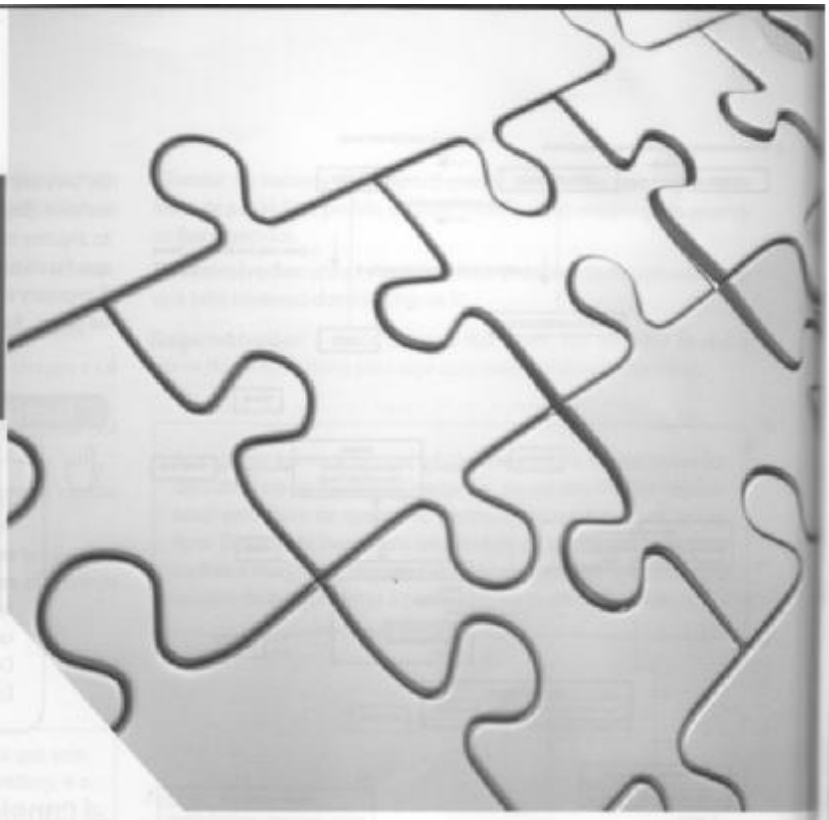
[artigo]



Eduardo Dias
(edurdias@gmail.com)
Especialista em desenvolvimento .NET, prestou serviços em empresas como WEG e Siemens. Atualmente, é arquiteto de software na Hartmann TI, empresa especializada em soluções de ECM (Enterprise Content Management). Entusiasta de metodologias ágeis e desenvolvimento baseados em testes, ele promete convencer os amigos a utilizarem XP.

Consumindo Web Services dinamicamente

Cada vez mais as empresas utilizam serviços Web Services para a integração de aplicações, impulsionando o desenvolvimento orientado a serviços (SOA). Com o aumento destes serviços, vemos a necessidade de consumi-los sem termos que adicionar uma referência física ao projeto em questão, como se faz normalmente. Para isso, iremos criar um proxy que nos permitirá consumir qualquer serviço, sem a necessidade destas referências.

Quando falamos de Web Services em uma aplicação .NET já "vem à cabeça" o termo Web Reference. Para os que não conhecem o termo, uma Web Reference é um recurso do .NET que nos permite consumir um serviço web de forma transparente ao código. Este recurso é muito útil pois elimina toda a complexidade existente em outras linguagens. Ao adicionar uma referência, o Visual Studio cria uma pasta dentro da App_WebReferences com o nome do serviço informado. Esta pasta contém os arquivos necessários para a geração do código que permitirá consumir este serviço. Quando o software em desenvolvimento consome poucos serviços, este recurso se mostra bastante eficaz, tanto na implementação, como comentamos, como também no gerenciamento.

Porém quando necessitamos consumir um volume maior de serviços, nos deparamos com algumas dificuldades que são até naturais. Falando em números, é muito mais trabalhoso o gerenciamento de 25 referências do que apenas duas ou três em uma aplicação, principalmente quando há a mudança de, por exemplo, o endereço de uma destas referências. Mas qual estratégia devemos adotar para consumir um volume maior de serviços em uma aplicação? Isso é o que veremos.

Para podermos manter de forma dinâmica e sem a necessidade de ter que abrir o projeto para cada alteração em uma referência, e isso acontece muito no dia-a-dia, iremos criar um proxy que nos permitirá obter as devidas informações de um serviço. Mas o que este proxy irá fazer realmente? Ele irá obter, através de um WSDL (um arquivo que contém informações tais como as operações disponíveis e a definição dos elementos envolvidos), a descrição completa deste serviço. Quando adicionamos uma referência pelo Visual Studio podemos observar a existência de um arquivo com a extensão .wsdl, nele está contida sua descrição. Com a descrição em mãos, iremos gerar o código que utilizaremos posteriormente em nosso proxy e através de reflexão poderemos obter os parâmetros de cada operação, seu retorno e também invocá-la. Assim teremos um código mais enxuto e mais fácil de gerir. No decorrer do artigo, veremos passo-a-passo como implementar este código e o porquê de cada etapa.

Preparando o ambiente

Antes de começar, devemos estar atentos a alguns detalhes em nossa implementação. Em nosso proxy (que chamaremos de WebServiceProxy), utilizaremos recursos como reflexão e geração de código em tempo de execução. Para podermos utilizar estes recursos, devemos inserir as devidas referências no início do arquivo que contém a nossa classe. Sabemos também que alguns serviços estão sob SSL e quando acessamos este serviço pelo browser aparece uma imagem solicitando uma confirmação para continuarmos (figura 1). A diferença é que um usuário não tem como confirmar esta solicitação, se ela é feita na aplicação que desenvolvemos.

Figura 1 Alerta de segurança do Internet explorer

Para resolver este problema, desenvolveremos um método que será responsável por esta tarefa. O método manipulará a chamada feita pela propriedade ServerCertificateValidationCallback da classe ServicePointManage.

Listagem 1. Inserindo a referência do arquivo.

```
using System;
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
using System.Web.Services.Description;
using System.IO;
using System.CodeDom;
using System.CodeDom.Compiler;
using System.Globalization;
using Microsoft.CSharp;
using System.Reflection;
using System.Xml.Serialization;
```

Após termos adicionado as devidas referências, estamos prontos para finalizar esta etapa de preparação do nosso ambiente. No construtor da classe, iremos associar à propriedade ServerCertificateValidationCallback o manipulador que comentamos anteriormente. Sua associação encontra-se na Listagem 2 e sua implementação na Listagem 3.

Listagem 2. Associando manipulador à propriedade ServerCertificateValidationCallback da classe ServicePointManager.

```
ServicePointManager.ServerCertificateValidationCallback =
        new RemoteCertificateValidationCallback(CertificateValidation);
```

Listagem 3. Código do método CertificateValidation.

```
public bool CertificateValidation(object sender,
        X509Certificate certificate, X509Chain chain,
        SslPolicyErrors sslPolicyErrors)
{
        return true;
}
```

Obtendo a descrição do serviço

Para podermos consumir um serviço, precisamos obter a sua descrição. Esta descrição está contida no arquivo WSDL do serviço. No caso de um serviço .NET, cuja extensão padrão é .asmx, a descrição pode ser obtida adicionando "?WSDL" ao final. O nosso proxy recebe um argumento em seu construtor, do tipo Uri, que contém o endereço deste serviço. Utilizaremos a classe WebRequest para fazer uma requisição deste endereço e obter seu conteúdo que virá em forma de Stream. Com este Stream de resposta em mãos, poderemos carregar a descrição do serviço para nosso código. A Listagem 4 contém o código necessário para obter esta informação.

Listagem 4. Código que obtém a descrição do serviço.

```
WebRequest request = WebRequest.Create(uri);
Stream stream = request.GetResponse().GetResponseStream();

ServiceDescription serviceDescription =
        ServiceDescription.Read(stream);
string serviceName = serviceDescription.Services[0].Name;
```

A descrição é armazenada na variável serviceDescription e o nome do serviço requisitado na variável serviceName. A primeira é utilizada para a geração da classe que usaremos para invocar as operações do serviço e a segunda para podermos, através de reflexão, instanciar esta classe.

Gerando a classe do serviço

Após obtermos a descrição e armazená-la na variável serviceDescription, iremos importá-la através de uma variável do tipo ServiceDescriptionImporter que irá gerar a estrutura da classe do serviço em questão dentro de um objeto do tipo CodeCompileUnit que será utilizado para gerar o código em si. A Listagem 5 contém o código que monta a variável serviceImporter (ServiceDescriptionImporter) que utilizaremos em seguida.

Ao criar a variável serviceImporter, informamos o nome do protocolo utilizado e também que desejamos que os tipos simples deste serviço sejam gerados como propriedades e não como simples campos.

Listagem 5. Criando a variável serviceImporter.

```
ServiceDescriptionImporter importer =
        new ServiceDescriptionImporter();
importer.AddServiceDescription(serviceDescription,
        string.Empty, string.Empty);
importer.ProtocolName = "Soap";
importer.CodeGenerationOptions =
        CodeGenerationOptions.GenerateProperties;
```

Gerando o grafo

Com o serviceImporter devidamente configurado podemos preparar os objetos que permitirão gerar o código. Estes objetos fazem parte do namespace System.CodeDom que contém as classes que permitem representar os elementos e as estruturas de um código-fonte. No nosso caso, iremos trabalhar com as classes CodeNamespace e CodeCompileUnit (esta já mencionada anteriormente). Um objeto do tipo CodeNamespace será adicionado a lista de namespaces do objeto da classe CodeCompileUnit e este último recebe de serviceImporter o grafo da classe do serviço para que possamos gerar o código. A Listagem 6 demonstra como é feita esta importação.

Listagem 6. Importando o grafo da classe para CodeCompileUnit.

```
CodeNamespace @namespace = new CodeNamespace();
CodeCompileUnit unit = new CodeCompileUnit();
unit.Namespaces.Add(@namespace);
ServiceDescriptionImportWarnings warnings =
        importer.Import(@namespace, unit);
```

Ao importarmos o grafo para a variável unit (CodeCompileUnit) é retornada uma variável do tipo ServiceDescriptionImportWarnings que contém os problemas com a geração do grafo. Nossa preocupação é que esta variável tenha valor diferente de NoCodeGenerated e NoMethodsGenerated.

Gerando o código

Com a geração do grafo realizada com sucesso e como nossa implementação se baseia em C#, utilizaremos a classe CSharpCodeProvider para gerarmos o código a partir do grafo. Para podermos gerar este código, necessitamos criar um objeto do tipo StringWriter com a cultura atual (por isso incluímos o namespace System.Globalization nas referências) para utilizá-lo ao gerar o código.

Listagem 7. Gerando o código da classe do serviço a partir do grafo.

```
StringWriter writer = new StringWriter(CultureInfo.CurrentCulture);
CSharpCodeProvider provider = new CSharpCodeProvider();
provider.GenerateCodeFromNamespace(@namespace, writer, null)
```

Compilando o código

Após termos gerado o código, podemos preparar sua compilação. Para isso, devemos informar os parâmetros de compilação e as referências básicas. Entre os parâmetros (Listagem 8) estão GenerateExecutable que deve estar com valor falso para que não seja gerado nenhum executável físico. E inversamente a esta opção, temos a opção GenerateInMemory que deve ter o valor verdadeiro para que seja compilado em memória. Devemos informar também que os avisos não devem ser tratados como erro atribuindo falso para o parâmetro TreatWarningsAsErros e 4 para o nível de aviso pelo parâmetro WarningLevel.

Listagem 8. Montagem dos parâmetros para compilação.

```
CompilerParameters parameters =
        new CompilerParameters(new string[]
        {
                "System.dll", "System.Xml.dll",
                "System.Web.Services.dll", "System.Data.dll"
        });
parameters.GenerateExecutable = false;
parameters.GenerateInMemory = true;
parameters.TreatWarningsAsErrors = false;
parameters.WarningLevel = 4;
```

Após termos preparado o grafo, código e os parâmetros, basta realizarmos a compilação em memória e, com o Assembly em mãos, obtermos o tipo da classe do serviço. Armazenaremos este tipo na variável global chamada serviceType. Esta variável será utilizada posteriormente para que, através de reflexão, possamos obter os nomes das operações disponíveis, seus parâmetros e também invocá-las.

Listagem 9. Compilação da classe.

```
CompilerResults results = provider
        .CompileAssemblyFromSource(parameters,
        writer.ToString());
Assembly serviceAssembly = results.CompiledAssembly;
serviceType = serviceAssembly.GetType(serviceName);
```

Obtendo as operações do serviço

Agora que temos o tipo do serviço armazenado na variável serviceType, podemos obter a lista de operações disponíveis no serviço informado. Para isso, criamos uma propriedade Methods que retorna um vetor de objetos do tipo MethodInfo (Listagem 10).

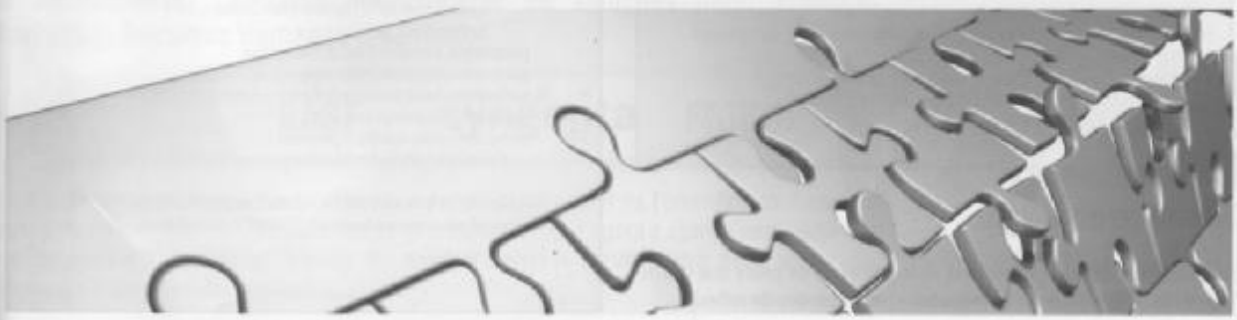
Utilizando reflexão conseguimos obter todas as operações existentes na classe que compilamos (classe do serviço). O tipo MethodInfo pode nos fornecer informações tais como nome, parâmetros e seu tipo de retorno.

Listagem 10. Propriedade que retorna a lista de operações do serviço.

```
public MethodInfo[] Methods
{
    get
    {
        if (serviceType != null)
            return serviceType.GetMethods(BindingFlags
                .DeclaredOnly | BindingFlags.IgnoreCase
                | BindingFlags.Instance
                | BindingFlags.InvokeMethod
                | BindingFlags.Public);
        return new MethodInfo[] { };
    }
}
```

Obtendo os parâmetros de uma operação

Para nos auxiliar na obtenção dos parâmetros de uma operação (método) do serviço, iremos criar um método que nos retornará um vetor de objetos do tipo ParameterInfo. Este vetor contém todos os parâmetros, com seus tipos, nomes e valores iniciais. A Listagem 11 contém o código que implementa este método.



Listagem 11. Código do método que obtém os parâmetros de uma operação (método).

```
public ParameterInfo[] GetParameters(string methodName)
{
    if (serviceType != null)
        return serviceType.GetMethod(methodName).GetParameters();
    return new ParameterInfo[] { };
}
```

Invocando uma operação

Agora que temos as operações e seus parâmetros, podemos invocá-las. Para isso, precisamos antes descobrir qual o tipo de retorno da operação desejada. Basicamente uma operação contém dois tipos de retorno:

- objeto – um objeto somente, podendo ser primário ou complexo;
- vetor de objetos – uma lista de objetos primários ou complexos.

Para nos auxiliar nesta tarefa, implementamos o método ReturnArray que (novamente por reflexão) obtém o tipo de retorno de uma operação e caso seja um vetor retorna verdadeiro (Listagem 12).

Agora podemos invocar uma operação de acordo com seu tipo de retorno. Para isso, iremos implementar dois métodos, um para cada tipo, que nos permitam invocar uma operação especifica.
O primeiro método retorna um objeto primário ou complexo utilizando a classe Activator para criar uma instância do serviço e recebendo o nome da operação e seus parâmetros, invocar a operação desejada. O segundo funciona da mesma forma, porém faz uma conversão do retorno para object[] conforme a Listagem 13.

Listagem 12. Método que verifica o tipo de retorno de uma operação.

```
public bool ReturnArray(string methodName)
{
    return serviceType.GetMethod(methodName)
            .ReturnType.IsSubclassOf(typeof(Array));
}
```

Listagem 13. Invocando operações.

```
public object Invoke(string methodName, object[] parameters)
{
    object instance = Activator.CreateInstance(serviceType);
    return serviceType.GetMethod(methodName)
            .Invoke(instance, parameters);
}

public object[] InvokeAsArray(string methodName, object[] parameters)
{
    object instance = Activator.CreateInstance(serviceType);
    return (object[])serviceType.GetMethod(methodName)
            .Invoke(instance, parameters);
}
```

Código completo

Na Listagem 14 podemos ter uma visão completa do proxy que desenvolvemos. A ligação entre os métodos e uso extensivo de reflexão.

Listagem 14. Código completo do WebServiceProxy.

```
using System;
using System.IO;
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
using System.Web.Services.Description;
using System.CodeDom;
using System.CodeDom.Compiler;
using System.Globalization;
using Microsoft.CSharp;
using System.Reflection;
using System.Xml.Serialization;

public class WebServiceProxy
{
    private Type serviceType;

    public WebServiceProxy(Uri uri)
    {
        ServicePointManager.ServerCertificateValidationCallback =
            new RemoteCertificateValidationCallback(
                CertificateValidation);

        WebRequest request = WebRequest.Create(uri);
        Stream stream = request.GetResponse().GetResponseStream();

        ServiceDescription serviceDescription =
            ServiceDescription.Read(stream);
        string serviceName = serviceDescription.Services[0].Name;

        ServiceDescriptionImporter importer =
            new ServiceDescriptionImporter();
        importer.AddServiceDescription(serviceDescription,
            string.Empty, string.Empty);
        importer.ProtocolName = "Soap";
        importer.CodeGenerationOptions =
            CodeGenerationOptions.GenerateProperties;

        CodeNamespace @namespace = new CodeNamespace();
        CodeCompileUnit unit = new CodeCompileUnit();
        unit.Namespaces.Add(@namespace);
        ServiceDescriptionImportWarnings warnings =
            importer.Import(@namespace, unit);

        if (warnings != ServiceDescriptionImportWarnings
                .NoCodeGenerated
            && warnings != ServiceDescriptionImportWarnings
                .NoMethodsGenerated)
        {
            StringWriter writer = new StringWriter(
                CultureInfo.CurrentCulture);
            CSharpCodeProvider provider = new CSharpCodeProvider();
            provider.GenerateCodeFromNamespace(
                @namespace, writer, null);
            CompilerParameters parameters =
                new CompilerParameters(new string[] { "System.dll",
                    "System.Xml.dll", "System.Web.Services.dll",
                    "System.Data.dll" });
            parameters.GenerateExecutable = false;
            parameters.GenerateInMemory = true;
            parameters.TreatWarningsAsErrors = false;
            parameters.WarningLevel = 4;
            CompilerResults results = provider
                .CompileAssemblyFromSource(parameters, writer.ToString());

            Assembly serviceAssembly = results.CompiledAssembly;
            serviceType = serviceAssembly.GetType(serviceName);
        }
    }
```

```
public MethodInfo[] Methods
{
  get
  {
    if (serviceType != null)
      return serviceType.GetMethods(BindingFlags.DeclaredOnly
        | BindingFlags.IgnoreCase | BindingFlags.Instance
        | BindingFlags.InvokeMethod | BindingFlags.Public);
    return new MethodInfo[] { };
  }
}

public ParameterInfo[] GetParameters(string methodName)
{
  if (serviceType != null)
    return serviceType.GetMethod(methodName).GetParameters();
  return new ParameterInfo[] { };
}

public object Invoke(string methodName, object[] parameters)
{
  object instance = Activator.CreateInstance(serviceType);

  return serviceType.GetMethod(methodName).Invoke(
    instance, parameters);
}

public object[] InvokeAsArray(string methodName,
  object[] parameters)
{
  object instance = Activator.CreateInstance(serviceType);

  return (object[])serviceType.GetMethod(methodName)
    .Invoke(instance, parameters);
}

public bool ReturnArray(string methodName)
{
  return serviceType.GetMethod(methodName)
    .ReturnType.IsSubclassOf(typeof(Array));
}
public bool CertificateValidation(object sender,
  X509Certificate certificate, X509Chain chain,
  SslPolicyErrors sslPolicyErrors)
{
  return true;
}
}
```

Considerações finais

Sendo uma alternativa para quem busca uma solução para o consumo e gerenciamento de um grande volume de serviços, esta técnica facilita o desenvolvimento e manutenção de uma aplicação. Seu nível de abstração traz uma grande flexibilidade ao desenvolvedor.

Uma situação prática de uso, e que é interessante ressaltar, é quando se tem um diretório de serviços ou mesmo um catálogo armazenado em banco de dados e se deseja realizar o consumo destes serviços. Nesse caso, o proxy que desenvolvemos apresenta-se como uma boa escolha.