**WCF .Настройка поведения сервисов**

Инфраструктура механизмов Windows Communication Foundation построена таким образом, что каждый ее элемент обладает определенным поведением. И разработчики, в процессе программирования сервиса, конфигурируют это поведение. В прошлой статье мы упоминали о настройках поведения конечных точек(endpoints), а сегодня рассмотрим параметры конфигурации поведения сервиса в целом – **Service Behavior.**

**Service Behavior**

Управление поведением в масштабах всего сервиса устанавливается при помощи свойств атрибута **ServiceBehavior.** Этим атрибутом может быть отмечен только  класс, который инкапсулирует в себе весь функционал сервиса, реализуя все контракты операций сервиса. Альтернативный способ – использование файла конфигурации. Мы рассмотрим первый способ. Итак, атрибут **ServiceBehavior**имеет следующие свойства:

1. *AddressFilterMode* – устанавливает тип фильтра, который используется механизмом маршрутизации WCF для направления входящих запросов на нужную конечную точку, путем применения данного фильтра для сравнения запрашиваемого адреса и реального адреса расположения точек. Свойство может принимать следующие значения из перечисления System.ServiceModel.AddressFilterMode: Exact, Prefix и Any;
2. *AutomaticSessionShutdown* – логическое свойство, которое определяет должен ли сервис автоматически завершать сеанс, когда клиент закрывает соединение с сервисом. По умолчанию установлено значение true и сеанс сервиса существует, пока открыто соединение с клиентом. Если установить свойство AutomaticSessionShutdown в false – то автоматическое прерывание отключается и можно контролировать время жизни сеанса в коде приложения сервиса;
3. *ConcurrencyMode* – устанавливает режим потоковой работы сервиса. В зависимости от значения данного свойства сервис может выполнять операции в однопоточном или многопоточном режимах. Свойство может принимать следующие значения из перечисления System.ServiceModel.ConcurrencyMode:
   * *Single* – значение по умолчанию, экземпляр сервиса работает в однопоточном режиме и не допускает обратных вызовов;
   * *Reentrant –*сервис работает в однопоточном режиме, но допускает повторные вызовы;
   * *Multiple –* сервис работает в многопоточном режиме, при этом механизмы WCF не утруждают себя синхронизацией, поэтому работая в этом режиме не забывайте, при необходимости, программно обеспечивать синхронизацию потоков.
4. *ConfigurationName* – задает имя соответствующей секции behavior в конфигурационном файле;
5. *IgnoreExtensionDataObject –* логическое свойство, которое определяет будет ли сервис обрабатывать данные, хранящиеся в свойстве ExtensionDataObject контракта данных. Данное свойство используется для обеспечения совместимости разных версий контрактов данных, в него записываются все неизвестные поля и свойства при десериализации сервисом входящего сообщения. По умолчанию установлено в true, рекомендуется устанавливать в значение false, если требуется строгое соответствие схемы при работе клиентов с сервисом(подробно об использовании интерфейса IExtensibleDataObject рассматривалось в статье[WCF – это просто: Продвинутое управление контрактами данных](http://dev.net.ua/blogs/evgeniymuzika/archive/2011/04/16/10773.aspx));
6. *IncludeExceptionDetailInFaults* – логическое свойство, которое определяет, должны ли ошибки возникающие на стороне сервиса преобразовываться к типу System.ServiceModel.FaultException и передаваться в ответном сообщении клиенту, который инициировал запрос. По умолчанию данная функция отключена, ее следует включать в целях отладки сервиса;
7. *InstanceContextMode* – определяет режим создания новых экземпляров сервиса. В конечном счете, сервис представляет собой нестатический класс, который реализует контракты операций сервиса. И, соответственно, перед тем как вызывать его методы, необходимо создать экземпляр данного класса. Данное свойство определяет каким образом механизм WCF будет создавать новые экземпляры класса-сервиса, свойство может принимать следующие значения из перечисления System.ServiceModel.InstanceContextMode:
   * *PerSession*– режим по умолчанию – для каждого сеанса(сессии) создается свой экземпляр сервиса(когда клиент открывает соединение с сервисом), который используется для всех вызовов во время данного сеанса и удаляется по его завершению;
   * *PerCall –* для каждого обращения клиента к сервису создается новый экземпляр класса-сервиса, который удаляется по завершению вызова;
   * *Single* – все время работы сервиса будет существовать только один экземпляр класса-сервиса. При первом обращении любого клиента создается новый экземпляр сервиса, который будет обрабатывать все последующие запросы и не удалиться никогда.
8. *MaxItemsInObjectGraph –*свойство определяет максимальное число полей во входящем сообщении, которые необходимо будет десериализовать;
9. *Name* и *Namespace* – эти два свойства определяют имя и пространство имен сервиса, которые будут отображаться в сгенерированном WSDL коде;
10. *ReleaseServiceInstanceOnTransactionComplete –* логическое свойство, которое определяет будет ли объект сервиса автоматически удаляться после завершения транзакции;
11. *TransactionAutoCompleteOnSessionClose*– логическое свойство, определяющее завершаются ли автоматически все транзакции при закрытии сессии(сеанса связи клиент-сервис);
12. *TransactionIsolationLevel* – свойство типа System.Transactions.IsolationLevel, которое определяет политику блокировок, применяемую к транзакциям, которые обрабатывает сервис(транзакции в WCF – достаточно обширная тема, поэтому я не буду описывать подробно атрибуты, связанные с транзакциями. В скором будущем я планирую написать об этом отдельную статью);
13. *TransactionTimeout* – свойство указывает период времени, в течении  которого транзакция должна завершиться. После его истечения транзакция будет автоматически прервана;
14. *UseSynchronizationContext*– логическое значение, определяющее должен ли сервис использовать текущее значение SynchronizationContext для выбора потока исполнения(значение по умолчанию – true);
15. *ValidateMustUnderstand – логическое свойство, определяющее необходимо ли строго контролировать, смог ли сервис понять все заголовки во входящем SOAP сообщении, которые предварительно были отмечены атрибутом MustUnderstand.*

Путем комбинации вышеперечисленных свойств можно создавать для сервиса правила поведения. Однако, будьте осторожны – здесь тоже есть свои подводные камни. Например, если установить оба свойства ConcurrencyMode и InstanceContextMode в значение Single – то в этом случае будет создан лишь один объект сервиса и он будет работать в однопоточном режиме. Такая конфигурация приведет к тому, что сервис будет способен одновременно обрабатывать лишь один запрос от клиента. В случае, когда сервис активно используется множеством клиентом, это приведет к созданию очереди запросов на стороне сервиса. Если при этом сервис еще выполняет долговременную, тяжелую операцию – то такая конфигурация сделает вашу систему полностью неработоспособной. Чтобы этого избежать, рекомендуется при установке свойства InstanceContextMode в значение Single, задавать в свойстве ConcurrencyMode многопоточный режим работы для сервиса(значение Multiple).

Рассмотрим пример, в котором устанавливаются вышеописанные свойства поведения для сервиса, при помощи атрибута ServiceBehavior:

[ServiceContract]

public interface ITestService

{

[OperationContract]

int CalculateSum(int a, int b);

}

[ServiceBehavior(

ConcurrencyMode = ConcurrencyMode.Single,

InstanceContextMode = InstanceContextMode.PerSession,

IncludeExceptionDetailInFaults = true,

ReleaseServiceInstanceOnTransactionComplete = true

)]

public class TestService : ITestService

{

public int CalculateSum(int a, int b)

{

return a + b;

}

}

В данном примере наш сервис будет работать в одном потоке, новый объект сервиса будет создаваться при открытии канала клиентом и будет удаляться по завершению каждой транзакции. Также, в целях отладки, включена опция отправки клиентам подробной информации об ошибках на сервисе.

**Расширение ServiceBehavior**

В примере выше мы использовали атрибут ServiceBehavior для задания параметров поведения сервиса. Класс, представляющий данный атрибут имеет следующее объявление:

public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior { ... }

Как видно, данный класс реализует интерфейс IServiceBehavior. Данный интерфейс предоставляет методы для внедрения расширений в параметры поведения служб WCF. Посмотрим на описание этого интерфейса:

public interface IServiceBehavior

{

void Validate(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase);

void AddBindingParameters(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase,

Collection<ServiceEndpoint> endpoints, BindingParameterCollection bindingParameters);

void ApplyDispatchBehavior(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase);

}

Методы интерфейса предназначены для следующих целей:

1. AddBindingParameters – предназначен для внедрения данных в элементы привязки текущего сервиса. Используется некоторыми механизмами безопасности для применения идентификационных данных клиентов. Данный метод отличается от любым других методов расширения поведения тем, что вызывается для каждой конечной точки(остальные методы вызываются только один раз);
2. ApplyDispatchBehavior – предоставляет возможности для вставки объектов настраиваемых расширений, например, обработчиков ошибок, перехватчиков параметров или сообщений, а также других объектов настраиваемых расширений;
3. Validate – обеспечивает проверку соответствия описания службы и основного приложения хостинга. Ярким примером использования является атрибут AspNetCompatibilityRequirementsAttribute, который также реализует интерфейс IServiceBehavior и использует метод Validate для проверки установленной политики режима совместимости с ASP.NET на сервисе и режима совместимости, предоставляемого основным приложением. Например, если на сервисе установлен атрибут AspNetCompatibilityRequirements в режим AspNetCompatibilityRequirementsMode.Required (в этом случае сервис требует обязательное наличие совместимости для хостинга в среде приложения ASP.NET), но такой режим совместимости не предоставляется хостингом(например, если сервис хоститься отдельным WCF проектом) – то при обращении к сервису возникнет исключение.

Теперь, когда мы познакомились с содержимым интерфейса IServiceBehavior, мы готовы написать свое расширение поведения сервиса. Для этого необходимо создать свой класс и реализовать интерфейс IServiceBehavior, внеся определенный код в необходимые методы:

public class MyServiceBehavior : IServiceBehavior

{

public void Validate(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase)

{

Type serviceType = serviceDescription.ServiceType;

MethodInfo[] methods = serviceType.GetMethods(BindingFlags.Instance | BindingFlags.Public)

.Where(m => m.DeclaringType == serviceType).ToArray();

if (methods.Length == 0)

throw new InvalidOperationException("Service does not have any public methods!");

foreach (MethodInfo method in methods)

{

foreach (ParameterInfo parameter in method.GetParameters())

{

if (parameter.ParameterType.IsByRef)

{

throw new InvalidOperationException("This service does not support methods with out or ref parameters!");

}

}

}

}

public void AddBindingParameters(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase,

Collection<ServiceEndpoint> endpoints, BindingParameterCollection bindingParameters)

{ }

public void ApplyDispatchBehavior(ServiceDescription serviceDescription, ServiceHostBase serviceHostBase)

{ }

}

В данном примере внесли код в метод Validate, при этом не дополняя методы AddBindingParameters и ApplyDispatchBehavior. Код в методе Validate выполняет две функции:

1. В массив methods  мы получаем список всех публичных методов текущего контракта операций. Если не было найдено ни одного метода – мы генерируем исключение с сообщением об этом;
2. Если методы найдены – мы перебираем в цикле все параметры каждого метода и если хотя бы один из них имеет модификатор ref или out – генерируется исключение(только для примера).

Теперь осталось только применить наш класс расширения поведения к сервису, сделать это можно в файле конфигурации:

<system.serviceModel>

<extensions>

<behaviorExtensions>

<add name="MyServiceBehavior" type="WCFMultiContract.MyServiceBehavior, HostApplication, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null"/>

</behaviorExtensions>

</extensions>

<behaviors>

<serviceBehaviors>

<behavior name="UsingMyServiceBehavior">

<MyServiceBehavior/>

</behavior>

</serviceBehaviors>

</behaviors>

<services>

<service name="WCFMultiContract.MyService" behaviorConfiguration="UsingMyServiceBehavior">

<endpoint />

...

</service>

</services>

</system.serviceModel>