<http://www.linhadecodigo.com.br/artigo/2314/wcf-introducao.aspx>

**WCF - Introdução**

A Microsoft disponibiliza várias tecnologias para o desenvolvimento de aplicações distribuídas. Cada uma delas é voltada para uma necessidade específica, e entre essas tecnologias temos: *ASP.NET Web Services*, *WSE - Web Services Enhancements*, *.NET Remoting*, *COM+ - Enterprise Services* e *MSMQ - Message Queue*. Cada uma delas possui sua própria API, com vários tipos que devem ser estudados para que possamos desenvolver uma aplicação que exponha ou consuma recursos destas tecnologias.

Quando iniciou a criação do .NET Framework 3.0, um entre quatro dos grandes pilares que havia dentro dele era o *Indigo* que mais tarde recebeu o nome de [*Windows Communication Foundation*, ou simplesmente WCF](http://www.devmedia.com.br/curso/introducao-ao-windows-communication-fundations/314). O WCF unificou as várias tecnologias de programação distribuídas na plataforma Microsoft em um único modelo, baseando-se na arquitetura orientada à serviços (SOA). Essa nova API facilita consideravelmente o aprendizado e desenvolvimento, já que o WCF está totalmente desacoplado das regras de negócios que serão expostas pelo serviço. A finalidade deste artigo é mostrar uma introdução ao WCF, construindo passo-à-passo um exemplo simples de como criar e consumir um serviço.

Para começar a fazer uso do WCF, tudo o que precisamos é referenciar em nossa aplicação o *assembly System.ServiceModel.dll*. Esse *assembly* possui a maioria dos tipos necessários para a construção de um serviço ou de um cliente. Ainda há outros *assemblies* que complementam o WCF, como é o caso do suporte à serviços baseados em REST, mas que serão abordados em artigos específicos.

### Estrutura

A estrutura de um serviço WCF não é muito complexa, pois devemos utilizar conceitos puros de programação .NET para a criação do contrato e da classe que representará o serviço. Além disso, o WCF também suporta a utilização de tipos complexos, como classes que criamos para atender uma determinada necessidade.

O primeiro passo na criação do serviço é a definição do contrato. É o contrato que determinará quais operações estarão expostas, quais informações essas operações necessitam para ser executadas e também qual será o tipo de retorno. O contrato nada mais é que uma *Interface* que, por sua vez, deverá possuir os métodos (apenas a sua assinatura) que serão expostos. A *Interface* que servirá como contrato deverá ser obrigatoriamente decorada com o atributo *ServiceContractAttribute* pois, caso contrário, uma exceção do tipo *InvalidOperationException* será disparada antes da abertura do *host*.

Nem sempre todos os membros expostos pela *Interface* devem ser expostos para o serviço e, justamente por isso, todas as operações que serão disponibilizadas devem ser decoradas com o atributo *OperationContractAttribute*. Vale lembrar que o WCF obriga a termos no mínimo uma operação definida com este atributo, já que não faz sentido publicar um serviço que não tenha nenhuma operação a ser executada. Caso a *Interface* não possua nenhuma operação definida com este atributo, uma exceção do tipo *InvalidOperationException*também será disparada antes da abertura do *host*. O código abaixo exibe uma *Interface* simples, que servirá como exemplo para o artigo:

using System;

using System.ServiceModel;

[ServiceContract]

public interface IContrato

{

    [OperationContract]

    Usuario RecuperarUsuario(string nome);

}

Para fins de exemplo, esta *Interface* apenas terá um único membro, mas ela poderá conter vários outros e, como já dito acima, você controla a visibilidade destes membros através do atributo *OperationContractAttribute*. Como podemos notar, o método *RecuperarUsuario* retorna uma instância da classe *Usuario*. Neste momento dois novos atributos entram em cena: *DataContractAttribute* e *DataMemberAttribute*, ambos contidos no *namespace System.Runtime.Serialization*, fornecido pelo *assembly System.Runtime.Serialization.dll*.

Os *data contracts* são uma forma que se tem de publicar possíveis estruturas de dados que podem ser trocadas durante o envio e recebimento de uma mensagem. A utilização do atributo *DataContractAttribute* determina que uma classe poderá ser exposta através de um serviço WCF, e deve ser aplicado a todas as classes que estão referenciadas, como parâmetro ou tipo de retorno, em um contrato (*Interface*). Já os tipos primitivos, como *String*, *DateTime*, *Int32*, não precisam disso, já que podem ser serializados diretamente.

Já o atributo *DataMemberAttribute* deve ser aplicado nos campos e propriedades que o tipo possui e que devem ser expostos através do serviço. Esse atributo irá controlar a visibilidade do campo ou da propriedade para os clientes que consomem o serviço, não importando o modificador de acesso (*public*, *private*, etc.) que possui. O código abaixo define a classe *Usuario*:

using System;

using System.Runtime.Serialization;

[DataContract] //Opcional com .NET 3.5 + SP1

public class Usuario

{

    [DataMember] //Opcional com .NET 3.5 + SP1

    public string Nome { get; set; }

}

**Observação:** A partir do *Service Pack 1* do .NET Framework 3.5 esse comportamento foi mudado. Visando o suporte ao POCO (*Plain Old C# Objects*), a Microsoft tornou mais flexível a utilização de *data contracts* em serviços WCF, não obrigando às classes, propriedades e campos serem decorados com os atributos acima citados. Com isso, apenas propriedades do tipo escrita/leitura serão serializadas. A partir do momento que você decora a classe com o atributo *DataContractAttribute*, você também deverá especificar, via *DataMemberAttribute*, quais campos deverão ser serializados.

Vale lembrar também que o atributo *XmlSerializableAttribute* (*namespace System.Xml.Serialization*) e as *Interfaces IXmlSerializable*(*namespace System.Xml.Serialization*) e *ISerializable* (*namespace System.Runtime.Serialization*) também continuam sendo suportadas, permitindo que você customize como o objeto será serializados/deserializados pelo WCF ou qualquer outro recurso fornecido .NET Framework.

Uma vez que o contrato do serviço esteja definido e os possíveis tipos que ele expõe também estejam devidamente configurados, o próximo passo é a criação da classe que representa o serviço. Esta classe deverá implementar todos os membros expostos pela *Interface* que define o contrato do serviço, inclusive aqueles que não estão marcados com o atributo *OperationContractAttribute*, lembrando que a implementação de uma *Interface* em uma classe é uma imposição da linguagem, e não do WCF.

A implementação dos métodos poderá conter a própria regra de negócio, bem como pode servir de *wrapper* para algum outro componente ou serviço. Além disso, as classes que representam o serviço também podem configurar alguns outros recursos fornecidos pelo WCF e que estão acessíveis através de *behaviors*, como por exemplo: transações, sessões, segurança, etc., mas veremos isso mais detalhadamente abaixo. O WCF desacopla totalmente a regra do negócio de sua API e, justamente por isso, que é possível notar no código abaixo que a classe que representa o serviço não possui nenhuma configuração do WCF:

using System;

public class Servico : IContrato

{

    public Usuario RecuperarUsuario(string nome)

    {

        return new Usuario() { Nome = nome };

    }

}

Por si só esta classe não trabalha, pois deverá ser consumida pelo WCF. Mas afinal, como se determina que é esta classe responsável por atender as requisições? Isso é realizado através do *host*, ou melhor, da classe *ServiceHost*. Logo no construtor desta classe, você deverá passar uma instância da classe *Type*, apontando para a classe que representa o serviço e, obrigatoriamente, deverá implementar todos os possíveis contratos que são expostos através dos *endpoints*. A configuração do *host* para este exemplo será abordada na seção seguinte.

Grande parte dos atributos que vimos nesta seção disponibilizam várias propriedades que nos permitem interagir com o serializador/deserializador da mensagem e, além disso, permitem especificarmos algumas regras que serão validadas antes da abertura do *host* e, caso não sejam atendidas, uma exceção será disparada. Como essas propriedades influenciam nas mais variadas funcionalidades expostas pelo WCF, elas serão detalhadamente abordadas nos artigos que correspondem à sua utilização. Para conhecer os artigos disponíveis, consulte a listagem na seção *Explorando outras Funcionalidades*.

### Hosting

Uma das grandiosidades do WCF é a possibilidade de utilizar qualquer tipo de aplicação como *host*, ou seja, ele não tem uma dependência de algum software, como o IIS (*Internet Information Services*), como acontecia com os *ASP.NET Web Services*. O WCF pode expor serviços para serem acessados através dos mais diversos tipos de protocolos, como por exemplo: HTTP, TCP, IPC e MSMQ.

Atualmente temos três alternativas de *hosting*: *self-hosting*, IIS e o WPAS. Como há vários detalhes na criação e gerenciamento do *hosting*, ficaria muito extenso publicar cada detalhe, vantagens e desvantagens que cada uma das técnicas possui. Para maiores detalhes, consulte [este artigo](http://www.linhadecodigo.com.br/Artigo.aspx?id=1269) que explora cada uma das funcionalidades expostas pelo WCF para a interação com o *hosting*.

O *host* é representado dentro do WCF pela classe *ServiceHost* ou uma de suas variações e, é através dela que efetuamos várias configurações, como *endpoints*, segurança, etc. Em seu construtor, ela espera a classe que representa o serviço, podendo ser definida através de seu tipo (classe *Type*) ou através de uma instância desta mesma classe anteriormente criada (*Singleton*). Para o exemplo utilizado neste artigo, a configuração parcial do *host* fica da seguinte forma:

using System;

using System.ServiceModel;

using (ServiceHost host = new ServiceHost(typeof(Servico),

    new Uri[] { new Uri("net.<tcp://localhost:9393>") }))

{

    //endpoints

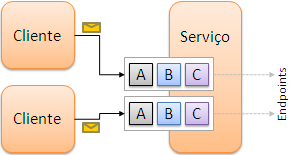
    host.Open();

    Console.ReadLine();

}

### Endpoints

Os *endpoints* são uma das características mais importantes de um serviço, pois é por onde toda a comunicação é realizada, pois fornece o acesso aos clientes do serviço WCF que está sendo disponibilizado. Para compor um *endpoint*, basicamente precisamos definir três propriedades que obrigatoriamente precisamos para poder trabalhar: *address* (A), *binding* (B) e *contract* (C) e, opcionalmente, definir alguns *behaviors*, que falaremos na sequência. A figura abaixo ilustra a estrutura dos *endpoints* e onde eles estão situados:



O *address* consiste em definir um endereço único que permitirá aos clientes saber onde o serviço está publicado. O endereço geralmente é definido através de uma instância da classe *Uri*. Essa classe fornece um construtor que recebe uma *string*, contendo o protocolo, o servidor, a porta e o endereço do serviço (usado para diferenciar entre muitos serviços no mesmo local), tendo a seguinte forma: *scheme://host[:port]/[path]*. Cada uma dessas configurações são representadas respectivamente pelas seguintes propriedades da classe *Uri*: *Scheme*, *Host*, *Port* e *AbsolutePath*.

O protocolo indica sob qual dos protocolos suportados pelo WCF o serviço será exposto. Atualmente temos os seguintes protocolos: HTTP (http://), TCP (net.tcp://), IPC (net.pipe://) e MSMQ (net.msmq://). O *host* refere-se à máquina onde o serviço irá ser executado, podendo inclusive referenciar o *localhost*. A porta permite especificarmos uma porta diferente do valor padrão e, quando omitida, ela sempre assumirá a porta padrão especificada pelo protocolo. E, finalmente, temos o *path*, que é utilizado quando desejamos diferenciar entre vários serviços expostos sob um mesmo protocolo, *host* e porta.

O *binding* indica como a comunicação será realizada com aquele *endpoint*, como por exemplo, qual transporte será utilizado (HTTP, TCP, etc), qual a codificação utilizada (*Binary* ou *Text*) para serializar a mensagem, segurança, suporte à transações, etc. O WCF disponibiliza vários *bindings*, e através da tabela abaixo podemos analisar as características de cada um deles (sendo as opções em negrito a configuração padrão):

|  |  |
| --- | --- |
| ***Binding*** | **Características** |
| *BasicHttpBinding* | * **Transporte:** HTTP. * **Segurança:** ***None***, *Transport*, *Message* e *Mixed*. * **Suporte à Transações:** Não. * ***Duplex*:** Não. * ***Sessions*:** Não. * ***Encoding*:** ***Text***. * **Modos de Transferência:** *Streaming* e ***Buffered***. |
| *WebHttpBinding* | * **Transporte:** HTTP. * **Segurança:** ***None*** e *Transport*. * **Suporte à Transações:** Não. * ***Duplex*:** Não. * ***Sessions*:** Não. * ***Encoding*:** ***Text*** e MTOM. * **Modos de Transferência:** *Streaming* e ***Buffered***. |
| *WSHttpBinding* | * **Transporte:** HTTP. * **Segurança:** *None*, *Transport*, ***Message*** e *Mixed*. * **Suporte à Transações:** Sim. * ***Duplex*:** Sim. * ***Sessions*:** Sim. * ***Encoding*:** ***Text*** e MTOM. * **Modos de Transferência:** ***Buffered***. |
| *WSDualHttpBinding* | * **Transporte:** HTTP. * **Segurança:** ***None***, ***Message*** e *Mixed*. * **Suporte à Transações:** Sim. * ***Duplex*:** Sim. * ***Sessions*:** Sim. * ***Encoding*:** ***Text*** e MTOM. * **Modos de Transferência:** ***Buffered***. |
| *WSFederationHttpBinding* | * **Transporte:** HTTP. * **Segurança:** ***None*** e *Message*. * **Suporte à Transações:** Sim. * ***Duplex*:** Não. * ***Sessions*:** Sim. * ***Encoding*:** ***Text*** e MTOM. * **Modos de Transferência:** ***Buffered***. |
| *NetTcpBinding* | * **Transporte:** TCP. * **Segurança:** *None*, ***Transport***, *Message* e *Mixed*. * **Suporte à Transações:** Sim. * ***Duplex*:** Sim. * ***Sessions*:** Sim. * ***Encoding*:** ***Binary****.* * ***Modos de Transferência:****Streaming e****Buffered****.* |
| *NetPeerTcpBinding* | * **Transporte:** Peer. * **Segurança:** ***Transport***. * **Suporte à Transações:** Não. * ***Duplex*:** Sim. * ***Sessions*:** Não. * ***Encoding*:** --. * **Modos de Transferência:** ***Buffered***. |
| *NetNamedPipeBinding* | * **Transporte:** IPC. * **Segurança:** *None* e ***Transport***. * **Suporte à Transações:** Não. * ***Duplex*:** Sim. * ***Sessions*:** Sim. * ***Encoding*:** ***Binary***. * **Modos de Transferência:** *Streaming* e ***Buffered***. |
| *NetMsmqBinding* | * **Transporte:** MSMQ. * **Segurança:** *None*, ***Transport*** e *Message*. * **Suporte à Transações:** Sim. * ***Duplex*:** Não. * ***Sessions*:** Sim. * ***Encoding*:** ***Binary***. * **Modos de Transferência:** ***Buffered***. |
| *MsmqIntegrationBinding* | * **Transporte:** MSMQ. * **Segurança:** ***Transport***. * **Suporte à Transações:** Sim. * ***Duplex*:** Não. * ***Sessions*:** Não. * ***Encoding*:** --. * **Modos de Transferência:** ***Buffered***. |

**Observação**: Além da lista de *bindings* que vimos acima, temos ainda a classe *CustomBinding* que, como o próprio nome indica, possibilita a criação de um *binding* customizado, definindo qual o meio de transporte, codificação, suporte ou não à transações, etc.

Finalmente, a última característica de um *endpoint* é o contrato. Como já vimos acima, o contrato é representado por uma *Interface* e, uma vez que ele é definido como um contrato de serviço, são esses membros que serão disponibilizados aos clientes em forma de operações, definindo os parâmetros de entrada e o tipo de retorno e o formato da mensagem (*request-reply*, *one-way* ou *duplex*).

O que diferencia uma *Interface* normal de uma *Interface* que será utilizada como contrato de serviço. Os atributos que devem ser decorados na mesma (*ServiceContractAttribute*) e também naqueles membros que farão parte do serviço (*OperationContractAttribute*), devendo também se atentar aos tipos que são expostos, pois tipos complexos também devem ser marcados com um atributo especial (*DataContractAttribute*), e as propriedades que ele irá expor devem ser decoradas com o atributo *DataMemberAttribute*, tudo como já foi explicado acima.

Depois de conhecer cada uma das características de um *endpoint*, é necessário entender como criar e configurar um *endpoint*. Cada *endpoint* é representado por uma classe chamada *ServiceEndpoint* (*namespace System.ServiceModel.Description*) e possui várias propriedades que expõem exatamente as configurações de um *endpoint* como *Address*, *Binding* e *Contract*. A criação pode ser realizada de duas formas: uma delas criando e configurando a instância da classe *ServiceEndpoint* e adicionando-a à coleção de *Endpoints* do *host*, e a segunda e mais convencional forma é através do método *AddServiceEndpoint* da classe *ServiceHost*. O código abaixo ilustra a configuração de um *endpoint*:

using System;

using System.ServiceModel;

using System.ServiceModel.Description;

using (ServiceHost host = new ServiceHost(typeof(Servico),

    new Uri[] { new Uri("net.<tcp://localhost:3933>") }))

{

    host.AddServiceEndpoint(typeof(IContrato), new NetTcpBinding(), "srv");

    host.Open();

    Console.ReadLine();

}

Entre os vários *overloads* que existem do método *AddServiceEndpoint*, um deles aceita uma instância da classe *Type* representando o contrato, a instância de um dos *bindings* disponíveis e, finalmente, o endereço que, por sua vez, poderá ser absoluto ou relativo. Outro detalhe importante é que precisamos ter um endereço (*base address*) definido no construtor da classe *ServiceHost* com o mesmo protocolo utilizado pelo *binding* que, no caso acima, deve ser net.tcp. E, para finalizar, você pode criar quantos *endpoints* desejar, inclusive com protocolos diferentes.

### Behaviors

Até o momento criamos o contrato, os tipos que fazem parte dele e também a classe que representa o serviço. Com exceção da configuração *host* e dos atributos, nenhuma outra funcionalidade do WCF foi adicionada ao serviço. Como o exemplo é muito simples, nenhuma configuração adicional foi necessária, mas e quando os serviços necessitam o suporte de sessões, transações, funcionalidades de segurança, etc.?

Através de *behaviors* podemos modificar como a mensagem será processada pelo WCF, e boa parte das funcionalidades do WCF são expostas por intermédio de *behaviors*, como segurança, transações, *throttle*, metadados, etc. Como os *behaviors* são aspectos de execução do WCF, a inserção ou remoção deles não afeta a comunicação entre as partes.

Atualmente temos três escopos diferentes de *behaviors*: *operation behavior*, *endpoint behavior* e *service behavior*. O primeiro deles afetará apenas a execução de uma operação específica. Os *endpoint behaviors* são usados exclusivamente para um *endpoint*. Finalmente, os *service behaviors* possibilita a execução de algo válido para o serviço como um todo, independentemente de quantos *endpoints* houverem. O WCF já traz vários *behaviors* criados prontos para serem utilizados e, possibilita também criarmos nossos próprios *behaviors*, apenas implementando as *Interfaces IOperationBehavior*, *IEndpointBehavior* ou *IServiceBehavior*. Os *behaviors*customizados serão abordados em um futuro artigo.

Alguns *behaviors* são implementados em forma de atributo, podendo ser aplicado diretamente no tipo ou membro onde ele deverá influenciar. Um exemplo disso são os atributos *OperationBehaviorAttribute* e *ServiceBehaviorAttribute*, onde o primeiro deles pode ser aplicado em uma operação e customiza informações relacionadas à segurança e transações. Já o atributo *ServiceBehaviorAttribute*fornece uma infinidade de detalhes relacionados ao serviço, tais como: modo de gerenciamento de instância, sincronização, transações, etc. Essas configurações não estão relacionadas ao contrato, mas sim à classe que representa o serviço, e justamente por isso que os *behaviors* devem ser aplicados nela, conforme é exibido abaixo:

using System;

using System.ServiceModel;

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]

public class Servico : IContrato

{

    [OperationBehavior(TransactionScopeRequired = true)]

    public Usuario RecuperarUsuario(string nome)

    {

        return new Usuario() { Nome = nome };

    }

}

Há ainda outros *behaviors* que não são expostos como atributos, como é o caso do *ServiceMetadataBehavior* que é um *behavior* que é aplicado no serviço como um todo, gerenciando como expor os metadados (mais detalhes sobre ele abaixo). Para criar um *behaviors*de serviço, você deverá instanciá-lo e adicionar na coleção de *Behaviors* da classe *ServiceHost*, coleção que é exposta através da propriedade *Description*.

A forma declarativa (via atributos) de adicionarmos os *behaviors* não se resume apenas a isso. O WCF permite que *behaviors* sejam adicionados a partir de arquivos de configuração, o que torna esse processo muito mais flexível. Veremos mais detalhes sobre esta técnica mais abaixo.

### Metadados - WSDL

Os metadados têm um papel importante dentro do WCF e de qualquer *web service*. Regidos por padrões mundialmente conhecidos, como WSDL (*Web Services Description Language*) ou o *WS-Policy/WS-MetadataExchange*, utilizam a linguagem XML/XSD para descrever um serviço, especificando suas operações e tipos de dados que ele suporta.

Por padrão, o WCF não publica os metadados, tendo que fazer isso explicitamente justamente para diminuir a superfície de ataque. Para efetuar essa publicação, basicamente adicionamos um *behavior* chamado *ServiceMetadataBehavior* que está contido no *namespace System.ServiceModel.Description* na coleção de *behaviors* do serviço. Esse *behavior* possui uma propriedade boleana chamada *HttpGetEnabled* que, quando definida como *True*, permite acessar o WSDL a partir de um navegador, ou melhor, de um endereço HTTP. Essa técnica é bastante utilizada para publicação deste documento, já que muitas vezes o serviço é exposto a partir de um protocolo que está inacessível devido às políticas de *firewall*.

Quando definimos a propriedade *HttpGetEnabled* como *True*, obrigatoriamente devemos ter um *base address* com um endereço HTTP especificado, caso contrário, uma exceção do tipo *InvalidOperationException* será disparada antes da abertura do *host*, informando a ausência de um endereço HTTP. Uma alternativa para isso é utilizar a propriedade *HttpGetUrl* da classe *ServiceMetadataBehavior*, especificando um endereço absoluto para o documento WSDL. O código a seguir exibe como efetuar essa configuração:

using System;

using System.ServiceModel;

using System.ServiceModel.Description;

using (ServiceHost host = new ServiceHost(typeof(Servico),

    new Uri[] { new Uri("net.<tcp://localhost:3933>") }))

{

    host.Description.Behaviors.Add(

        new ServiceMetadataBehavior()

        {

            HttpGetEnabled = true,

            HttpGetUrl = new Uri("<http://localhost:9393/>")

        });

    host.AddServiceEndpoint(typeof(IContrato), new NetTcpBinding(), "srv");

    host.Open();

    Console.ReadLine();

}

No exemplo acima, o WSDL estará disponível a partir do endereço *http://localhost:9393/*, ou seja, estamos publicando o WSDL em um protocolo diferente do qual o serviço será disponibilizado. É importante dizer que os metadados não necessariamente precisam ser expostos via HTTP. Serviços podem disponibilizar a sua descrição a partir de TCP ou IPC. A única exceção é quando utilizamos o *Message Queue* que, por sua vez, não permite a publicação de metadados, obrigando-nos a escolher um protocolo diferente, como o HTTP.

Há ainda uma outra forma de publicar o documento WSDL, que é criando um *endpoint* específico para isso, conhecido como *Metadata Exchange Endpoint* ou simplesmente *MEX Endpoint*. Como qualquer outro *endpoint*, este tipo especial exige também um endereço, um contrato e um *binding* e, com exceção do endereço, as duas outras características possuem algumas considerações.

A primeira delas, o contrato, deverá ser sempre a *Interface IMetadataExchange* que, por sua vez, também está contido no *namespace System.ServiceModel.Description*. Os membros desta *Interface* são irrelevantes para o desenvolvedor do serviço, já que ela será usada exclusivamente pelo *runtime* do WCF. É importante notar que, mesmo que ela faça parte do serviço, ela não deve ser implementada na classe que o representa mas, mesmo utilizando esta técnica, é necessário que o *behavior ServiceMetadataBehavior* também esteja adicionado.

Para expor os metadados também é necessário efetuar algumas mudanças no *binding*. Para evitar a configuração do *binding* a cada *endpoint* de metadados que criamos, a Microsoft disponibilizou uma classe estática chamada *MetadataExchangeBindings*, que possui vários métodos que retornam *bindings* devidamente configurados para suportar a publicação dos metadados. Entre estes métodos temos: *CreateMexHttpBinding*, *CreateMexHttpsBinding*, *CreateMexNamedPipeBinding* e *CreateMexTcpBinding*, e escolher um deles dependerá do protocolo onde se deseja publicar o documento. O código abaixo ilustra como efetuar a configuração de um *MEX Endpoint*:

using System;

using System.ServiceModel;

using System.ServiceModel.Description;

using (ServiceHost host = new ServiceHost(typeof(Servico),

    new Uri[] { new Uri("net.<tcp://localhost:3933>") }))

{

    host.Description.Behaviors.Add(new ServiceMetadataBehavior());

    host.AddServiceEndpoint(typeof(IContrato), new NetTcpBinding(), "srv");

    host.AddServiceEndpoint(

        typeof(IMetadataExchange),

        MetadataExchangeBindings.CreateMexTcpBinding(),

        "mex");

    host.Open();

    Console.ReadLine();

}

**Observação:** Ambas as formas de publicar os metadados também podem ser configuradas de forma declarativa, ou seja, através do arquivo de configuração. Veremos mais detalhes sobre isso na próxima seção deste artigo.

### Configuração Declarativa vs. Imperativa

Grande parte das configurações que vimos no decorrer deste artigo pode ser feita de duas formas: declarativa ou imperativa, onde cada uma das duas tem suas vantagens e desvantagens. Com a forma declarativa temos uma maior flexibilidade, já que qualquer alteração a nível de configuração do serviço pode ser realizada sem a necessidade de recompilar a aplicação. Podemos abrir e editar o arquivo de configuração através de um editor de texto, mas isso está propício à erros, já que não é fortemente tipado. Para contornar esse problema, podemos utilizar um software fornecido pela própria Microsoft, chamado *Microsoft Service Configuration Editor*, que fornece uma *interface* gráfica para isso.

Já no modo imperativo, toda criação e manipulação de qualquer configuração, seja do *binding*, *endpoints* ou *behaviors*, será realizada diretamente via código (C# ou VB.NET). Entre as vantagens que temos com este modo é a facilidade para montar as configurações de forma dinâmica, podendo essas configurações virem de algum repositório, efetuar condicionais, etc., situações que o modo declarativo não suporta. Como essa técnica não coloca nenhuma informação no arquivo de configuração, configurações acidentais não danificam o sistema.

Como todo o exemplo foi baseado na configuração de forma imperativa, esta seção mostrará como efetuar a configuração do *binding*, *behaviors* e *endpoints* a partir do arquivo de configuração (*App.config* ou *Web.config*). O fato de utilizá-lo não elimina a necessidade de criar e abrir um *host* (*ServiceHost*); ao detectar que existe um arquivo contendo as configurações do serviço, elas serão carregadas a partir dele. Essa "amarração" é efetuada a partir do atributo *name* do elemento *service*, como vemos abaixo:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <configuration>    <system.serviceModel>      <services>        <service name="Host.Servico" behaviorConfiguration="srvBehaviorConfig">          <host>            <baseAddresses>              <add baseAddress="net.<tcp://localhost:9393/>"/>            </baseAddresses>          </host>          <endpoint            name="Srv"            address="srv"            contract="Host.IContrato"            binding="netTcpBinding"            bindingConfiguration="bindingConfig" />          <endpoint            name="Mex"            address="mex"            contract="IMetadataExchange"            binding="mexTcpBinding" />        </service>      </services>      <bindings>        <netTcpBinding>          <binding name="bindingConfig">            <security mode="None" />          </binding>        </netTcpBinding>      </bindings>      <behaviors>        <serviceBehaviors>          <behavior name="srvBehaviorConfig">            <serviceMetadata              httpGetEnabled="true"              httpGetUrl="<http://localhost:2322/>"/>          </behavior>        </serviceBehaviors>      </behaviors>    </system.serviceModel>  </configuration> |

No código acima, devemos notar que logo após o elemento *system.serviceModel* há um sub-elemento chamado *services* que é uma coleção de serviços, e cada serviço é representado pelo elemento *service*. Esse elemento possui um atributo chamado *name* que deve refletir o *full name* da classe que representa o serviço, incluindo o *namespace*. Na sequência definimos os possíveis *base address* do serviço e, como essa é uma configuração relacionada à um serviço específico, ela deverá estar dentro do elemento *service*.

Ainda dentro do elemento *service* há um sub-elemento chamado *endpoint*, onde podemos definir cada *endpoint* (*address*, *binding* e *contract*), podendo adicionar quantos forem necessários. Além da configuração do ABC, o elemento *endpoint* possui um atributo chamado *bindingConfiguration* que podemos apontar para uma outra seção dentro do mesmo arquivo de configuração, contendo a configuração do *binding*, como segurança, modos de transferência, etc. No exemplo acima, o *endpoint* "srv" define o atributo *bindingConfiguration* como *bindingConfig*.

O elemento *service* disponibiliza um atributo chamado *behaviorConfiguration* e que permite apontar para uma seção dentro do mesmo arquivo de configuração, contendo os possíveis *behaviors* de serviço e *endpoint* que, no exemplo acima, habilitamos a publicação dos metadados. A finalidade da configuração dos *bindings* e dos *behaviors* estar fora do elemento *service* é justamente por questões de reutilização, já que podemos ter a mesma configuração para vários serviços que são executados nesta mesma aplicação.

Como dito anteriormente, ainda é necessária a criação da classe *ServiceHost*, mas com uma pequena mudança. A classe que representa o serviço deve continuar sendo informada, enquanto o segundo parâmetro, um *array* de objetos do tipo *Uri*, deverá ser informado como um *array* vazio, caso contrário, uma exceção do tipo *ArgumentNullException* será disparada. Podemos notar através do código abaixo que não há mais a necessidade de configurar os *endpoints* ou os *behaviors*, pois eles serão carregados do arquivo de configuração.

using System;

using System.ServiceModel;

using (ServiceHost host = new ServiceHost(typeof(Servico), new Uri[] {  }))

{

    host.Open();

    Console.ReadLine();

}

### Configuração do Cliente

Para que possamos fazer o uso do serviço em aplicações cliente, é necessário efetuar a referência deste serviço. A referência consiste em criar uma classe que encapsulará todo o acesso para o serviço, que também é conhecida como *proxy*. Este processo irá ler os metadados do serviço, criando o *proxy* com os mesmos membros expostos, dando a impressão ao consumidor que está chamando uma classe local mas, em tempo de execução, a requisição será encaminhada para o serviço remoto.

Podemos efetuar a criação do *proxy* de três formas diferentes. A primeira delas é através da referência do serviço por meio da IDE do *Visual Studio .NET* que, por sua vez, fornece uma opção chamada *Add Service Reference* que exige a referência para o serviço. A segunda opção é fazer uso do utilitário **svcutil.exe** que, dada uma URL até o serviço, ele também é capaz de gerar a classe que representará o *proxy*. Ambas as opções também automatizam a criação do arquivo de configuração, que fornece de forma declarativa todas as configurações do serviço.

Ao efetuar a referência, será criada uma representação local do contrato (*Interface*), bem como os tipos complexos que fazem parte do serviço. Além disso, a classe que representa o *proxy* herda diretamente da classe abstrata e genérica ClientBase<TChannel> que fornece toda a implementação necessária para permitir aos clientes se comunicarem com o respectivo serviço. Essa classe irá configurar em tempo de execução as propriedades necessárias para efetuar a requisição, extraindo tais informações do arquivo de configuração. É a partir desta classe que começamos a criar o código cliente para efetuar a requisição:

using System;

using Client.Servico;

using (ContratoClient proxy = new ContratoClient())

{

    Usuario usuario = proxy.RecuperarUsuario("Israel Aece");

    Console.WriteLine(usuario.Nome);

}

**Observação:** Se envolvermos o *proxy* em um bloco *using* (lembre-se de que ele será transformado em *try/finally*), é necessário tomarmos um certo cuidado. Não porque o método *Dispose* não será chamado, mas sim onde a execução deste método afetará a aplicação cliente. Vamos supor que algum erro ocorra durante a execução da operação e, como já era de se esperar, o bloco *finally* será disparado, chamando o método *Dispose* do *proxy*. Neste caso, o método *Dispose* não faz mais nada a não ser invocar o método *Close*. O problema aqui é que o método *Close* poderá exigir algumas atividades extras, necessitando fazer alguma outra comunicação com o serviço e, se neste momento algum erro ocorrer, uma exceção do tipo *CommunicationObjectFaultedException* será disparada, mascarando o real problema. Finalmente, a opção para contornar esse possível problema é a chamada do método *Abort*, que encerra imediatamente a comunicação entre o cliente e o serviço, assim como é demonstrado [neste endereço](http://msdn.microsoft.com/en-us/library/aa355056.aspx).

A terceira e última forma que existe para efetuar a comunicação entre o cliente e o serviço é realizar toda a configuração de forma manual. Isso obriga a conhecer exatamente o *binding* e suas respectivas configurações, qual o endereço onde o serviço está publicado e, principalmente, a *Interface* do contrato e os tipos que ela expõe devem estar compartilhados entre o cliente e o serviço. Apesar deste compartilhamento ser simples de realizar, pois basta isolar os tipos em um *assembly* (DLL), o problema é quando o número de clientes aumenta consideravelmente, dificultando a distrubuição. De qualquer forma, abaixo consta um exemplo de como efetuar essa configuração:

using System;

using System.ServiceModel;

using (ChannelFactory<IContrato> srv =

    new ChannelFactory<IContrato>(new NetTcpBinding(),

    new EndpointAddress("net.<tcp://localhost:9393/>")))

{

    Usuario u = srv.CreateChannel().RecuperarUsuario("Israel Aece");

    Console.WriteLine(u.Nome);

}

### Explorando outras Funcionalidades

O WCF fornece diversas funcionalidades que visam performance, segurança, disponibilidade, entre outros que podemos incorporar em nossos serviços para torná-los muito mais ricos, flexíveis e de fácil manipulação. Cada uma das principais funcionalidades fornecidas por ele possui um artigo exclusivo que abordará na íntegra cada uma delas. Para acessar esses artigos, consulte a listagem abaixo:

[**Tipos de Mensagens**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2154): Tradicionalmente, em qualquer tipo de aplicação, podemos criar um método que faz alguma tarefa. Ao criá-lo, podemos consumí-lo na mesma aplicação ao até mesmo referenciar a classe em que ele está contido e também consumí-los nos mais variados projetos. Ao realizar a chamada para este método, devemos esperar a sua execução e, quando finalizada, damos continuidade na execução do programa. Ao criar uma operação em um serviço WCF, ela também se comportará da mesma forma. Mas essa não é a única alternativa fornecida pelo WCF. O foco deste artigo é explorar tais alternativas e como elas influenciam na configuração e implementação e execução do serviço.

[**Hosting**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1269): Como já sabemos, WCF - Windows Communication Foundation - é parte integrante do .NET Framework 3.0. Ele fornece um conjunto de classes para a construção e hospedagem de serviços baseando-se na arquitetura SOA (Service Oriented Architecture), podendo expor tais serviços para serem acessados através dos mais diversos tipos de protocolos, como por exemplo: HTTP, TCP, IPC e MSMQ. Atualmente existem três tipos de hosts para serviços construídos em WCF: self-hosting (através da classe ServiceHost), IIS (Internet Information Services) e WAS (Windows Activation Services - Windows Vista) e, é exatamente isso que este artigo abordará, ou seja, como configurar cada um desses hosts para expor os serviços construídos em WCF.

[**Error Handling**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2158): Independentemente de que tipo de aplicação estamos criando, erros sempre podem acontecer. O mesmo serve para serviços, não estando isentos disso. Em se tratando de serviços, os erros podem ser os mais variados possíveis, havendo problemas a nível de transporte (protocolo), na entrega/recebimento da mensagem, no runtime ou até mesmo (e é o mais comum) na execução da operação (método). O WCF fornece várias técnicas para analisar e tratar os possíveis erros que ocorrem durante a execução do serviço. O grande desafio aqui é como fazer com que este problema (erro) seja passado para o cliente que o consome independentemente da plataforma, dando à ele a capacidade de saber o que ocorreu e como contorná-lo, mantendo a aplicação cliente e proxy estáveis. Esse artigo abordará como devemos proceder para disparar erros, notificar o cliente e, como ele pode fazer para tratar os erros que ocorrem.

[**Gerenciamento de Instâncias**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1987): O gerenciamento de instância é uma técnica que é utilizada pelo WCF ou qualquer outra tecnologia de computação distribuída que determina como e por quem as requisições dos clientes serão atendidas. A escolha do modo de gerenciamento de instâncias interfere diretamente na escalabilidade, performance e transações de um serviço/componente, além de termos algumas mudanças à nível de implementação de contrato, que precisamos nos atentar para garantir que o mesmo funcione sob o modelo de gerenciamento escolhido. A finalidade do artigo consiste, basicamente, em mostrar cada uma das três técnicas disponíveis pelo WCF mas, também, abordando os seus respectivos benefícios e algumas técnicas que circundam esse processo e que, de alguma forma, estão ligadas e influenciam na escolha e/ou implementação. O artigo também abordará os cuidados que devemos ter na escolha e implementação de cada uma das técnicas fornecidas.

[**Sincronização**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2153): Ao expor um serviço para que ele seja consumido, devemos nos atentar à possibilidade deste serviço ser acessado simultaneamente. Isso ocorre quando múltiplas requisições (threads) tentam acessar o mesmo recurso ao mesmo tempo. A possibilidade de acessos simultâneos poderá acontecer dependendo do tipo de gerenciamento de instância escolhido para o serviço. A finalidade deste artigo é mostrar as três opções fornecidas pelo WCF para tratar a concorrência e, além disso, exibir algumas das várias técnicas de sincronização fornecidas pelo .NET Framework e que poderão ser utilizadas em conjunto com o WCF.

[**Transferência e Codificação de Dados**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1983): Um dos maiores benefícios que serviços da Web tem em relação à tecnologias de comunicação distribuídas é o uso de XML como base da codificação, permitindo assim, a interoperabilidade entre as mais diversas plataformas. Entretanto, a escolha do padrão de codificação e a forma de transferência destas informações a serem adotados pelo serviço influenciará diretamente na performance e interoperabilidade do mesmo e também daqueles que o consomem. O WCF fornece algumas técnicas e alternativas que podemos adotar durante a configuração ou criação de um determinado serviço. A finalidade deste artigo é mostrar como implementar tal configuração/criação e analisar os impactos (inclusive a nível de contrato), benefícios e restrições de cada uma destas técnicas.

[**Chamadas Assíncronas**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1967): Muitas vezes desenvolvemos um método para desempenhar alguma tarefa e, depois de devidamente codificado, invocamos o mesmo a partir de algum ponto da aplicação. Dependendo do que este método faz, ele pode levar certo tempo para executar e, se o tempo for consideravelmente alto, podemos começar a ter problemas na aplicação, pois como a chamada é sempre realizada de síncrona, enquanto o método não retornar, a execução do sistema que faz o uso do mesmo irá congelar, aguardando o retorno do método para dar seqüência na execução. A finalidade deste artigo é mostrar como implementar o processamento assíncrono tanto do lado do cliente (*proxy*) bem como do lado do servidor (contrato) em serviços WCF.

[**Throttling e Pooling**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1996): Através do gerenciamento de instância de um serviço podemos definir qual a forma de criação de uma instância para servir uma determinada requisição. Essa configuração que fazemos à nível de serviço, através de um *behavior*, não impõe nenhuma restrição na quantidade de instância e/ou execuções concorrentes que são realizadas e, dependendo do volume de requisições que o serviço tenha ou até mesmo a quantidade de recursos que ele utiliza, podemos degradar consideravelmente a performance. O *Throttling* possibilita restringirmos a quantidade de sessões, instâncias e chamadas concorrentes que são realizadas para um serviço. Além do *Throttling*, ainda há outra funcionalidade que pode ser utilizada em um serviço, que é o *Pooling* de objetos. Este artigo explicará como proceder para efetuar a configuração do *Throttling* e suas implicações; também falaremos supercialmente sobre a estrutura do *Pooling* e como implementá-lo.

[**Transações**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2199): Uma necessidade existente em muitas aplicações é assegurar a consistência dos dados durante a sua manipulação. Ao executar uma tarefa, precisaremos garantir que, se algum problema ocorrer, os dados voltem ao seu estado inicial. Dentro da computação isso é garantido pelo uso de transações. As transações já existem há algum tempo, e a finalidade deste artigo é mostrar as alternativas que temos para incorporá-las dentro de serviços e clientes que fazem o uso do WCF como meio de comunicação.

[**Transações**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2199): Uma necessidade existente em muitas aplicações é assegurar a consistência dos dados durante a sua manipulação. Ao executar uma tarefa, precisaremos garantir que, se algum problema ocorrer, os dados voltem ao seu estado inicial. Dentro da computação isso é garantido pelo uso de transações. As transações já existem há algum tempo, e a finalidade deste artigo é mostrar as alternativas que temos para incorporá-las dentro de serviços e clientes que fazem o uso do WCF como meio de comunicação.

[**Reliable Messages**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2198): Ao consumir um serviço WCF podemos interagir com o mesmo através de diferentes mecanismos, tais como, *request-reply* ou *one-way*. Sabemos que, independente do tipo que você utilize, a mensagem trafega entre o cliente e o serviço através da rede, utilizando o protocolo especificado pelo *binding*. Com isso, uma das principais preocupações que se tem é com relação a garantia de entrega da mensagem ao seu destinatário, pois problemas com a rede podem acontecer, fazendo com que a mensagem seja interceptada ou simplesmente perdida. A finalidade deste artigo é apresentar uma técnica disponibilizada pelo WCF, para evitar que problemas como estes comprometam a consistência e execução de um serviço.

[**Message Queue**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2201): Ao efetuar uma chamada para uma operação de um determinado serviço, desejamos que ela seja sempre executada. Mas nem sempre há como garantir isso, já que o serviço que atende as requisições, por algum motivo, está indisponível naquele momento. Isso fará com que as requisições sejam rejeitadas e o cliente somente conseguirá executá-la quando o serviço estiver novamente no ar. Para garantir a entrega da mensagem e o processamento assíncrono da operação (mesmo quando o serviço estiver *offline*), o WCF faz uso do *Microsoft Message Queue*. Este artigo irá explorar as funcionalidades e, principalmente, os benefícios fornecidos por essa integração.

[**Serviços RESTFul**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1933): A versão 3.5 do Windows Communication Foundation introduziu uma nova forma de expor e consumir serviços. Esse novo modelo, também conhecido como Web Programming Model, permite o consumo destes serviços através dos mais variados clientes, como é o caso dos navegadores. A finalidade deste artigo é explorar os tipos que estão disponíveis para tornar isso possível.

[**Syndication**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2306): *Web Syndication* é uma forma popularmente conhecida que temos para publicar um conteúdo de um determinado site para outros sites ou pessoas. Esta técnica fornece aos seus consumidores um pequeno sumário ou, às vezes, em sua íntegra, conteúdos que foram recentemente adicionados ao site. A partir da versão 3.5, o WCF disponibiliza uma API para suportar a criação de serviços que expõem o seu conteúdo em um dos formatos conhecidos para o *syndication* e esta API que será abordada no decorrer deste artigo.

[**Tracing**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1300): Toda e qualquer tipo de aplicação sempre exige uma forma de armazenar possíveis erros que possam acontecer durante a sua execução. Isso irá ajudar imensamente para diagnosticarmos problemas que ocorrem e, conseqüentemente, facilitar na sua solução. Isso não é diferente em serviços WCF. Esse artigo tem a finalidade de demonstrar a integração que os serviços WCF possibilitam para a captura e persistência dos erros para uma posterior análise.

[**Know Types**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1338): É muito comum em qualquer linguagem orientada a objetos, criarmos uma classe base e que, a partir dela, criar classes derivadas. Além disso, um dos grandes benefícios que temos com a orientação a objetos é a possibilidade de declararmos uma variável do tipo da classe base e atribuirmos a ela uma instância de uma classe concreta e, da mesma forma, podemos ter uma função em que em seus parâmetros os seus tipos são especificados com o tipo da classe base e, conseqüentemente, podemos também passar instâncias das classes derivadas. Infelizmente não funciona da mesma forma quando falamos de serviços que são expostos a partir do WCF. Neste cenário, por padrão, você não pode usar uma classe derivada ao invés de uma classe base. É justamente esta funcionalidade disponibilizada pelo WCF que iremos analisar neste artigo.

[**Segurança**](http://www.linhadecodigo.com.br/Artigo.aspx?id=2304): Um dos grandes desafios de um software é a segurança do mesmo. Em qualquer software hoje em dia a segurança não consiste apenas em autenticar um usuário, mas também que direitos ele tem dentro do software. As coisas ficam mais complicadas quando falamos de um ambiente distribuído, tornando o processo de autenticação e autorização um pouco mais complexo e, como se não bastasse, temos que nos preocupar com a proteção das requisições que viajam entre o cliente e o servidor. A finalidade deste artigo é exibir todas as possibilidades que temos para manipular a segurança em serviços WCF.

[**Integração com MembershipProvider e RoleProvider**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1297): Por padrão, serviços WCF utilizam as identidades e grupos do Windows para autenticação e autorização, respectivamente. Um dos grandes problemas é quando temos isso sendo disponibilizado através de uma aplicação web, pois iria requerer que todos os clientes que acessam via web estivessem devidamente cadastrados dentro do Windows; além disso, há o problema com relação aos grupos de usuários, já que muitas vezes não temos acesso para cadastrá-los e, quando isso não é um problema, podemos ter um problema adicional quando estivermos rodando em culturas de servidores diferentes. Com isso, dificilmente uma aplicação ou serviços que são expostos para a internet utilizam as contas e grupos do Windows. A solução é que felizmente o ASP.NET 2.0 fornece uma infraestrutura completa para o gerenciamento de autenticação e autorização, chamada de *Provider Model*. Essa integração é o tema deste artigo.

[**Autenticação e Autorização Customizada**](http://www.linhadecodigo.com.br/artigo/2314/wcf-introducao.aspx): O WCF fornece várias possibilidades de gerenciar a autenticação e autorização dentro dos serviços. Uma dessas possibilidades é customizar como o WCF deverá autenticar e autorizar o cliente, analisando as suas credenciais, verificando se essas são válidas em um determinado repositório, determinar quais são os direitos que o cliente tem no serviço e, finalmente, conceder ou negar o acesso à alguma operação baseando-se em seus privilégios. A finalidade deste artigo é analisar os passos necessários para essa customização.

[**Partial Trust**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1812): Na primeira versão do WCF - .NET Framework 3.0 - ele não era suportado em ambientes que estavam sob Partial Trust, o que obrigava muitos clientes a conceder mais direitos do que o necessário para poder executar/invocar um serviço escrito em WCF. Depois de muitas requisições, a Microsoft decidiu afrouxar essa segurança com o lançamento do .NET Framework 3.5, permitindo (com várias restrições) que serviços sejam invocados a partir de um ambiente parcialmente confiável. A finalidade deste artigo é exibir como criar um proxy para um serviço que expõe um endpoint não suportado neste ambiente.

[**Consumindo serviços no AJAX**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1383): Uma das grandes partes do .NET Framework 3.0 foi o WCF. Quando ele foi lançado, várias formas de acessar os serviços WCF também foram disponibilizadas. Entre as formas, conhecidas como *endpoints*, podemos citar algumas, tais como: HTTP, TCP e MSMQ. Com a vinda do ASP.NET AJAX, surgiu a necessidade de consumir serviços WCF diretamente dentro deste tipo de aplicação. Através do Visual Studio .NET 2008 e o .NET Framework 3.5, a Microsoft se preocupou com a necessidade de consumir serviços WCF no AJAX e aproveitou esta oportunidade para criar um *binding*. Este *binding*, chamado de *WebHttpBinding*, trata-se de um novo tipo de *binding* que permite a criação de um *endpoint* para ser consumido por aplicações AJAX e que será discutido neste artigo.

[**Expondo componente COM+**](http://www.linhadecodigo.com.br/Artigo.aspx?id=1903): Com o surgimento do WCF, uma plataforma de comunicação unificada, a Microsoft não se esqueceu do legado, ou seja, de componentes grandes e complexos hospedados no COM+ e, possibilita a utilização do WCF para expor esse componente através do HTTP (ou qualquer outra forma). Ao contrário do que acontecia anteriormente com Web Services, não precisamos recorrer ao *Component Services* para isso. Junto com o SDK do .NET Framework 3.X, a Microsoft disponibiliza uma ferramenta chamada *Microsoft Service Configuration Editor* que, dentre todas as funcionalidades disponibilizadas, uma delas é a possibilidade de integração de um componente COM+ a um serviço WCF, que será tema deste artigo.

### Conclusão:

O WCF fornece uma grande quantidade de funcionalidades que facilmente podem ser adicionadas em serviços. Além disso, grande parte dessas funcionalidades podem ser configuradas de forma declarativa, através de arquivos de configuração que, na maioria dos casos, traz uma enorme flexibilidade. O artigo mostrou os conceitos básicos necessários para a criação e consumo de um serviço WCF, que são informações importantes para dar sequência na leitura dos artigos acima, que exploram cada uma das principais funcionalidades.