Get More Refcardz! Visit refcardz.com

DZone Refcardz

CONTENTS INCLUDE:

Getting Started with Windows Communication Foundation 4.0

By Scott Seely

ABOUT WCF

Since .NET 3.0, Windows Communication Foundation (WCF) is the preferred messaging system in .NET applications. It provides an abstraction layer over transports. This abstraction allows developers to focus on the types of messages their applications need to send and receive. It removes the need for developers to have intimate knowledge of how the messages themselves are sent and received.

Instead, developers focus on a concept called an endpoint which listens for and receives messages. Endpoints are built using three items: Address, Binding, and Contract.

The address defines where a message is sent. http://www.dzone.com/ is an address.

A binding describes how to send the message. The binding contains information such as the transport to use, how to encode messages, and security requirements.

A contract defines how a message is structured. The contract defines the message exchange pattern (MEP) used for each exchange as well as which messages initiate a conversation or stop a conversation. All message exchanges fall into one of three message exchange patterns, or MEPs: one-way, request-response, and duplex. One-way messages can act as event notifications. Request-response messages send some data and expect a response in a particular format. Duplex messages are a conversation. In duplex, when one caller initiates a conversation, that caller promises to make a set of one-way and request-response messages available to the callee. Typically, duplex MEPs make use of sessions as well.

CONFIGURATION OVERVIEW

Configuration is a big feature in WCF. It permeates much of the messaging framework. This section provides a map of configuration. The configuration section group, <system.configuration>, contains all WCF configuration settings. Within this configuration section group, 14 different configuration sections exist. Some of those sections have relationships with other sections. One, system.serviceModel/extensions, is a configuration section for configuration only!

behaviors	Contains configuration for shared behaviors that can be applied to endpoints and services.
bindings	Contains information for shared bindings used to create message processing pipelines. A computer program that can run independently and can propagate a complete working version of itself onto other hosts on a network.
client	Contains information for clients to communicate with services. Lookups are performed using a two part key: the name of the configuration and the name of the contract.
comContracts	An integration point used between COM+ and WCF to configure the name and namespace for any hosted services. This mimics the functionality normally provided by the [ServiceContract] attribute via the Name and Namespace properties.
commonBehaviors	Defines a collection of behaviors applied to all services and endpoints. These behaviors add to any others already present on a service.
diagnostics	Diagnostic settings for WCF, including WMI, performance counter, message filters, and other settings.
extensions	Defines configuration extensions for the bindings, behaviors, bindingElement (for extending CustomBinding configuration), and standardEndpoints.
machineSettings	Allows the user to log personally identifiable information in traces and message logs via the XML attribute, enableLoggingKnownPii. This section can only be edited in machine.config, located in the framework directory.
protocolMapping	Used to map a protocol to a binding for easier configuration.
serviceHostingEnvironment	Used to configure services hosted in ASP.NET.
services	Defines which contracts a service instance will listen for messages on and any base addresses the service will use. A service may also add metadata and discovery endpoints via this mechanism.
standardEndpoints	Contains configuration for endpoints specified by the kind attribute on any service endpoint configuration.
routing	Provides message routes that a routing service uses to move messages closer to its final destination.
tracking	Defines tracking settings for a workflow service.

WCF CONTRACTS

WCF developers create two types of contracts: service contracts and data contracts. Service contracts inform the WCF runtime how to read, write, and dispatch received messages. Data contracts inform the serialization infrastructure how to translate CLR objects to and from an XML Infoset representation.

Service Contracts

The WCF runtime creates the infrastructure to host services and dispatch messages. Developers declare the pieces of infrastructure they need by marking up classes with attributes. The WCF runtime then uses this information to allow for communication with callers and the hosted classes. This information is also shared via Web Services Description Language (WSDL) that WCF services can generate. To declare a class or interface represents the contract for a WCF service, mark the class with [ServiceContract].

```
[System.ServiceModel.ServiceContract]
public interface IMyService {
}
```

Through [ServiceContract], you will normally set the following properties about the collection of operations on the service:

Namespace	Sets the default namespace for the XML Schema Documents (XSD) for the request and response messages. Default is http://tempuri.org.
SessionMode	One of three values: Allowed, Required, NotAllowed. By default, SessionMode is Allowed. If your contract requires session semantics, set this value to System.ServiceModel.SessionMode.Required. If your contract will fail with sessions, set it to System.ServiceModel.SessionMode. NotAllowed.
CallbackContract	CallbackContract: If the contract implements a duplex MEP, set this to the interface representing the other side of the duplex conversation.

To make methods visible to external callers, mark the methods with [OperationContract].

```
[System.ServiceModel.ServiceContract(
  Namespace = "http://www.dzone.com/WCF")]
public interface IMyService
{
  [System.ServiceModel.OperationContract]
  string SayHi(string name);
}
```

The commonly set properties on [OperationContract] are:

IsOneWay	Can only be set on methods that return void. Use this to indicate that the method does not send a response. Default is false.
IsInitiating	Use this to state that a given method can be called to instantiate a new service. Default value is true.
IsTerminating	Use this to state that when a given method is called, the current instance can be disposed. Default value is false. If this value is set to true, you must also set ServiceContractAttribute.SessionMode to SessionMode.Required.

Your code may also return exceptions to callers. In SOAP messaging, errors are returned as Fault messages. A given operation may return zero or more faults. One declares the types of faults being returned through [FaultContract]. [FaultContract] has a constructor that accepts a type describing what the fault details will look like. This description is used by the callers to read any faults your service might return.

```
[System.ServiceModel.ServiceContract(
  Namespace = "http://www.dzone.com/WCF")]
public interface IMyService
{
  [System.ServiceModel.OperationContract()]
  [FaultContract(typeof(FaultDetails))]
  string SayHi(string name);
}
```

Data Contracts

WCF reads and writes objects to and from different formats using serialization (write) and deserialization (read). WCF supports serialization through several mechanisms: System.Xml.Serialization, System.ISerializable, [System.Serializable], and through System.Runtime.Serialization. The first three mechanisms exist to support legacy code. When developing your own declarations, you will normally use System.Runtime.Serialization to describe your data contracts. In WCF 4.0, an unattributed type will automatically read and write any public fields or properties using the name of the property in the generated format. To control what gets written, mark the type with [DataContract] and any members to be written with [DataMember].

```
[System.Runtime.Serialization.DataContract]
public class Name
{
  [System.Runtime.Serialization.DataMember]
  public string FirstName { get; set; }

  [System.Runtime.Serialization.DataMember]
  public string LastName { get; set; }
}
```

When you explicitly mark a member with [DataMember], that field will be read and written regardless if the member is private, protected, or public.

The commonly set properties on [DataContract] are:

Namespace	Defines the URL used as the targetNamespace in the XSD describing the type and used when serializing the data contract as an XML Infoset.
Name	Defines the name of the item when read and written to an XML Infoset.

The commonly set properties on [DataMember] are:

Name	Defines the name of the item when read and written to an XML Infoset.
IsRequired	Set to true if the field must be present when the type is deserialized. Default value is false.
EmitDefaultValue	Indicates whether or not to write the member when set to the default value. The default value of this member is true.
Order	An integer used to alter the ordering of values when read or written. By default, values are read and written in alphabetic order. The default value of Order is 0. Members with the same Order value are serialized alphabetically. Order is sorted ascending.

BINDINGS

WCF supports messaging through a messaging pipeline. The pipeline itself must have stages that represent the transport and the serialization mechanism. In WCF, the pipeline is created by a type derived from System.ServiceModel.Channels.Binding and the stages are created by types derived from System.ServiceModel.Channel.BindingElement. Besides the transport and serialization mechanism, the BindingElement also adds stages for security, reliability, and transactions. When creating a Binding, the developer of the Binding decides how to surface the underlying BindingElement properties. Some settings will be easily set, others made private, and still others surfaced as different concepts.

Every binding has settings for timeouts: OpenTimeout, ReceiveTimeout, CloseTimeout. These set the amount of time the user of the Binding will wait for the pipeline to Open, Receive a message, or Close. A Binding also identifies its URL scheme, which is the scheme used by its transport. WCF ships with Bindings supporting the following schemes: http, https, net.msmq, net.pipe, net.p2p, and net.tcp. WCF also supports soap.udp, but only in an internal class used to support WS-Discovery.

Each of the Bindings has a common set of properties not required by the base class, Binding:

EnvelopeVersion	Sets the version of any SOAP Envelope. Normally, this is set to None, Soap11, Soap12. Use None for REST or other non-SOAP messaging; Soap11 to send a SOAP 1.1 envelope and related headers; Soap12 to send a SOAP 1.2 envelope and related headers.
MaxBufferPoolSize	Most messages are received into memory. The buffer pool is used to allocate memory for receiving messages. By default, each buffer in the pool is 65536 bytes. Adjust this value to change the size of the individual buffers.
ReaderQuotas	This sets quotas on reading inbound messages. The type, System.Xml.XmlDictionaryReaderQuotas is found in System.Runtime.Serialization. The quotas set limits which limit the damage a denial of service attack or poorly formed XML document can do. By default, arrays can be no longer that 16384 elements (MaxArrayLength), strings must be less than 8192 characters (MaxStringContentLength), and XML nodes must be no more than 32 levels deep (MaxDepth).
Scheme	Identifies the scheme used by the underlying transport protocol.

Depending on which protocol you want to use, you have different bindings available. Most of the bindings in WCF support HTTP. The rest of the supported protocols are represented by one binding. The following table shows which bindings support duplex communications, message level security, reliable messaging, flowing transactions, and workflow context. Transport level security is available through all bindings in WCF.

Binding	Duplex	Message level security	Reliable	Transactions
BasicHttpBinding		✓*		
BasicHttpContextBinding		✓*		
MsmqIntegrationBinding				
NetMsmqBinding		✓		
NetNamedPipeBinding	✓	✓		✓
NetPeerTcpBinding	✓	✓		
NetTcpBinding	✓	✓	✓	✓
NetTcpContextBinding	✓	✓	✓	✓
WS2007FederationHttpBinding		✓	✓	✓
WS2007HttpBinding		✓	✓	✓
WSDualHttpBinding	✓	✓	✓	✓
WSFederationHttpBinding		✓	✓	✓
WSHttpBinding		✓	✓	✓
WSHttpContextBinding		✓	✓	✓

* The BasicHttpBinding and BasicHttpContextBinding will sign the timestamp on the message, but not sign and encrypt the body. For full message level signing and encryption, use WSHttpBinding and WSHttpContextBinding.

Bindings appear in configuration in the system.serviceModel/bindings section under configuration that matches a camelCased version of the binding name. For example, the collection of WSHttpBinding can be found in system.serviceModel/bindings/wsHttpBinding. Each configured binding has a name. A binding without a name becomes the default set of settings for that particular binding. For example, to create a reliable WSHttpBinding, use the following configuration:

```
<system.serviceModel>
    <bindings>
      <wsHttpBinding>
        <binding name="reliable">
          <reliableSession enabled="true"/>
        </binding>
      </wsHttpBinding>
    </bindings>
  </system.serviceModel>
```

BEHAVIORS

Behaviors influence the hosting environment for a service. They are used to handle instancing, expose metadata, enhance discoverability, and more. There are four types of behaviors: service, contract, endpoint, and operation. All behaviors may be applied in code. Service and endpoint behaviors may also be applied by configuration. (Note: There is no notion of contract configuration anywhere in WCF.) Service and contract behaviors are normally applied via attributes in code.

A service behavior applies to a service instance and may alter aspects of the service, all endpoints, and all contracts. A contract behavior applies to a contract and all implementations of the contract. Endpoint behaviors apply to a specific endpoint instances. Finally, operation behaviors apply to specific operations.

Attribute-based Behaviors

There are two service behaviors: [AspNetCompatibilityRequirements] and [ServiceBehavior]. [AspNetCompatibilityRequirements] has a single property, RequirementsMode, that allows a service to declare that ASP features such as identity impersonation are Required, Allowed, or NotAllowed (default). When set to Required, configuration must enable compatibility too:

```
  <system.serviceModel>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
  </system.serviceModel>
```

[ServiceBehavior] contains several properties. The most commonly used are:

ConcurrencyMode	Controls the internal threading model to enable support for reentrant or multithreaded services. By default, concurrency is single threaded per instance.
InstanceContextMode	Controls how instances are created for the service. By default, each session gets a new service instance. Set this property to PerSCall to allow for a new instance for every method call. Set this property to Single if you need singleton behavior for the service.
IncludeExceptionDetailsInFaults	Returns exception details to clients when debugging services. This property requires a request-response or duplex capable binding.

[DeliveryRequirements], a contract behavior, allows a contract to specify information about reliable messaging requirements imposed on the service delivery.

[CallbackBehavior], an endpoint behavior, provides configuration settings for the callback contract on a duplex service. The settings are the same as the [ServicBehavior] except for settings for instancing and transactions.
In configuration, you can declare information about security, diagnostics, discovery, and throttling.

WCF contains many configuration elements. Here, we focus on the commonly configured behaviors for services. The following behaviors allow you to configure your service behaviors as part of configuration within system.serviceModel/behaviors/serviceBehaviors/:

serviceCredentials	Allows the service to pick how it authenticates itself using X.509, Windows, username/password, and other valid tokens.
serviceDebug	Allows you to set up the HttpHelp page and to include exception details in faults.
serviceDiscovery	Enables the WS-Discovery endpoint.
serviceMetadata	Enables the WS-MetadataExchange endpoint.

On an endpoint, you can configure other behaviors. The following commonly used behaviors allow you to configure your endpoint behaviors as part of configuration within system.serviceModel/behaviors/endpointBehaviors/:

callbackDebug	Allows you to state if a callback contract should include exception details in faults.
clientCredentials	Allows the client to pick how it authenticates itself using X.509, Windows, username/password, and other valid tokens.
enableWebScript	Enables a Web Script endpoint on the service. This allows the service to return JavaScript when the URL end in /js or /jsdebug. This behavior is automatically included if using the System.ServiceModel.Activation.WebScriptServiceHostFactory on a .SVC.
webHttp	Allows the endpoint to dispatch messages based on URL. This behavior is automatically included if using the System.ServiceModel.Activation.WebScriptServiceHostFactory or System.ServiceModel.Activation.WebServiceHostFactory as the factory name on a .SVC.

HOSTING SERVICES

WCF services can listen for messages anywhere: Windows Services, desktop applications, and Internet Information Services (IIS). All environments use the same configuration elements. In all cases, an instance of a ServiceHost will be used to reflect over a service implementation to determine service requirements. The ServiceHost then marries the code with any configuration to produce an entity that can listen for, dispatch, and respond to messages that arrive over the various transports.

To demonstrate hosting, we use the following contract:

```
namespace DZone.Contracts
{
  [System.ServiceModel.ServiceContract(
    Namespace = "http://www.dzone.com/WCF")]
  public interface IMyService
  {
    [System.ServiceModel.OperationContract]
    string SayHi(string name);
  }
}
Implemented by the following class:
namespace DZone.Services
{
  public class MyService : DZone.Contracts.IMyService
  {
    public string SayHi(string name)
    {
      return string.Format("Console: Hello, {0}", name);
    }
  }
}
```

and use the following configuration:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MyServiceBehavior">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name="DZone.Services.MyService"
        behaviorConfiguration="MyServiceBehavior">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost/Demo"/>
          </baseAddresses>
        </host>
        <endpoint contract="DZone.Contracts.IMyService"
          binding="basicHttpBinding" address="/MyService"/>
        <endpoint kind="mexEndpoint" address="/mex"
          binding="mexHttpBinding" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

The configuration declares that there is a serviceBehavior named MyServiceBehavior which supports metadata exchange. This behavior is attached to a service instance whose name is DZone.Services.MyService and matches the name of the service implementation. The host has a base address of http://localhost/Demo. This base address is used for any bindings that support the http scheme. The service exposes two endpoints. One endpoint exposes an implementation of the DZone.Contracts.IMyService contract listening off the http base address at /MyService. The other endpoint hosts metadata exchange using a predefined binding named mexHttpBinding listening off the http base address at /Mex.

Hosting in a Console/GUI application

Using this configuration, we can host the service in a Console application with the following code:

```
var host = new ServiceHost(typeof(MyService))

  host.Open();
  Console.WriteLine("Press [Enter] to exit");
  Console.ReadLine();

try
{
  ((IDisposable)host).Dispose();
}
catch (CommunicationObjectFaultedException ex)
{
  // TODO: add code to log
}
```

Once the using block exits, host.Dispose() is called. You can do something similar in a WinForm/WPF application by opening the service on window Load and explicitly calling ServiceHost.Dispose() when the window is closed/unloaded.

Hosting in a Windows Service

A Windows Service requires you to be able to respond to Start and Resume events very quickly. In order to do this, you do not want to block in the start if at all possible. Creating a single WCF ServiceHost does not normally take much time. However, we should always be prepared for things to take a while and have the services behave nicely. To host the same WCF service in a Windows Service, write the following code:

```
  private ServiceHost _host;

  protected override void OnStart(string[] args)
  {
    ThreadPool.QueueUserWorkItem(StartListening, this);
  }

  static void StartListening(object state)
  {
    var service = state as DemoService;
    if (service != null)
    {
      service._host = new ServiceHost(typeof(MyService));
      service._host.Faulted +=
        (s, e) =>
        {
          service.StopListening();
          service._host = null;
          StartListening(service);
        };
    }
  }

  protected override void OnStop()
  {
    StopListening();
  }

  void StopListening()
  {
    try
    {
      ((IDisposable)_host).Dispose();
    }
    catch (CommunicationObjectFaultedException ex)
    {
      // TODO: add code to log
    }
  }
```

The preceding code executes the initialization logic on a separate thread via ThreadPool.QueueUserWorkItem. The ServiceHost will run for a long time. If the ServiceHost gets into a situation where it can no longer listen for messages by entering the Faulted state, the code should start up a new instance of the ServiceHost. When the service is done listening, a CommunicationObjectFaultedException may be thrown on Dispose. Because this exception is expected, the code will log the exception. When the service stops, we call StopListening. If you want to support Pause and Resume, add the following:

```
  protected override void OnPause()
  {
    StopListening();
    base.OnPause();
  }

  protected override void OnContinue()
  {
    ThreadPool.QueueUserWorkItem(StartListening, this);
    base.OnContinue();
  }
```

Hosting in IIS

To host a service in IIS, you create a .svc file which is hosted at a path of your choosing in your web application. That will give you a file that looks like this:

```
<%@ ServiceHost Service="DZone.Services.MyService" %>
```

The config remains largely the same. IIS will automatically set the http base address for the host to be the address of the .svc file.

In IIS 7 and later, you can also host services with NetTcpBinding and NetNamedPipeBinding. To do this, run the following commands:

TCP Activation:

1. Run the following command (on one line):

```
%windir%\system32\inetsrv\appcmd.exe set site "Default Web Site" -+bindings.
[protocol='net.tcp',bindingInformation='808:*']
```

2. Run the following command to enable the http and net. pipe protocol on your site (on one line):

```
%windir%\system32\inetsrv\appcmd.exe set app
"Default Web Site/[your v-dir]" /enabledProtocols:http,net.tcp
```

Named Pipe Activation

1. Run the following command (on one line):

```
%windir%\system32\inetsrv\appcmd.exe set site "Default Web Site"
-+bindings.[protocol='net.pipe',bindingInformation='*']
```

2. Run the following command to enable the http and net. pipe protocol on your site (on one line):

```
%windir%\system32\inetsrv\appcmd.exe set app
"Default Web Site/[your v-dir]" /enabledProtocols:http,net.pipe
```

With Vista SP1 and Server 2008, you can also enable these protocols in the IIS Manager.

For the demo application, remove the <host> base addresses and add in these two endpoints:

```
<endpoint contract="DZone.Contracts.IMyService" address="/MyService"
  binding="netTcpBinding" />
<endpoint contract="DZone.Contracts.IMyService" address="/MyService"
  binding="netNamedPipeBinding" />
```

OperationContext

Every incoming message is associated with an OperationContext. Think of OperationContext as WCF's version of HttpContext. OperationContext.Current yields the current operation context with pointers to the following commonly used properties:

IncomingMessageHeaders	Headers on the incoming message.
OutgoingMessageHeaders	Can use this to add more headers to the response.
IncomingMessageProperties	The message properties serve as a mechanism to send information in between layers within the message processing pipeline about a specific message.
ServiceSecurityContext	Gain access to the identity of the currently logged in user. Also available through ServiceSecurityContext.Current. Contains a property, AuthorizationContext, where you can investigate the ClaimSets for the current user.

The OperationContext also has two often used methods:

SetTransactionComplete()	Allows the service to complete a transaction in code instead of automatically on exit.
GetCallbackChannel<T>()	For duplex services, allows the service to send messages back to the service.

CONSUMING SERVICES

When consuming services, you will typically create a proxy via the Visual Studio 'Add Service Reference' tool. Enter the WSDL or MetadataExchange endpoint for the service, typically just the address of the .svc file for IIS hosted services, and then pick out the service. Visual Studio will do the rest of the work, including adding configuration. The developer may need to edit the application configuration file (named [app name].exe.config if the client is an executable or web.config if the client is a web application) to enter in security credentials, but otherwise the work is done.

When consuming the previous service, write the following code, catching the TimeoutException, CommunicationException, and FaultException since any of these may be returned. For both the CommunicationException and TimeoutException, the code may want to have some retry logic built in to get a new client instance and try again up to n times, logging the exception each time. In all cases, the code should Abort the connection to release all resources on the client machine.

When the configuration is generated for the client, the endpoints will have names like binding_Contract. For IMyService, the client endpoint is named BasicHttpBinding_IMyService in configuration. The endpoint can then be created by instantiating the generated proxy using the configured name. A call to a IMyService implementation looks like this:

```
var client = new DZoneService.MyServiceClient(
"BasicHttpBinding_IMyService");
try
{
  Console.WriteLine(client.SayHi("DZone"));
}
catch (TimeoutException timeoutException)
{
  client.Abort();
}
catch (FaultException<KnownFaultType> faultException)
{
  client.Abort();
}
catch (FaultException faultException)
{
  client.Abort();
}
catch (CommunicationException communicationException)
{
  client.Abort();
}
```

DIAGNOSTICS

WCF also comes with a rich set of diagnostics information. Unlike other aspects of WCF, diagnostics can only be set in configuration. The diagnostics allows for tracing, message logging, WMI inspection, and performance counters.

Tracing/WMI

All production applications should be deployed with System. ServiceModel tracing configured but turned off. WMI should be enabled so that an administrator can enable tracing through WMI. The configuration looks like this:

```
  <system.diagnostics>
    <sources>
      <source name="System.ServiceModel"
        switchValue="Off"
        propagateActivity="true">
        <listeners>
          <add name="ServiceModelTraceListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add initializeData=".\logs\web_tracelog.svclog"
        type="System.Diagnostics.XmlWriterTraceListener, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        name="ServiceModelTraceListener"
        traceOutputOptions="Timestamp" />
    </sharedListeners>
  </system.diagnostics>
  <system.serviceModel>
    <diagnostics wmiProviderEnabled="true" />
  </system.serviceModel>
```

If, in production, something goes wrong, the administrator can run the following Powershell commands to update the switchValue, capture data, change the switchValue back to Off, and send the traces off to development:

```
$appDomain = Get-WmiObject -Namespace root\ServiceModel
  AppDomainInfo

$appDomain.TraceLevel = "Verbose, ActivityTracing"

$appDomain.put()
```

Note: the line to get the WMI object will change depending on how many AppDomains hosting WCF endpoints are running on the computer. Each AppDomainInfo has a corresponding ProcessID and AppDomainId to help you pick the right instance. Once that is done, you can set the trace level as above and call put() to save the data. Above turns on ActivityTracing which allows WCF to assign IDs to related traces and show how one group of activity flows from another. Use SvcTraceViewer.exe (part of the Windows SDK and ships as a part of Visual Studio) to view and interpret the traces. When the administrator is done collecting data, turn tracing back off:

```
$appDomain.TraceLevel = "Off"

$appDomain.put()
```

Note that changes to the trace level do not get saved to the configuration file as this would cause the AppDomain to restart on IIS.

With WMI on, you can also inspect all the running endpoints including binding data, behaviors, and many other items.

Message Logging

Message logging is enabled in two places. First, create the message logging trace source. If you are logging and tracing, you can get all the traces into one file with the following configuration:

```
<system.diagnostics>
  <sources>
    <source name="System.ServiceModel"
      switchValue="Off"
      propagateActivity="true">
      <listeners>
        <add name="ServiceModelTraceListener" />
      </listeners>
    </source>
    <source name="System.ServiceModel.MessageLogging"
      switchValue="Off"
      propagateActivity="true">
      <listeners>
        <add name="ServiceModelTraceListener" />
      </listeners>
    </source>
  </sources>
  <sharedListeners>
    <add initializeData=".\logs\web_tracelog.svclog"
      type="System.Diagnostics.XmlWriterTraceListener, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
      name="ServiceModelTraceListener"
      traceOutputOptions="Timestamp" />
  </sharedListeners>
</system.diagnostics>
```

In the ServiceModel configuration, you can set what you want to log:

```
<diagnostics>
  <messageLogging
       logMessagesAtTransportLevel="true"
       logMessagesAtServiceLevel="true" />
</diagnostics>
```

Typically, you will log messages at the transport level to see an XML 1.0 text representation of what the ServiceModel saw including message headers added by the message pipeline. You log at the service level to see what the message looked like after the message is decrypted.

Performance Counters

To enable performance counters, viewed in tools like PerfMon, you add the following configuration to system.serviceModel:

```
<diagnostics performanceCounters="ServiceOnly" />
```

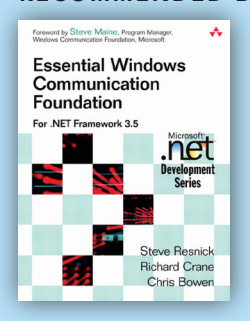
Performance Counters can have one of four values:

Default	WCF_Admin is created and SQM data is collected. The values are never updated.
Off	Performance counters are disabled.
ServiceOnly	Only service level counters are enabled.
All	Counters for the service, endpoints, and operations are created.

ABOUT THE AUTHOR

Scott Seely is the author of several books on Web Services and an instructor for Pluralsight. He is a Microsoft Regional Director. Right now, he is working on Essential Windows Communication Foundation, 2nd Edition for Addison Wesley. He helped found the Chicago Code Camp with several other local developers. He helps run the Lake County .NET Users' Group with Tim Stall. Throughout the year, Scott can be found speaking at the user groups throughout northern Illinois and Wisconsin. From 2000 through 2006, Scott worked for Microsoft. He spent his first two years as a developer/writer for MSDN and then moved over to the Indigo/WCF team as a developer. Scott is also founder of Friseton, LLC. Friseton specializes in creating highly scalable parallelized systems, solving application performance problems, and building distributed applications that utilize REST and WS-*.

RECOMMENDED BOOK

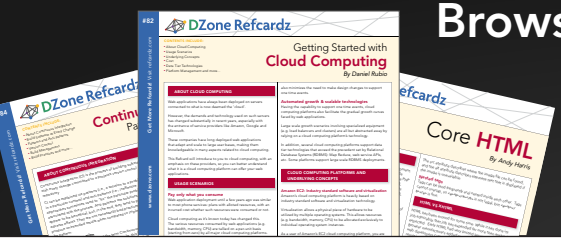


Essential Windows Communication Foundation For .NET Framework 3.5

Written by three experts at the Microsoft Technology Center in Boston, this guide answers developers' questions about WCF. Throughout the book's 13 chapters, authors Steve Resnick, Richard Crane, and Chris Bowen offer best practices, key advice, tips, and problem-solving tricks. They solve developers' problems with WCF through in-depth explanations and an extensive amount of code samples.

BUY NOW

books.dzone.com/books/essential-wcf



Browse our collection of 100 Free Cheat Sheets

Free PDF

Upcoming Refcardz
Apache Ant
Hadoop
Spring Security
Subversion

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-75-2
ISBN-10: 1-934238-75-9
50795
9 781934 238752
$7.95

Version 1.0

Copyright © 2010 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.