`EasyRESTClient`: Uma proposta para abstração de comunicações entre microsserviços .NET

Guilherme Ramos da Silva

Orientador: Ricardo Terra
Coorientadora: Elena A. Araujo

Universidade Federal de Lavras, Lavras, Brasil

guilherme.ramos@sistemas.ufla.br

Abstract. *The microservice architecture consists of a set of independent services that execute well-defined functionalities, allowing each microservice to be developed in different programming languages, use different frameworks and be hosted on different servers. In order to provide tools for the development of microservices in the Java language, the Spring Cloud Netflix project provides a set of frameworks and libraries to solve common problems in scalable distributed systems. However, for the most part, these services cannot be used by applications developed in languages other than Java, such as C#. In this scenario, the present article proposes an approach to communications among .NET. This approach consists of a solution that allows to abstract the definitions of communication among microservices and to carry out the routing of such requisitions. In addition, the present work also proposes a C# code refactoring tool for the context of the proposed abstractions.*

Resumo. *A arquitetura de microsserviços é composta por um conjunto de serviços independentes que executam funcionalidades bem definidas, permitindo que cada microsserviço seja desenvolvido em diferentes linguagens de programação, utilize diferentes frameworks e esteja hospedado em diferentes servidores. Com o objetivo de fornecer ferramentas para o desenvolvimento de microsserviços na linguagem Java, o projeto Spring Cloud Netflix disponibiliza um conjunto de frameworks e bibliotecas para resolver os problemas comuns em sistemas distribuídos escaláveis. Contudo, em sua maioria, esses serviços não podem ser utilizados por aplicações desenvolvidas em linguagens diferentes de Java, tal como C#. Diante desse cenário, o presente artigo propõe uma abordagem de abstração das comunicações entre microsserviços .NET que consiste em uma solução que permite abstrair as definições de comunicação entre microsserviços e realizar o encaminhamento de tais requisições. Além disso, o presente trabalho propõe também uma ferramenta de refatoração de código C# para o contexto das abstrações propostas.*

1. Introdução

A Arquitetura de Microsserviços surgiu recentemente como uma promessa de estilo arquitetural que visa promover a reutilização e facilitar a manutenção de aplicações por meio de serviços [Di Francesco 2017]. Esse estilo arquitetural compreende à uma variação da Arquitetura Orientada a Serviços (SOA) a nível de desenvolvimento e implantação de aplicações distribuídas [Zimmermann 2016]. Newman reforça ainda que a arquitetura de microsserviços surgiu com o objetivo de melhor compreender e aplicar as práticas propostas por SOA, adotado por empresas como Netflix e Amazon [Newman 2015].

Fowler e Lewis definem a arquitetura de microsserviços como uma abordagem para desenvolver uma única aplicação como uma suíte de serviços independentes, cada um rodando em seu próprio processo e se comunicando por meio de mecanismos leves através de uma API que utiliza o protocolo HTTP [Fowler and Lewis 2014]. Os microsserviços são caracterizados por serem entidades autônomas e distribuídas, podendo estar hospedados em diferentes servidores, serem implementados em diferentes linguagens de programação, utilizar diferentes *frameworks* e serem gerenciados por persistência de dados poliglota (utilização de múltiplos bancos de dados na mesma aplicação) [Araujo et al. 2017].

Com o objetivo de fornecer ferramentas para o desenvolvimento de aplicações distribuídas e que sigam as premissas da arquitetura de microsserviços, diversos *frameworks* e bibliotecas têm sido propostos pelo projeto Spring Cloud Netflix [Pivotal 2017]. Esse projeto fornece uma integração entre o Spring Boot e o projeto Netflix a fim de resolver problemas comuns em sistemas distribuídos escaláveis, tais como *(i)* descoberta de serviços através do microsserviço `Eureka` [Netflix 2018b], roteamento de requisições por meio do serviço `Zuul` [Gonigberg 2018] e *(iii)* criação de clientes web declarativos através do cliente `Feign` [Netflix 2018a]. Por meio do cliente `Feign`, é possível habilitar e configurar aplicações através de anotações a fim de integrar os serviços que compõe a aplicação. Contudo, tais definições são disponíveis apenas para serviços implementados na linguagem Java, tornando sua utilização inacessível a microsserviços desenvolvidos em outras linguagens, tal como C#.

Com o objetivo de aplicar os conceitos de abstração de requisições encontrados na linguagem Java para a linguagem C#, foram realizadas buscas na literatura sobre abordagens que disponibilizassem tais recursos. Porém, não foram encontradas até o presente momento registros de ferramentas que propusessem propostas de tais abstrações para o ambiente de desenvolvimento em .NET.

Nesse contexto, o presente trabalho apresenta a abordagem `EasyRESTClient`, a qual visa fornecer uma abstração para criação de clientes web declarativos para os microsserviços implementados na linguagem C# de maneira rápida e fácil. Para isso, inicialmente, o código fonte de cada microsserviço que compõe a aplicação, como apresenta a Figura 1a, é analisado e todas as requisições do tipo `HttpClient` são mapeadas. Posteriormente, o código fonte onde tais definições foram mapeadas é refatorado por meio da ferramenta proposta `RefactoryEasyClient` (Figura 1b), gerando uma abstração para a realização de comunicações entre microsserviços, definida como `EasyRESTClient` (Figura 1c). Por fim, quando os serviços da aplicação forem executados, a ferramenta `EasyRESTClient` será responsável pelo encaminhamento das requisições entre serviços (Figura 1d).

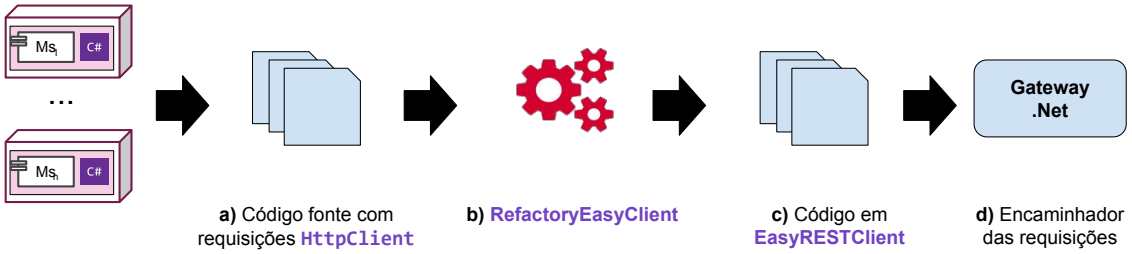

Figura 1. Exemplo da abordagem proposta.

O restante deste trabalho está organizado da seguinte maneira. A Seção 2 introduz os conceitos fundamentais para o entendimento deste trabalho, incluindo as ferramentas relacionadas. A Seção 3 descreve a abordagem proposta. A Seção 4 descreve uma avaliação controlada para critérios de validação da abordagem proposta e, por fim, a Seção 5 conclui o trabalho.

2. *Background*

A descoberta de serviço é uma proposta para aumentar a reutilização dos microsserviços disponíveis e por facilitar encontrar o endereço onde estão hospedados. O padrão de descoberta de serviços exige que na arquitetura de microsserviços tenha algum meio que centralize e realize a pesquisa dos locais dos serviços, ou seja, esse mecanismo deve entregar às aplicações clientes o endereço em que os microsserviços estejam hospedados. Nesse contexto, essa abordagem é denominada como descoberta de serviços.

2.1. Descoberta de serviços

A localização de aplicações que se encontram na nuvem pode não ser conhecida estaticamente pelos clientes em tempo de execução. Essa característica é reflexo da dinamicidade causada pela possibilidade das aplicações serem escaladas e realocadas em diferentes servidores [Balalaie et al. 2016]. Com o intuito de solucionar tais desafios, mecanismos para a descoberta de serviços têm sido propostos. Montesi e Weber apresentam duas formas de solucionar esse problema por meio da descoberta de serviço do lado do cliente e do lado do servidor [Montesi and Weber 2016].

A Figura 2 ilustra a abordagem de descoberta de serviço do lado do cliente. Inicialmente, cada microsserviço deve se registrar no serviço de descoberta para que ele mantenha as informações de cada serviço. O cliente, por sua vez, comunica-se diretamente com o serviço de descoberta a fim de obter as informações do serviço desejado e realizar a requisição.

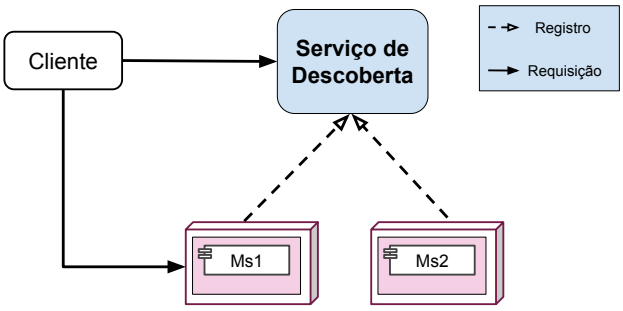

Figura 2. Descoberta de serviço do lado do cliente. Adaptado de [Montesi and Weber 2016].

Já na abordagem de descoberta no lado do servidor, ilustrado na Figura 3, um intermediador encaminha as requisições do cliente ao serviço de descoberta. Nesse tipo de arquitetura, toda vez que um cliente solicitar informações de um determinado serviço, a requisição é direcionada ao encaminhador. O encaminhador, por sua vez, realiza a comunicação com o serviço de descoberta para obter informações do microsserviço desejado.

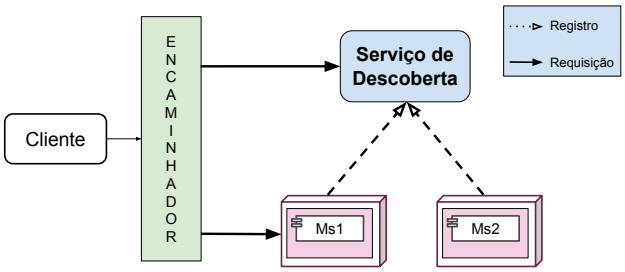

Figura 3. Descoberta de serviço do lado do servidor. Adaptado de [Montesi and Weber 2016].

O encaminhador é uma aplicação que atua como um único ponto de entrada, sendo responsável pelo roteamento, composição e conversão do protocolo de requisição, centralizando as requisições realizadas pelas aplicações clientes e encaminhando as chamadas para os microsserviços adequados [Canaver 2017]. Centralizar as lógicas de requisição no encaminhador isenta o aplicativo cliente de implementar sua lógica de requisição ao serviço de descoberta, porém exige que o aplicativo faça a configuração do ambiente do encaminhador.

Com o objetivo de prover serviços de descoberta por meio de nomes, o projeto Netflix propõe o microsserviço Eureka. Para realizar a comunicação com o Eureka, o projeto Pivotal *Cloud Foundry*[1] disponibiliza a biblioteca Steeltoe para a plataforma .NET.

2.1.1. Descoberta de serviços por meio do Eureka

O microsserviço Eureka é um serviço fornecido pelo projeto Netflix para descoberta de serviços na nuvem baseado em requisições REST que fornece além da descoberta, o serviço de balanceamento de cargas e de tolerância a falhas [Netflix 2014]. O Eureka atua

[1] O Cloud Foundry é uma plataforma de software livre que oferece a opção de nuvens, estruturas de desenvolvimento e serviços de aplicativo [Pivotal 2018b].

como um serviço de nomes, ou seja, cada microsserviço registrado é autenticado com um nome através de suas configurações. Dessa maneira, é possível que cada microsserviço possua diferentes instâncias e seja acessado apenas com a informação do nome registrado.

2.1.2. Descoberta de serviços por meio do Steeltoe

Steeltoe é uma biblioteca de código fonte aberto que fornece integração entre microsserviços .NET e os *frameworks* disponibilizados pelo projeto Spring Cloud Netflix [Pivotal 2018a]. Os *frameworks* disponibilizados incluem descoberta de serviços (através do microsserviço Eureka), gerenciamento de configuração (por meio do serviço *Config Server*) e padrão *circuit braker* (por meio do serviço Hystrix). O padrão de descoberta de serviços utilizando o Steeltoe é similar ao uso do *framework* Spring e o microsserviço Eureka. Ao registrar cada microsserviço no Eureka utilizando o Steeltoe, é necessário que cada aplicação possua um arquivo de configuração único em seu diretório raiz, o qual é identificado como `appsettings` no formato JSON.

Na Listagem 1, é possível identificar algumas propriedades de configuração que o Steeltoe utiliza para preparar o ambiente de microsserviços em .NET. A propriedade principal é a identificação do nome do microsserviço no Eureka, definido como `Ms1` (linha 4). Posteriormente, as propriedades de identificação do servidor Eureka são definidas (linhas 6 a 9). Por fim, a identificação da localização desta instância do microsserviço é definida, tendo como IP e porta os valores descritos nas linhas 11 e 12, respectivamente. É válido ressaltar que o arquivo de configuração pode possuir diversas propriedades, não limitando apenas as apresentadas na listagem abaixo.

```
1  {
2    "spring": {
3      "application": {
4        "name": "Ms1" }
5    },
6    "eureka": {
7      "client": {
8        "serviceUrl": "http://192.168.254.21:8761/eureka/" }
9    },
10   "instance": {
11     "hostname": "52.201.200.150",
12     "port": 5001 }
13 }
```

Listagem 1. Arquivo de Configuração Steeltoe.

Para que os microsserviços possam se registrar e se autenticar no ambiente Eureka, é necessário inicializar antes a biblioteca Steeltoe. Para isso, é apresentada na Listagem 2, duas das maneiras definidas pelo projeto .NET Core para inicialização da biblioteca Steeltoe. A classe `Startup` (classe iniciadora em aplicações web em .NET Core) possui dois métodos de configurações: `ConfigureServices` e o `Configure` (linhas 2 e 6, respectivamente). O método `ConfigureServices` é opcional e é executado antes do método `Configure`. Nele é possível adicionar serviços dentro do *container* de serviços e disponibiliza-los dentro do método `Configure`, o qual é definido com o intuito de especificar a maneira como o aplicativo responderá às solicitações HTTP [Smith et al. 2018].

```
1  public class Startup{
2     public void ConfigureServices(IServiceCollection services){
3        services.AddDiscoveryClient(Configuration);
4        ...
5     }
6     public void Configure(IApplicationBuilder app, IHostingEnvironment env){
7        ...
8        app.UseDiscoveryClient();
9     }
10 }
```

Listagem 2. Inicialização da descoberta de serviços em projetos Steeltoe.

2.2. Microsserviços em .NET Core

O cenário para aplicar a arquitetura de microsserviços em .NET Core se apresenta muito favorável, pois é possível encontrar na própria plataforma de desenvolvimento ferramentas nativas que auxiliam o processo de codificação de aplicações em microsserviços. Para que uma aplicação cliente realize requisições em um microsserviço, é necessário apenas uma requisição HTTP para o endereço do serviço desejado.

A Listagem 3 apresenta como é realizada a comunicação entre microsserviços por meio de requisições HTTP padrões. Inicialmente, um objeto do tipo `HttpClient` deve ser definido a fim de representar a requisição a ser realizada (linha 3). Essa requisição é executada em um método assíncrono estabelecido pelo indicador `async` (linha 5). Posteriormente a requisição HTTP é realizada através do método `GetAsync` (linha 8) para a `url` desejada (linha 7). Após a verificação do *status* da requisição, o resultado da chamada ao microsserviço `Ms2` é obtido (linha 11).

```
1  class ProgramMs1 : Controller
2  {
3     static HttpClient clientHttp = new HttpClient();
4
5     public async Task<ClientMs2> GetMs2Info(...)
6     {
7        string url = "http://192.168.254.21:8080/Foo";
8        HttpResponseMessage response = await clientHttp.GetAsync(url);
9        if (response.IsSuccessStatusCode)
10       {
11          return await response.Content.ReadAsAsync<ClientMs2>();
12       }
13       ...
14    }
15    ...
16 }
```

Listagem 3. Comunicação por meio de requisições HTTP padrão.

Como pode ser observado, toda a definição da requisição HTTP deve ser descrita em muitos detalhes, tornando-a extensa e repetitiva, em casos em que chamadas a microsserviços são frequentes. Como alternativa para abstrair e simplificar toda a definição de requisições HTTP na linguagem C#, foi implementada uma biblioteca para codificação desse tipo de cliente definida como `EasyRESTClient`. Essa biblioteca é baseada no cliente `REST` declarativo `FeignClient`, microsserviços *Zuul* e *Eureka*, definidos pelo projeto Spring Cloud Netflix [Netflix 2018c].

2.3. Microsserviços em Java com `Feign Client`

A abordagem `EasyRESTClient` é baseada no cliente web declarativo *Feign Client*, disponibilizado no *framework* Spring Cloud e microsserviço *Zuul* do projeto Net-

flix [Netflix 2018a, Gonigberg 2018], ambos disponibilizados para aplicações desenvolvidas na linguagem Java.

Feign Client é um serviço que disponibiliza a criação de clientes web de maneira simples que, em conjunto com o microsserviço *Zuul*, fornecem um cliente HTTP com os recursos de encaminhamento e balanceamento de carga [Netflix 2018a]. Por meio do uso de anotações e de interfaces, o *Feign Client* realiza as requisições de comunicação entre microsserviços [Netflix 2018c]). O *Zuul* atua como encaminhador de requisições para a comunicação dos serviços disponibilizados pelo projeto Netflix, garantindo segurança no encaminhamento das requisições e permitindo roteamento dinâmico, monitoramento e resiliência em suas chamadas [Pivotal 2018c].

A Listagem 4 apresenta como é realizada a comunicação entre os microsserviços utilizando o cliente *Feign*. Inicialmente é necessário declarar uma interface de comunicação *@FeignClient*, informando qual o nome do microsserviço (*Ms2*) que se deseja realizar a comunicação (linha 1). Posteriormente são definidos os métodos de acesso a essas informações, por exemplo, através do método `getFoo()` (linha 4). Para cada método, é definida uma anotação *@RequestMapping* (linha 3). Nessa anotação, deve-se informar qual será a interface acessada em tal microsserviço (*"/foo"*).

```
1 @FeignClient("Ms2")
2  public interface Ms2Client {
3     @RequestMapping(method = RequestMethod.GET, value = "/foo")
4     List<String> getFoo();
5 }
```

Listagem 4. Interface utilizando notações Feign.

Na Listagem 5, é apresentada a utilização da interface `Ms2Client` (Listagem 4). Para utilizar essa interface de comunicação, é necessário declarar o objeto `ms2Client` (linha 4), injetando uma dependência por meio da anotação *@Autowired* (linha 3), para solicitar informações do microsserviço *Ms2*. Essa solicitação é invocada por meio do método `bar()` (linha 6), através das informações retornadas pelo método `getFoo()` (linha 7).

```
1  public class Ms1{
2     ...
3     @Autowired
4     private Ms2Client ms2Client;
5     ...
6     public String bar(){
7        return ms2Client.getFoo();
8     }
9     ...
10 }
```

Listagem 5. Realização da chamada ao microsserviço utilizando Feign.

É válido ressaltar que tais definições são disponíveis apenas para serviços implementados na linguagem Java, limitando a utilização das abstrações fornecidas pelo cliente *Feign* para microsserviços desenvolvidos em outras linguagens, tal como C#. Nesse contexto, foram realizadas pesquisas sobre aplicações que propusessem as abstrações definidas no cliente *Feign* para a linguagem C# na plataforma .NET. Contudo, não foi encontrada nenhuma abordagem ou ferramenta que aplicasse tais abstrações. Sendo assim, o presente trabalho propõe a abordagem `EasyRESTClient`, um cliente REST declarativo para microsserviços implementados na linguagem C#.

3. Abordagem proposta

Com o objetivo de fornecer aos desenvolvedores um mecanismo para definição de cliente REST declarativo e encaminhador de requisições entre microsserviços, é proposta a abordagem `EasyRESTClient`, automatizada pela ferramenta `EasyRESTClient`[2]. A Figura 4 apresenta como a abordagem realiza o encaminhamento das comunicações para microsserviços C#, cuja descrição pode ser vista na Seção 3.1. Como contribuição do trabalho, foi desenvolvida também uma ferramenta de refatoração de código para aplicações que realizam requisições HTTP padrão para a abstração `EasyRESTClient`, a qual é descrita na Seção 3.2 .

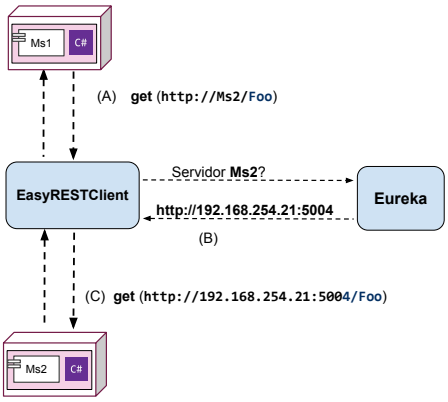

Figura 4. Abordagem `EasyRESTClient`.

3.1. `EasyRESTClient`

A abordagem `EasyRESTClient` é descrita em três etapas principais, definidas como: *(A)* Requisição do serviço cliente ao serviço desejado; *(B)* Descoberta da localização do serviço desejado; e *(C)* Resposta da execução do serviço desejado ao cliente solicitador.

A: Requisição do serviço cliente ao serviço desejado

Nessa etapa, quando requisições do tipo REST (métodos `get`, `put`, `post` ou `delete`) forem executadas em um microsserviço, a ferramenta proposta obtém a informação do nome do serviço desejado e sua respectiva interface. No exemplo apresentado na Figura 4, a ferramenta obtém *Ms2* como o nome do microsserviço desejado e `Foo` como a interface de comunicação. A abordagem `EasyRESTClient` estabelece a definição de uma interface do tipo `IMicroservice` (Listagem 6) e sua respectiva utilização (Listagem 7).

```
1 [MicroServiceHost("Ms2")]
2 public interface IMs2Service : IMicroService
3 {
4     [MicroService("Foo", TypeRequest.Get)]
5     Task<HttpResponseMessage> Foo(...);
6 }
```

Listagem 6. Comunicação por meio do `EasyRESTClient` - Interface.

[2]Disponível em: https://github.com/PqES/EasyRESTClient/

A interface a ser definida (`IMs2Service`, linha 2) deve ser anotada pelo tipo `MicroserviceHost`, como apresenta a Listagem 6. A anotação `MicroserviceHost` define o nome do microsserviço a ser requisitado (*Ms2*, linha 1) . O tipo `IMicroservice` (linha 2) identifica que em tal interface será realizada uma injeção de dependência quando seus métodos forem invocados. Esses métodos são anotados pelo atributo `MicroService` (linha 4) e representam a interface de comunicação do microsserviço desejado.

A Listagem 7 apresenta a utilização da interface definida (`IMs2Service`) na classe `ProgramMs1` (linha 1). Inicialmente, um objeto do tipo `IMs2Service` é definido (linha 3). Posteriormente, no construtor da classe controladora é executada a injeção de dependência para o objeto criado (linhas 4 a 6). Por fim, para realizar a requisição a algum método definido na interface, basta apenas invocar o método desejado. No exemplo, o método `Foo` é invocado (linha 9).

```
1  class ProgramMs1 : Controller
2  {
3      private IMs2Service _ms2Service;
4      public ProgramMs1(IMs2Service ms2Service){
5          _ms2Service = ms2Service;
6      }
7      public async Task<ClientMs2> GetMs2Info(...){
8          ...
9          _ms2Service.Foo();
10         ...
11     }
12 }
```

Listagem 7. Comunicação por meio do `EasyRESTClient`.

B: Descoberta da localização do serviço desejado

Após definir qual o microsserviço é requisitado, a ferramenta `EasyRESTClient` realiza uma requisição ao serviço de descoberta *Eureka*, solicitando o endereço do serviço desejado. O *Eureka*, por sua vez, devolve `EasyRESTClient` endereço de alguma instância disponível.

C: Resposta da execução do serviço desejado

Com a localização e a interface de comunicação em memória, a ferramenta `EasyRESTClient` executa a requisição direta para o microsserviço desejado. Após o processamento do método que é anotado pela interface de comunicação desejada, a ferramenta `EasyRESTClient` retorna o resultado da execução para o serviço solicitador.

É perceptível que a abstração da abordagem `EasyRESTClient` diminui a verbosidade bem como a complexidade para a definição das requisições a outros serviços quando comparada as definições HTTP padrão (Listagem 3). Além disso, a abordagem proposta abstrai a instância do microsserviço a ser executada, uma vez que apenas o nome do serviço é considerado e não mais o respectivo endereço IP, como apresenta a linha 7 da Listagem 3. Nesse caso, fica a cargo do *Steeltoe* realizar o balanceamento de carga, decidindo qual instância do serviço será executada. É válido ressaltar que a proposta `EasyRESTClient` é totalmente independente de ferramentas externas, atuando como uma biblioteca das aplicações que desejam utilizar as abstrações de requisições REST na linguagem C#.

3.2. Ferramenta de refatoração

Outra contribuição deste trabalho compete à ferramenta `RefactoryEasyClient`[3] apresentada na Figura 5.

Essa ferramenta recebe como entrada o código fonte de cada microsserviço que compõe a aplicação. As informações presentes em cada classe do microsserviço são analisadas por meio de *Abstract Syntax Tree* (AST). Todos os nós representados pelos métodos `GetAsync`, `PostAsync`, `PutAsync` e `DeleteAsync` são analisados e suas informações são armazenadas.

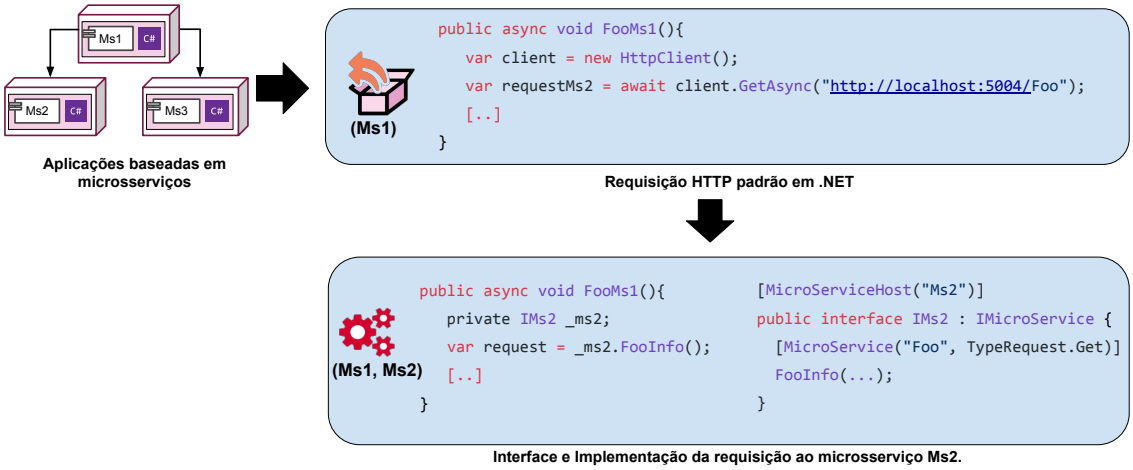

Figura 5. Ferramenta `RefactoryEasyClient` de refatoração proposta.

A ferramenta `RefactoryEasyClient` gera os arquivos que representam a interface do tipo `IMicroservice`. Na interface definida, as informações do nome do serviço são inseridas no atributo `MicroserviceHost` e as interfaces de comunicação são adicionadas ao atributo `MicroService` com a respectiva assinatura da interface de comunicação e tipo da requisição, como previamente ilustrado na Listagem 6.

É importante mencionar que até o presente momento, esta ferramenta de refatoração analisa as requisições codificadas seguindo o padrão de requisições HTTPClient, recomendado pela empresa Microsoft.[4] Porém, como trabalhos futuros, planeja-se estender a identificação de outros tipos de requisições presentes na linguagem C#, tal como WebRequest.[5]

4. Avaliação Controlada

Como parte do presente trabalho, foi realizada uma avaliação controlada para a validação da abordagem proposta. Para isso, foram desenvolvidos cinco microsserviços[6], compostos por meio de orquestração [Nanda et al. 2004]. Nesse tipo de composição, uma entidade central, denominada orquestrador, é responsável pela execução e gerenciamento

[3]https://github.com/PqES/RefactoryEasyRestClient

[4]Disponível em: http://docs.microsoft.com/pt-br/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client

[5]https://msdn.microsoft.com/pt-br/library/system.net.webrequest(v=vs.110).aspx

[6]Disponível em: https://github.com/PqES/Microsservicos.NET

das chamadas e fluxos dos microsserviços conhecidos [Miranda et al. 2016]. A Figura 6 apresenta a arquitetura da comunicação entre os microsserviços implementados.

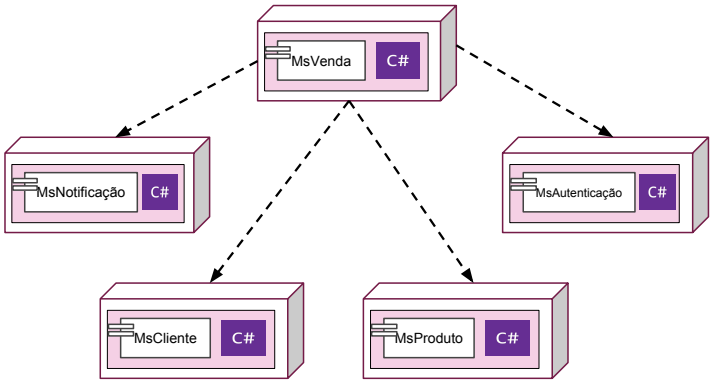

Figura 6. Comunicação entre microsserviços da aplicação de controle de vendas.

O microsserviço MsVenda é responsável por realizar venda de produtos a determinado cliente, sendo esse microsserviço caracterizado como orquestrador. O microsserviço MsAutenticação é responsável por autenticar um funcionário. Os microsserviços `MsCliente` e `MsProduto` gerenciam suas respectivas entidades. Já o microsserviço `MsNotificacao` é responsável por realizar o envio de e-mails para os clientes cadastrados no microsserviço `MsCliente` quando um novo produto é cadastrado. É válido ressaltar que os microsserviços desenvolvidos utilizaram abordagens e tecnologias diferentes para o gerenciamento de banco de dados para demonstrar o total desacoplamento das dependências de comunicação entre eles, sendo eles o `EntityFramework`[7] e banco de dados de armazenamento em arquivos textos.

As Listagens 8 e 10 demonstram o comparativo da requisição de autenticação de um funcionário no microsserviço MsVenda para o microsserviço MsAutenticação em requisições .NET padrão e requisições `EasyRESTClient`. A Listagem 8 apresenta o método `Autenticar` (linha 1) definido para autenticação de um funcionário. Nesse método, são definidos os parâmetros para a autenticação (linhas 4 e 5), uma requisição `post` ao microsserviço *MsAutenticacao* localizado no endereço `http://52.201.200.150` e porta `5001` com a interface `Authenticate` (linha 7).

```
1  public async Task<IActionResult> Autenticar(string username, string password){
2     var client = new HttpClient();
3     var requestParams = new List<KeyValuePair<string, string>>{
4        new KeyValuePair<string, string>("username", username),
5        new KeyValuePair<string, string>("password", password)};
6     var requestParamsFormUrlEncoded = new FormUrlEncodedContent(requestParams);
7     var tokenServiceResponse = await
          client.PostAsync("http://52.201.200.150:5001/Autenticacao/Autenticar",
          requestParamsFormUrlEncoded);
8     string responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
9     var responseCode = tokenServiceResponse.StatusCode;
10    var responseMsg = new HttpResponseMessage(responseCode){
11       Content = new StringContent(responseString, Encoding.UTF8, "application/json")
12    };
13    return View();
14 }
```

Listagem 8. Autenticação realizada com requisições .NET padrão.

[7]Disponível em: https://docs.microsoft.com/pt-br/ef/

O resultado da refatoração do código apresentado na Listagem 8 realizado pela ferramenta `RefactoryEasyClient` está disponível na Listagem 9. A interface `IAutenticacaoService` (linha 2) define as funcionalidades disponibilizados pelo microsserviço *MsAutenticacao* através do método `Autenticar` (linha 5).

```
1 [MicroServiceHost("MsAutenticacao")]
2 public interface IAutenticacaoService : IMicroService
3 {
4   [MicroService("Autenticacao/Autenticar", TypeRequest.Post)]
5   Task<HttpResponseMessage> Autenticar(List<KeyValuePair<string, string>> parameters
        = null);
6 }
```

Listagem 9. Interface para comunicação com o serviço MsAutenticacao.

A Listagem 10 apresenta a implementação do método `Autenticar` (linha 4) no contexto da abstração `EasyRESTClient`. Esse método recebe o retorno da execução do método `Autenticar` da interface `IAutenticacaoService` por meio do objeto `_autenticacaoService` (linha 11).

```
1  private IAutenticacaoService _autenticacaoService;
2  ...
3  public async Task<IActionResult> Autenticar(string username, string password)
4  {
5     var requestParams = new List<KeyValuePair<string, string>>
6     {
7        new KeyValuePair<string, string>("username", username),
8        new KeyValuePair<string, string>("password", password)
9     };
10
11    _autenticacaoService.Autenticar(requestParams);
12
13    return View();
14 }
```

Listagem 10. Autenticação realizada com `EasyRESTClient`.

Os códigos refatorados pela ferramenta `RefactoryEasyClient` das requisições do microsserviço *MsVenda* aos serviços *MsCliente*, *MsProduto* e *MsNotificacao* estão disponíveis no Apêndice A.

5. Conclusão

A arquitetura de microsserviços tem sido amplamente utilizada na indústria e academia para o desenvolvimento de aplicações distribuídas e escaláveis [Newman 2015, Netflix 2018c, Balalaie et al. 2016]. Para isso, um conjuntos de *frameworks* e bibliotecas têm sido propostos pelo projeto Spring Cloud Netflix para auxiliar no desenvolvimento de aplicações que sigam as premissas da arquitetura de microsserviços. Contudo, tais definições estão atualmente disponíveis apenas para microsserviços implementado na linguagem Java, como é o caso do cliente *Feign Client* e do microsserviço *Zuul*.

Com o objetivo de utilizar as definições de abstração de comunicação presente nos serviços *Feign Client* e *Zuul* para microsserviços implementados na linguagem C#, foi realizada uma busca exaustiva e, até o presente momento, não foi encontrado nenhum trabalho que propusesse tais definições. Nesse contexto, o presente trabalho apresenta a abordagem `EasyRESTClient`, uma solução que permite abstrair as definições de comunicação padrão em microsserviços C# desenvolvidos na plataforma .NET e realizar

o encaminhamento de tais requisições. Como contribuição adicional, o presente trabalho apresenta a ferramenta `RefactoryEasyClient` como uma ferramenta que realiza a refatoração de código fonte das comunicações HTTP padrão para o contexto da abordagem `EasyRESTClient`.

Para demonstrar a aplicabilidade da abordagem `EasyRESTClient` e da ferramenta `RefactoryEsyClient` foi desenvolvida uma aplicação composta de cinco microsserviços compostos por meio de orquestração para o contexto de venda de produtos. Nessa arquitetura foram especificados os microsserviços *MsVenda*, *MsProduto*, *MsAutenticacao*, *MsNotificacao* e *MsVenda*, no qual o serviço *MsVenda* é o orquestrador. Esses serviços foram inicialmente implementados em requisições HTTP padrão. A ferramenta `RefactoryEasyClient` foi executada e todo o código do serviço *MsVenda*, qual continha as requisições ao demais serviços, foi refatorado para o contexto da abordagem `EasyRESTClient`. No momento em que as aplicações foram novamente executadas, a ferramenta *EasyRESTClient* realizou o correto encaminhamento das requisições de comunicação aos microsserviços definidos com o auxílio da biblioteca Steeltoe e Eureka.

Como trabalhos futuros, planeja-se realizar a validação da abordagem e ferramentas propostas em ambientes de desenvolvimento real. Mais a fundo, pretende-se identificar qual o impacto que a inserção das abstrações `EasyRESTClient` causam em aplicações que recebam várias requisições em determinado intervalo de tempo. Por fim, planeja-se também estender a identificação de tipos de requisições diferentes do HTTPClient, tal como WebRequest na ferramenta `RefactoryEasyClient`.

Referências

[Araujo et al. 2017] Araujo, E. A., Jr., E. R., Pinto, A. F., and Terra, R. (2017). Em busca de uma abordagem de conformidade arquitetural para arquitetura de microsserviços. In *V Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, pages 68–75.

[Balalaie et al. 2016] Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

[Canaver 2017] Canaver, F. Z. (2017). Uma API para sincronização de dados, baseada em micro serviços, para o suporte ao desenvolvimento de aplicações multiplataforma offline. Master's thesis. Universidade Federal de São Carlos, Brazil.

[Di Francesco 2017] Di Francesco, P. (2017). Architecting microservices. In *1st International Conference on Software Architecture Workshops (ICSAW)*, pages 224–229.

[Fowler and Lewis 2014] Fowler, M. and Lewis, J. (2014). Microservices: a definition of this new architectural term. Acesso em: 29 jun. 2018. Disponível em: <https://martinfowler.com/articles/microservices.html>.

[Gonigberg 2018] Gonigberg, A. (2018). Zuul. Acesso em: 08 jul. 2018. Disponível em: ⟨https://github.com/Netflix/zuul/wiki⟩.

[Miranda et al. 2016] Miranda, S., Rodrigues Jr, E., Valente, M. T., and Terra, R. (2016). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 15(3):1–34.

[Montesi and Weber 2016] Montesi, F. and Weber, J. (2016). Circuit breakers, discovery, and API gateways in microservices. Technical report, Department of Mathematics and Computer Science, University of Southern, Dinamarca.

[Nanda et al. 2004] Nanda, M. G., Chandra, S., and Sarkar, V. (2004). Decentralizing execution of composite web services. In *19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 170–187.

[Netflix 2014] Netflix, S. C. (2014). Eureka at a glance. Acesso em: 11 jun. 2018. Disponível em: <https://github.com/Netflix/eureka/wiki>.

[Netflix 2018a] Netflix, S. C. (2018a). Declarative REST client: Feign. Acesso em: 18 jun. 2018. Disponível em: <https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-feign.html>.

[Netflix 2018b] Netflix, S. C. (2018b). Service discovery: Eureka clients. Acesso em: 19 jun. 2018. Disponível em: <https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_service_discovery_eureka_clients>.

[Netflix 2018c] Netflix, S. C. (2018c). Spring Cloud Netflix. Acesso em: 19 jan. 2018. Disponível em: <https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html>.

[Newman 2015] Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. Sebastopol, EUA.

[Pivotal 2017] Pivotal, S. (2017). Projects spring boot. Acesso em: 05 jul. 2017. Disponível em: <https://projects.spring.io/spring-boot>.

[Pivotal 2018a] Pivotal, S. (2018a). Cloud-native .net microservices. Acesso em: 29 maio 2018. Disponível em: <https://steeltoe.io/docs/>.

[Pivotal 2018b] Pivotal, S. (2018b). Pivotal cloud foundry. Acesso em: 31 out. 2018. Disponível em: <https://docs.pivotal.io/pivotalcf/2-2/pas/intro.html>.

[Pivotal 2018c] Pivotal, S. (2018c). Routing and filtering. Acesso em: 18 jun. 2018. Disponível em: <https://spring.io/guides/gs/routing-and-filtering>.

[Smith et al. 2018] Smith, S., Dykstra, T., and Latham, L. (2018). Inicialização do aplicativo no ASP.NET Core. Acesso em: 07 jul. 2018. Disponível em: <https://docs.microsoft.com/pt-br/aspnet/core/fundamentals/startup?view=aspnetcore-2.1>.

[Zimmermann 2016] Zimmermann, O. (2016). Microservices tenets: agile approach to service development and deployment. *Computer Science-Research and Development*, 32(3):301–310.

A. Apêndice

Esta seção apresenta o resultado gerado no código fonte do microsserviço *Ms-Venda* após a refatoração realizada pela ferramenta `RefactoryEasyClient` para o contexto das abstrações propostas na abordagem `EasyRESTClient`. O código refatorado refere-se às requisições aos microsserviços *MsCliente, MsProduto* e *MsNotificação*, podendo ser encontrados no repositório GitHub[8]. É válido ressaltar que a refatoração do microsserviço *MsAutenticação* foi previamente apresentado na Seção 4.

A.1. Requisições ao microsserviço *MsCliente*

No código fonte referente às requisições ao microsserviço *MsCliente* são definidos os métodos `AddCliente`, `GetTodosClientes`, `GetClientePorCpf` e `DeletarCliente` para o gerenciamento da entidade cliente, representado nas Listagens 11, 12, 13 e 14, respectivamente. Nessas listagens, é possível perceber a complexidade para definição das requisições HTTP padrão em .NET.

```
public async Task<IActionResult> AddCliente(string name, string cpf, string email){
  using (var client = new HttpClient()){
    var requestParams = new List<KeyValuePair<string, string>>
    {
        new KeyValuePair<string, string>("name", name),
        new KeyValuePair<string, string>("cpf", cpf),
        new KeyValuePair<string, string>("email", email)
    };

    var requestParamsFormUrlEncoded = new FormUrlEncodedContent(requestParams);
    var tokenServiceResponse = await
        client.PostAsync("http://52.201.200.150:5002/Cliente/AddCliente",
        requestParamsFormUrlEncoded);
    var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();

    var responseCode = tokenServiceResponse.StatusCode;
    var responseMsg = new HttpResponseMessage(responseCode)
    {
        Content = new StringContent(responseString, Encoding.UTF8,
            "application/json")
    };
    return View();
  }
}
```

Listagem 11. Método adicionar cliente descrito através de requisição HTTP padrão.

```
public async Task<IActionResult> GetTodosClientes()
{
  using (var client = new HttpClient())
  {
    var tokenServiceResponse = await
        client.GetAsync("http://52.201.200.150:5002/Cliente/GetTodosClientes");
    var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
    var responseCode = tokenServiceResponse.StatusCode;
    var responseMsg = new HttpResponseMessage(responseCode)
    {
      Content = new StringContent(responseString, Encoding.UTF8, "application/json")
    };
    return View();
  }
}
```

Listagem 12. Método que retorna todos os clientes cadastrados descrito através de requisição HTTP padrão.

[8]Disponível em: https://github.com/PqES/MicrosservicosRefactory

```
1  public async Task<IActionResult> GetClientePorCpf(string cpf)
2  {
3    using (var client = new HttpClient())
4    {
5      var url = "http://52.201.200.150:5002/Cliente/GetClientePorCpf?cpf=";
6      var tokenServiceResponse = await client.GetAsync(url + cpf);
7      var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
8      var responseCode = tokenServiceResponse.StatusCode;
9      var responseMsg = new HttpResponseMessage(responseCode)
10     {
11       Content = new StringContent(responseString, Encoding.UTF8, "application/json")
12     };
13     return View();
14   }
15 }
```

Listagem 13. Método que retorna o cpf de um cliente cadastrado, descrito através de requisição HTTP padrão.

```
1  public async Task<IActionResult> Delete(string cpf)
2  {
3    using (var client = new HttpClient())
4    {
5      var url = String.Format("http://52.201.200.150:5002/Cliente/Delete?cpf={0}", cpf);
6      var tokenServiceResponse = await client.DeleteAsync(url);
7      var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
8      var responseCode = tokenServiceResponse.StatusCode;
9      var responseMsg = new HttpResponseMessage(responseCode)
10     {
11       Content = new StringContent(responseString, Encoding.UTF8, "application/json")
12     };
13     return View();
14   }
15 }
```

Listagem 14. Método que deleta um cliente cadastrado, descrito através de requisição HTTP padrão.

O resultado da refatoração dos códigos apresentados nas Listagens 11, 12, 13 e 14 realizado pela ferramenta `RefactoryEasyClient` está disponível na Listagem 15. A interface `IClienteService` (linha 2) define as funcionalidades disponibilizados pelo microsserviço *MsCliente* através dos métodos `AddCliente` (linha 5), `GetTodosClientes` (linha 8), `GetClientePorCpf` (linha 11) e `Delete` (linha 14).

```
1  [MicroServiceHost("MsCliente")]
2  public interface IClienteService : IMicroService
3  {
4    [MicroService("Cliente/AddCliente", TypeRequest.Post)]
5    Task<HttpResponseMessage> AddCliente(List<KeyValuePair<string, string>> parameters
         = null);
6
7    [MicroService("Cliente/GetTodosClientes", TypeRequest.Get)]
8    Task<HttpResponseMessage> GetTodosClientes(List<KeyValuePair<string, string>>
         parameters = null);
9
10   [MicroService("Cliente/GetClientePorCpf", TypeRequest.Get)]
11   Task<HttpResponseMessage> GetClientePorCpf(List<KeyValuePair<string, string>>
         parameters = null);
12
13   [MicroService("Cliente/Delete", TypeRequest.Delete)]
14   Task<HttpResponseMessage> Delete(List<KeyValuePair<string, string>> parameters =
         null);
15 }
```

Listagem 15. Interface `IClienteService`.

A Listagem 16 apresenta a implementação do método `GetClientePorCpf` (linha 7) no contexto da abstração `EasyRESTClient`. Esse método recebe o retorno da execução do método `GetClientePorCpf` da interface `IClienteService` por meio do objeto `_clienteService` (linha 12). É válido ressaltar que a utilização dos demais métodos disponibilizados pelo microsserviço *MsCliente* segue a mesma lógica apresentada na linha 12.

```
1  private IClienteService _clienteService;
2  ...
3  public ClienteController(IClienteService clienteService){
4    _clienteService = clienteService;
5  }
6
7  public async Task<IActionResult> GetClientePorCpf(string cpf)
8  {
9    var requestParams = new List<KeyValuePair<string, string>>{
10     new KeyValuePair<string, string>("cpf",cpf)
11   };
12   _clienteService.GetClientePorCpf(requestParams);
13
14   return View();
15 }
```

Listagem 16. Consulta de cliente por CPF, descrito por meio da abstração `EasyRESTClient`.

A.2. Requisições ao microsserviço *MsProduto*

No código fonte referente às requisições ao microsserviço *MsProduto*, são definidos os métodos `AddProduto`, `GetTodosProdutos` e `AtualizarEstoque` para o gerenciamento da entidade produto, conforme apresenta as Listagens 17, 18 e 19, respectivamente.

```
1  public async Task<IActionResult> AddProduto(string name, string description, int
       number, double value)
2  {
3    using (var client = new HttpClient())
4    {
5
6      var requestParams = new List<KeyValuePair<string, string>>
7      {
8        new KeyValuePair<string, string>("name", name),
9        new KeyValuePair<string, string>("cpf", description),
10       new KeyValuePair<string, string>("number", number.ToString()),
11       new KeyValuePair<string, string>("value", value.ToString())
12     };
13
14     var requestParamsFormUrlEncoded = new FormUrlEncodedContent(requestParams);
15     var tokenServiceResponse = await
           client.PostAsync("http://52.201.200.150:5004/Produto/AddProduto",
           requestParamsFormUrlEncoded);
16     var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
17
18     var responseCode = tokenServiceResponse.StatusCode;
19     var responseMsg = new HttpResponseMessage(responseCode)
20     {
21       Content = new StringContent(responseString, Encoding.UTF8, "application/json")
22     };
23
24     return View();
25   }
26 }
```

Listagem 17. Método adicionar produto descrito através de requisição HTTP padrão.

```
1  public async Task<IActionResult> GetTodosProdutos()
2  {
3    using (var client = new HttpClient())
4    {
5      var URL = "http://52.201.200.150:5004/Produto/GetTodosProdutos";
6      var tokenServiceResponse = await client.GetAsync(URL);
7      var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
8
9      var responseCode = tokenServiceResponse.StatusCode;
10     var responseMsg = new HttpResponseMessage(responseCode)
11     {
12       Content = new StringContent(responseString, Encoding.UTF8, "application/json")
13     };
14
15     return View();
16   }
17 }
```

Listagem 18. Consulta a todos os produtos cadastrados, descrito através de requisição HTTP padrão.

```
1  public async Task<IActionResult> AtualizarEstoque(string id, string qtds)
2  {
3    using (var client = new HttpClient())
4    {
5      var requestParams = new List<KeyValuePair<string, string>>
6      {
7        new KeyValuePair<string, string>("ids", id.ToString()),
8        new KeyValuePair<string, string>("qtds", qtds.ToString())
9      };
10
11     var url = "http://52.201.200.150:5004/Produto/AtualizaEstoque";
12     var requestParamsFormUrlEncoded = new FormUrlEncodedContent(requestParams);
13     var tokenServiceResponse = await client.PostAsync(url,
           requestParamsFormUrlEncoded);
14     var responseString = await tokenServiceResponse.Content.ReadAsStringAsync();
15
16     var responseCode = tokenServiceResponse.StatusCode;
17     var responseMsg = new HttpResponseMessage(responseCode)
18     {
19       Content = new StringContent(responseString, Encoding.UTF8, "application/json")
20     };
21
22     return View();
23   }
24 }
```

Listagem 19. Atualização do estoque de produtos, descrito através de requisição HTTP padrão.

De acordo com a Listagem 19, para concretizar uma determinada venda, o método `AtualizarEstoque` (linha 1) é executado e uma requisição ao microsserviço *MsProduto* é realizada para atualização do estoque (linha 13), enviando uma lista contendo os identificadores (`id`) dos produtos e as quantidades (`qtds`) que serão debitadas no estoque.

O resultado da refatoração dos códigos apresentados nas Listagens 17, 18 e 19 realizado pela ferramenta `RefactoryEasyClient` está disponível na Listagem 20. A interface `IProdutoService` (linha 2) define as funcionalidades disponibilizadas pelo microsserviço *MsProduto* através dos métodos `AddProduto` (linha 4), `GetTodosProdutos` (linha 7) e `AtualizarEstoque` (linha 10).

```
1  [MicroServiceHost("MsProduto")]
2  public interface IProdutoService : IMicroService{
3    [MicroService("Produto/AddProduto", TypeRequest.Post)]
4    Task<HttpResponseMessage> AddProduto(List<KeyValuePair<string, string>> parameters
         = null);
5
6    [MicroService("Produto/GetTodosProdutos", TypeRequest.Get)]
7    Task<HttpResponseMessage> GetTodosProdutos(List<KeyValuePair<string, string>>
         parameters = null);
8
9    [MicroService("Produto/AtualizarEstoque", TypeRequest.Post)]
10   Task<HttpResponseMessage> AtualizarEstoque(List<KeyValuePair<string, string>>
         parameters = null);
11 }
```

Listagem 20. Interface `IProdutoService`.

A Listagem 21 apresenta a implementação do método `AtualizarEstoque` (linha 6) no contexto da abstração `EasyRESTClient`. Esse método recebe o retorno da execução do método `AtualizarEstoque` da interface `IProdutoService` por meio do objeto `_produtoService` (linha 11). A lista de parâmetros definida nas linhas 7 a 10 é enviada na requisição através do parâmetro opcional `parameters` da assinatura do método. É válido ressaltar que a utilização dos demais métodos é semelhante ao apresentado na linha 11.

```
1  private IProdutoService _produtoService;
2  public ProdutoController(IProdutoService produtoService){
3    _produtoService = produtoService;
4  }
5
6  public async Task<IActionResult> AtualizarEstoque(int id, int qtds)
7    var requestParams = new List<KeyValuePair<string, string>>{
8      new KeyValuePair<string, string>("ids", id.ToString()),
9      new KeyValuePair<string, string>("qtds", qtds.ToString())
10   };
11   var httpResponse = _produtoService.AtualizarEstoque(requestParams);
12   ...
13 }
```

Listagem 21. Consulta a todos os produtos descrito por meio da abstração `EasyRESTClient`.

A.3. Requisição ao microsserviço *MsNotificação*

No código fonte referente às requisições ao microsserviço *MsNotificacao*, é definido o método `Notificar` responsável por enviar e-mails aos clientes quando um produto está novamente disponível em estoque, conforme apresenta a Listagem 22.

```
1  public async Task<IActionResult> Notificar(){
2    using (var client = new HttpClient()){
3      var tokenServiceResponseNotificacao = await
           client.PostAsync("http://52.201.200.150:5003/Notificacao/Notificar", null);
4      var responseStringNotificacao = await
           tokenServiceResponse.Content.ReadAsStringAsync();
5      var responseCodeNotificacao = tokenServiceResponse.StatusCode;
6      var responseMsgNotificacao = new HttpResponseMessage(responseCode){
7        Content = new StringContent(responseString, Encoding.UTF8, "application/json")
8      };
9
10     return View();
11   }
12 }
```

Listagem 22. Método Notificar descrito através de requisição HTTP padrão.

O resultado da refatoração do código apresentado na Listagem 22 realizado pela ferramenta `RefactoryEasyClient` está disponível na Listagem 23. A interface `INotificacaoService` (linha 2) define a funcionalidade disponibilizada pelo microsserviço *MsNotificacao* através do método `Notificar` (linha 5).

```
1 [MicroServiceHost("MsNotificacao")]
2 public interface INotificacaoService : IMicroService
3 {
4   [MicroService("Notificacao/Notificar", TypeRequest.Post)]
5   Task<HttpResponseMessage> Notificar(List<KeyValuePair<string, string>> parameters =
        null);
6 }
```

Listagem 23. Interface `INotificacaoService`.

A Listagem 24 apresenta a implementação do método `Notificar` (linha 7 a 12) no contexto da abstração `EasyRESTClient`. Esse método recebe o retorno da execução do método `Notificar` da interface `INotificacaoService` por meio do objeto `_notificacaoService` (linha 9).

```
1  private INotificacaoService _notificacaoService;
2  public NotificacaoController(INotificacaoService notificacaoService)
3  {
4    _notificacaoService = notificacaoService;
5  }
6
7  public async Task<IActionResult> Notificar()
8  {
9    _notificacaoService.Notificar();
10
11   return View();
12 }
```

Listagem 24. Método Notificar descrito por meio da abstração `EasyRESTClient`.