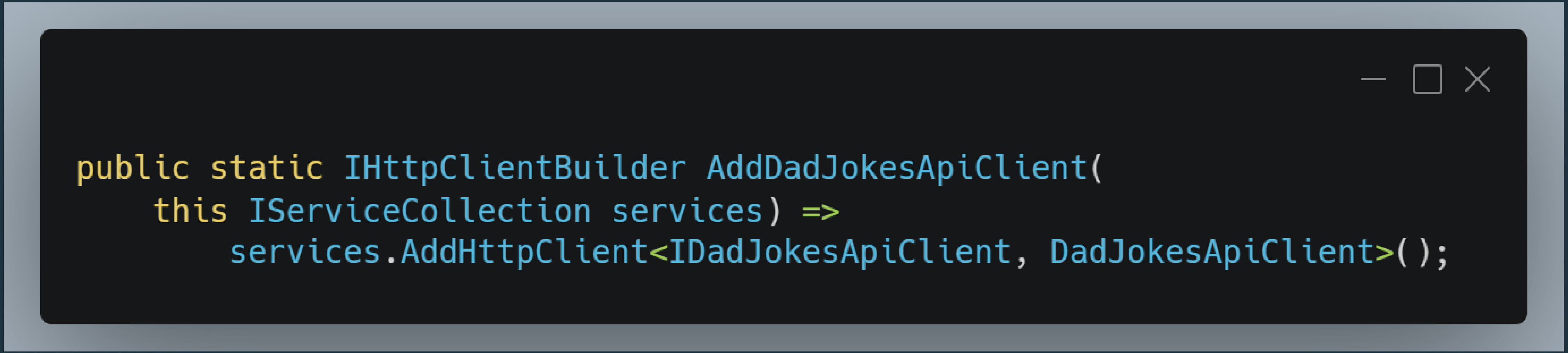
❌ Need to write a lot of repetitive code

❌ Someone should maintain a codebase in case of API changes and bugs

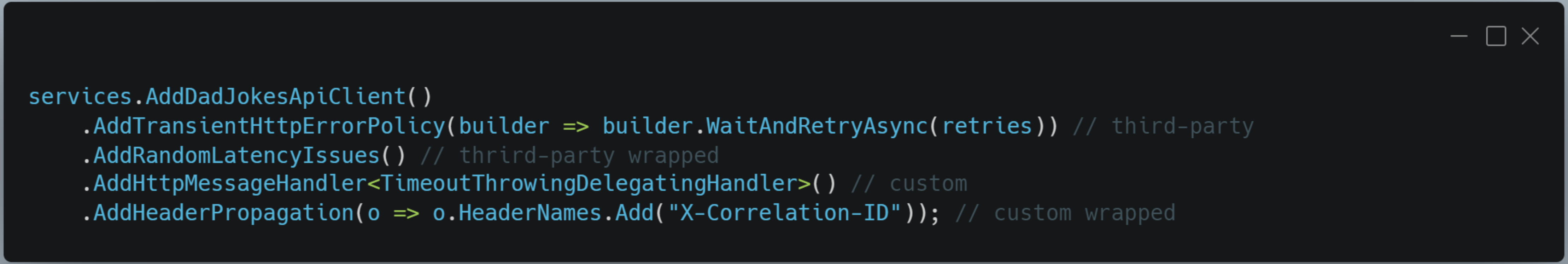
03 Extension points

<p>A wide variety of problems could be expressed as cross-cutting concerns. A message handler receives an HTTP request and returns an HTTP response</p>

DEMO



```
public static IHttpClientBuilder AddDadJokesApiClient(
    this IServiceCollection services) =>
        services.AddHttpClient<IDadJokesApiClient, DadJokesApiClient>();
```



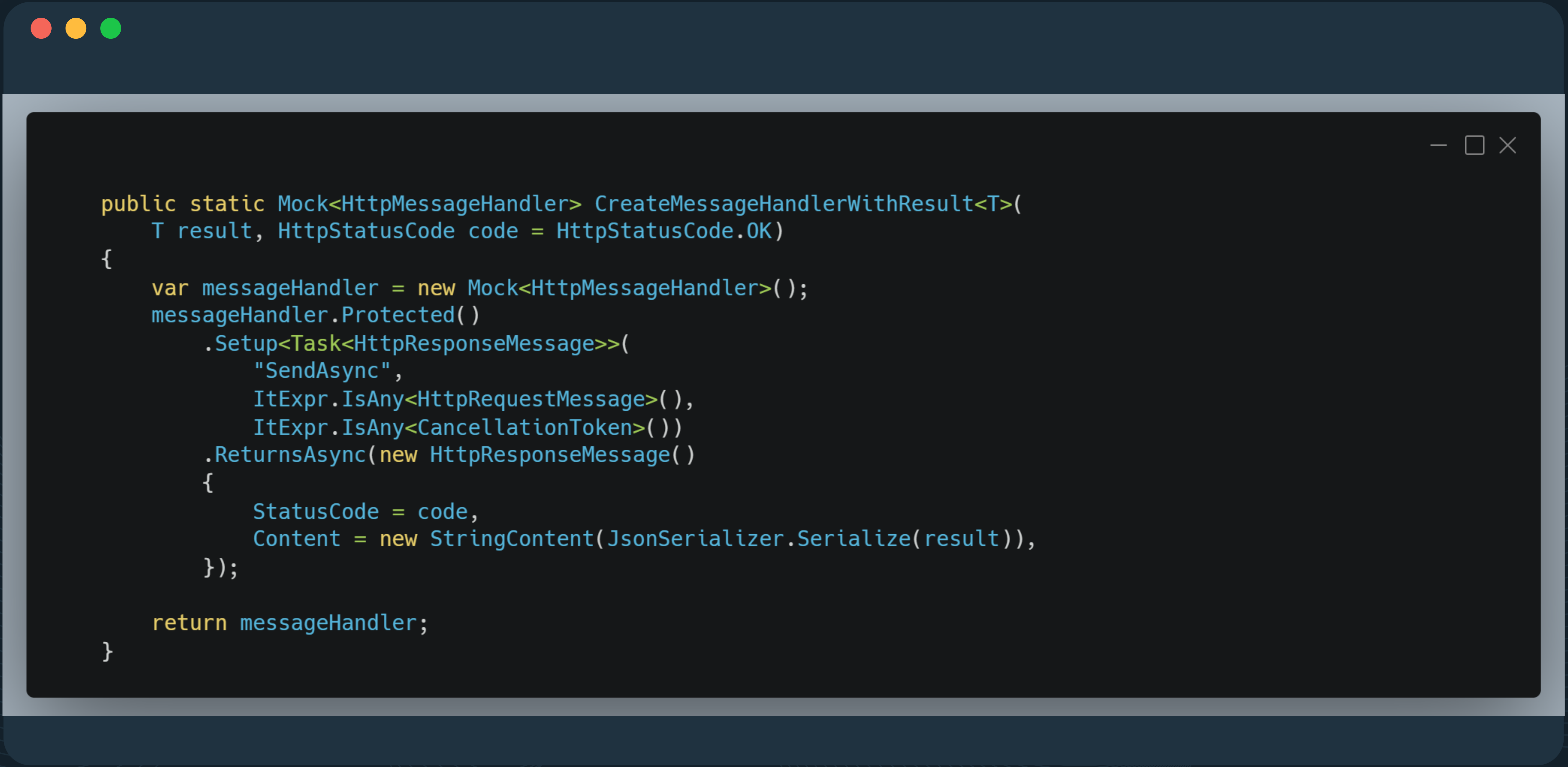
```
services.AddDadJokesApiClient()
    .AddTransientHttpErrorPolicy(builder => builder.WaitAndRetryAsync(retries)) // third-party
    .AddRandomLatencyIssues() // thrird-party wrapped
    .AddHttpMessageHandler<TimeoutThrowingDelegatingHandler>() // custom
    .AddHeaderPropagation(o => o.HeaderNames.Add("X-Correlation-ID")); // custom wrapped
```

`IHttpClientBuilder is a DI-friendly way to add message handlers.`

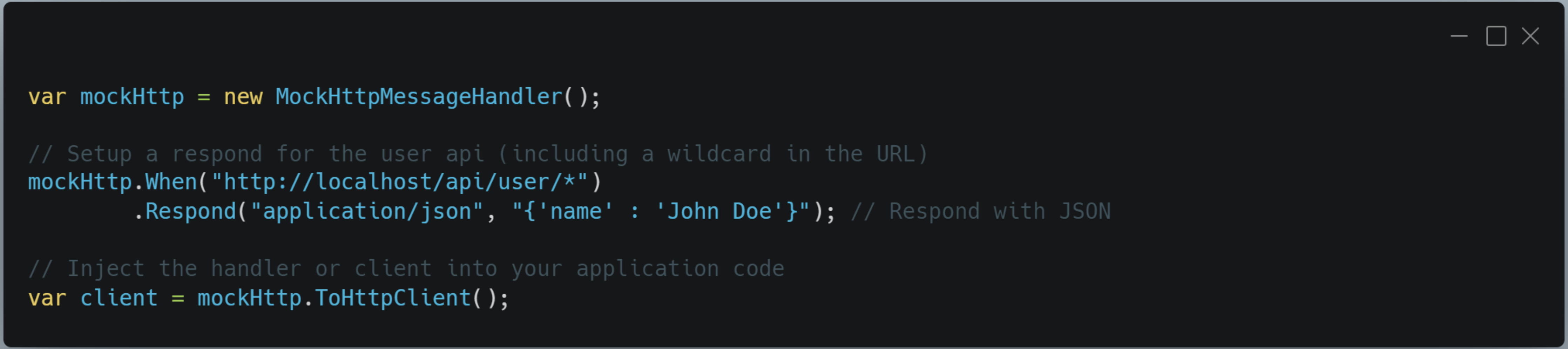
04

Test

<p> Add a brief introduction of your section here: Let’s dive in and get to know some interesting facts about animals! </p>

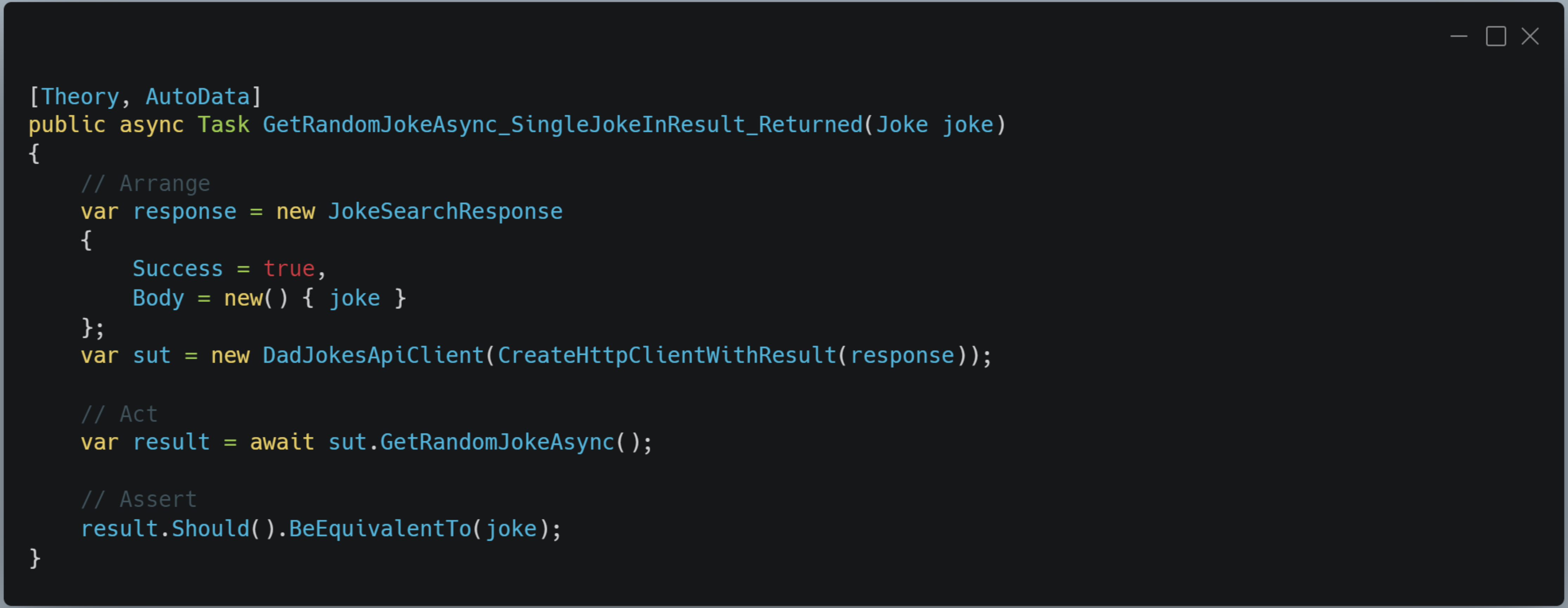


```
public static Mock<HttpMessageHandler> CreateMessageHandlerWithResult<T>(
    T result, HttpStatusCode code = HttpStatusCode.OK)
{
    var messageHandler = new Mock<HttpMessageHandler>();
    messageHandler.Protected()
        .Setup<Task<HttpResponseMessage>>(
            "SendAsync",
            ItExpr.IsAny<HttpRequestMessage>(),
            ItExpr.IsAny<CancellationToken>())
        .ReturnsAsync(new HttpResponseMessage()
        {
            StatusCode = code,
            Content = new StringContent(JsonSerializer.Serialize(result)),
        });

    return messageHandler;
}
```



```
var mockHttp = new MockHttpMessageHandler();

// Setup a respond for the user api (including a wildcard in the URL)
mockHttp.When("http://localhost/api/user/*")
        .Respond("application/json", "{'name' : 'John Doe'}"); // Respond with JSON

// Inject the handler or client into your application code
var client = mockHttp.ToHttpClient();
```

https://github.com/richardszalay/mockhttp

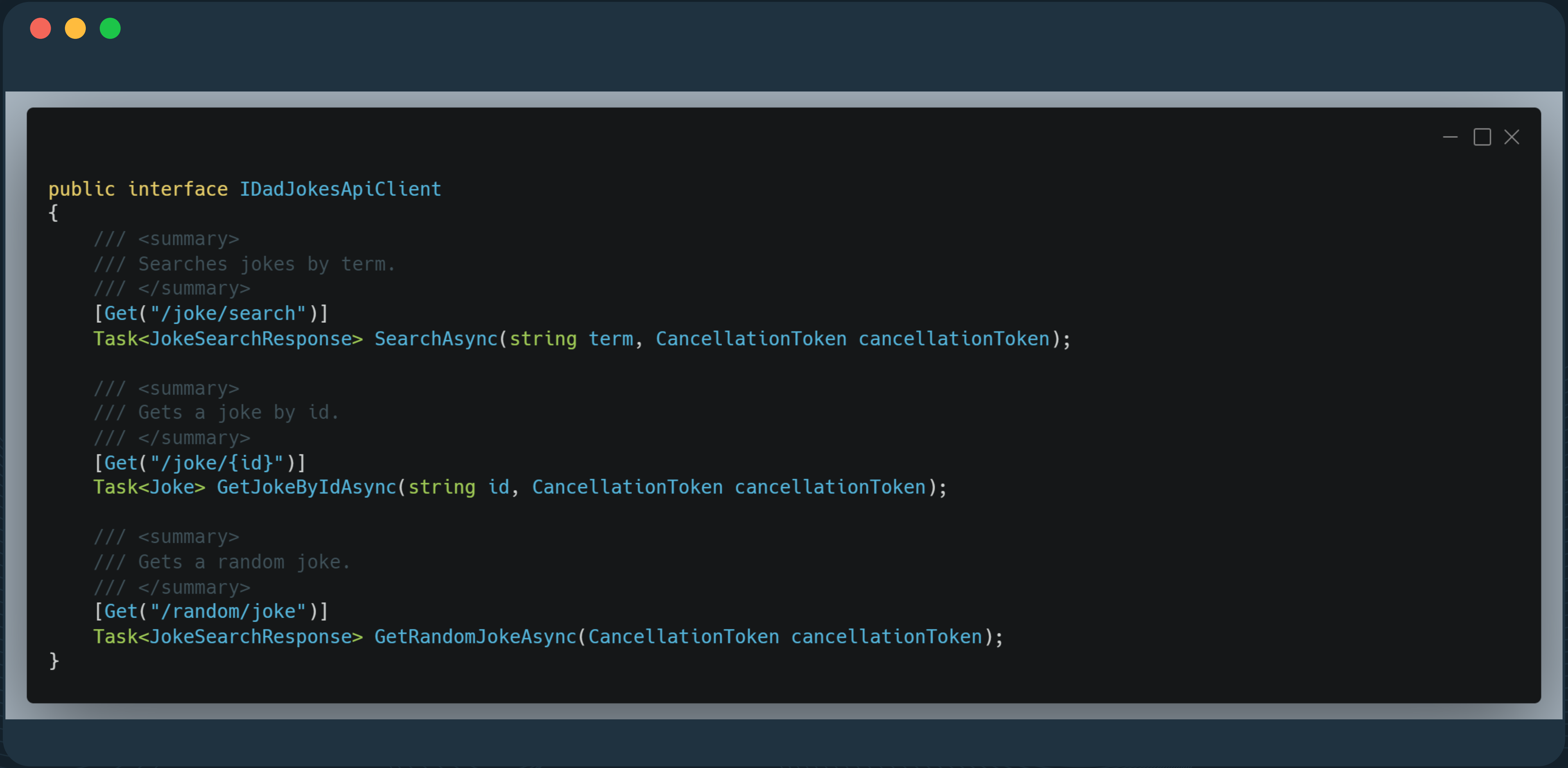


```
[Theory, AutoData]
public async Task GetRandomJokeAsync_SingleJokeInResult_Returned(Joke joke)
{
    // Arrange
    var response = new JokeSearchResponse
    {
        Success = true,
        Body = new() { joke }
    };
    var sut = new DadJokesApiClient(CreateHttpClientWithResult(response));

    // Act
    var result = await sut.GetRandomJokeAsync();

    // Assert
    result.Should().BeEquivalentTo(joke);
}
```

05

How to write declarative clients

<p>Refit is an automatic type-safe REST library for .NET. It turns your REST API into a live interface. </p>

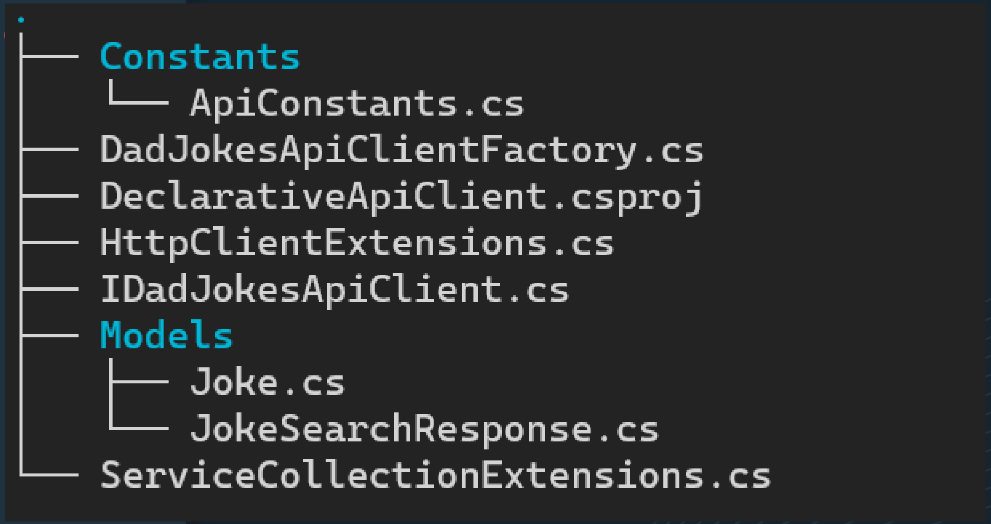


```
public interface IDadJokesApiClient
{
    /// <summary>
    /// Searches jokes by term.
    /// </summary>
    [Get("/joke/search")]
    Task<JokeSearchResponse> SearchAsync(string term, CancellationToken cancellationToken);

    /// <summary>
    /// Gets a joke by id.
    /// </summary>
    [Get("/joke/{id}")]
    Task<Joke> GetJokeByIdAsync(string id, CancellationToken cancellationToken);

    /// <summary>
    /// Gets a random joke.
    /// </summary>
    [Get("/random/joke")]
    Task<JokeSearchResponse> GetRandomJokeAsync(CancellationToken cancellationToken);
}
```

It is a good idea to start from the contract

DEMO

The code is automatically generated based on attribute-based configuration. This concept is known as metaprogramming.



```
.
├── Constants
│   └── ApiConstants.cs
├── DadJokesApiClientFactory.cs
├── DeclarativeApiClient.csproj
├── HttpClientExtensions.cs
├── IDadJokesApiClient.cs
├── Models
│   ├── Joke.cs
│   └── JokeSearchResponse.cs
└── ServiceCollectionExtensions.cs
```

Pros and Cons

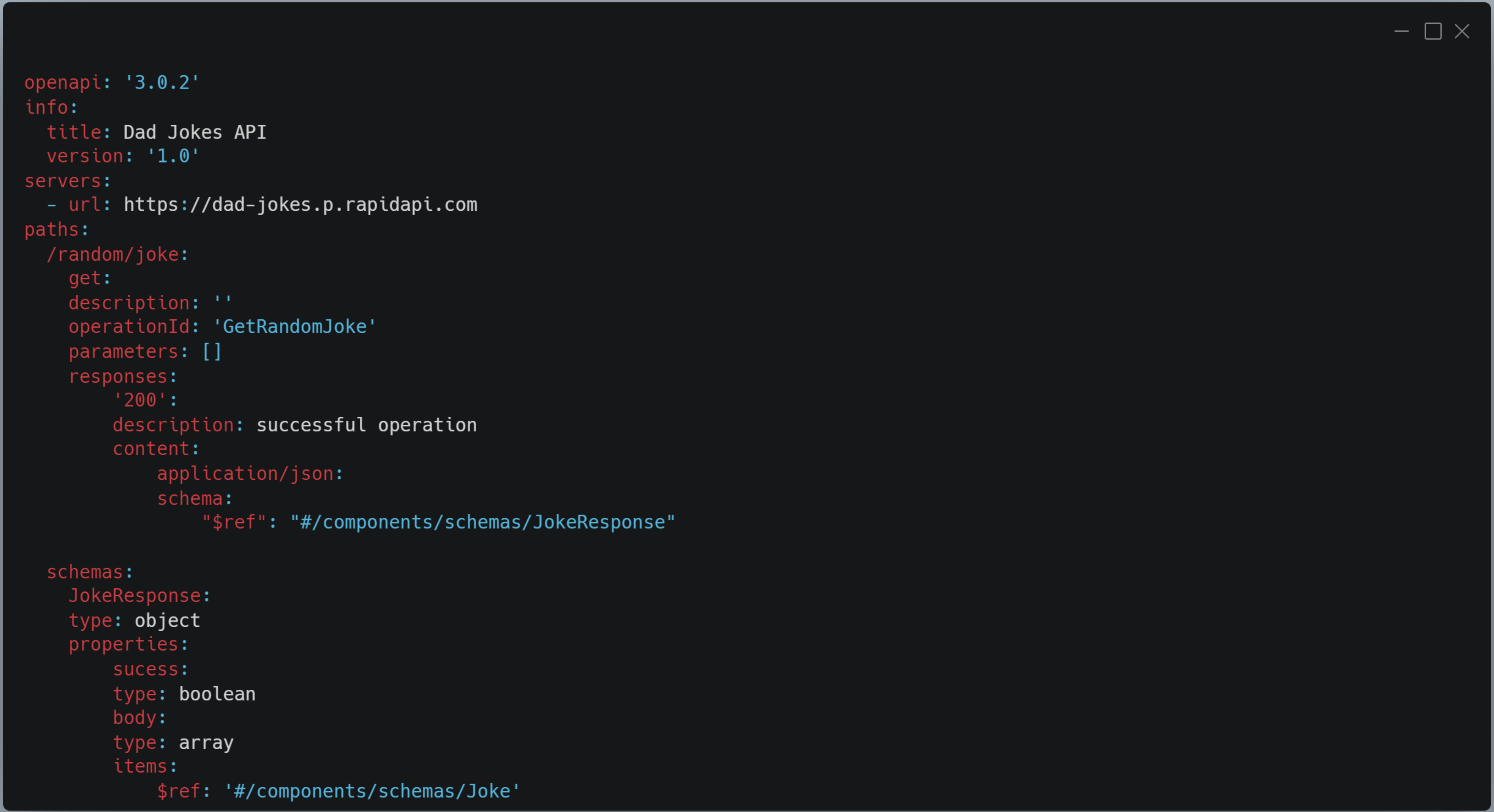
- ✓ Easy to use and develop API clients
- ✓ Highly configurable. Flexible enough to get things done
- ✓ No need for additional unit testing

- ✕ Hard to troubleshoot
- ✕ Requires other team members to understand the tool.
- ✕ Still consumes some time for medium/large APIs.

06

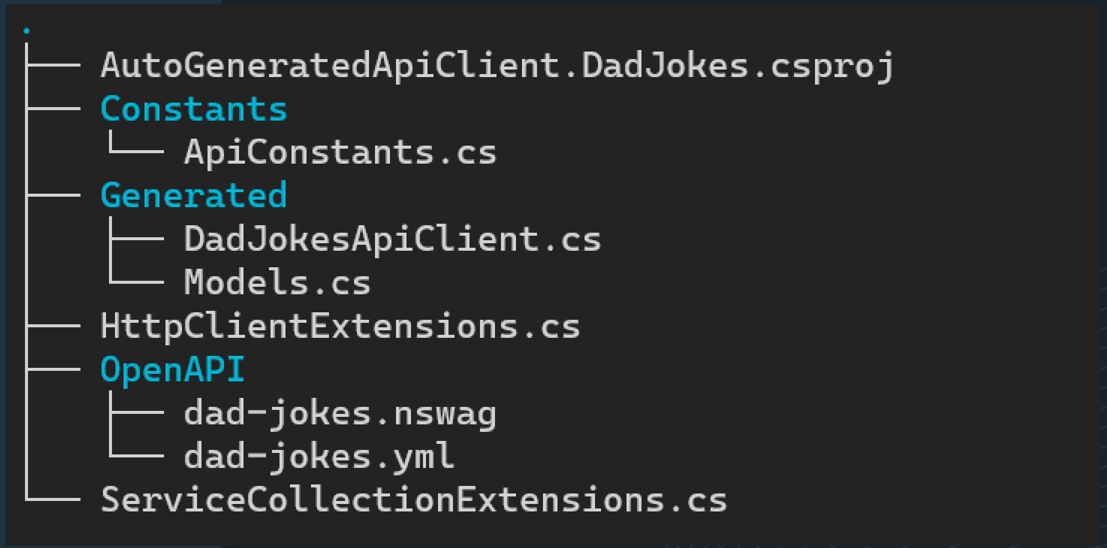
How to generate clients?

<p> There is a way to automate HTTP Client SDKs fully. The OpenAPI/Swagger specification uses JSON and JSON Schema to describe an API </p>


```yaml
openapi: '3.0.2'
info:
  title: Dad Jokes API
  version: '1.0'
servers:
  - url: https://dad-jokes.p.rapidapi.com
paths:
  /random/joke:
    get:
    description: ''
    operationId: 'GetRandomJoke'
    parameters: []
    responses:
        '200':
        description: successful operation
        content:
            application/json:
            schema:
                "$ref": "#/components/schemas/JokeResponse"

  schemas:
    JokeResponse:
    type: object
    properties:
        sucess:
        type: boolean
        body:
        type: array
        items:
            $ref: '#/components/schemas/Joke'
```

DEMO

The NSwag project provides tools to generate client code from these OpenAPI specifications.



```
.
├── AutoGeneratedApiClient.DadJokes.csproj
├── Constants
│   └── ApiConstants.cs
├── Generated
│   ├── DadJokesApiClient.cs
│   └── Models.cs
├── HttpClientExtensions.cs
├── OpenAPI
│   ├── dad-jokes.nswag
│   └── dad-jokes.yml
└── ServiceCollectionExtensions.cs
```

Pros and Cons

- ✓ Based on the well-known specification
- ✓ May be integrated into CI/CD process
- ✓ Multi-language support
- ✓ Relatively easy to troubleshoot

- ✗ Hard to customize and control the contract of generated API Client
- ✗ Can't be applied without proper OpenAPI specification

THANK YOU!

Do you have any questions?

@nikiforovall

https://nikiforovall.github.io/