Desafio Técnico – Desenvolvimento de Cliente REST Genérico em .NET 9.0

1. Introdução

Este documento apresenta o desafio técnico proposto para candidatos e membros do time de engenharia de integração. O objetivo é guiar o desenvolvimento de um **cliente REST genérico** em C# com base na versão **.NET 9.0**, com foco em boas práticas de arquitetura de software, modularidade e capacidade de reutilização em projetos reais de integração de APIs.

O desafio também serve como **material educativo** para o time, permitindo o aprofundamento nos seguintes temas:

- Mecanismos de autenticação em APIs modernas (API Key, Bearer Token, OAuth 2.0)
- Estratégias de controle de vazão de requisições (rate limiting)
- Políticas de retry com backoff para resiliência e robustez
- Uso correto de `HttpClient` com `IHttpClientFactory` e `DelegatingHandlers`

Público-alvo

Este desafio é direcionado a:

- **Candidatos técnicos** em processos seletivos para vagas com foco em integrações, desenvolvimento de middlewares ou SDKs.
- **Desenvolvedores do time interno**, como forma de estudo, nivelamento técnico e aprofundamento em aspectos avançados de construção de clientes HTTP.

O que será avaliado

Os participantes serão avaliados com base nos seguintes critérios:

- Qualidade arquitetural da solução
- Clareza, organização e legibilidade do código
- Cobertura dos requisitos técnicos e funcionais
- Uso adequado de padrões e boas práticas
- Capacidade de extensão e manutenção da biblioteca
- Documentação, clareza de explicação e atenção a detalhes

Como este desafio será utilizado

- Como exercício de avaliação técnica em processos seletivos
- Como ponto de partida para decisões arquiteturais internas sobre consumo de APIs
- Como material de referência didática para novos integrantes do time

2. Visão Geral do Desafio

Neste desafio, você deverá desenvolver uma **biblioteca genérica para consumo de APIs REST**, com suporte a mecanismos comuns e fundamentais para comunicação segura, resiliente e eficiente com serviços externos. A biblioteca será construída em **C# sobre a plataforma .NET 9.0**, e deve ser adequada para uso em integrações profissionais.

🧩 Objetivo Técnico

Criar um cliente REST configurável, reutilizável e extensível, com as seguintes capacidades:

1. Autenticação Configurável

- O cliente deve permitir diferentes estratégias de autenticação via configuração externa:
  - **API Key** (via cabeçalho ou query string)
  - **Bearer Token** (token fixo)
  - **OAuth 2.0** com fluxo client credentials (incluindo **renovação automática de token**)

2. Rate Limiting

- A biblioteca deve aplicar **controle de vazão de requisições** baseado em um limite configurável, como por exemplo:
  - 60 requisições por minuto
- O rate limit deve ser respeitado inclusive em chamadas concorrentes.

3. Retry com Backoff

- Em caso de falhas transitórias (como status HTTP 429, 5xx ou exceções de rede), o cliente deve tentar novamente a requisição.
- O comportamento de retry deve:
  - Ser configurável
  - Usar backoff exponencial com ou sem jitter
  - Permitir definir o número máximo de tentativas

🛠️ Estrutura Esperada

A implementação deverá usar:

- **HttpClient** com `IHttpClientFactory` para gerenciamento adequado de instâncias
- **DelegatingHandlers** para encapsular as responsabilidades de autenticação, rate limit e retry
- Configuração externa padronizada (via JSON)
- Uma **interface comum e padronizada** para os métodos de chamada REST (detalhada no Capítulo 4)

🎯 O que se espera do resultado

O resultado deste desafio será uma biblioteca com:

- Uma interface clara e reaproveitável para consumir endpoints HTTP REST
- Comportamento configurável sem reescrever código
- Tratamento adequado de autenticação e falhas
- Uma arquitetura modular, com separação de responsabilidades
- Leitura clara e didática, pensada tanto para uso quanto para manutenção futura

👩‍💻 Estilo de Solução

A proposta deve refletir **boas práticas de engenharia de software**, como:

- Responsabilidade única por classe ou componente
- Uso de injeção de dependência
- Isolamento e reaproveitamento de lógica
- Padronização e clareza no código

Este projeto é mais do que uma prova de conceito. Ele deve ser visto como uma **base para bibliotecas reais**, com qualidade de engenharia e capacidade de evolução.

3. Requisitos Técnicos

Este capítulo detalha as **tecnologias obrigatórias**, **restrições**, **liberdades técnicas permitidas** e **boas práticas recomendadas** para a implementação da biblioteca REST genérica.

✅ Tecnologias Obrigatórias

Item	Requisito
Plataforma .NET	**.NET 9.0**
Linguagem	**C#** (versão 10 ou superior)
Cliente HTTP	`HttpClient` (com uso de `IHttpClientFactory`)
Configuração	Arquivo JSON externo (ou via `IOptions`)
Formato de resposta	JSON (serialização e desserialização com System.Text.Json ou equivalente nativo)

📚 Bibliotecas e Recursos Permitidos

O uso das seguintes ferramentas e bibliotecas está **permitido**, desde que bem justificado e usado com critério:

Categoria	Exemplos Permitidos

Retry/Backoff	Polly (altamente recomendado)
Serialização	`System.Text.Json`, `Newtonsoft.Json`
Logging	`ILogger`, `Serilog` (opcional)
Testes	xUnit, NUnit, MSTest
Injeção de Dependência	`Microsoft.Extensions.DependencyInjection`

> ❗ Bibliotecas de terceiros que substituam o papel do `HttpClient` (ex: RestSharp, Refit) **não devem ser usadas**, pois o objetivo é desenvolver um cliente HTTP genérico de forma mais próxima do metal.

🔗 Integração e Modularidade

A biblioteca deve ser pensada para uso modular e integrável a outros sistemas. Para isso:

- As configurações devem ser injetadas via `IOptions<T>`, `ConfigurationManager` ou similar.
- Os comportamentos (auth, rate limit, retry) devem estar separados por responsabilidade e serem facilmente substituíveis ou estendidos.
- Os métodos principais da biblioteca devem ser acessados por meio de uma **interface comum** (`IRestClient`), conforme definido no Capítulo 4.

👷 Boas Práticas Esperadas

Prática	Espera-se que o candidato...
Separação de responsabilidades	Encapsule lógica de autenticação, retry e rate limit em handlers separados.
Reuso e extensibilidade	Modele os componentes para permitir substituições futuras.
Injeção de dependência	Utilize construtores e serviços para injetar dependências como `HttpClient`, configurações e provedores.
Tratamento de exceções	Diferencie falhas transitórias de falhas permanentes, e exponha erros úteis.
Código limpo e legível	Use nomenclatura clara, evite duplicações e mantenha consistência.
Testabilidade	Estruture o código para permitir escrita de testes automatizados, mesmo que os testes não sejam entregues.

⚠️ Restrições Importantes

- **Não utilizar SDKs prontos ou wrappers de APIs específicas**.

- **Não utilizar bibliotecas que abstraem completamente o `HttpClient`**, pois o objetivo é exercitar sua correta aplicação.
- **A biblioteca deve funcionar com múltiplas estratégias de autenticação e configuração** sem exigir reescrita de código.

4. Interface Padrão

Todas as implementações do cliente REST genérico devem expor uma **interface pública padronizada** que defina as principais operações de consumo de APIs REST. Essa interface servirá como ponto único de entrada para os consumidores da biblioteca, abstraindo os detalhes de autenticação, configuração, controle de taxa e políticas de retry.

🧩 Finalidade da Interface

- Promover **padronização** entre diferentes implementações
- Garantir **interoperabilidade** para testes comparativos
- Abstrair complexidades de infraestrutura da lógica de negócio
- Facilitar integração com outras partes da aplicação

✅ Interface Obrigatória: `IRestClient`

```
public interface IRestClient
{
    Task<T?> GetAsync<T>(
        string endpoint,
        CancellationToken cancellationToken = default);

    Task<TResponse?> PostAsync<TRequest, TResponse>(
        string endpoint,
        TRequest body,
        CancellationToken cancellationToken = default);

    Task<TResponse?> PutAsync<TRequest, TResponse>(
        string endpoint,
        TRequest body,
        CancellationToken cancellationToken = default);

    Task DeleteAsync(
        string endpoint,
        CancellationToken cancellationToken = default);
}
```

Regras gerais:

- Todos os métodos devem ser **assíncronos** (`Task`-based)

- O `endpoint` deve ser **relativo à `BaseUrl`** definida na configuração
- O parâmetro `CancellationToken` deve ser aceito em todos os métodos
- A serialização e desserialização devem seguir padrão JSON
- Os métodos devem respeitar as políticas ativas de autenticação, retry e rate limit

---

🔁 Semântica dos Métodos

Método	Finalidade
`GetAsync<T>`	Requisição HTTP GET e desserialização da resposta para tipo `T`
`PostAsync<TReq, TRes>`	Envia JSON no corpo via HTTP POST e retorna `TRes`
`PutAsync<TReq, TRes>`	Envia JSON via HTTP PUT e retorna `TRes` (se houver)
`DeleteAsync`	Executa HTTP DELETE no endpoint especificado

🧱 Possíveis Extensões Futuras (Opcional)

Implementações que desejarem podem expandir a interface para incluir:

- `PatchAsync<TRequest, TResponse>()`
- Métodos com headers customizáveis
- Envio de arquivos (multipart/form-data)
- Versão com `HttpResponseMessage` bruto para casos especiais

Essas extensões devem ser feitas **fora da interface padrão**, via interfaces adicionais ou métodos de conveniência.

---

5. Formato de Configuração Padrão

A biblioteca REST desenvolvida neste desafio deve ser **totalmente configurável** por meio de um arquivo externo. Esse arquivo será responsável por fornecer as informações necessárias para ajustar o comportamento do cliente REST sem exigir mudanças no código.

📃 Exemplo de Configuração (JSON)

```
{
  "ApiClient": {
    "BaseUrl": "https://api.exemplo.com/",
    "Authentication": {
      "Type": "OAuth2",
      "ClientId": "abc123",
      "ClientSecret": "xyz456",
      "TokenEndpoint": "https://auth.exemplo.com/oauth2/token",
      "ApiKey": "minha-chave-api",
      "ApiKeyHeader": "x-api-key"
    },
    "RateLimit": {
      "Enabled": true,
      "RequestsPerMinute": 60
    },
    "Retry": {
      "Enabled": true,
      "MaxRetries": 3,
      "BaseDelayMilliseconds": 500,
      "UseExponentialBackoff": true
    }
  }
}
```

> 🔁 A configuração acima cobre os três pilares da biblioteca: autenticação, controle de taxa (rate limit) e retry.

📑 Dicionário de Campos

Seção: `ApiClient`

Campo	Tipo	Obrigatório	Descrição
`BaseUrl`	`string`	✅	URL base da API externa. Todas as requisições partirão deste endereço. Deve terminar com `/` ou considerar concatenação correta nos métodos.

Seção: `ApiClient.Authentication`

Campo	Tipo	Obrigatório	Descrição
`Type`	`string`	✅	Define o tipo de autenticação. Valores suportados: `ApiKey`, `Bearer`, `OAuth2`.
`ApiKey`	`string`	Condicional	Chave da API. Obrigatório se `Type` for `ApiKey`.
`ApiKeyHeader`	`string`	Condicional	Nome do cabeçalho onde a `ApiKey` será enviada. Se omitido, pode ser enviado via query string.

`ClientId`	`string`	Condicional	Identificador do cliente OAuth2. Obrigatório se `Type` for `OAuth2`.
`ClientSecret`	`string`	Condicional	Segredo do cliente OAuth2. Obrigatório se `Type` for `OAuth2`.
`TokenEndpoint`	`string`	Condicional	Endpoint do servidor de autenticação para obtenção do `access_token`. Obrigatório se `Type` for `OAuth2`.

Seção: `ApiClient.RateLimit`

Campo	Tipo	Obrigatório	Descrição
`Enabled`	`bool`	✅	Indica se o controle de taxa estará ativado.
`RequestsPerMinute`	`int`	Condicional	Número máximo de requisições permitidas por minuto. Obrigatório se `Enabled` for `true`.

Seção: `ApiClient.Retry`

Campo	Tipo	Obrigatório	Descrição
`Enabled`	`bool`	✅	Indica se a lógica de retry estará ativa.
`MaxRetries`	`int`	Condicional	Número máximo de tentativas após falhas transitórias. Obrigatório se `Enabled` for `true`.
`BaseDelayMilliseconds`	`int`	Condicional	Tempo inicial de espera entre tentativas, em milissegundos.
`UseExponentialBackoff`	`bool`	Opcional	Se `true`, o tempo de espera aumentará exponencialmente. Se `false` (ou omitido), usará delay fixo.

🛡️ Validações Obrigatórias

- O campo `Authentication.Type` deve corresponder a um valor suportado (`ApiKey`, `Bearer`, `OAuth2`). Qualquer valor inválido deve gerar erro claro.
- Campos condicionais devem ser validados com base no tipo de autenticação ou nas flags de ativação (`Enabled`).
- O cliente não deve iniciar com configurações faltantes ou inválidas.

🧠 Observações

- A configuração pode ser lida via `IOptions<ApiClientOptions>` ou deserializada diretamente, a critério do desenvolvedor.

- A estrutura deve ser preparada para **adicionar novos campos** no futuro, sem quebrar a compatibilidade.
- O cliente deve se comportar corretamente mesmo com múltiplas instâncias usando configurações diferentes (ex: diferentes endpoints ou tokens).
- Se desejar, o desenvolvedor pode permitir **sobrescrita programática** das configurações no código para facilitar testes.

6. Orientações para Implementação

Este capítulo oferece orientações práticas e técnicas para estruturar a solução de forma modular, testável e aderente às boas práticas da plataforma .NET. Não se trata de um roteiro obrigatório, mas de um conjunto de recomendações que servirão como referência técnica para a construção do cliente REST genérico.

🧱 Arquitetura Recomendada

A biblioteca deve ser composta por camadas separadas por responsabilidade:

```
IRestClient
   │
   └── HttpClient (via IHttpClientFactory)
         └── DelegatingHandlers:
               ├── AuthenticationHandler
               ├── RateLimitHandler
               └── RetryHandler
```

Esse padrão permite encapsular o comportamento de cada funcionalidade em componentes independentes e reaproveitáveis.

📦 Organização de Projeto (Sugerida)

```
RestClient/
│
├── Core/
│   ├── IRestClient.cs
│   └── RestClient.cs
│
├── Auth/
│   ├── IAuthProvider.cs
│   ├── ApiKeyAuthProvider.cs
│   ├── BearerTokenProvider.cs
│   └── OAuth2AuthProvider.cs
│
├── Handlers/
```

```
|   ├── AuthenticationHandler.cs
|   ├── RateLimitHandler.cs
|   └── RetryHandler.cs
|
├── Configuration/
|   └── ApiClientOptions.cs
|
└── SampleApp/ (opcional para demonstração de uso)
```

🔐 Autenticação

A estratégia de autenticação deve ser **abstraída via interface** para que seja possível trocar a implementação facilmente:

```
public interface IAuthProvider
{
    Task<string?> GetAccessTokenAsync();
}
```

A `AuthenticationHandler` usará essa interface para injetar os headers de autenticação nas requisições.

Dicas:

- No caso de OAuth2, armazene o token em memória com controle de expiração.
- No caso de Bearer token fixo, o provider pode simplesmente retornar o valor diretamente.
- No caso de API Key, o handler pode injetar diretamente o header ou alterar a URL (query string).

⏱️ Rate Limiting

Implemente o controle de requisições utilizando `SemaphoreSlim`, `Timers` ou outras estruturas de sincronização.

Dicas:

- Uma opção simples é usar uma janela deslizante (ex: N requisições a cada minuto).
- Evite soluções globais que conflitam com instâncias paralelas do cliente.
- Pode-se optar por usar bibliotecas auxiliares como Polly + Token Bucket (opcional).

🔁 Retry com Backoff

Use uma política configurável de retry que trate:

- Códigos de status 429 e 5xx
- Exceções transitórias (como timeouts ou falha de DNS)

Dicas:

- Utilize Polly com `WaitAndRetryAsync` (com ou sem jitter)
- Leia os headers de `Retry-After` quando disponíveis
- Permita configuração de quantidade máxima de tentativas e tempo inicial

---

🧰 HttpClient + DelegatingHandlers

Utilize `IHttpClientFactory` para registrar o cliente REST com os handlers necessários:

```
services.AddHttpClient<IRestClient, RestClient>()
    .AddHttpMessageHandler<AuthenticationHandler>()
    .AddHttpMessageHandler<RateLimitHandler>()
    .AddHttpMessageHandler<RetryHandler>();
```

Garanta que os handlers sejam registrados corretamente no pipeline e injetados com suas dependências.

📄 Configuração

A biblioteca deve ler as configurações do arquivo JSON padronizado descrito no Capítulo 5, preferencialmente via `IOptions<ApiClientOptions>`.

Dicas:

- Mantenha uma classe de modelo (`ApiClientOptions.cs`) refletindo a estrutura do JSON.
- A configuração pode ser reusada por múltiplas instâncias, se necessário.
- Considere validações antecipadas das configurações no startup.

🧪 Testabilidade

Mesmo que não sejam exigidos testes automatizados como parte da entrega, a arquitetura deve permitir:

- Substituição dos handlers por fakes ou mocks
- Testes unitários dos providers de autenticação

- Simulação de falhas e limites de taxa
- Testes com endpoints simulados (ex: `httpbin.org`)

🧩 Modularidade e Extensibilidade

Construa a solução como se ela fosse uma **biblioteca interna reutilizável**. Por isso:

- Use interfaces para permitir substituições
- Evite código acoplado diretamente ao app de demonstração (caso exista)
- Não exponha implementações concretas para o consumidor da biblioteca
- Mantenha os métodos públicos simples e seguros para uso direto

7. Entrega Esperada

Este capítulo descreve os itens obrigatórios e opcionais que devem compor a entrega final do desafio. O foco está em garantir **clareza**, **reprodutibilidade** e **padronização** para facilitar a avaliação e comparação entre diferentes soluções.

📂 Estrutura Esperada do Projeto

A entrega deverá conter:

```
Root/
│
├── RestClient/               → Código-fonte da biblioteca
│
├── SampleApp/ (opcional)     → Projeto de demonstração de uso
│
├── README.md                 → Documento explicando a solução
│
├── appsettings.json          → Exemplo funcional de configuração
│
└── .sln ou .csproj           → Solução pronta para build
```

✅ Itens Obrigatórios

Item	Descrição
Biblioteca funcional	A implementação completa do cliente REST, com os requisitos definidos nos capítulos anteriores.

Interface `IRestClient`	Interface com os métodos obrigatórios definidos no Capítulo 4.
Leitura de configuração externa	Utilização do formato padrão de configuração em JSON conforme Capítulo 5.
Encapsulamento modular	Código organizado por responsabilidade: autenticação, retry, rate limit e chamadas HTTP.
README.md	Deve conter: visão geral do projeto, instruções de execução, descrição da arquitetura e explicações dos conceitos aplicados.
Exemplo de configuração (`appsettings.json`)	Com todos os campos exigidos no desafio, ajustados para uma API simulada, de teste ou pública.

🎁 Itens Opcionais (Diferenciais)

Item	Benefício
✅ **Testes automatizados**	Demonstra domínio de testabilidade e atenção à qualidade.
✅ **Projeto de exemplo (SampleApp)**	Mostra como usar a biblioteca com casos reais ou simulados.
✅ **Logs estruturados**	Facilita o entendimento do fluxo de execução e problemas durante requisições.
✅ **Documentação adicional**	Diagramas, comentários no código e observações de design enriquecem a entrega.

📤 Como entregar

Você pode escolher uma das seguintes formas de entrega:

1. **Repositório público (GitHub, GitLab, etc.)**
   - Recomendado: permite versionamento, documentação online e facilidade de execução
2. **Arquivo `.zip` com a solução**
   - Deve conter todos os arquivos necessários para execução
   - Evite arquivos binários, pastas `bin/` e `obj/`

🧪 Validação da Entrega

A solução deve estar pronta para ser:

- Restaurada (`dotnet restore`)
- Compilada (`dotnet build`)
- Executada (em caso de projeto de demonstração)

  Observação: se for necessária alguma dependência ou passo adicional, isso deve estar documentado no `README.md`.

8. Anexos

Este capítulo reúne exemplos, recursos visuais e materiais complementares que reforçam a compreensão do desafio, servindo como guia para consulta rápida durante o desenvolvimento.

📘 A. Exemplo Completo de Configuração (`appsettings.json`)

```
{
  "ApiClient": {
    "BaseUrl": "https://api.dadospublicos.gov.br/",
    "Authentication": {
      "Type": "OAuth2",
      "ClientId": "seu-client-id",
      "ClientSecret": "seu-client-secret",
      "TokenEndpoint": "https://api.dadospublicos.gov.br/oauth2/token"
    },
    "RateLimit": {
      "Enabled": true,
      "RequestsPerMinute": 60
    },
    "Retry": {
      "Enabled": true,
      "MaxRetries": 4,
      "BaseDelayMilliseconds": 500,
      "UseExponentialBackoff": true
    }
  }
}
```

📊 B. Diagrama: Pipeline de Handlers

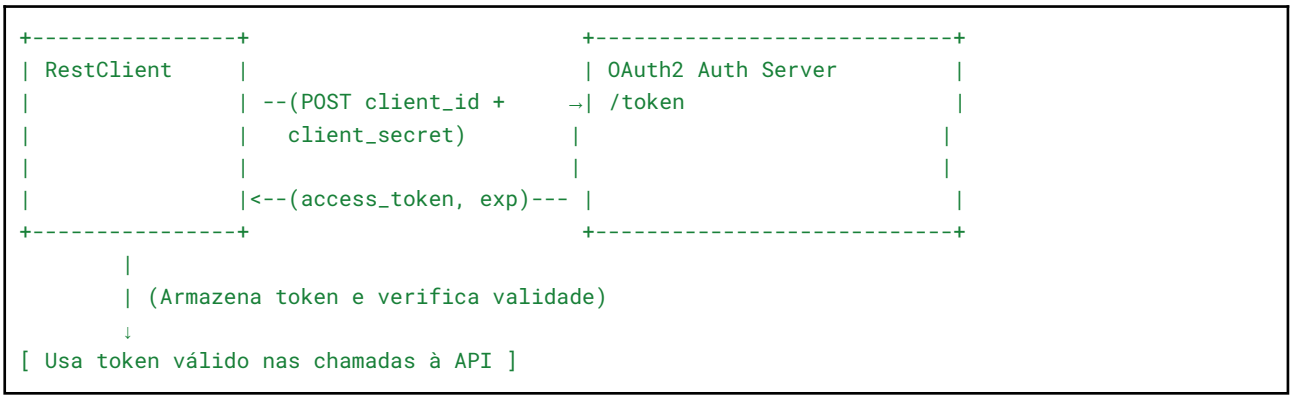
```
Request → [ RetryHandler ]
        → [ RateLimitHandler ]
        → [ AuthenticationHandler ]
        → [ HttpClient ]
```

```
        → Internet
```

Cada handler atua de forma independente e compõe a cadeia de execução do `HttpClient`. Essa arquitetura permite:

- **Testes isolados**
- **Substituição modular**
- **Baixo acoplamento e alta coesão**

🔄 C. Fluxo: Autenticação OAuth2 Client Credentials



Considerações:

- Tokens devem ser renovados **antes da expiração real** (ex: 1 minuto antes).
- Tokens podem ser compartilhados entre múltiplas chamadas simultâneas (use cache in-memory com segurança).
- A lógica de obtenção e renovação do token deve estar **totalmente encapsulada no AuthProvider**.

🛠️ D. Exemplos de Código Simulado (pseudocódigo)

Injeção do cliente REST

```
services.AddHttpClient<IRestClient, RestClient>()
    .AddHttpMessageHandler<AuthenticationHandler>()
    .AddHttpMessageHandler<RateLimitHandler>()
    .AddHttpMessageHandler<RetryHandler>();
```

Uso do cliente

```
var user = await _restClient.GetAsync<UserDto>("users/123");
```

Retry com Polly (modelo)

```
Policy
    .Handle<HttpRequestException>()
    .OrResult<HttpResponseMessage>(r => r.StatusCode == HttpStatusCode.TooManyRequests)
    .WaitAndRetryAsync(3, retryAttempt =>
        TimeSpan.FromMilliseconds(500 * Math.Pow(2, retryAttempt)));
```

📚 E. Recursos de Referência

- Documentação oficial do IHttpClientFactory
- Polly – Biblioteca de Resiliência para .NET
- OAuth 2.0 Client Credentials Flow (RFC 6749)
- Rate Limiting Explained (Cloudflare)