

WCF

- Eine Einführung -

Yvette Teiken

Agenda

- Einführung
- WCF Hosting
- Das WCF ABC
- Endpoint
- Clients

19. Februar 2009



2 / 67

Was ist WCF?

- WCF: Windows Communication Foundation
- Kommunikations-Plattform für verteilte Systeme
- Zusammenführung von Kommunikationstechnologien
- Dient der Entwicklung von SOA-Architekturen
- Ab .NET 3.0



WCF Hosting

Hosting

- WCF können nicht alleine existieren
- Müssen im Windows Host Process gehostet werden
- Ein Service kann in verschiedenen in verschiedenen Host Processes gehostet werden
- Arten von Hosting:
 - IIS Hosting
 - WAS Hosting
 - Eigenes
 - „Dublin“ Hosting

IIS Hosting

- IIS managed Lifecycle des Host Prozesses
- Nur HTTP
- Alle Services auf gleichem Port
- Ähnliches Vorgehen wie beim *.asmx hosten
 - Virtuelles Verzeichnis
 - *.svc (Code behind identifizieren)
 - VS Unterstützung

```
<%@ Servicehost
  Language="C#"
  Debug = "true"
  CodeBehind = "~\App\MyService.cs"
  Service = "MyService"
%>
```

Self Hosting

- Entwickler verantwortlich für Lebenszyklus
- Provider kann jeder Windows Prozess sein
 - Forms-Anwendung
 - Console
 - ...
- Muss vor dem ersten Aufruf laufen
- Alles selber machen

WAS-Hosting

- Teil von IIS 7.0
- Gibt's nur in Vista und Server 2008
- *.svc wie bei IIS
- WAS kann alle Formate, Protokolle, Ports und Queues
- Erlaubt zudem
 - Pooling
 - Identity Management
 - Isolation
- Wenn möglich, verwenden

Dublin Hosting

- Dublin Applikationsserver vorgestellt auf PDC 08
- Vom Prinzip ähnlich wie BizTalk, soll ihn aber nicht ablösen
- Ähneln IIS bloß mit kompletter WCF Protokoll-Unterstützung
- Verwaltung und Überwachung integriert
- Soll kurz nach .NET 4.0 veröffentlicht werden
- Erweiterung von Vista und Server 2008

Hosting Demo

- Self Hosting inklusive WCF Hello World



Das WCF ABC

Das ABC

- Adresse
- Binding
- Contract



Adresse

Adressen

- Jeder Service besitzt eindeutige Adresse
- Adresse spezifiziert:
 - Ort des Services (Maschine, Webseite,...)
 - Transportprotokoll
- Unterstützte Transport Schemata in WCF
 - HTTP
 - TCP
 - Peer Network
 - IPC (Inter Process Communication via Named Pipes)
 - MSMQ

Adressen-Format

- Adress-Schemata:
 - [base adress]/[optional URI]
- Base address:
 - [transport]://[machine or domain][:optional port]
- Beispiele:
 - <http://localhost:8001>
 - <http://localhost:8001/MyService>
 - net.tcp://localhost:8002/MyService
 - net.pipe://localhost/MyPipe
 - net.msmq://localhost/MyService

Adressen I

- TCP-Adressen:
 - TCP Adressen verwenden net.tcp und beinhalten eine Portnummer:
 - net.tcp://localhost:8002/MyService
 - Kein Port, default Port 808
- HTTP-Adressen :
 - Nicht nur http sondern auch https
 - Nutzung wenn's übers Internet geht (Firewall)
 - Kein Port, default 80
- IPC-Adressen (Pipes)
 - Net.pipe
 - In WCF nur Aufrufe von eigener Maschine

Adressen II

- MSMQ-Adressen:
 - net.msmq
- Peer Network Adresse
 - net.p2p
 - Zusätzlich spezifizieren:
 - Peer Network name
 - Eindeutiger Pfad
 - Eindeutiger Port



Binding

Bindings

- **Bestimmt**
 - Art der Nachrichtenübertragung
 - Transportprotokoll
 - Security
 - Art der Kommunikation
- 9 Standard Bindings werden „mitgeliefert“
- Definition / Implementierung eigener Bindings möglich

Die Standard-Bindings

- **Basic Binding**
 - Eigentlich Legacy asmx
 - Alte Clients können mit neuen Services arbeiten
- **TCP Binding**
 - TCP für Inter-Maschinen-Kommunikation
 - Unterstützt
 - Reliability
 - Transaktionen
 - Security Features
 - Optimiert für WCF2WCF
 - → Client und Server müssen WCF unterstützen

Standard-Bindings II

- Peer Network Binding
 - Gleichberechtigung von Client und Server
 - Alle sind mit gleichem Grid verbunden und senden Nachrichten an Grid
- IPC Binding
 - Verwendung von Named Pipes
 - Gleiche Maschine
 - Sehr sicher, da keine Aufrufe von außen angenommen werden
- Web Service (WS) Binding
 - Transport: HTTP oder HTTPS
 - Internet-Kommunikation
 - Inter-Plattform Kommunikation

Standard-Bindings III

- Federated WS Binding
 - Erweitert WS Binding
 - Unterstützt Federated Security
- Federated Security
 - Unterscheidung von Service-Aufrufen und Authentifizierung
 - Ermöglicht Zusammenarbeit zwischen verschiedenen
 - Systemen
 - Netzwerken
 - Organisationen
 - Nicht weiter betrachtet
 - Quelle: <http://msdn.microsoft.com/en-us/library/aa355045.aspx>

Standard-Bindings IV

- Duplex WS Binding
 - Wie WS
 - Plus: bidirektionale Kommunikation
- MSMQ Binding
 - MSMQ Transport
 - Unterstützt disconnected queued calls
- MSMQ Integration Binding
 - Konvertiert WCF Aufrufe ins MSMQ Aufrufe
 - Entwickelt zur Legacy Unterstützung von MSMQ Clients

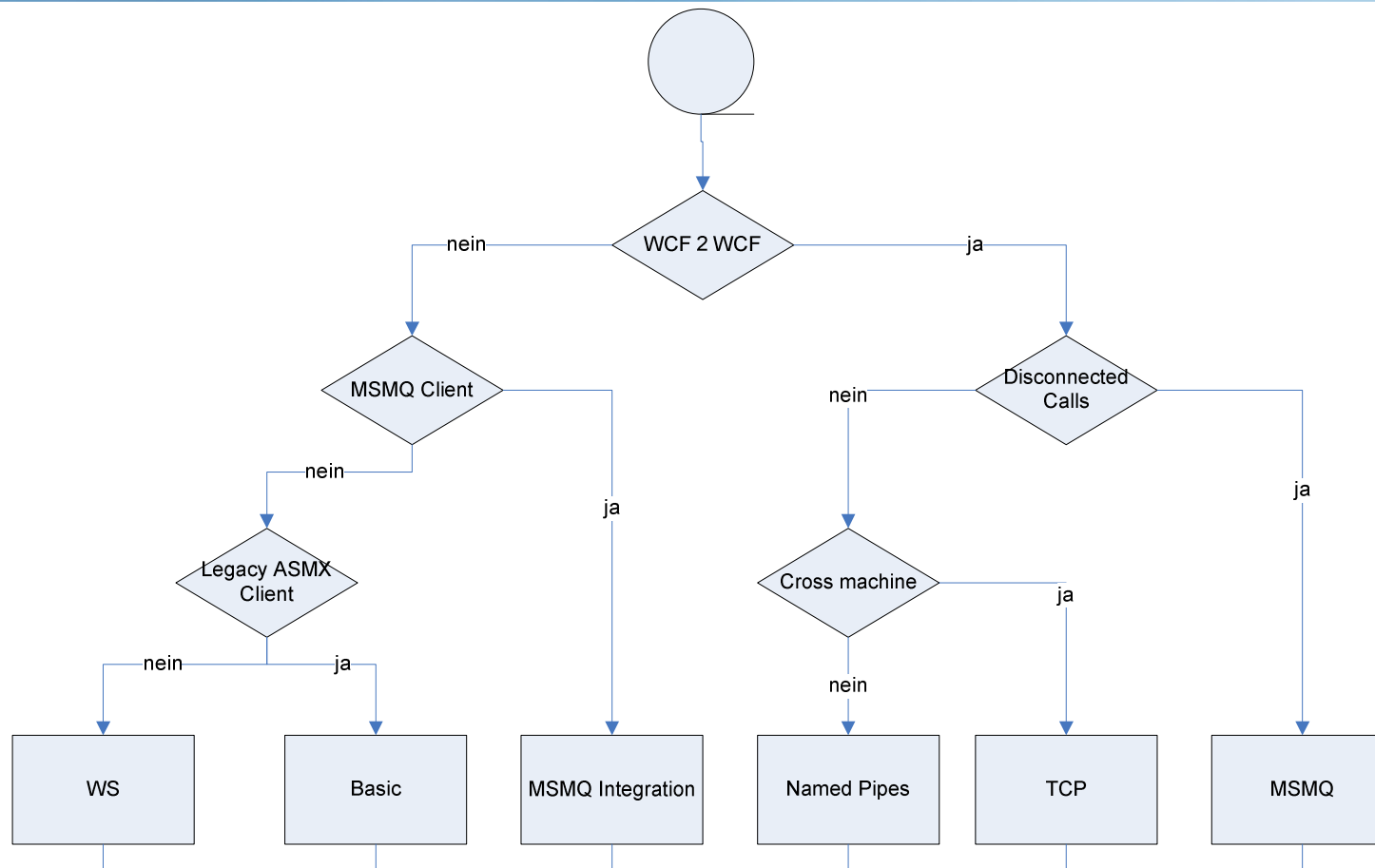
Bindings und Encoding

Name	Transport	Encoding	Interoperational
Basic HTTP	HTTP/HTTPS	Text	Ja
NetTcp	TCP	Binär	Nein
Net Peer	P2P	Binär	Nein
Named Pipe	IPC	Binär	Nein
WS HTTP	HTTP/HTTPS	Text	Ja
WS Federation	HTTP/HTTPS	Text	Ja
WS Dual HTTP	HTTP/HTTPS	Text	Ja
Net MSMQ	MSMQ	Binär	Nein
MSMQ Integration	MSMQ	Binär	Ja

Die Wahl des richtigen Bindings

- Ist wichtig
- Treffen von Entscheidungen
- Wichtigste Entscheidung: WCF ja oder nein
- Passendes wählen ggf. selber erweitern
- Erweiterung meist nur bei Framework- und nicht bei Anwendungsentwicklung
- [DEMO](#)

Die Wahl des richtigen Bindings





Contract

Verträge I

- Service Contract
 - Beschreibt, welche Operationen ein Client bei einem Service erwarten kann
- Data Contract
 - Beschreibt welchen Daten übertragen werden
 - Implizite Verträge für built-in types (int, string,...)
- Fault Contract
 - Beschreibt Fehlerarten, die beim Host auftreten können
 - Behandlung von Fehlern
 - Weitergabe an Clients

Verträge II

- Message Contract
 - Kommunikation zwischen Services direkt über Messages
 - Sinnvoll, wenn bestehendes Nachrichten Format unterstützt werden soll
 - Nicht wirklich angebracht für WCF, eher selten benutzen

Service Contracts I

```
using System.ServiceModel;

namespace HelloWorld1
{
    [ServiceContract]
    public interface IMyContract
    {
        [OperationContract]
        string MyMethod( string text );

        //nicht Teil des Contracts
        string myMethod( string
        something );
    }
}
```

```
public class MyService : IMyContract
{
    public string MyMethod(string text)
    {
        return "Meine Methode ist
        super";
    }

    public string myMethod(string
    something)
    {
        return "nicht über WCF";
    }
}
```

Service Contract II

- Attribute:
 - [ServiceContract] für Interfaces / Klassen
 - [OperationContract] für Methoden
- Definition:
 - Compiler Attribut → Technologie-neutrale Beschreibung
 - Typen-Sichtbarkeit unwichtig für WCF
 - Ohne Compiler Attribut nicht sichtbar für WCF
 - Service-Grenzen sind explizit
 - Attribute gelten nur für Methoden und nicht für Properties, Indexers, Events
 - Ein Contract kann als Parameter nur primitive Typen oder Data Contracts haben

Realisierung von Service Contracts

- Interfaces können implizit oder explizit implementiert werden
 - Interface nicht zwingend notwendig
- Eine Klasse kann mehrere Contracts realisieren
- Parametrisierte Konstruktor vermeiden, denn WCF ruft den default Konstruktor auf

```
[ServiceContract]
public class MyService
{
    [OperationContract]
    string myMethod()
    {
        //do something
    }
}
```

- Geht zwar, aber vermeiden (ReUse)

Überladen von Methoden I

- Überladen in OO-Programmiersprachen üblich

```
public interface Calculator
{
    int Add( int a, int b );
    double Add( double a, double b );
}
```

- In WCF:

```
[ServiceContract]
public interface Calculator
{
    [OperationContract]
    int Add( int a, int b );
    [OperationContract]
    double Add( double a, double b );
}
```

- Führt zu
InvalidOperationException

Überladen von Methoden II

- Abhilfe schafft das Name Attribut des OperationContracts

```
[ServiceContract]
public interface Calculator
{
    [OperationContract(Name = "AddInt")]
    int Add( int a, int b );
    [OperationContract(Name="AddDouble")]
    double Add( double a, double b );
}
```

- Generierter Proxy ersetze Namen mit Contract Namen
- Oder Client selber umschreiben (überladen erlauben)



Data Contracts

Serialisierung I

- Motivation
 - WCF ermöglicht, Services als CLR Interfaces zu nutzen
 - Auch Definition von Services als CLR Interfaces
 - CLR Interfaces .NET spezifisch
 - → Eigene Technologie zur CLR-Abstraktion
 - Z.B. XML
 - In WCF: Data Contracts
- Was ist Serialisierung?
 - Umwandlung von Daten in externe Repräsentation
 - Meist über Technologie-Grenzen
 - Persistenz

Data Contracts mittels Serialisierung

- WCF hat speziellen Serialisierungs-Mechanismus
 - Typ-Informationen nicht übertragen
 - Unterstützt nicht IFormator
- WCF serialisiert selbständig, solange Typen serialisierbar sind
- Primitive Typen werden automatisch serialisiert
- Rest, spezielle Attribute:
 - [DataContract] für Klassen, Enums und Structs
 - Führt nicht dazu, dass Member von WCF serialisiert werden
 - [DataMember] für Felder und Properties
 - Sind im Namespace System.Runtime.Serialization enthalten

Data Contract Beispiel

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

    [DataMember]
    string Address;

    double Turnover;
}
```

```
[ServiceContract]
public class ContactManager
{
    [OperationContract]
    public int GetNumberOfCustomers(){
        return 0;}

    [OperationContract]
    Customer GetCustomerByName(string aName)
    { ... }
}
```

Data Set und Tabellen

.NET Framework Class Library

DataSet Class

Represents an in-memory cache of data.

Namespace: [System.Data](#)

Assembly: [System.Data](#) (in [System.Data.dll](#))

Syntax

Visual Basic (Declaration)

```
<SerializableAttribute> _  
Public Class DataSet _  
    Inherits MarshalByValueComponent _  
    Implements IListSource, IXmlSerializable, ISupportInitializeNotification, ISupportInitialize, _  
    ISerializable
```

- →

```
[DataContract]  
struct Contact  
{  
    [DataMember]  
    public ContactType type;  
    [DataMember]  
    string FirstName;  
    [DataMember]  
    private DataSet buys;  
}
```

- Nur DataTable Contract, keine spezifischen Informationen zum Schema (kein ADO.NET)
- Abhilfe: einfach im generierten Client ADO.NET Referenz setzen

Data Set und Tabellen

- Besser: Typisierte DataSets und DataTable
- Erzeugung mit Hilfe von Visual Studio
 - Typisierte Zeilen
 - Typisierte Spalten

Beispiel Vertrag mit typisierten DataSets

```
public class MyMovieService : IMovie
{
    public MyDataSet GetMovies()
    public void AddMovies(MyDataSet.MovieDataTable aMovieTable)
}

[ServiceContract]
public interface IMovie
{
    [OperationContract]
    MyDataSet GetMovies();

    [OperationContract]
    void AddMovies( MyDataSet.MovieDataTable aMovieTable );
}
```

```

namespace Kannst_Du_löschen.ServiceReference2 {

[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
[System.ServiceModel.ServiceContractAttribute(ConfigurationName="ServiceReference2.IMovie")]
public interface IMovie {

[System.ServiceModel.OperationContractAttribute(Action="http://tempuri.org/IMovie/GetMovies", ReplyAction="http://tempuri.org/IMovie/GetMovi
System.Data.DataSet GetMovies();
}

[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
public interface IMovieChannel : Kannst_Du_löschen.ServiceReference2.IMovie, System.ServiceModel.IClientChannel {
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
public partial class MovieClient : System.ServiceModel.ClientBase<Kannst_Du_löschen.ServiceReference2.IMovie>, Kannst_Du_löschen.ServiceReferenc

public MovieClient() {
}

public MovieClient(string endpointConfigurationName) :
base(endpointConfigurationName) {
}

public MovieClient(string endpointConfigurationName, string remoteAddress) :
base(endpointConfigurationName, remoteAddress) {
}

public MovieClient(string endpointConfigurationName, System.ServiceModel.EndpointAddress remoteAddress) :
base(endpointConfigurationName, remoteAddress) {
}

public MovieClient(System.ServiceModel.Channels.Binding binding, System.ServiceModel.EndpointAddress remoteAddress) :
base(binding, remoteAddress) {
}

public System.Data.DataSet GetMovies() {
return base.Channel.GetMovies();
}
}

```

Arrays statt DataSet und DataTable I

- DataTable und DataSet dank WCF und Visual Studio total einfach
- Aber
 - ADO.NET ist .NET-Technologie
 - Erzeugtes Serialisierungsschema sehr komplex
 - Evt. Inperformant
 - Gibt u.U. eigene Interna raus
 - Probleme bei Änderungen der Datenbank
 - Andere Plattformen unpraktikabel
- Besser Daten nur auf Datenebene beschreiben
- Neutrale Datenstruktur wie das Array

Arrays statt DataSet und DataTable II

- Beispiel

```
[DataContract]
public struct Movie
{
    [DataMember]
    public int ID;

    [DataMember]
    public string title;

    [DataMember]
    public int rank;
}
```

```
public void AddMovies(Movie[] newMovies)
{
    MovieTableAdapter dataSetTableAdapters =
        new MovieTableAdapter();

    for (int i = 0; i < newMovies.Length; i++)
    {
        dataSetTableAdapters.Insert
            (newMovies[i].ID,newMovies[i].rank ,...)
    }
}
```

Generics

- Generics sind .NET Spezifika
- Nutzung von Generics würde Service-Gedanken verletzen
- → Verwendung von Bounded Generics
- Beispiel:
 - Wird bounded exportiert

```
[System.Runtime.Serialization.DataMemberAttribute()]
public int mMyMember {
    get {
        return this.mMyMemberField;
    }
    set {
        if ((this.mMyMemberField.Equals(value) != true)) {
            this.mMyMemberField = value;
            this.RaisePropertyChanged("mMyMember");
        }
    }
}
```

```
[DataContract]
public class MyClass<T>
{
    [DataMember]
    public T mMyMember;
}

[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod( MyClass< int > obj );
}

public class Bla : IMyContract
{
    public void MyMethod(MyClass<int> obj)
    {
    }
}
```

Collections I

- Definition: Jeder Typ (Klasse), die IEnumerable oder IEnumerable<T> implementieren
- Arrays, List und Stack sind Built-In Collections
- Collections sind .NET spezifisch
- Deswegen kein WCF-Unterstützung
- Collections werden als Arrays exportiert
- Beispiel → nächste Folien

Collections II

```
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    IEnumerable< Contact > GetContact();
}

public class ContactManager : IContactManager
{
    public IEnumerable<Contact> GetContact()
    {
        List<Contact> myConacts = new List< Contact >();
        return myConacts;
    }
}
```

Collections III

- Wird exportiert als

```
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    Contact[] GetContacts();
}
```

- Wie geht man mit Arrays um?
- Fachbegriff: automatisches Marshalling

```
Contact[] contact = new Contact[123];
List< Contact > aList = contact.ToList();
```

Eigene Collections I

- Auch eigene Collection unterstützen
automatisches Marshalling, nicht nur Build-In
Collections
- Dictionaries: Gehen auch, obwohl gegen das
Service Paradigma
- Sog. [CollectionDataContract]



Fault Contract (Exceptions)

Exception Handling

- Finden von Fehlern manchmal sinnvoll ☺
- Bekanntes Konzept: Exception Handling
 - Erneut CLR Konzept
 - Nicht einfach übertragbar
 - Client bekommt Fault Exception
- Stufen von Exception Handling innerhalb von WCF
 1. Error Codes
 2. Fault Contract
 3. Fault Contract mittels Exceptions

Beispiel Error Codes

```
//normaler Code Server
  throw new ArgumentException();
} catch (ArgumentException e) {
  throw new FaultException(
    new FaultReason(e.Message),
    new FaultCode("Argument")
  ); }
```

```
catch (FaultException e)
{
  if (e.Code.Name == "Argument") {
    Console.WriteLine("Behandle Argument Exception: " + e.Reason);
    Console.Read();
  }
} catch (Exception e) { //...
}
```

Beispiel Fault Contract mit Exception I

```
try
{
    //normaler Code
    throw new ArgumentException();
}
catch (ArgumentException e)
{
    throw new FaultException<ArgumentException>(e);
}
```

```
[ServiceContract]
public interface IService
{
    [OperationContract]
    [FaultContract(typeof(ArgumentException))]
    void ThrowException(string typeName);
}
```

Beispiel Fault Contract mit Exception II

```
catch (FaultException<ArgumentException> e)
{
    Console.WriteLine("Bekomme Fehler: "+e.Message);
    Console.Read();
}
```

WCF Extensions

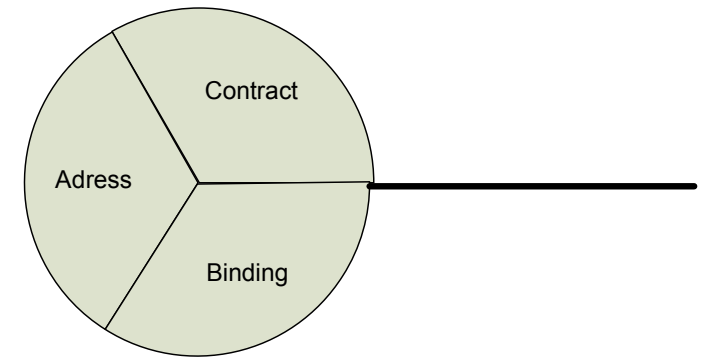
- Nicht das was man möchte
- Besser nicht merken, dass WCF
- → WCF Extensions



Endpoint

Endpoints

- Fusion des ABC
- Jeder Service verfügt über einen Endpoint
- Ein Endpoint muss über alle drei Elemente verfügen
- Endpoint = Interface des Services
- Endpoint immer extern zum Service
- Arten der Konfiguration:
 - Programmatisch
 - Administrativ



Programmatische Endpoint Konfiguration

- Konfiguration kann innerhalb des Programms erfolgen

```
ServiceHost myHost = new ServiceHost( typeof ( HelloWorldImplementierung ) );  
  
Binding wsBinding = new WSHttpBinding();  
Binding tcpBinding = new NetTcpBinding();  
  
myHost.AddServiceEndpoint( typeof ( IHelloWorld ), wsBinding,  
    "http://localhost:8080/HelloService" );  
myHost.AddServiceEndpoint( typeof ( IHelloWorld ), tcpBinding,  
    "net.tcp://localhost:8081/HelloService" );  
  
host.Open();
```

Administrative Endpoint Konfiguration I

- Endpoint ist extern zum Service → Konfiguration auch auslagern
- In app.config

```
<system.serviceModel>
  <services>
    <service name="SelfHostinh.HelloWorldImplementierung">
      <endpoint
        address="http://localhost:8080/HelloService"
        binding="wsHttpBinding"
        contract="„HostingDemo.IHelloWorld">
      </endpoint>
    </service>
  </services>
</system.serviceModel>
```

Administrative Endpoint Konfiguration II

- Definition multipler Endpunkte
- Mit gleicher Basisadresse möglich
- URI muss unterschiedlich sein
- Administrative Konfiguration ist vorzuziehen, da flexibler

```
<services>
  <service name=„Hosting.HelloWorldImplementierung“>
    <endpoint
      address="http://localhost:8080/HelloService"
      binding="wsHttpBinding"
      contract="SelfHosting.IHelloWorld">
    </endpoint>
    <endpoint
      address="net.tcp://localhost:8081/HelloService"
      binding="netTcpBinding"
      contract="SelfHosting.IHelloWorld"
    ></endpoint>
  </service>
</services>
```

Austausch von Metadaten

- Metadaten notwendig zur Kommunikation (Beschreibung des Services, ähnlich zu WSDL)
- Ermöglicht Verwendung über Visual Studio
- Veröffentlichen auf zwei Arten:
 - HTTP-Get
 - Eigener Endpoint
- HTTP-Get Variante:
 - Fügt automatisch Metadaten hinzu
 - Explizit Verhalten hinzufügen
 - Realisierung: Programmtisch oder administrativ

Metadaten Konfiguration Administrativ

```
<services>
  <service name="SelfHostinh.HelloWorldImplementierung" behaviorConfiguration="MEXGET">
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost:8080"/>
      </baseAddresses>
    </host>
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name="MEXGET">
      <serviceMetadata httpGetEnabled="true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Metadaten Konfiguration Programmatisch

```
ServiceMetadataBehavior metaData;  
metaData = host.Description.Behaviors.Find<ServiceMetadataBehavior>();  
if (metaData == null)  
{  
    metaData = new ServiceMetadataBehavior();  
    metaData.HttpGetEnabled = true;  
    metaData.HttpGetUrl = new Uri("http://localhost:8080/MyService/MEX");  
    host.Description.Behaviors.Add(metaData);  
}
```



Clientseitige Programmierung

Proxy I

- Clients müssen Verträge importieren
- Allgemeines Vorgehen in WCF mittels Proxy-Generierung
- Proxy abstrahiert alles vom Services, im Speziellen:
 - Ort, an dem der läuft
 - Implementierung
 - Implementierungstechnologie
 - Runtime Plattform
 - Transport
- Erzeugung eines Proxy
 - Visual Studio
 - svcUtil
- Bearbeiten der Konfiguration
 - Svcconfigeditor

Proxy II

- Konfiguration auch programmatisch möglich
- Aspekte administrativ:
 - Kein Rebuild
 - Kein Redeployment
 - Nicht Typsicher
 - Fehlererkennung zur Laufzeit
- Aspekte programmatisch:
 - Flexibel bei dynamischer Konfiguration
 - Statische Beschreibung (keine Änderung)



Vielen Dank für die
Aufmerksamkeit!