Pequeña introducción a los conceptos de Windows Communication Foundation (WCF) es el modelo de programación unificado de Microsoft para generar aplicaciones orientadas a servicios

Introducción a Windows Communication Foundation WCF

Oscar Alvarez Guerras-
http://geeks/blogs/oalvarez

Contenido

Arquitectura Orientada a Servicios (SOA)

Actualmente en el mundo IT se habla bastante de la arquitectura orientada aservicios, conocida por sus siglas SOA en inglés, y como casi siempre suele pasar, este concepto que puede parecer nuevo, no lo es.
SOA puede ser definido como un estilo de arquitectura para sistemas de información que habilita la creación de aplicaciones que son construida combinando servicios desacoplados e interoperables. Un servicio es una unidad de trabajo ofrecida por un proveedor de servicios que alcanza los resultados deseados por el cliente del servicio. Tanto el proveedor como elcliente de servicios son roles asumidos por agentes de software en nombre de sus propietarios.
Esta definición puede resultar un poco abstracta pero la realidad no dice que SOA está por todas partes. Veamos un ejemplo donde descubriremos como SOA podemos encontrarlo perfectamente en un salón. Coger un DVD por un instante. Si queremos reproducir la película que contiene, pondremos este DVD en un reproductor de DVD y este lo reproducirá por nosotros. El reproductor de DVD está ofreciendo un servicio de reproducción de películas en DVD. Es perfecto porque podemos reemplazar el DVD introducido por otroy verlo también sin ningún tipo de problema. Podemos reproducir el mismo DVD en un reproductor de DVD portátil o en nuestro caro sistema de cine encasa. Ambos ofrecen el mismo servicio de reproducción de DVD pero lógicamente la calidad de este último será diferente a la del portátil.
La idea de SOA se distancia significativamente de la programación orientada a objetos, la cual sugiere que tanto los datos como su proceso deben ir de la mano. Por tanto, en el estilo de programación orientado a objetos, todo DVD debería venir con su propio reproductor y no se conciben uno separado del otro. Aunque pueda parecer extraño es como hemos realizado muchos sistemas de software hasta ahora. La razón por la queremos que otros hagan el trabajo por nosotros es por aqueellos son expertos en su campo, especialistas. Utilizar un servicio es normalmente más barato y más eficiente que hacer nosotros mismos elt rabajo. La mayoría de nosotros somos bastante inteligentes como para pode rrealizar el trabajo nosotros mismos pero no lo suficientemente inteligentes para ser expertos en todo, nadie es experto en todo. Lo bueno, es que estaregla también aplica a la construcción de sistemas de software. ¿Cómo hace SOA para conseguir desacoplar agentes de software? Pues empleando dos restricciones de arquitectura que vamos a ver a continuación:

1. Un pequeño conjunto de interfaces simples y omnipresentes en todos los agentes de software en los que participan. Solo la semántica genérica es codificada en los interfaces. Los interfaces deben estar disponibles para todo el mundo.
2. Mensajes descriptivos limitados por esquemas extensibles disponibles a través de interfaces. El esquema limita el vocabulario y la estructura de los mensajes. Un esquema extensible permite añadir nuevas versiones de servicios sin romper los existentes.

Si las interfaces no funcionan, el sistema no funciona. Una interfaz necesita prescribir comportamientos de sistemas, siendo esto muy complejo de implementar correctamente a través de las diferentes plataformas y lenguajes. Cabe destacar que los interfaces remotos suelen ser la parte más lenta de las aplicaciones distribuidas.

Para poder decir que una arquitectura está orientada a servicios, esta debe cumplir unas reglas. A continuación se enumeran estas reglas

1. Los mensajes han de ser descriptivos en vez de instructivos, porquees el proveedor de servicios el responsable de solucionar el problema. Si vas a un restaurante, al camarero le dices lo que quieres comer pero nunca te pones a explicarle como debe cocinar tu plano paso a paso.
2. El proveedor del servicio no podrá entender tus peticiones si tus mensajes no van escritos en un determinado formato, estructura y vocabulario que sea el que entienden todas las partes implicadas. Por tanto, limitar el vocabulario y la estructura de un mensaje es necesario para que las comunicaciones sean eficientes.
3. La extensibilidad es lo más importante, y no es muy difícil de entender porqué. Los clientes demandan cambios continuamente en los sistemas de software, y si un servicio no es extensible estos se quedarán bloqueados en una versión para siempre
4. SOA debe proveer de un mecanismo por el cual un cliente de un servicio pueda descubrir su funcionalidad

Windows Communication Foundation

Windows Communication Foundation (anteriormente conocida con el nombre en clave "Indigo") es un conjunto de tecnologías .NET para la creación y puesta en marcha de sistemas interconectados. Propone la unificación del modelo de programación para diferentes tecnologías, proporcionando la posibilidad de construir aplicaciones que son independientes del mecanismo usado para conectar los servicios y las aplicaciones. Es realmente muy difícil, si no imposible, separar totalmente la estructura programática de una aplicación o de un servicio de su infraestructura de comunicaciones, sin embargo WCF permite acercase mucho a la consecución de esto.
Adicionalmente, usar WCF permite mantener compatibilidad hacia atrás con muchas de las tecnologías anteriores. Por ejemplo, una aplicación cliente WCF puede fácilmente comunicarse con un Servicio Web que haya sido creado usando Web Services Enhancements (WSE).

Características de WCF

En esta sección veremos rápidamente cuales son aquellas características de WCF que lo hacen tan potente para la construcción de aplicaciones distribuidas, y en posteriores capítulos serán abordadas de manera más detallada.

Transacción

Una transacción es una unidad de trabajo. Una transacción asegura que todo dentro de dicha transacción funcionará en su totalidad o fallará todo ello. Por ejemplo, si la transacción contiene cuatro unidades de trabajo a realizar y durante la ejecución de las mismas, alguna de estas unidades falla, fallarán las

cuatro. Esta transacción se podrá considerar como que ha funcionado únicamente cuando las cuatro unidades de trabajo hayan finalizado su trabajo correctamente. WCF incorpora este proceso transaccional en sus comunicaciones. Como desarrollador, se podrán agrupar comunicaciones dentro de una transacción.
A nivel empresarial, esta característica permitirá ejecutar procesos transaccionales a través de diferentes plataformas

Alojamiento

WCF permite hospedar servicios en una gran cantidad de entornos diferentes, como Servicios Windows NT, Windows Forms, aplicaciones consola, IIS y Windows Activation Services (WAS).
Hospedar un servicio en IIS permite beneficiarse por parte del servicio de muchas de las características nativas de IIS. Por ejemplo, el IIS puede controlar el arrancar y parar un servicio automáticamente.

Seguridad

No es posible concebir Windows Communication Foundation sin seguridad. Realmente no es precisamente de seguridad de lo que pueda carecer WCF. Todo, desde los mensajes a los clientes, pasando por los servidores están autenticados, y WCF tiene la característica de asegurarse que los mensajes no son manipulados durante su transferencia. WCF incluye integridad y confidencialidad en los mensajes.
Además, WCF permite integrar cualquier aplicación dentro de una infraestructura de seguridad ya existente, incluyendo lo que se extiende más allá de los entornos únicamente Windows, usando mensajes de tipo SOAP seguros.

Colas de Mensajes

WCF provee colas de mensajes, permitiendo que los mensajes sean guardados de forma segura proporcionando un estado consistente de comunicación. Las colas colectan y guardan mensajes enviados desde una aplicación emisora y los redirige luego a la aplicación receptora. Esto proporciona un mecanismo de entrega de mensajes seguro y fiable

Contratos, Mensajes y Servicios

Una de las ideas fundamentales en Windows Communication Fundation son los contratos (son parte del ABC de WCF - Address, Binding, Contract).

El ABC de WCF

El ABC en WCF es una regla nemotécnica que nos permite recordar de manera fácil tres conceptos importantísimos en WCF.

Address

Nos indica dónde está el servicio, esto se traduce en una URI del tipo de los servicios Web *http:\\localhost\MiServicio.asmx* pero ahora estas URIs pueden direccionar los siguientes protocolos:

- HTTP
- TCP
- Named Pipe
- Peer 2 Peer
- MSMQ

Binding

Un binding nos especifica cómo se accede al servicio, esto es:

- Protocolo que se utiliza
- Codificación (texto, binario...)
- Protocolos WS-* que permiten mensajes transaccionales

Contract

El contrato nos indica qué ofrece el servicio, es decir, que métodos y operaciones expone

Definición de Contratos

Los contratos en Windows Communication Foundation proveen de la interoperabilidad necesaria para comunicarse con las aplicaciones cliente. Este contrato es el que establecen los clientes y servicios para convenir los tipos de operaciones y estructuras que se van a utilizar mientras dure la comunicación entre ambas partes. Existen tres contratos básicos que son usados para definir un servicio WCF:

- **Contrato de Servicio (Service Contracts)**Define los métodos de un servicio, es decir, que operaciones podrárealizar el cliente.
- **Contrato de Datos (Data Contracts)** Define los tipos de datos disponibles de los métodos del servicio.
- **Contrato de Mensaje (Message Contracts)** Provee la habilidad de controlar las cabeceras de los mensajesdurante la generación de estos

Los Contratos y la relación con el CLR

Los tipos de los contratos se traducen fácilmente entre los tipos de .NET existentes internamente y la representación externa XML compartida. En la siguiente lista se muestra la relación de equivalencia de la que estamos hablando:

- **Service Contracts** CLR y WSDL (Web Service Description Language)
- **Data Contracts** CLR y XSD (XML Schema Definition)
- **Message Contracts** CLR y SOAP (Simple Object Access Protocol

Contratos de Servicio

Como ya se ha explicado, un Service Contract define las operaciones, o métodos, que están disponibles en un servicio y que son expuestas al mundo exterior. Además, definen patrones de intercambio básicos de mensajes, como la manera de comportarse de los mensajes en request/reply, one-way o dúplex.
Un Service Contract expone la información necesaria al cliente, el cual habilita a este a entender lo que ofrece el servicio. Esta información incluye lo siguiente:

- Tipos de datos en los mensajes
- La localización de las operaciones
- Información del protocolo y el formato de serialización que asegureuna comunicación satisfactoria.
- Agrupación de Operaciones
- Patrón de intercambio de mensajes (MEPs)

Un Service Contract es definido simplemente aplicando el atributo [ServiceContract] a una clase o interfaz. En el siguiente ejemplo se muestra como definir una interfaz como Service Contract

```
[ServiceContract]
public interface IProductService
{
  //…
 }
```

Las operaciones del servicio se especifican aplicando el atributo [OperationContract] a los métodos de una interfaz

```
[OperationContract]
List<string> ListProducts();

[OperationContract]
Product GetProduct(string productNumber);
```

Juntando ambos ejemplos conseguimos un completo Service Contract, como
se muestra a continuación

```
[ServiceContract]
public interface IProductService
{
  [OperationContract]
  List<string> ListProducts();

  [OperationContract]
  Product GetProduct(string productNumber);
```

Una vez definida la interfaz del servicio, entonces es cuando puede ser implementada. En el siguiente ejemplo se implementa la interfaz IProductService definida en el código de ejemplo anterior

```
public class ProductService : IProductService
{
public List<string> ListProducts()
{
//...
}

public Product GetProduct(string productNumber)
{
//...
}
}
```

En los anteriores ejemplos hemos usado atributos como [ServiceContract] sin parámetros. Sin embargo, tanto el atributo [ServiceContract] como el [OperationAttribute] tienen varios parámetros que pueden ser usados para especificar detalles especiales

El atributo [ServiceContract]

El atributo *[ServiceContract]* identifica una clase o interfaz como un contrato de servicio. Este atributo define explícitamente la interfaz como una interfaz CLR y la habilita para realizar operaciones de contracto de WCF, donde *[ServiceContract]* es mapeado con una declaración equivalente del tipo puerto WSDL.

La siguiente lista contiene los parámetros disponibles para ser usados con el *[ServiceContract]*. Se puede aplicar más de un parámetro y pueden ser aplicados en cualquier orden

- CallbackContract
- ConfigurationName
- Name
- Namespace
- ProtectionLevel
- SessionMode

CallbackContract

El parámetro CallbackContract establece el tipo del contrato de tipo callback cuando el contrato se comunica en modo dúplex. Este parámetro contiene el tipo que especifica el contrato callback. El tipo debe contener un *CallbackContract = typeof(ClientContract)* que representa el contrato requerido en el lado cliente en una operación dúplex de intercambio de mensajes, es decir, en un contrato cliente callback. Especificando este parámetro informa al cliente que es necesario que "escuche" llamadas entrantes procedentes del servicio

```
[ServiceContract(CallbackContract = typeof(IClientContract))]
public interface IProductService
{
  //…
}
```

ConfigurationName

El parámetro ConfigurationName establece el nombre usado para localizar el elemento servicio en el fichero de configuración. Este valor es de tipo *string*. Si no es especificado, el nombre por defecto es el nombre de la clase que implementa el servicio.

```
[ServiceContract(ConfigurationName = "service")]
public interface IProductService
{
  //…
}
```

El fichero de configuración de la aplicación para el anterior ejemplo sería parecido a lo siguiente

```
<configuration>
  <system.servicemodel>
       <services>
         <service name = "ProductService">
       </service>
    </services>
  </system.servicemodel>
  </configuration>
```

Name

El parámetro Name establece el nombre para el elemento <*portType*> del WSDL (Web Service Description Language). El valor por defecto para este parámetro es el nombre de la clase o interfaz del servicio. El elemento <portType> en el WSDL contiene toda la información necesaria para un servicio, como las operaciones que pueden realizar y los mensajes que participan en estas.

```
[ServiceContract(Name = "IProductService")]
public interface IProductService
{
//…
}
```

Namespace

El parámetro Namespace establece el namespace para el elemento *<portType>* del WSDL (Web Service Description Language). El valor por defecto para este parámetro es *"http://www.tempuri.org"*.

El parámetro Namespace para el atributo *[ServiceContract]* establece el namespace de este elemento

```
[ServiceContract(Namespace = "http://AdventureWorks.Contracts"]
public interface IProductService
{
//…
}
```

ProtectionLevel

El parámetro ProtectionLevel especifica el nivel de protección requerido para utilizar el servicio. Esto incluye encriptación, firma digital, o ambos por cada punto que expone el contrato.El valor de este parámetro viene del enumerador *System.Net.SecurityLevel.ProtectionLevel*. Los siguientes valores están disponibles

- **EncryptAndSign** Encriptar y firmar datos aseguran la confidencialidad e integridad de los datos transmitidos.
- **None** Solo autenticación
- **Sign** Firmar los datos para ayudar a mantener la integridad de los datos transmitidos, pero sin encriptarlos

```
[ServiceContract(ProtectionLevel =
System.Net.Security.ProtectionLevel.None]
public interface IProductService
{
   //…
}
```

SessionMode

El parámetro SessionMode especifica el tipo de soporte para sesiones seguras que un contrato requiere o soporta. El valor de este parámetro viene del enumerador *SessionMode*. Los siguientes valores están disponibles:

- **Allowed** Especifica que el contrato soporta sesiones seguras si la conexión entrante las soporta también.
- **NotAllowed** Especifica que el contrato no soporta sesiones seguras.
- **Required** Especifica que el contrato requiere sesiones seguras todo el tiempo

```
[ServiceContract(SessionMode = SessionMode.Required]
public interface IProductService
{
   //…
}
```

Las conexiones seguras están implementadas a través del protocolo *WS- Reliable* de mensajería.

El atributo [OperationContract]

El atributo *[OperationContract]* incluye los métodos marcados como parte del contrato del servicio e identifica el método marcado como una operación del servicio. Un método marcado como una operación de un contrato es expuesto al público, por lo que los métodos no marcados con este atributo no son expuestos externamente. El atributo *[OperationContract]* es mapeado con la operación WSDL equivalente. En la siguiente lista se muestran los parámetros disponibles para el atributo *[OperationContract].*

- Action
- AsyncPattern
- IsInitiating
- IsOneWay
- IsTerminating
- Name
- ProtectionLevel
- ReplyAction

Action

El parámetro Action establece la acción del WS-Addressing para un mensaje solicitado. Esto define la acción que identifica la operación actual.

```
[OperationContract(Action = true)]
void ChangeStockLevel(string productNumber, int newStockLevel, string
shelf, int bin);
```

AsyncPattern

El parámetro AsyncPattern especifica que operación es una operación asíncrona. Las operaciones del servicio pueden ser síncronas o asíncronas. Las operaciones asíncronas son implementadas usando el par de métodos que comienzan por *Begin* y *End*. Windows Communication Foundation redirige los mensajes entrantes al método *Begin* y el resultado del método *End* es enviado como mensaje de salida.

```
[OperationContract(AsyncPattern = true)]
void BeginChangeStockLevel(string productNumber, int newStockLevel,
string
shelf, int bin, AsyncCallback callback, object state);

void EndChangeStockLevel(IAsyncResult ar);
```

IsInitiating

El parámetro IsInitiating especifica si una operación implementada por el método asociado puede o no iniciar una sesión en el servidor. El valor por efecto para este parámetro es *true*.

```
[ServiceContract]
public interface IProductService
{
[OperationContract(IsInitiating = true, IsTerminating = false)]
void Login(user);
```

```
[OperationContract(IsInitiating = false, IsTerminating = false)]
void ChangeStockLevel(string productNumber, int newStockLevel, string
shelf, int bin);

[OperationContract(IsInitiating = false, IsTerminating = true)]
void Logout(user);
}
```

IsOneWay

La comunicación entre servicios puede ser unidireccional y bidireccional. Por defecto la comunicación es bidireccional, esto significa que la operación del servicio puede recibir mensajes y enviarlos. El parámetro IsOneWay especifica cuando una operación de un servicio devuelve un mensaje de respuesta. El valor por defecto para este parámetro es *false*.

```
[ServiceContract]
public interface IProductService
{
  [OperationContract(IsOneWay = true)]
  void Login(user);

  [OperationContract(IsOneWay = false)]
  void ChangeStockLevel(string productNumber, int newStockLevel,
  string
shelf, int bin);
}
```

En una comunicación de tipo one-way, el cliente inicia la comunicación y continúa la ejecución del código sin esperar una respuesta por parte del servicio. Sin embargo, en una comunicación de tipo two-way, el cliente espera una respuesta del servicio antes de continuar la ejecución del código

IsTerminating

La propiedad IsTerminating especifica que la llamada a una operación delservicio va a terminar la sesión de comunicación que está abierta.

```
[ServiceContract]
public interface IProductService
{
[OperationContract(IsInitiating = true, IsTerminating = false)]
void Login(user);

[OperationContract(IsInitiating = false, IsTerminating = false)]
void ChangeStockLevel(string productNumber, int newStockLevel, string
shelf, int bin);

[OperationContract(IsInitiating = false, IsTerminating = true)]
void Logout(user);
}
```

Name

La propiedad *Name* es usada para establecer el nombre de la operación. La propiedad sobrescribe el

elemento <*operation*> del WSDL.

```
[ServiceContract]
public interface IProductService
{
[OperationContract(Name = “UserLogin”)]
void Login(user);
}
```

ProtectionLevel

El parámetro ProtectionLevel especifica el nivel de encriptación para el mensaje de la operación. Esto incluye encriptación, firma digital, o ambos por cada punto que expone el contrato. El valor de este parámetro viene del enumerador *System.Net.SecurityLevel.ProtectionLevel*. Los siguientes valores están disponibles

- **EncryptAndSign** Encriptar y firmar datos aseguran la confidencialidad e integridad delos datos transmitidos.
- **None** Solo autenticación.
- **Sign** Firmar los datos para ayudar a mantener la integridad de los datos transmitidos, pero sin encriptarlos.

ReplyAction

Esta propiedad especifica la acción de respuesta para los mensajes entrantes.
El valor para este parámetro ha de ser una URL.

```
[ServiceContract]
public interface IProductService
{
[OperationContract(ReplyAction =
“http://AdventureWorks.Contracts/IProductService/RefreshStock”)]
}
```

Contrato de Datos

Un *Data Contract* describe los datos que van a ser intercambiados. Es un contrato formal entre dos partes que contiene información relacionada con los datos que van a ser intercambiados.

Como nota importante recordar que para que dos partes puedan comenzar a comunicarse, no es necesario que compartan los tipos de datos, solo sería necesario que compartieran los *Data Contracts*. El *Data Contract* define específicamente como será serializado y desearializado cada parámetro y tipo de respuesta, en qué orden van a ser intercambiado.

Es muy importante conocer y entender como aplica WCF la serialización y deserialización. La serialización es el acto de tomar estructuras de datos y convertirlas a un formato que pueda ser enviado a través de un cable. La desearialización es el proceso inverso

El atributo [DataContract]

Los *Data Contracts* son definidos declarativamente como los *Service Contracts* y los *Operation Contracts*.

Para definir un *Data Contract* solo hay que decorar una clase o una enumeración con el atributo *[DataContract]*. No se puede decorar un interfaz como Data Contract

```
[DataContract]
public class Product
{
…
}
```

Una vez definido el Data Contract, podemos definir los miembros de dichocontrato. Esto miembro son campos o propiedades de la clase.

```
[DataContract]
                public class Product
                {
                [DataMember]
                public string Name;

                [DataMember]
                public string ProductNumber;

                [DataMember]
                public string Color;

                [DataMember]
                public decimal ListPrice;
                }
```

El atributo [DataContract] tiene la opción de definir una serie de parámetros que permiten especificar detalles sobre el contrato.

Name

El parámetro Name es usado para obtener o establecer el nombre del data contract. Esta propiedad es usada como el nombre del tipo en el esquema

XML

```
[DataContract(Name = "SaleProduct")]
public class Product
{
…

}
```

Namespace

El parámetro Namespace especifica el namespace para el data contract.

```
[DataContract(Namespace = "http://www.AdventureWorks.com")]
public class Product
{
…
}
```

El atributo [DataMember]

El atributo [DataMember] es aplicado a todos los miembros de un tipo DataContract para identificar que es un data member. Como el atributo *[DataContract],* el atributo *[DataMember]* tiene varios parámetros que permiten especificar detalles

EmitDefaultValue

La propiedad *EmitDefaultValue* especifica si se serializa o no el valor por defecto de un campo o propiedad que está siendo serializada.

IsRequired

El parámetro IsRequired es usado para informar al motor de serialización que un data member debe estar presente

```
[DataContract]
public class Product
{
  [DataMember(IsRequired = “false”)]
  public string Name;
}
```

Name

La propiedad Name es usada para establecer el nombre del data member. Esta propiedad sobrescribe el nombre por defecto la del data member.

```
[DataContract]
public class Product
{
 [DataMember(Name = “NameProduct”)]
 public string Name;
 }
```

Order

El parámetro Order es usado para especificar el orden por el cual los data members son serializados y deserializados. Es decir, el orden en el que los datos son enviados y recibidos en el XML

```
[DataContract]
public class Product
{
[DataMember(Order=0)]
public string Name;
[DataMember(Order=1)]
public string ProductNumber;
[DataMember(Order=2)]
public string Color;
[DataMember(Order=3)]
public decimal ListPrice;

}
```

Contrato de Mensaje

El contrato de mensaje provee la manera de controlar el formato de los mensajes SOAP. En la mayoría de los casos no será necesario utilizar este nivel de control ya que el data contract ya provee prácticamente todo el control necesario. Sin embargo, es bueno tener presente que está disponible este tipo de control para según

qué necesidades.

Seguramente la mejor razón por la que necesitemos de este nivel de control sea por la interoperabilidad. El message contract provee el nivel de interoperabilidad necesario para comunicarse con clientes y otros servicios en los que existe un esquema o WSDL particular.

El atributo [MessageContract]

El message contract es definido aplicando el atributo [MessageContract]. A continuación, habrá que especificar por cada mensaje el atributo *[MessageBodyMember]* y *[MessageHeader]*.

```
[MessageContract]
public class Product
{
   [MessageHeader]
   Public string Name;

   [MessageBodyMember]
   Public decimal Price;

 }
```

El atributo [MessageContract] define clases fuertemente tipadas que están asociadas con el mensaje SOAP. Esto permite crear un mensaje tipado que habilita el pasar mensajes entre servicios, lo que son operaciones de servicios.

A continuación se tratan los parámetros disponibles para el atributo *[MessageContract]*.

HasProtectionLevel

El parámetro HasProtectionLevel especifica el nivel de protección del mensaje. El mensaje puede ir encriptado, firmado o ambas cosas. Los valores disponibles serían true o false. Si se especifica true, el mensaje puede ir encriptado, firmado o ambas cosas. En caso contrario, si se especifica false, se está especificando que no es necesario ningún nivel de protección. El valor por defecto es false.

```
[MessageContract(HasProtectionLevel = true)]
public class Product
{
  …
}
```

IsWrapped

El mensaje puede ser formateado de varias formas. Una de las formas es document/wrapper. En este tipo de documento XML, el elemento de salida es llamado elemento wrapper

```
[MessageContract(IsWrapped = True)]
public class Product
{
}
```

ProtectionLevel

El parámetro ProtectionLevel especifica el nivel de protección requerido para utilizar el servicio. Esto incluye encriptación, firma digital, o ambos por cada punto que expone el contrato.

El valor de este parámetro viene del enumerador *System.Net.SecurityLevel.ProtectionLevel*. Los siguientes valores están disponibles:

- **EncryptAndSign** Encriptar y firmar datos aseguran la confidencialidad e integridad de los datos transmitidos.
- **None** Solo autenticación.
- **Sign** Firmar los datos para ayudar a mantener la integridad de los datos transmitidos, pero sin encriptarlos

```
[MessageContract(ProtectionLevel =
System.Net.Security.ProtectionLevel.None)]
public class IProduct
{
  …
 }
```

WrapperName

El parámetro WrapperName trabaja en conjunto con el parámetro descripto anteriormente IsWrapped, en caso de que este sea True. El parámetro especifica el nombre del elemento XML de salida

```
[MessageContract(IsWrapped = True, WrapperName=”Order”)]
public class Product
{
  …
 }
```

WrapperNamespace

El parámetro WrapperNamespace también trabaja en conjunto con el parámetro IsWrapped

```
[MessageContract(IsWrapped = True, WrapperName=”Order”,
WrapperNamespace=”http://www.adventureworks.com”)]
public class Product
{
  …
}
```

El atributo [MessageHeader]

El atributo *[MessageHeader]* mapea los campos y propiedades de la cabecera del mensaje SOAP con los marcados por el atributo *[MessageHeader]*.

```
[MessageContract]
public class Product
{
  [MessageHeader]
  Public string Name;
```

```
}
```

A continuación se detallan los parámetros disponibles para el atributo *[MessageHeader]*.

Actor

Como los mensajes son enviados desde un emisor a un receptor, WCF provee un mecanismo para especificar qué mensaje en específico es el objetivo de un endpoint específico.

```
[MessageContract]
public class Product
{
  [MessageHeader(Actor=”http://www.AdventureWorks.com/Product2”)]
  Public string Name;
}
```

MustUnderstand

El parámetro MustUnderstand es usado en conjunción con el parámetro Actor. Los valores que permite son true o false. Si el parámetro es True, el URI especificado por el parámetro Actor, debe entender la cabecera de entrada para poder procesar apropiadamente la cabecera

```
[MessageContract]
public class Product
{
[MessageHeader(Actor=”http://www.AdventureWorks.com/Product2”,
MustUnderstand=”true”)]
Public string Name;
}
```

Name

El parámetro Name especifica el nombre del elemento message contract. El valor debe ser un nombre de elemento válido para XML

```
[MessageContract]
public class Product
{
[MessageHeader(Name=“Title”)]
}
```

Namespace

El parámetro Namespace especifica el namespace del elemento del message contract.

```
[MessageContract]
public class Product
{
[MessageHeader(Namespace=“http://www.AdventureWorks.com/Product/Name
”)]
}
```

Relay

El parámetro Relay le dice al endpoint que está procesando el mensaje, que lo redirija al siguiente endpoint especificado por el parámetro Actor una vez que el endpoint actual procese el mensaje

```
[MessageContract]
public class Product
{
  [MessageHeader(Actor="http://www.AdventureWorks.com/Product2",
Relay="true")]
  Public string Name;
}
```

El atributo [MessageBodyMember]

El atributo [MessageBodyMember] identifica los miembros que van a ser serializados como cuerpo del elemento SOAP

```
[MessageContract]
public class Product
{
  [MessageHeader]
  Public string Name;

  [MessageBodyMember]
  Public decimal Price;
}
```

Name

La propiedad Name del atributo *[MessageBodyMember]* define el nombre del elemento para este miembro

```
[MessageContract]
public class Product
{
  [MessageHeader]
  Public string Name;

  [MessageBodyMember(Name="ProductPrice")]
  Public decimal Price;

 }
```

Order

La propiedad Order del atributo *[MessageBodyMember]* especifica la posición ordinal en la que cada miembro del cuerpo del mensaje será serializado en el mensaje SOAP.

```
[MessageContract]
public class Product
{
[MessageHeader]
Public string Name;

[MessageBodyMember(Order=1)]
Public decimal Price;
}
```

El atributo [MessageProperty]

El atributo *[MessageProperty]* especifica los miembros que no van a ser serializados en el mensaje SOAP, pero que contienen datos que son incluidos en un mensaje personalizado

```
[MessageContract]
public class Product
{
   [MessageHeader]
   Public string Name;

   [MessageProperty]
   Public string Color
   {
        get{ return _Color;}
        set{ _Color = value;}
   }
}
```

Name

Permite dar un nombre al MessageProperty.

```
[MessageContract]
public class Product
{
  [MessageHeader]
  Public string Name;

  [MessageProperty(Name=”ColorName”]
  Public string Color
  {
    get{ return _Color;}

    set{ _Color = value;}

   }

}
```

Comportamiento de los Servicios

Existen dos tipos de Behaviors en Windows Communication Foundation, los de servicio y operación

- Service Behavior
- Operation Behavior

Estos se aplican de la misma manera que el resto de objetos de WCF. Mediante los atributos *[ServiceBehavior]* y *[OperationBehavior*

El Atributo [ServiceBehavior]

El atributo [ServiceBehavior] se aplica directamente sobre la clase que implementa el servicio. Viene de la clase ServiceBehaviorAttribute de System.ServiceModel y especifica cuál será el comportamiento en tiempo de ejecución de una implementación de un contrato de servicio.

Disponemos de las siguientes propiedades para definir en detalle el atributo *[ServiceBehavior]:*

- Name
- Namespace
- AddressFilterMode
- AutomaticSessionShutdown
- ConcurrencyMode
- ConfigurationName
- IgnoreExtensionDataObject
- IncludeExceptionDetailInFaults
- InstanceContextMode
- ReleaseServiceInstanceOnTransactionComplete
- TransactionAutoCompleteOnSessionClose
- TransactionIsolationLevel
- TransactionTimeout
- UseSynchronizedContext
- ValidateMustUnderstand

AddressFilterMode

Una manera fácil de cambiar el filtro de mensajes usado por un servicio es a través de la propiedad AddressFilterMode de *[ServiceBehavior]*.

AddressFilterMode tiene tres valores: Any, Exact y Prefix. Cada uno de estos valores se asigna a la implementación de MessageFilter correspondiente

```
 [ServiceBehavior(AddressFilterMode=AddressFilterMode.Any)]
public class ProductService : IProductService
{
   ...
}
```

AutomaticSessionShutdown

Especifica si la sesión es cerrada automáticamente cuando el cliente cierra una sesión saliente

```
[ServiceBehavior(AutomaticSessionShutdown=false)]
public class ProductService : IProductService
{
   ...
 }
```

ConcurrencyMode

Especifica el tipo de concurrencia soportado por el servicio. Los valores disponibles son

- **Single** Mientras haya un mensaje procesándose, no se aceptarán más mensajes, por lo que quedará en espera hasta que se termine de procesar el mensaje actual. No acepta llamadas reentrantes.
- **Multiple** El servicio es multihilo. Hay que manejar manualmente la consistencia de los estados y la sincronización.
- **Reentran** Es como el tipo Single pero acepta llamadas reentrantes

ConfigurationName

Especifica el valor a usar para localizar el elemento service en el fichero de configuración.

```
[ServiceBehavior(ConfigurationName="WcfService")]
public class ProductService : IProductService
```

IgnoreExtensionDataObject

Especifica cuando enviar datos serializados desconocidos en el cliente y el servicio.

```
[ServiceBehavior(IgnoreExtensionDataObject=true)]
public class ProductService : IProductService
{
...
}
```

IncludeExceptionDetailInFaults

Cuando una excepción no manejada es lanzada, este parámetro especifica si es encapsulada en una FaultExcepcion y relanzada. Es utilizada durante el proceso de desarrollo del servicio. El valor por defecto es false.

```
[ServiceBehavior(IncludeExceptionDetailInFaults=true)]
public class ProductService : IProductService
{
...
}
```

InstanceContextMode

Utilizar la propiedad de InstanceContextMode para especificar cuando se crean los nuevos objetos del servicio. Debido a que el objeto del servicio no está conectado directamente con el canal con el que se comunica, la vida de los objetos del servicio es independiente de la del canal entre un cliente y el servicio.

La propiedad puede tomar los siguientes valores

- **PerCall** Un nuevo objeto InstanceContext es creado y reciclado por cada llamada sucesiva.
- **PerSession** Establece en el servicio el crear un nuevo objeto cuando una nueva sesión se haya establecido entre un cliente y el servicio. Las llamadas siguientes en la misma sesión son manejadas por el mismo objeto.
- **Single** Solamente un objeto de InstanceContext se utiliza para todas las llamadas entrantes y no es reciclado a las siguientes llamadas. Si no existe un objeto de servicio, se crea uno

El valor por defecto es PerSession.

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall)]
public class ProductService : IProductService
{
   ...
}
```

ReleaseServiceInstanceOnTransactionComplete

Específica si el objeto de servicio es reciclado una vez la transacción actual es completada.

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete=true)]
public class ProductService : IProductService
{
   ...
}
```

TransactionAutoCompleteOnSessionClose

Cuando esta propiedad es verdadera, toda transacción pendiente es completada cuando la sesión actual es cerrada.

```
[ServiceBehavior(TransactionAutoCompletoOnSession=true)]
public class ProductService : IProductService
{
   ...
```

TransactionIsolationLevel

Especifica el nivel del aislamiento de la transacción para las nuevas transacciones creadas en el servicio, y las transacciones entrantes desde un cliente.

Los valores posibles son:

- **Chaos** Los cambios pendientes de las transacciones más visibles no pueden ser sobrescritos.
- **ReadCommitted** Los datos temporales no se pueden leer durante la transacción, sino pueden ser modificados.
- **ReadUncommitted** Los datos temporales se pueden leer y modificar durante la transacción.
- **RepeatableRead** Los datos temporales se pueden leer pero no modificar durante la transacción. Los nuevos datos se pueden agregar durante la transacción.
- **Serializable** Los datos temporales se pueden leer pero no modificar durante la transacción. Ninguno de los datos nuevos se podrán agregar durante la transacción.
- **Snapshot** Los datos temporales pueden ser leídos. Antes de modificar cualquier dato, la transacción comprueba si los datos han sido cambiados por otra transacción. Se lanza un error si se han cambiado los datos. El proceso permite el acceso de la transacción actual a los datos previamente establecidos.
- **Unspecified** Un nivel diferente del aislamiento se está utilizando a excepción del especificado, y el nivel es indeterminado

Los datos temporales son aquellos afectados por una transacción

```
[ServiceBehavior(TransactionIsolationLevel=System.Transaction.IsolationLe
ve
l.ReadUncommitted)]
public class ProductService : IProductService
{
   ...
```

TransactionTimeout

En esta propiedad especifica el tiempo que tiene una transacción para completarse

```
[ServiceBehavior(TransactionTimeout=”00:01:00”)]
public class ProductService : IProductService
{
...
}
```

UseSynchronizedContext

Con esta propiedad se especifica si se usará el contexto actual de sincronización. El valor por defecto es true.

```
[ServiceBehavior(UseSynchronizedContext=true)]
public class ProductService : IProductService
{
   ...
}
```

ValidateMustUnderstand

Especifica si se fuerza el procesar la cabecera SOAP MustUnderstand

```
[ServiceBehavior(ValidateMustUnderstand=false)]
public class ProductService : IProductService
```

```
{
   ...
```

El Atributo [OperationBehavior]

El atributo *[OperationBehavior]* se aplica directamente sobre un método de la clase que implementa el servicio Viene de la clase OperationBehaviorAttribute de System.ServiceModel y especifica cuál será el comportamiento en tiempo de ejecución de una operación del servicio.

Las siguientes p`ropiedades están disponibles `para ela atributo *[OperationBehavior]*

- AutoDisposeParameters
- Impersonation
- ReleaseInstanceMode
- TransactionAutoComplete
- TransactionScopeRequired

AutoDisposeParameters

Si esta propiedad es true, especifica que los parámetros del método asociado son automáticamente disposed.

Por defecto es true.

```
[OperationBehavior(AutoDisposeParameters=true)]
public double Division(double o1, double o2)
{
  return o1 / o2;
}
```

Impersonation

La propiedad Impersonation especifica el comportamiento de “impersonar” para una operación en la instancia actual del servicio. Permite identificar qué métodos se ejecutarán bajo una identidad suplantada. El valor por defecto es NotAllowed.

Existen tres posibles valores para esta propiedad

- **Allowed** La “impersonación” es realizada si las credenciales están disponibles y ImpersonateCallerForAllOperations es true.
- **NotAllowed** La “impersonación” no está permitida.
- **Required** La “impersonación” es requerida

```
[OperationBehavior(Impersonation=Impersonation.Allowed)]
public double Division(double o1, double o2)
{
    return o1 / o2
 }
```

ReleaseInstanceMode

Es usada para determinar cuándo se recicla el objeto del servicio durante el curso de una operación. Es recomendable usar esta propiedad en conjunción de InstanceContextMode con el valor PerCall, para asegurarnos que siempre obtendremos un nuevo objeto cuando el servicio sea ejecutado. Posibles valores de esta propiedad

- **AfterCall** Recicla el objeto una vez completada la operación.
- **BeforeAndAfterCall** Recicla el objeto antes de llamar a la operación y una vez completada la misma.
- **BeforeCall** Recicla el objeto antes de llamar a la operación
- **None** Recicla el objeto acorde con la propiedad InstanceContextMode

```
[OperationBehavior(ReleaseInstanceMode=ReleaseInstanceMode.AfterCall)
]
public double Division(double o1, double o2)
{
   return o1 / o2;
}
```

TransactionAutoComplete

Especifica si la transacción actual se finaliza automáticamente en caso de que no haya habido errores. En caso de algún error, la transacción se cancela.

```
[OperationBehavior(TransactionAutoComplete=true)]
public double Division(double o1, double o2)
{
   return o1 / o2;
 }
```

TransactionScopeRequired

Especifica cuando el método asociado requiere una transacción. En caso de haber una transacción, el método se ejecutar dentro de ella, en caso de que no exista, se creará una transacción y será la usada por el método.

```
[OperationBehavior(TransactionScopeRequired=true)]
public double Division(double o1, double o2)
{
return o1 / o2;
}
```

Alojamiento de Servicios

El dónde y cómo se alojan los servicios es una parte clave de estos. Windows Communication Foundation (WCF) no incluye su propio host, pero sí una clase llamada *ServiceHost* que permitirá alojar fácilmente los servicios WCF en su propia aplicació. No tiene que pensar en los aspectos específicos sobre el transporte de red para asegurarse de que sus servicios son accesibles. Se trata de configurar los endpoint de sus servicios mediante programación o declaración, y llamar al método *Open* de *ServiceHost*. Podríamos resumir en tres las opciones que disponemos donde alojar servicios WCF:

- Auto alojamiento en una aplicación .NET
- Alojamiento como servicio Windows
- Alojamiento en cualquier versión de IIS

Auto alojamiento en una aplicación .NET

De las tres manera que existen de alojamiento, esta es la más sencilla y flexible que existe. El único "problema" es que nosotros somos los responsables de iniciar y parar el servicio. A continuación se muestran las ventajas y desventajas de utilizar este tipo de alojamiento de servicios WCF

Ventajas

- **Fácil de usar** Con sólo unas pocas líneas de código, tiene su servicio en funcionamiento.
- **Flexible** Control de la vida de los servicios mediante los métodos proporcionados por la clase *ServiceHost<T>*.
- **Fácil de depurar** Se podrá depurar desde un entorno muy familiar.
- **Fácil de implementar**.
- **Compatible con todos los enlaces y transportes**

Desventajas

- **Disponibilidad limitada**
- El servicio es accesible sólo cuando la aplicación está en ejecución
- **Características limitadas** Las aplicaciones auto alojadas disponen de compatibilidad limitada con una alta disponibilidad, una sencilla administración, estabilidad, capacidad de recuperación, versiones e implementación de escenarios. Al menos, el WFC original no las ofrece, de modo que en un escenario auto alojado debe implementar estas características usted mismo; IIS, por ejemplo, incluye varias de estas características de forma predeterminada

Alojamiento como servicio Windows

Con Visual Studio 2005 es muy sencillo crear un servicio Windows utilizando la plantilla que se provee. Tan solo habrá que implementar los métodos *Start()* y *Stop()* del servicio Windows realizando las funciones de *Open()* y *Close()* del *ServiceHost<T>* respectivamente. A continuación se muestran las ventajas y desventajas de utilizar este tipo de alojamiento de servicios WCF.

Ventajas

- **Inicio automático** El Administrador de control de servicios de Windows le permite establecer el tipo de inicio a automático, para que el servicio se inicie cuando lo haga Windows, sin un inicio de sesión interactivo en el equipo
- **Recuperación** El Administrador de control de servicios de Windows cuenta con compatibilidad integrada para reiniciar los servicios cuando se produzcan errores.
- **Identidad de seguridad** El Administrador de control de servicios de Windows le permite elegir una identidad específica de seguridad bajo la que desea que se ejecute el servicio, incluidas cuentas de servicio de red o sistema integradas.
- **Capacidad de administración** En general, los operadores de Windows saben mucho acerca del Administrador de control de servicios de Windows y otras herramientas de administración que pueden funcionar con la instalación y configuración de servicios de Windows. Esto mejorará la aceptación de los servicios de Windows en entornos de producción; no obstante, para hacer que los servicios sean de mantenimiento sencillo, probablemente tendría que agregar algunas características de instrumentación y registro.
- **Compatibilidad con todos los enlaces y transportes**.

Desventajas

- **Implementación** Los servicios deben instalarse con la utilidad *Installutil.exe* de .NET Framework o a través de una acción personalizada de un paquete de instalación.
- **Características limitadas** Los servicios de Windows todavía tienen un conjunto limitado decaracterísticas integradas para admitir una alta disponibilidad, una sencilla administración, versiones e implementación de escenarios. Básicamente, debe satisfacer estos requisitos usted mismo a través de código personalizado mientras, por ejemplo, IIS incluye varias de estas características de forma predeterminada. Los servicios de Windows agregan capacidad de recuperación y algunas características de seguridad, pero no las suficientes por lo que habrá que realizar algunas tareas

Alojamiento en cualquier versión de IIS

Cuando Microsoft lanzó ASP.NET 1.0 ya venía preparado para hacer realidad los servicios web. El problema estuvo en la estrecha dependencia con el protocolo HTTP que limitaba en gran medida la implementación orientada a servicios. Más tarde salió a la luz WSE (Web Services Enhancements) cubriendo algunas limitaciones de los servicios web ASP.NET y permitiendo abordar algunas implementaciones WS-*.

Para alojar un servicio WCF en IIS necesitaremos de un nuevo fichero con la extensión .svc. En este archivo se asocia un servicio con su implementación y permite a IIS crear el *ServiceHost* necesario. Con esto dejamos la responsabilidad de instanciar e iniciar el servicio a IIS, despreocupándonos de esto, al contrario que ocurre en las anteriores formas de alojamiento que hemos visto. Los Endpoints del servicio son definidos en un fichero de configuración como norma general, este fichero en el alojamiento IIS será un Web.config. Así sería un ejemplo del contenido del un fichero .svc

```
<%@ServiceHost Service="AdventureWorks.Services.ProductService" %>
 <%@Assembly Name="AdventureWorks.Services" %
```

Bindings, Transporte y Endpoints

Hay múltiples aspectos en la comunicación con cualquier servicio dado, al igual que hay muchos posibles patrones de comunicación: Mensajes síncronos, asíncronos, bidireccionales, mensajes que se envían inmediatamente o se encolan, las colas pueden ser volátiles o duraderas. También hay muchos protocolos de transporte posibles para los mensajes, como HTTP, HTTPS, TCP, P2P, IPC o MSMQ. Al igual que hay muchas opciones de codificación de mensajes pudiendo elegir entre texto plano para habilitar la interoperabilidad, codificación binaria para optimizar el rendimiento, o MTOM cuyo significado en inglés es Mecanismo Transporte de Mensajes Optimizado.
Tenemos algunas opciones para proteger nuestros mensajes pudiendo elegir entre intercambiar mensajes sin protección, proteger los mensajes a nivel de transporte solo, proveer al mensaje de nivel de seguridad y privacidad, o numerosos mecanismos de autenticación y autorización. Si nos ponemos a enumerar todas las posibles opciones que tenemos de comunicación e interacción el número de permutaciones posibles es grandísimo. Pero ante esto lo claro es que tanto el cliente como el servicio han de estar alineados en todas estas opciones para poder establecer comunicaciones de forma correcta.

Bindings y Transporte

Para simplificar todas estas opciones y hacerlas más manejables, WCF agrupa todas ellas como aspectos de comunicación en bindings. Por tanto un Binding es un sistema consistente de opciones relacionadas con el protocolo de transporte, el tipo de codificación para el mensaje, el patrón de comunicación, la seguridad, la fiabilidad, la propagación de transacciones y la interoperabilidad. Lo ideal es extraer todo esto del código fuente del servicio y permitir al servicio solamente fijarse en la implementación de la lógica de negocio. Con los bindings podemos usar prácticamente la misma lógica de servicio sobre diferentes configuraciones

Bindings Estandar

A continuación se describen los nuevo bindings estándar que define WCF y que podemos usar directamente:

Basic Binding

Ofrecido por la clase *BasicHttpBinding,* está diseñado para exponer un servicio WCF como un servicio web ASMX, por lo que todos los antiguos clientes pueden trabajar con el nuevo servicio. Cuando es usado por un cliente, le permite trabajar con los antiguos web services.

TCP Binding

Ofrecido por la clase *NetTcpBinding,* usa peer networking como transporte. Este tipo de transporte habilita tanto al cliente como al servicio a suscribirse al mismo grupo de mensajes.

IPC Binding

Ofrecido por la clase *NetNamedPipeBinding,* usa named pipes como transporte para comunicaciones en la misma máquina. Es el tipo de binding más seguro porque no permite comunicaciones del exterior de la máquina donde esta alojado además de soportar una gran variedad de características similares al TCP binding.

Web Service (WS) Binding

Ofrecido por la clase WSHttpBinding, usa los protocolos HTTP o HTTPS como transporte y está diseñado para ofrecer una variedad de características como fiabilidad, transacciones, y seguridad sobre internet.

Federated WS Binding

Ofrecido por la clase WSFederationHttpBinding, es una especialización del WS binding ofreciendo soporte para seguridad federada

Duplex WS Binding

Ofrecido por la clase WSDualHttpBinding, es similar al WS binding salvo que soporta comunicaciones bidireccionales desde el servicio al cliente.

MSMQ Binding

Ofrecido por la clase NetMsmqBinding, usa MSMQ como transporte estando diseñado para ofrecer soporte para llamadas a colas desconectadas.

MSMQ integration Binding

Ofrecido por la clase MsmqIntegrationBinding, cubre mensajes de WCF hacia y desde mensajes MSMQ. Está diseñado para interoperar con clientes MSMQ.

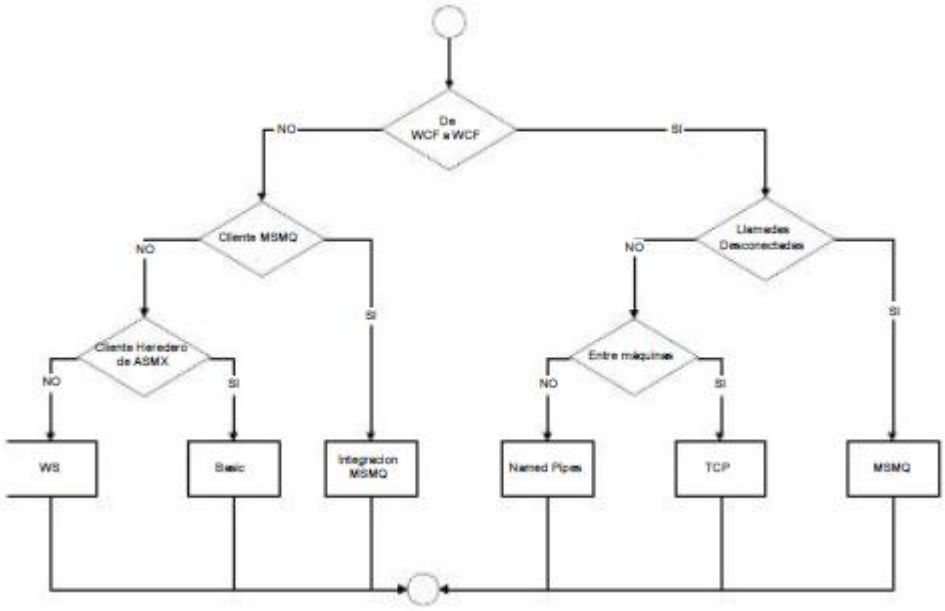
Formatos y Codificación

Cada binding estándar que hemos visto antes utiliza un tipo de transporte y codificación diferente:

Binding	Transporte	Codificación	Interop
BasicHtpBinding	HTTP/HTTPS	Texto, MTOM	Si
NetTcpBinding	TCP	Binaria	No
NetPeerTcpBinding	P2P	Binaria	No
NetNamedPipeBinding	IPC	Texto, MTOM	No
WSHttpBinding	HTTP/HTTPS	Texto, MTOM	Si
WsFederationHttpBinding	HTTP/HTTPS	Texto, MTOM	Si
WSDualHttpBinding	HTTP	Binaria	Si
NetMsmqBinding	MSMQ	Binaria	No

Seleccionando un tipo de Binding

Para elegir un tipo de binding para un servicio a desarrollar podemos seguir el siguiente diagrama de decisión que nos ayudará a guiarnos y decidir



La primera pregunta que debemos realizarnos es si el servicio WCF que vamos a desarrollar va a interactuar

con otro servicio WCF. En caso que la respuesta sea afirmativa, y el cliente utilice MSMQ usaremos el binding MsmqInterationBinding.

Si necesitas operar con clientes no WCF y estos clientes esperan el protocolo básico de servicios web, elegiremos el binding BasicHttpBinding, que expone nuestro servicio WCF al mundo exterior como si fuera un servicio web ASMX.

El problema de este protocolo es que no podemos aprovechar las ventajas que suponen los protocolos del tipo WS-*. Por tanto, si el cliente es capaz de entender este tipo de protocolos, mejor utilizar los bindings de tipo WS (WSHttpBinding, WSFederationBinding o WSDualHttpBinding

Cuando asumimos que el cliente será un cliente WCF debemos preguntarnos si requiere interacción desconectada, si es así seleccionaremos el binding NetMsmqBinding. En caso contrario, si el cliente requiere comunicaciones conectadas y puede ser utilizado desde diferentes máquinas utilizaremos NetTcpBinding. Pero si el servicio y el cliente están en la misma máquina, utilizaremos NetNamedPipeBinding

Usando los Bindings

Como hemos visto, cada binding ofrece una gran variedad de configuraciones. Podemos utilizar los bindings de tres maneras. Usar el binding como está, siempre y cuando cumpla con nuestros requerimientos de negocio. Configurar algunas de las propiedades que tenga el binding para adaptarlo a nuestras necesidades: fiabilidad, seguridad, etc. O construirnos nuestro propio binding. La forma de trabajar habitual es la de utilizar un binding ya existente y configurar algunos de sus aspectos para lograr que alcance nuestras expectativas

```
<system.serviceModel>
  <services>
    <service name="AdventureWorks.Services.ProductService"
behaviorConfiguration="ProductService">
      <endpoint address="" binding="basicHttpBinding"
contract="AdventureWorks.Contracts.IProductService" />
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="ProductService">
        <serviceMetadata httpGetEnabled="true" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

EndPoints

En el ABC de los servicios, todo servicio tiene asociado un Address (Define dónde está el servicio), un Binding (Define cómo hay que comunicarse con el servicio) y un Contract (Define qué hace el servicio). Pues

bien, un Endpoint sería la fusión del Address, del Binding y del Contract
Por tanto, todo Endpoint ha de tener estos tres elementos y será el host del servicio el que exponga el endpoint. Desde el punto de vista lógico, podemos definir el endpoint como la interfaz del servicio.
Como regla general, todo servicio debe exponer al menos un endpoint y cada uno de estos tendrá exactamente un contrato. Todo endpoint en un servicio tiene una dirección única, pudiendo un único servicio exponer varios endpoints. Estos endpoint no tienen por qué tener relación alguna entre ellos pudiendo exponer el mismos contrato o no. Es importante recalcar que los endpoint son siempre externos al código del servicio WCF, pudiéndose administrar usando el fichero de configuración o mediante programación

En fichero de configuración

A continuación se muestra un ejemplo de cómo configurar un endpoint utilizando para ello el fichero de configuración:

```
<system.serviceModel>
  <services>
    <service name="AdventureWorks.Services.ProductService">
    <endpoint address="http://localhost/ProductService"
            binding="basicHttpBinding"
            contract="AdventureWorks.Contracts.IProductService" />
    </service>
  </services>
<system.serviceModel>
```

Cuando en el fichero de configuración se provee una dirección base, esta sobrescribe cualquier otra dirección base que el host del servicio pudiera haber provisto.
Cuando es el host que el provee la dirección base, se puede dejar el campo address en blanco.

```
<system.serviceModel>
  <services>
    <service name="AdventureWorks.Services.ProductService">
    <endpoint address=""
            binding="basicHttpBinding"
            contract="AdventureWorks.Contracts.IProductService" />
    </service>
  </services>
```

Pero si no proveemos dirección y el host tampoco, se lanzará una excepción

Programáticamente

La opción de configurar un endpoint programáticamente es equivalente a utilizar el fichero de configuración. Esto será posible utilizando el método *AddServiceEndpoint()* de la instancia del *ServiceHost*.
A continuación se muestra un ejemplo de configurar un endpoint de forma programática

```
// Crear el objeto ServiceHost
productsServiceHost = new ServiceHost(typeof (ProductService));
```

```
  // Suscribirse al evento Faulted
  productsServiceHost.Faulted += new
  EventHandler(productsServiceHost_Faulted);

  NetNamedPipeBinding binding = new NetNamedPipeBinding();
  binding.TransactionFlow = true;
  binding.TransactionProtocol = TransactionProtocol.OleTransactions;
  productsServiceHost.AddServiceEndpoint(typeof (IProductService),
              binding, "net.pipe://localhost/ProductServicePipe");

  // Arrancar el Servicio
productsServiceHost.Open();
```

Propagación y gestión de Excepciones

En la programación .NET pura, cuando una excepción no manejada ocurre, el proceso donde ha tenido lugar es terminado inmediatamente. El comportamiento de WCF ante excepciones no manejas no es exactamente igual.

Cuando una llamada a un servicio en nombre de un cliente causa una excepción, esta no tiene los permisos necesarios para parar el proceso donde se hospeda el servicio, por que el resto de clientes que estén utilizando el servicio no tienen porque verse afectados. Como consecuencia de esto, cuando una excepción no manejada ocurre, es serializada y devuelta como mensaje al cliente que ha realizado la llamada al servicio. Cuando el mensaje con la excepción llega al proxy cliente, este lanza una excepción del lado del cliente.

Podemos encontrar hasta tres tipos de errores diferentes del lado del cliente cuando invocamos un servicio WCF:

- **Errores de Comunicación** Disponibilidad de red, direcciones erróneas, el proceso que aloja el servicio no funciona, etc.
- **Errores en el Cliente** Estado del proxy y de los canales, contratos erróneos, etc.
- **Errores en el Servicio WCF**

Los errores de comunicación se manifiestan en el lado cliente como CommunicationExceptions, los de cliente cuando es problema de proxy se ven como ObjectDisposedException y los ocurridos en el servicio WCF por defecto llegarán al cliente como FaulExceptions

El modo de instanciación y las excepciones

Según haya sido instanciado el servicio WCF, cuando ocurre una excepción en el servicio esta afectara de una u otra manera a la instancia del servicio y a la posibilidad del cliente de seguir usando el proxy o el canal del servicio

Servicios Per-Call

Si la llamada al servicio encuentra una excepción, la instancia del servicio es finalizada después de la excepción y el proxy lanza una *FaultException*, en el lado del cliente.

Por defecto todas la excepciones lanzadas desde el servicio (con la excepción de las que derivan de *FaultException*) provocan un error en el canal por lo que incluso cuando el cliente captura la excepción no puede publicar las siguientes llamadas ya que estas dependen de *CommunicationObjectFaultException*. El cliente solo podrá cerrar el proxy

Servicios Per-Session

Cuando se usa este modo de instanciación, todas las excepciones (con la excepción de las que derivan de *FaultException*) terminan la sesión. WCF finaliza la instancia y el cliente obtiene un *FaultException*. Incluso si el cliente captura la excepción, este no podrá seguir usando el proxy porque las siguientes llamadas dependen de *CommunicationObjectFaultedException* La única acción que puede realizar el cliente de manera segura es cerrar el proxy, porque una vez que la instancia del servicio que participa en la sesión encuentra un error, ya no es posible seguir usando dicha sesión.

Servicios Single

Cuando es usado este modo de instanciación y se encuentra una excepción, la instancia singleton no se termina y continúa ejecutándose. Por defecto, todas las excepciones (salvo las excepciones que derivan de *FaultException*) provocan un fallo en el canal de comunicación y el cliente no puede publicar las siguientes llamadas salvo la que cierra el proxy. Si el cliente tiene una sesión de este tipo, es finalizada

Faults

El problema fundamental de las excepciones es que son específicas dependiendo de la tecnología utilizada para crear el aplicativo donde han sido lanzadas por lo que no deben ser compartidas fuera de ese ámbito. Para mantener la interoperabilidad de un servicio necesitamos una manera de compartir estas excepciones, de una tecnología en específico, mapeándolas con una forma neutral de presentar el error. Para esto existe lo que es llamado Soap Faults.

Las Soap Faults están basadas en un estándar de la industria que es independiente de la tecnología específica de las excepciones como excepciones de CLR, Java, C++, etc. Por tanto, cuando un servicio tiene que lanzar una excepción no utilizará las los tipos de excepciones del CLR sino que lanzará una excepción del tipo *FaultException<T>*.

```
[DataContract]
public class ConfigFault
{
   [DataMember]
   public string ConfigOperation;

   [DataMember]
   public string ConfigReason;

   [DataMember]
   public string ConfigMessage;
}
```

```
// Read the configuration information for connecting to the
AdventureWorks
database
Database dbAdventureWorks;
try
{
  dbAdventureWorks =
DatabaseFactory.CreateDatabase("AdventureWorksConnection");
}
catch(Exception e)
{
   ConfigFault cf = new ConfigFault();
   cf.ConfigOperation = "CreateDatabase";
   cf.ConfigReason = "Excepción leyendo la información de configuración
   de
la base de datos AdventureWorks database";
   cf.ConfigMessage = e.Message;
   throw new FaultException<ConfigFault>(cf);
}
```

FaultException<T> es una especialización de FaultException, por lo que cualquier cliente que programe contra FaultException será capaz de lanzar FaultException<T> también. El parámetro T para FaultException<T> conlleva los detalles del error. El tipo del detalle puede ser cualquiera, no siendo necesario que derive de la clase Exception. El único requerimiento es que sea serializable o un data contract. El usar una excepción con detalle nos permite estar más alineados con las prácticas convencionales de programación en .NET y por tanto resultará en un código de lectura más sencilla

Contratos de tipo Fault

Por defecto como hemos visto toda excepción lanzada desde el servicio al cliente llega como una *FaultException*. La razón es que cualquier cosa más allá de los errores de comunicación que el servicio quiere compartir con el cliente deben ser parte del comportamiento contractual del servicio.

WCF provee los fault contracts que es una forma de listar los tipos de errores que el servicio puede lanzar. La idea es que ese tipo de errores deben ser los mismos que los tipos de parámetros usados con *FaultException<T>*, y listándolos en un fault contract, el cliente WCF será capaz de distinguir entre los errores que pertenezcan al fault contract y del resto de errores

Para definir un Fault Contract existe el atributo *[FaulContract]* que ha de ser aplicado en la operación del contrato del servicio

```
[ServiceContract(Namespace = "http://AdventureWorks.Service")]
public interface IProductService
{
   [FaultContract(typeof(ConfigFault))]
   [FaultContract(typeof(DatabaseFault))]
   [OperationContract]
   List<string> ListProducts();

   [OperationContract]
   Product GetProduct(string productNumber);
}
```

La acción del atributo [FaulContrat] se limita a método que decora. Solo este método será el que puede lanzar este tipo de excepciones y sea el que tenga que propagarlas al cliente. Además, si la operación lanza una excepción que no forma parte del control esta llegará al cliente como una FaulException. Para propagar la excepción, el servicio debe lanzar exactamente el mismo tipo de listado de detalle que es definido en el fault contract. En el caso del ejemplo

```
[FaultContract(typeof(ConfigFault))]
```

Es importante tener en cuenta que no podremos utilizar fault contracts en una operación de tipo one-way porque en teoría nada debería ser devuelto en este tipo de operaciones.

Capturar Faults

Los fault contracts son publicados con el resto de los metadatos del servicio. Cuando un cliente WCF importa los metadatos, las definiciones de lo contratos contienen los fault contracts al igual que las definiciones de los tipos de detalle incluyendo información relevante. El cliente por tanto estaría preparado para capturar y lanzar los tipos de excepciones.

Hay que tener en cuenta que el cliente aun puede encontrar excepciones de comunicación. El cliente puede elegir en todo momento tratar todas las excepciones que no sean de comunicación simplemente capturando las excepciones basadas en *FaultException*.

Cuando el servicio lanza una excepción del tipo del faul contract, la excepción no provocará un fallo en el canal de comunicación. De esta manera, el cliente podrá capturar la excepción y continuar usando el proxy o cerrarlo de manera segura

Errores en Callbacks

Callbacks en el cliente puede fallar debido a errores en las comunicaciones o porque el propio callback lance una excepción de algún tipo. Al igual que las operaciones de los service contract, las operaciones de los callback contract pueden definir fault contracts.

Sin embargo, a diferencia de una llamada normal a un servicio necesitamos conocer qué es propagado al servicio y el cómo se manifiesta el error es el producto de:

- Cuando el callback es invocado.
- El uso de bindings.
- El tipo de la excepción lanzada.

Podemos esperar un callback fault contract porque los errores son propagados al host de acuerdo al contrato. Si el cliente lanza una excepción que aparece en el fault contract, o si el callback lanza un FaultException, no provocará el fallo del canal de comunicación y por tanto podremos capturar la excepción

y continuar usando dicho canal de tipo callback. Sin embargo, como en las llamadas a los servicios, después de una excepción que no forma parte de un fault contract, evitemos el uso del canal de callback

Asincronismo y Comunicaciones Duplex

WCF soporta operaciones asíncronas para englobar los servicios que no se ajustan al esquema de realizar un proceso y devolver un dato como resultado a la aplicación cliente. También soporta el patrón de diseño IAsyncResult. WCF habilita la implementación del patrón de diseño IAsyncResult de dos maneras distintas:

- En la invocación de la aplicación cliente
- En la forma en la que el servicio WCF implementa la operación

Comunicaciones Bidireccionales

Debemos prestar especial atención al tipo de bindings que vamos a usar si queremos que el cliente pueda soportar callbacks, es concreto, el binding a utilizar ha de soportar comunicaciones bidireccionales. Cualquiera de los extremos en la comunicación ha de ser capaz de iniciar comunicaciones y de aceptarla. Por ejemplo, los tipos de transporte TCP y named pipes son por naturaleza bidireccionales y podemos usar tanto el binding NetTcpBinding como el NetNamedPipeBinding con un cliente callback.
Sin embargo, el modelo de implementación usado en el protocolo HTTP hace no soportable el modo de operaciones bidireccionales. Por ejemplo, no podríamos usar el binding BasicHttpBinding o WSHttpBinding. Pero no hay que alarmarse si lo que estamos buscando es construir un sistema basado en el transporte HTTP, ya que WCF trae el binding WSDualHttpBinding para permitir este tipo de operaciones sobre HTTP. Para hacer posible esto, WSHttpBinding establece dos canales HTTP, uno que envía mensajes y otro que los recibe del cliente al servidor y viceversa. Esto es transparente para el desarrollador y es posible su uso como si se tratara de un canal único bidireccional.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <wsDualHttpBinding>
            <binding name="WSDualHttpBinding_IProductService"
closeTimeout="00:01:00" openTimeout="00:01:00"
receiveTimeout="00:10:00"
sendTimeout="00:01:00" bypassProxyOnLocal="false"
transactionFlow="false"
hostNameComparisonMode="StrongWildcard"
maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
messageEncoding="Text" textEncoding="utf-8" useDefaultWebProxy="true">
               <readerQuotas maxDepth="32"
maxStringContentLength="8192" maxArrayLength="16384"
maxBytesPerRead="4096" maxNameTableCharCount="16384" />
               <reliableSession ordered="true"
inactivityTimeout="00:10:00" />
               <security mode="Message">
                  <message clientCredentialType="Windows"
negotiateServiceCredential="true"
algorithmSuite="Default" />
               </security>
            </binding>
         </wsDualHttpBinding>
      </bindings>
      <client>
         <endpoint
address="http://localhost/ProductService/ProductService.svc"
binding="wsDualHttpBinding"
bindingConfiguration="WSDualHttpBinding_IProductService"
contract="AdventureWorks.ProductClient.ProductServiceClient.IProductSer
vice
" name="WSDualHttpBinding_IProductService">
         </endpoint>
      </client>
   </system.serviceModel>
</configuration>
```

Callback Contract

Las operaciones Callback forman parte del contrato del servicio y esto permite al Service Contract definir su propio Callback Contract. Un Service Contract tiene por lo menos un Callback contract. Una vez definido, para poder usarse el cliente ha de soportar callbacks y proveer un endpoint callback al servicio en cada llamada. La propiedad a usar es CallbackContract.

La propiedad CallbackContract asigna el tipo del contrato de tipo callback cuando el contrato se comunica en modo dúplex. Esta propiedad contiene el tipo que especifica el contrato callback. El tipo debe contener un *CallbackContract = typeof(ClientContract)* que representa el contrato requerido en el lado cliente en una operación dúplex de intercambio de mensajes, es decir, en un contrato cliente callback. Especificando esta propiedad informa al cliente que es necesario que "escuche" por llamadas entrantes procedentes del servicio.

En el siguiente código fuente se puede apreciar un ejemplo de lo visto

```
[ServiceContract(Namespace = "http://AdventureWorks.Service",
                 CallbackContract = typeof(IProductServiceCallback))]
public interface IProductService
{
   [FaultContract(typeof(ConfigFault))]
   [FaultContract(typeof(DatabaseFault))]
   [OperationContract]
   List<string> ListProducts();

   [OperationContract]
   Product GetProduct(string productNumber);

   [OperationContract]
   bool ChangePrice(string productNumber, decimal price);

   [OperationContract]
   bool SubscribeToPriceChangedEvent();

   [OperationContract]
   bool UnsubscribeFromPriceChangedEvent();
}

public interface IProductServiceCallback
{
   [OperationContract(IsOneWay = true)]
   void OnPriceChanged(Product product);

   }
```

La interfaz IProductServiceCallback no necesita ser marcada como Service Contract ya que es asignada como CallbackContract en el Service Contract IProductService. De esta manera será incluido en el metadata del servicio. A pesar de esto, los métodos de la interfaz que contiene las operaciones del Callback (IProductServiceCallback), han de ser marcados con el atributo OperationContract

Aplicaciones Cliente usando Asincronismo

Para conectar una aplicación cliente a un servicio que permita operaciones asíncronas, WCF provee una funcionalidad añadida en la herramienta de generación de proxies *svcutil.exe.* Para ello, deberemos añadir la etiqueta */async* a la herramienta cuando vayamos a generar el proxy. Esta etiqueta provoca la generación del par de métodos que comienza por *begin* y *end* por cada operación asíncrona que se encuentre en la metadata del servicio.

Una vez generado el proxy, la aplicación cliente puede invocar el método *begin* para iniciar la operación. Para poder invocar este método es necesario proveer el nombre del método callback. Gracias a esto, cuando el servicio finaliza la operación y devuelve el resultado al cliente, se ejecuta en un nuevo hilo este método callback. Además se puede usar el método *end* para indicar que la respuesta ha sido procesada.

Cabe destacar después de lo anterior, que no es necesario modificar el servicio para que soporte este tipo de programación asíncrona. Incluso, ni siquiera el servicio ha de ser un servicio WCF, pudiendo haber sido implementado usando cualquier otra tecnología. Entendamos pues, que el código necesario para este tipo de programación asíncrona es encapsulado en el proxy cliente generado por la herramienta svcutil.exe y el

framework .NET

Transaccionalidad de Servicios WCF

Antes de comenzar a adentrarnos en la transaccionalidad de servicios WCF, debemos definir que es una transacción.
Una transacción se puede definir como un grupo de una o más operaciones ejecutadas como un todo, es decir, las transacciones nos permite agrupar procesos unitarios como si fuesen uno solo.
Un buen ejemplo de transacción es la transferencia de fondos de una cuenta bancaria a otra. En este caso, hay que retirar los fondos de la primera cuenta e ingresarlos en la segunda para que se pueda considerar que la operación se ha realizado correctamente. Si se retiran correctamente los fondos, pero no se consigue realizar el ingreso, habrá que deshacer la retirada de fondos de la primera cuenta para dejar las dos cuentas en un estado correcto y coherente.
La solución para esto es incluir ambas operaciones en una única unidad de ejecución llamada transacción. La transacción provocará que si falla alguna de las operaciones, fallarán toda. Es decir, si se retiran fondos de una cuenta y cuando se van a ingresar en la cuenta destino falla el proceso, ambas operaciones será marcadas como falladas y todos los cambios realizados serán deshechos

ACID

Detrás de este acrónimo se descubren las cuatro características, en inglés, que debe poseer una transacción pura

- Atomic
- Consistent
- Isolated
- Durable

ATOMIC

Atómico, del latín *atomum*, tiene como significado: la parte más pequeña e indivisible, que no se puede dividir en más partes. Las transacciones deben ser atómicas, es decir que todas las operaciones incluidas en la transacción se ejecutan correctamente o si fallan, fallarán todas.

CONSISTENT

Significa que el resultado de salida será exactamente el que se espera. Es decir, en el ejemplo de la transferencia de fondos, si en la cuenta de origen hay 1000 euros y transferimos 250, lo esperado cuando

finalice con éxito la transacción es que en la cuenta de origen el saldo será de 750 euros, habiéndose incrementado el saldo de la cuenta de destino en 250 euros

ISOLATED

Las transacciones están aisladas del resto, son privadas en definitiva, es decir, nada tiene constancia de la transacción hasta que esta ha sido terminada. Siguiendo con el ejemplo de la transferencia de fondos, si mientras se está realizando la transferencia de los 250 euros a la cuenta destino hacemos una consulta del saldo en la cuenta de origen, el saldo será de 1000 euros hasta que la transacción no se haya completado.

DURABLE

Las transacciones serán durables, es decir, deben ser resistentes a fallos. Si ocurre un error grave, la transacción mantendrá el estado.

Transacciones en System.ServiceModel

System.Transactions es el namespace necesario para poder soportar transacciones. Fue incluido con la versión 2.0 de la framework de .NET. Este namespace es el utilizado por Windows Communication Foundation para proveerse de todo lo necesario a la hora de construir servicios WCF y aplicaciones clientes que soporten transacciones

El Atributo [ServiceBehavior]

El atributo [ServiceBehavior] se aplica directamente sobre la clase que implementa el servicio. Las siguientes propiedades hacen referencia al comportamiento transaccional de un servicio:

- **TransactionAutoCompleteOnSessionClose** Especifica cuando las transacciones pendientes son completadas cuando la sesión actual es cerrada.
- **TransactionIsolationLevel** Determina el nivel de aislamiento de la transacción.
- **TransactionTimeout** Especifica el periodo que tiene la transacción para completarse

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession,
IncludeExceptionDetailInFaults = true,
TransactionIsolationLevel = IsolationLevel.RepeatableRead,
TransactionTimeout = "00:01:00")]
public class ProductServiceTx : IProductServiceTx
{
  public List<string> ListProducts()
  {
    // Read the configuration information for connecting to the
AdventureWorks database
    Database dbAdventureWorks;
    try
    {
       dbAdventureWorks =
DatabaseFactory.CreateDatabase("AdventureWorksConnection");
    }
    catch (Exception e)
    {
       ConfigFault cf = new ConfigFault();
       cf.ConfigOperation = "CreateDatabase";
       cf.ConfigReason =
        "Excepción leyendo la información de configuración de la
        base de
datos AdventureWorks database";
       cf.ConfigMessage = e.Message;
       throw new FaultException<ConfigFault>(cf);
    }
    //(..Resto Codigo fuente..)
  }

  }
```

El Atributo [OperationBehavior]

El atributo *[OperationBehavior]* se aplica directamente sobre un método de la clase que implementa el servicio. Las siguientes propiedades hacen referencia al comportamiento transaccional de la operación.

- **TransactionAutoComplete** Especifica que la transacción se auto completará si no ocurre ninguna excepción.
- **TransactionScopeRequired** Especifica cuando el método asociado requiere una transacción.

```
[OperationBehavior(TransactionScopeRequired=true,
              TransactionAutoComplete=false)]
public bool ChangePrice(string productNumber, decimal price)
{
  // Connect to the AdventureWorks database
  Database dbAdventureWorks;
  try
  {
    dbAdventureWorks =
DatabaseFactory.CreateDatabase("AdventureWorksConnection");
  }
  catch (Exception e)
  {
    throw new FaultException(
      "Excepción leyendo la información de configuración de la base de
datos AdventureWorks database: " +
    e.Message, new FaultCode("CreateDatabase"));
  }
}
```

El Atributo [TransactionFlow]

Mediante este atributo se especifica a qué nivel de respuesta ante una situación con transacciones en la operación donde se aplica. Los valores que admite este atributo vienen del enumerador *TransactionFlowOption*

- **Allowed** La operación se puede o no usar en una transacción
- **Mandatory**Solamente se podrá utilizar la operación en una transacción.
- **NotAllowed** Nunca se podrá usar la operación en una transacción.

A continuación se muestra un ejemplo de uso de este atributo

```
[ServiceContract(Namespace = "http://AdventureWorks.Service",
SessionMode = SessionMode.Required)]
public interface IProductServiceTx
{
   [FaultContract(typeof (ConfigFault))]
   [FaultContract(typeof (DatabaseFault))]
   [OperationContract]
   List<string> ListProducts();

   [TransactionFlow(TransactionFlowOption.Allowed)]
   [OperationContract]
   Product GetProduct(string productNumber);

   [TransactionFlow(TransactionFlowOption.Allowed)]
   [OperationContract]
   bool ChangePrice(string productNumber, decimal price);
}
```

WS-AtomicTransaction

WS-AtomicTransaction es un protocolo para transacciones interoperables. Permite habilitar las transacciones distribuidas usando mensajes de servicios web y coordinar de manera interoperable entre infraestructuras heterogéneas. WS-AtomicTransaction usa el protocolo Two-Phase para dirigir transacciones entre aplicaciones distribuidas

La implementación de WS-AtomicTransaction Communication Foundation (WCF) incluye un protocolo de servicio construido en el Microsoft Distributed Transaction Coordinator (MSDTC). Usando WS-AtomicTransaction, las aplicaciones WCF pueden pasar transacciones a otras aplicaciones, incluyendo servicios web interoperables construidos usando otras tecnologías. Cuando una transacción pasa entre la aplicación cliente y la aplicación servidor, el protocolo para transacciones usado es determinado por el binding que expone el servicio en el endpoint seleccionado por el cliente. La elección de un protocolo para transacciones dentro de un binding dado puede ser modificado programáticamente.

Para la elección de dicho protocolo, pueden intervenir dos factores separados

- El formato de la cabecera del mensaje usado para fluir transacciones

- desde el cliente al servidor
- El protocolo de red usado para ejecutar el protocolo Two-Phase entre el administrador de las transacciones del cliente y las transacciones del servidor

En caso de que tanto el cliente como el servicio estén escritos usando WCF, no es necesario usar WS-AtomicTransaction. En su lugar, podemos usar los parámetros por defecto de *NetTcpBinding* con el atributo *TransactionFlow* habilitado, el cual usará el protocolo OleTransactions

Transacciones en el cliente

En la parte cliente, la manera de habilitar el flujo de transacciones es mediante el binding. En el siguiente ejemplo se puede observar cómo queda la configuración donde se habilitan las transacciones

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.serviceModel>
      <bindings>
         <wsDualHttpBinding>
            <binding name="WSDualHttpBinding_IProductServiceTx"
closeTimeout="00:01:00" openTimeout="00:01:00"
receiveTimeout="00:10:00"
sendTimeout="00:01:00" bypassProxyOnLocal="false"
transactionFlow="true"
hostNameComparisonMode="StrongWildcard" maxBufferPoolSize="524288"
maxReceivedMessageSize="65536" messageEncoding="Text"
textEncoding="utf-8"
useDefaultWebProxy="true">
               <readerQuotas maxDepth="32"
maxStringContentLength="8192" maxArrayLength="16384"
maxBytesPerRead="4096" maxNameTableCharCount="16384" />
               <reliableSession ordered="true"
inactivityTimeout="00:10:00" />
               <security mode="Message">
                  <message clientCredentialType="Windows"
negotiateServiceCredential="true"
algorithmSuite="Default" />
               </security>
            </binding>
         </wsDualHttpBinding>
      </bindings>
      <client>(Aquí el ABC del Servicio)</client>
   </system.serviceModel>
</configuration>
```