

Die Geschichte von Session und Instanz, Teil 2

Eigene Instance-Context-Provider auf der Basis von Custom Service Behaviors implementieren

von Damir Dobric

Die Windows Communication Foundation (WCF) bietet eine umfangreiche und sehr modular aufgebaute Architektur. Die bestehenden Session-Funktionalitäten decken eine Vielzahl von Szenarien ab, die in der Praxis einfach angewendet werden können. In diesem zweiten Teil wird auf Basis von Custom Service Behaviors gezeigt, wie ein eigener Instance-Context-Provider implementiert werden kann. Auf Basis von einigen Beispielen werden mächtige, interessante und bisher wenig dokumentierte Eigenschaften hinter den Kulissen der WCF erläutert.

Wenn es um die Erweiterung von bestehenden WCF-Funktionalitäten geht, ist es wichtig zu wissen, dass es erweiterbare „Artefakte“ gibt. Dazu zählen z.B. *ContractBehavior*, *ServiceBehavior*, *OperationBehavior*, *CustomChannel*, *CustomBinding* usw. Bei der Implementierung einer Erweiterung gibt es eine goldene Regel, welche bei der Entscheidung hilft, ob es sich um eine Erweiterung einer *Behavior* oder eines *Channel* handelt. Wenn eine Funktionalität nur Auswirkungen auf der Service- oder der Client-Seite hat, sollte diese als *Custom Behavior* (*ContractBehavior*, *ServiceBehavior* oder *OperationBehavior*) implementiert werden. Ein Beispiel wäre z.B. eine XSLT-Transformation von der Message, die nur auf Service-Seite durchgeführt werden muss.

Eine Funktionalität soll dann als *Channel* implementiert werden, wenn sie pro-

tokollrelevante Auswirkungen auf beiden Seiten hat, z.B. Komprimieren auf der Source-Seite und Dekomprimieren auf der Destination-Seite. In der Praxis gibt es Fälle, die sowohl als *Channel* als auch *Behavior* implementiert werden können. Ein solcher Kandidat ist z.B. die Implementierung von Instanz-Verhalten. In diesem Artikel wird gezeigt, wie Custom-Instance-Context-Provider als ein *ServiceBehavior* implementiert werden kann, weil die entsprechende Channel-Variante wesentlich komplexer wäre.

ServiceBehavior erweitern

Mal angenommen, es gäbe einen Service *Hello*, der ein *Contract IHello* implementiert (Listing 1). Wie bereits Teil 1 dieses Artikels in der Ausgabe 1/2.07 des *dot.net magazin* gezeigt hat – durch das Attribut *ServiceBehavior* ist dieser Service in der Lage die *Session* zu verwalten. Wie wäre es, wenn auch Folgendes möglich wäre:

```
...
[DOTNETMagazineBehavior]
public class Hello : IHello
```

Unabhängig davon, was die semantische Bedeutung des Attributs *DOTNETMagazineBehavior* wäre, ist dieses eine Klasse, die die Schnittstelle *IServiceBehavior* implementieren muss – diese Schnittstelle ist in Listing 2 zu sehen.

Wenn der Service Host gestartet und der Service initialisiert wird, erzeugt *DispatcherBuilder* (eine interne Klasse der WCF) die Instanzen von deklarierten *ServiceBehavior*-Klassen, die diese Schnittstelle implementieren. Diese werden – lange bevor die Service-Instanz erzeugt wird – erstellt. Danach wird die *Validate*-Methode aufgerufen. Damit ist es möglich, die schon initialisierte Description und den Host zu validieren. Dann wird die Methode *AddBindingParameters* aufgerufen. Sie kann verwendet werden, wenn bestimmte Parameter für Binding eingetragen oder geändert werden sollen. Am Ende wird die Methode *ApplyDispatchBehavior* aufgerufen, um das *CustomBehavior* zu installieren.

Listing 1

```
public interface IHello
{
    [OperationContract]
    string DoSomething();
}

[ServiceBehavior(InstanceContextMode =
    InstanceContextMode.PerSession)]
public class Hello : IHello
{
    public string DoSomething()
    {
        ...
    }
}
```

kurz & bündig

Inhalt

WCF-Sessions für Fortgeschrittene

Zusammenfassung

Dieser Artikel beschreibt wie die vorhandenen Session-Provider durch eigene Implementierungen ersetzt werden können

Quellcode

C#



Quellcode auf CD

Listing 3 zeigt wie eine von *IServiceBehavior* abgeleitete Klasse *CustomInstanceAttribute* implementiert werden kann. Das Beispiel führt keine Validierung vom Host, Dispatcher und den Properties durch, denn es installiert lediglich die zwei Behaviors *CustomInstanceContextProvider* und *CustomInstanceProvider*. Um dieses Beispiel zum Laufen zu bringen, muss nur noch die gewünschte Service-Klasse mit dem Attribut versehen werden:

```
[CustomInstance]
public class Hello : IHello
```

Mit diesem Code wird die Klasse *CustomInstanceAttribute* vom Dispatcher instan-

ziiert und ihre Methoden aufgerufen, bevor die Instanz vom Service erzeugt wird.

Custom Instance Provider

Die eigentliche Logik passiert im Grunde in der *Behavior*-Klasse. Wie schon in Teil 1 des Artikels erwähnt, gibt es seit der July-CTP keine Möglichkeit mehr, eine Instanz zwischen mehreren Clients zu teilen. Das Teilen der *Service*-Instanz zwischen mehreren Clients ist interessant, unter anderen in Gaming-Szenarien, wenn mehrere Spieler eine Service-Instanz verwenden sollen. Da *Instance Sharing* ursprünglich (vor der July-CTP) durch *Referenz Sharing* implementiert war, ist dieses Feature zu Recht aus dem *Scope* gestrichen wor-

den. Die Referenz von einer Instanz im Context von Web Services ist zum Teil ein Widerspruch zum SOA-Gedanken. Eine Referenz (Pointer) ist eine physikalische Erscheinung, die in heterogenen Systemen schwer abzubilden ist. Ein gutes Beispiel sind die Welten von CORBA und COM.

Um eine Instanz doch zwischen mehreren Clients verteilen zu können, könnte man eine *Custom Service Behavior* verwenden. An dieser Stelle ist es wichtig zu betonen, dass man vom *Sharen* von Instanzen spricht und nicht vom *Sharen* von Sessions. In diesem Szenario wird jeder Client (bzw. Proxy-Instanz im Client) eine eigene *Reliable Session* und eine gemeinsame Service-Instanz besitzen. Alle Session-Modi, die im ersten Teil des Artikels beschrieben wurden, sind im Grunde als

Listing 2

```
public interface IServiceBehavior
{
    void AddBindingParameters(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase,
        Collection<ServiceEndpoint> endpoints, BindingParameterCollection bindingParameters);

    void ApplyDispatchBehavior(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase);
    void Validate(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase);
}
```

Listing 3

Eine Implementierung von Custom ServiceBehavior

```
class CustomInstanceAttribute : Attribute, IServiceBehavior
{
    public void AddBindingParameters(ServiceDescription description, ServiceHostBase serviceHostBase,
        System.Collections.ObjectModel.Collection<ServiceEndpoint> endpoints, BindingParameterCollection parameters)
    {
        // Perform some property operations
    }

    //Apply the custom IInstanceContextProvider to the EndpointDispatcher.DispatchRuntime
    public void ApplyDispatchBehavior(ServiceDescription description, ServiceHostBase serviceHostBase)
    {
        CustomInstanceContextProvider instanceContextProvider = new CustomInstanceContextProvider();
        CustomInstanceProvider instanceProvider = new CustomInstanceProvider();

        foreach (ChannelDispatcherBase dispatcherBase in serviceHostBase.ChannelDispatchers)
        {
            ChannelDispatcher dispatcher = dispatcherBase as ChannelDispatcher;
            foreach (EndpointDispatcher endpointDispatcher in dispatcher.Endpoints)
            {
                // Following lines install two same Custom Behaviors

                endpointDispatcher.DispatchRuntime.InstanceContextProvider = instanceContextProvider;
                endpointDispatcher.DispatchRuntime.InstanceProvider = instanceProvider;
            }
        }
    }

    public void Validate(ServiceDescription description, ServiceHostBase serviceHostBase)
    {
        // Perform the validation of the dispatcher and host.
    }
}
```

Listing 4

```
public interface IInstanceContextProvider
{
    InstanceContext GetExistingInstanceContext(Message message, IContextChannel channel);
    void InitializeInstanceContext(InstanceContext instanceContext, Message message,
        IContextChannel channel);
    bool IsIdle(InstanceContext instanceContext);
    void NotifyIdle(InstanceContextIdleCallback callback,
        InstanceContext instanceContext);
}
```

Listing 5

```
public override InstanceContext
    GetExistingInstanceContext(Message message,
        IContextChannel channel)
{
    ServiceChannel channel1 =
        base.GetServiceChannelFromProxy(channel);
    if (channel1 == null)
    {
        return null;
    }
    return channel1.InstanceContext;
}
```

Listing 6

```
public override void InitializeInstanceContext
    (InstanceContext instanceContext,
        Message message, IContextChannel
        channel)
{
    ServiceChannel channel1 =
        base.GetServiceChannelFromProxy(channel);
    if ((channel1 != null) && channel1.HasSession)
    {
        instanceContext.BindIncomingChannel(channel1);
    }
}
```

Instance Context Providers implementiert: *PerCallInstanceContextProvider*, *PerSessionInstanceContextProvider* und *SingletonInstanceContextProvider*. Diese Klassen sind *internal*-Klassen und somit nicht dokumentiert. Es ist interessant zu wissen, wie diese Klassen die Sessions abbilden, um die eigene Klasse besser implementieren zu können. Alle hier erwähnten Provider implementieren die Schnittstelle *IInstanceContextProvider* (Listing 4).

Die wichtigsten Methoden sind *GetExistingInstanceContext* und *InitializeInstanceContext*. Die erste wird vor der eigentlichen Service-Operation aufgerufen. Sollte diese Methode *NULL* zurückliefern, wird sofort vom Dispatcher *InitializeInstanceContext* aufgerufen, die den vom Dispatcher neu erstellten *InstanzContext* initialisieren soll. Das nächste Beispiel zeigt, was hinter den Kulissen im jeweiligen *build-in*-Provider passiert. Z.B. liefert *PerCallInstanceContextProvider* immer eine *NULL* zurück:

```
public override InstanceContext
    GetExistingInstanceContext(Message
        message, IContextChannel channel)
{
    return null;
}
```

Dies erzwingt den Aufruf der Methode *InitializeInstanceContext*, die nichts tut (keine Implementierung). Somit wird bei jedem Aufruf ein neuer *Channel* alokiert und eine neue *Service*-Instanz erzeugt. Ein *Channel* ist in diesem Fall hier für den Austausch von Messages zuständig. *PerSessionInstanceContextProvider* ist etwas interessanter. Er liefert den *Instance Context (ServiceChannel.InstanceContext)* vom *Channel* selbst (Listing 5). Dieser *Channel* wurde in *InitializeInstanceContext* an den Kontext angebunden (Listing 6).

Listing 7 und 8 zeigen im Detail wie das Erzeugen und Liefern vom *InstanceContext* implementiert werden kann. Wenn eine Operation vom *Client-Proxy* aufgerufen wird, wird zuerst auf der Service-Seite die Methode *GetExistingInstanceContext* aufgerufen. Sie überprüft, ob ein bestimmter *Header* in der Message vorhanden ist. Dies bedeutet, der Client muss dafür sorgen, dass dieser *Header* mitgeliefert wird. Wenn der *Header* vorhanden ist, wird überprüft, ob ein *InstanceContext* zu dem Key, der im *Header* mitgeschickt wird, schon existiert. Der Key im *Header* ist ein String, welcher z.B. eine Game-Session bezeichnet. Da der Kontext beim ersten Aufruf noch nicht existiert,

liefert diese Methode eine *NULL* zurück. Dadurch wird *InitializeInstanceContext* aufgerufen. Sie erhält eine vom Dispatcher erzeugten *Instance Context*, fügt den *Channel* in die Liste von *Incoming Channels* und in eine interne Liste *m_Sessions* hinzu. Damit hat man einen *Session Key* und einen *Instance Context* zugewiesen. Ein *Instance Context* hat eine Eigenschaft *Incoming Channels* – diese enthält die Liste von allen *Channels*, die sich diesen *Context* teilen. Eine genauere Beschreibung vom *Channel* würde mit Sicherheit den Rahmen des Artikels sprengen. Man kann sich aber in diesem Fall einen *Channel* als Abstraktion vom *Client-Proxy* vorstellen. Exemplarisch ist dies, wenn zwei verschiedene Proxies in einem oder zwei Clients eine Operation aufrufen, allokiert *Service Model* (WFC-Framework) einen *Channel* für jeden Proxy. Jeder *Channel* verwaltet intern eine Session (z.B. *Reliable Session*).

Beim nächsten Aufruf von beliebigen Operationen vom selben Proxy wird im zugehörigen *Instance Context* der *Channel* von diesem Proxy gefunden. Hier wird der gefundene *Channel* nicht noch einmal in die Liste von Incoming-Channels aufgenommen, sondern einfach sein übergeordneter *Session Context* zurückgegeben.

Listing 7

Das Liefern von Custom Shared Instance Context

```
public InstanceContext GetExistingInstanceContext(Message message, IContextChannel
    channel)
{
    //Check if the incoming request has the InstanceContext id it wants to connect with.
    if (message.Headers.FindHeader(CustomHeader.HeaderName,
        CustomHeader.HeaderNamespace) != -1)
    {
        string sessionId = message.Headers.GetHeader<string>(CustomHeader.HeaderName,
            CustomHeader.HeaderNamespace);
        if (sessionId != null && this.m_Sessions.ContainsKey(sessionId))
        {
            //Retrieve the InstanceContext from the map
            InstanceContext context = m_Sessions[sessionId];
            if (context != null)
            {
                if (!context.IncomingChannels.Contains(channel))
                {
                    context.IncomingChannels.Add(channel);
                }
                return m_Sessions[sessionId];
            }
        }
        return null;
    }
}
```

Listing 8

Erzeugen eines Custom Shared Instance Context

```
public void InitializeInstanceContext(InstanceContext instanceContext, Message
    message, IContextChannel channel)
{
    int headerIndex = message.Headers.FindHeader(CustomHeader.HeaderName,
        CustomHeader.HeaderNamespace);
    //AddressableInstanceContextExtension extension = null;
    if (headerIndex != -1)
    {
        instanceContext.IncomingChannels.Add(channel);
        string sessionKey = message.Headers.GetHeader<string>(headerIndex);
        m_Sessions[sessionKey] = instanceContext;
    }
    instanceContext.Closing += this.RemoveInstanceContext;
}
```

Die Client-Implementierung

Bei der Implementierung von *Instance Context Provider* wurde angenommen, dass der Client einen bestimmten *Session Key* im *Header* überträgt (Listing 3 und 7). Um einen *Custom Header* zu erzeugen, gibt es einige Möglichkeiten – Listing 9 zeigt wie dies normalerweise implementiert werden kann. Das Problem dabei ist, dass Sie sich selbst um das Erstellen von Messages kümmern müssen. Dies ist nicht gerade einfach und es wäre zu aufwändig auf diese Art den Client zu implementieren. Aus diesem Grund gibt es eine weitere Möglichkeit, die nicht gerade elegant, aber sehr effektiv ist. Grundsätzlich kann ein *Header* an alle ausgehenden Messages im *OperationContext* wie folgt angehängt werden:

```
OperationContext.Current.OutgoingMessageHeaders.  
    Add(sessionKeyHeader);
```

Wenn diese einfache Linie ausgeführt wird, sollen alle Messages mit dem gegebenen *sessionKey*-Header zum Service gesendet werden – das gilt für alle Messages

im *OperationContext.Current*. Bedauerlicherweise gibt es auf dem Client kein *OperationContext.Current*. Dieser Wert ist einfach eine *NULL*-Referenz. Um diesem Problem auszuweichen, schlägt Microsoft die Lösung aus Listing 10 vor. Wie deutlich wird, ist es nicht notwendig die Messages selbst zu bauen. Man erzeugt einfach einen Context im definierten Scope. Dies ist nicht gerade die schönste Technik, aber sie funktioniert.

Custom Instance Provider

Bisher wurde gezeigt, wie ein *Session Context* implementiert werden kann. Was in diesem Puzzle noch fehlt, ist wie eine Service-Instanz manipuliert werden kann. So könnten manche Services sehr komplexe Objekte *wrappen*, wie z.B. COM, Datenbank-Verbindungen usw. Wenn solche Fälle in der Praxis auftreten, möchte man mit Sicherheit selbst die Kontrolle über das Erzeugen einer Instanz übernehmen. Dafür bietet WCF die Schnittstelle *IInstanceProvider* an. Eine einfache Implementierung dieser Schnittstelle ist unter [1] zu finden, die Installation ist in Listing 1 zu sehen.

Auch WCF ist erweiterbar

Dieser Artikel hat gezeigt, wie bestehende WCF-Funktionalitäten durch neue Funktionalitäten ersetzt werden können. Von zahlreichen möglichen Erweiterungen wurde lediglich die Erweiterung von *Behaviors* erläutert. Diese Erweiterungen sind erforderlich, um viele fortgeschrittene Funktionalitäten zu implementieren oder die bestehenden besser zu verstehen. Ein gutes Beispiel war die Funktionsweise vom *Session Context* hinter den Kulissen. Eine solche modulare Architektur, wie sie WCF besitzt, hat man in der Industrie bislang nur selten gesehen. Abgesehen von einigen nicht perfekt gelungenen Konzepten wird sie einen wichtigen Meilenstein in der Entwicklung zukünftiger Anwendungen darstellen. Falls Sie mutig genug sind, in die kaum dokumentierte Welt der WCF einzutauchen, wird das mit Sicherheit der beste Weg sein, zu verstehen, wie die WCF wirklich funktioniert. Der Autor wünscht Ihnen dabei sehr viel Geduld und Spass.

.....
Damir Dobric ist Geschäftsführer der Firma DAE-NET GmbH aus Frankfurt a.M., deren Schwerpunkt die Entwicklung von Unternehmenslösungen ist.

Listing 9

```
Message msg = Message.CreateMessage(MessageVersion.Soap12WSAddressing10, "DoSomething");  
MessageHeader sessionKeyHeader2 = MessageHeader.CreateHeader("SessionKey", "http://daenet.eu/SessionSharing",  
    "Game1");  
msg.Headers.Add(sessionKeyHeader2);
```

Listing 10

```
ChannelFactory<HelloService.IHello> cnFactory = new ChannelFactory  
    <HelloService.IHello>("helloClient");  
HelloService.IHello chn1 = cnFactory.CreateChannel();  
using (new OperationContextScope((IContextChannel)chn1))  
{  
    MessageHeader sessionKeyHeader = MessageHeader.CreateHeader("SessionKey",  
        "http://daenet.eu/SessionSharing", "Game1");  
    OperationContext.Current.OutgoingMessageHeaders.Add(sessionKeyHeader);  
    chn1.DoSomething();  
}
```

Listing 11

Client erstellt Custom Header

```
static void Main()  
{  
    ChannelFactory<HelloService.IHello> cnFactory = new ChannelFactory  
        <HelloService.IHello>("helloClient");  
    HelloService.IHello chn1 = cnFactory.CreateChannel();  
    using (new OperationContextScope((IContextChannel)chn1))  
    {  
        MessageHeader sessionKeyHeader = MessageHeader.CreateHeader("SessionKey",  
            "http://daenet.eu/SessionSharing", "Game1");  
        OperationContext.Current.OutgoingMessageHeaders.Add(sessionKeyHeader);  
        chn1.DoSomething();  
    }  
}
```

● Links & Literatur

- [1] Ein Beispiel für die Implementierung von Custom Service Behaviors: developers.de/files/9/damir_dobric/entry986.aspx