L Überblick über die 11. Übung

- .NET-Anwendungen bauen und binden
  - Assemblies
  - Anwendungskonfigurationsdatei
  - Signieren, Versionierung, Position der Assemblies
- .NET Remoting
  - Objekterzeugung und Fernaufruf
    - Aktivierung durch den Server
    - Aktivierung durch den Client
  - Marshalling
  - Objektserialisierung
  - Channels und Formatter

L.1 .NET-Anwendungen bauen und binden

1 Assemblies

- Anwendung oder Bibliothek
  z.B. unter Windows: ausführbare Datei!?
- entspricht einer Sammlung von Klassen, Strukturen und Typen
- Assembly enthält plattformunabhängigen Code.
  - *Common Intermediate Language* (CIL) oder *MSIL*
  - wird von JIT-Compiler übersetzt
  - → *portable execution file* (PE)

1 Assemblies

- private und gemeinsam benutzte Assemblies
- Metadaten werden im *Manifest* dem Assembly beigefügt.
  → Assembly ist selbstbeschreibend.
- Manifest enthält Metadaten der enthaltenen Typen und die folgenden Informationen:
  - Version
  - Umgebungsinformationen (Sprache)
  - referenzierte Typen
  - Abhängigkeiten
- *multifile Assembly*: besteht aus mehreren Dateien

2 Manifest

- enthält Metainformationen und Referenzen auf andere Assemblies
- Manifest eines Assemblies betrachten:

```
Mono> monodis Customer.exe
Win>  ildasm Customer.exe
```

- Ausgabe:

```
.assembly extern mscorlib
{
    .ver 1:0:3300:0
}
.assembly extern Bank
{
    .ver 1:0:0:1
}
.assembly 'Customer'
{
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.namespace Customer
{ ...
```

2 Die Anwendungskonfigurationsdatei

- Ort: Verzeichnis der Anwendung
- Name: <Name der Anwendung>.config
  - Beispiel: **`Customer.exe.config`**
- Format: XML
- Aufgabe: enthält z.B. Informationen für den Bindevorgang
- Beispiel: **`Customer.exe.config`**

```
<configuration>
   <runtime>
      <assemblyBinding
         xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="libs;libs/morelibs" />
      </assemblyBinding>
   </runtime>
</configuration>
```

  - Auflösen der Abhängigkeiten:
    Suchen der referenzierten Assemblies (*probing*)

3 Gemeinsam genutzte Assemblies

- Identifikation: *starker Name* bestehend aus:
  - einfacher Textname (*friendly name*)
  - Umgebungsinformationen (*culture*)
  - Versionsnummer
  - öffentlicher Schlüssel
  - digitale Signatur
- Umgebungsinformation: “English”, “German”, meist aber “neutral”
  Nur bei *satellite assemblies* (kein Code, nur Ressourcen) ist die Sprache relevant.
- Vorteile eines starken Namens
  - eindeutiger Name
  - mehrere Versionen möglich
  - Integritätsprüfung möglich
  - Herkunft überprüfbar

3 Gemeinsam genutzte Assemblies

- Position: *Global Assembly Cache* (GAC)
  z.B.: **`c:/windows/assembly`**, **`/usr/lib/mono/gac`**
- Assembly in GAC installieren:
  - mit dem Windows Explorer (drag & drop)
  - Mit **`gacutil`**

```
Win>  gacutil /i Bank.dll
Mono> gacutil -i Bank.dll
```


4 Signieren

- Schlüsselpaar erzeugen mittels *Strong Name Utility*

```
Win> sn -k MyKey.snk
```

- in einem beliebigen Source-File (z. B. in AssemblyInfo.cs) die Schlüssel-Datei mittels **`AssemblyKeyFile`**-Attribut angeben:

```
using System.Reflection;
[assembly: AssemblyKeyFile(@"MyKey.snk")]
```

```
Win> csc /target:library Bank.cs
```

- *oder* mittels Assemblylinker hinzufügen:

```
Win> csc /target:module Bank.cs
...
Win> al /out:Bank.dll /keyfile:MyKey.snk Bank.netmodule
```

- Zum Signieren wird aus dem Inhalt der Assembly ein Hash-Wert berechnet und dieser mit dem privaten Schlüssel verschlüsselt.
  - Zur Überprüfung kann die Signatur mit dem öffentlichen Schlüssel entschlüsselt, der Hash-Wert neu berechnet und beide verglichen werden.

4 Signieren

- Manifest der Bibliothek vor dem Signieren:

```
.assembly Bank
{
    .hash algorithm 0x00008004
    .ver 1:0:0:1
}
```

- ... und danach:

```
.assembly Bank
{
.publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00
              00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00
              6F 9A D3 D0 71 02 DA C8 AB B1 56 30 E7 30 2D 72
              3F 89 01 86 9A BB 7A 14 9D D6 C2 E4 17 D4 73 7F
              DD D4 0F C1 7A 0D 5F 3A 7E 5A 08 B3 B5 7F BD A6
              BA AD BC 80 8F 5F 73 E7 B0 4F 12 84 D5 CD 72 3B
              E2 93 6F 02 FE 23 C1 31 2E FE 40 DA 77 95 23 A9
              3B 37 1E F8 2B 0F 45 A8 1C 87 CF B5 84 24 12 5E
              60 6B 97 9D C1 73 FC 8F 3A 41 12 C0 89 87 CA E3
              EF F6 94 D6 E0 37 64 21 BA 06 E5 3B 05 6D 82 C2)
.hash algorithm 0x00008004
.ver 1:0:0:1
}
```


4 Signieren

- Manifest einer Anwendung...
  - ...mit einer Referenz auf eine private Bibliothek:

```
.assembly extern Bank
{
    .ver 1:0:0:1
}
```

  - ... mit einer Referenz auf eine signierte Bibliothek:

```
.assembly extern Bank
{
    .publickeytoken = (B4 26 2B 47 22 84 93 46 )
    .ver 1:0:0:1
}
```

- Der *public key token* ist ein Hash-Wert aus dem öffentlichen Schlüssel der referenzierten Assembly.

5 Versionierung

- Versionsnummer besteht aus 4 Teilnummern durch Punkt (.) oder Doppelpunkt (:) getrennt:
  <major version>.<minor version>.<buildnumber>.<revision>
- wird durch ein Attribut festgelegt

```
[assembly: AssemblyVersion("1.0.0.1")]
```

- kann auch nur teilweise festgelegt werden (automatische Erzeugung):

```
[assembly: AssemblyVersion("1.0.*")]
```

- Keine Versionskontrolle bei privaten Assemblies
- Versionskontrolle beim Binden von existierenden Anwendungen beeinflussen durch:
  - die Anwendungskonfigurationsdatei
  - die systemweite Konfiguration (<*install path*>\Config\Machine.config)
  - eine Richtliniendatei des Herstellers (*publisher policy*, aus XML-Konfiguration erzeugte Assembly)

6 Position der Assemblies

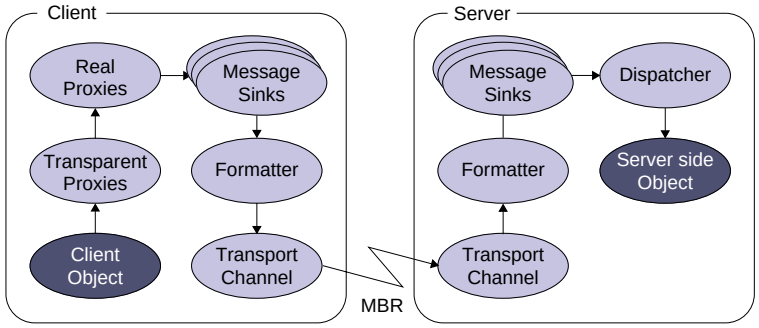
- GAC
- Anwendungsverzeichnis oder ein Unterverzeichnis
  - `<probing privatePath="libs" />`
- beliebiges Verzeichnis oder Web-Seite:

```
[...]
    <dependentAssembly>
        <assemblyIdentity name="Bank"
                          publicKeyToken="b4262b4722849346" />
        <bindingRedirect oldVersion="1.0.0.1-1.0.0.2"
                         newVersion="1.0.0.3" />
        <codeBase version="1.0.0.3"
                  href="file://c:/Bank.dll"/>
    </dependentAssembly>
[...]
```

  - Private Assemblies müssen im Anwendungsverzeichnis oder in einem Unterverzeichnis liegen (<codeBase> dann ohne Versionsnummer).

L.2 .NET Remoting

1 Überblick



- Wann wird das Serverobjekt erzeugt?
- Steht es einem oder mehreren Clients zur Verfügung?

1 Überblick

- Proxies
  - intelligente Proxies: Vorverarbeitung, Caching usw.
- Message Sinks: erlauben es, Nachrichten vor dem Versenden bzw. vor der Zustellung am Server zu bearbeiten
  - Kompression, Verschlüsselung usw.
- Formatter
  - Wie werden Objekte serialisiert (dargestellt)?
- Channel
  - Transportschicht
- Öffentliche und private Serverobjekte
- Lebenszeit wird durch *Lease-based Distributed Garbage Collection* vorgegeben.

2 Fernerzeugung öffentlicher und privater Objekte

- Fernerzeugung eines öffentlichen Objekts (server-activated / well-known objects)
  - Besitzen einen eindeutigen, bekannten Namen (URI): *well-known object*, diese Referenz auf das Serverobjekt ist öffentlich
  - Objekt kann über mehrere Aufrufe erhalten bleiben (“Singleton”) oder nach einem Aufruf automatisch gelöscht werden (“SingleCall”).
  - Dynamische Instantiierung des Objekts mittels Standardkonstruktor
  - “Veröffentlichung” eines bereits erzeugten Objekts möglich
- Fernerzeugung eines privaten Objekts (client-activated / private objects)
  - Für jeden Client wird dynamisch eine Instanz mit eigener, privater Kennung erzeugt.
  - Objekt kann über mehrere Aufrufe erhalten bleiben (stateful).
  - können vom Client mit beliebigem Konstruktor erstellt werden

2 Serveraktivierte Objekte

- Aktivierungsmodus bestimmt die Lebenszeit:
  - *SingleCall*: wird bei jedem Aufruf neu erzeugt und anschließend wieder zerstört (stateless)
  - *Singleton*: bleibt nach einem Aufruf am Leben (statefull)
- Singleton
  - Lebenszeit (für beide Varianten) durch Lease bestimmt
  - Anforderungen werden in separaten Threads ausgeführt (CLR stellt ThreadPool zur effizienten Ausführung bereit).
  - bei modifizierenden Zugriffen ist evtl. Synchronisation notwendig! (**`System.Threading`** oder Schlüsselwort **`lock`**)
- SingleCall
  - Für jede Anforderung wird ein eigenes Objekt erzeugt (Overhead!).
  - keine Zustandsübermittlung zwischen Clients/Aufrufen

2 Serveraktivierte Objekte - Server

- Als Beispiel dient die Klasse **`RemoteBank`** (s. **`/proj/i4mw/pub/net`**).
- **`System.Runtime.Remoting.dll`** wird benötigt
  ➔ eine Referenz auf das entsprechende Assembly hinzufügen

```
mcs -r:System.Runtime.Remoting.dll ...
```

- Klasse von **`MarshalByRefObject`** ableiten und implementieren

```
namespace BankLibrary {
    public class RemoteBank : MarshalByRefObject { ... }
}
```

- Kanal auswählen und mittels **`ChannelService.RegisterChannel`** registrieren
- Die Klasse registrieren / aktivieren (s. nächste Folien)
- Auf Anfragen von Clients warten

2 Serveraktivierte Objekte - Server

- Beispiel (es wird noch kein Objekt erstellt!)

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace BankServer {

  class ServerMain {
    public static void Main (String[] args) {
      HttpChannel channel = new HttpChannel(4711);
      ChannelServices.RegisterChannel(channel);

      RemotingConfiguration.RegisterWellKnownServiceType(
                          typeof(BankLibrary.RemoteBank),
                          "TestBank.soap",
                          WellKnownObjectMode.Singleton);
      // analog: WellKnownObjectMode.SingleCall

      Console.WriteLine("Server started.");
      Console.ReadLine();
    }
  }
}
```


2 Serveraktivierte Objekte - Client

- Der Client benötigt folgende Informationen:
  - Name des Servers
  - Typ des verwendeten Kanals
  - Port-Nummer, an der der Server wartet
  - Die URI des entfernten Objekts
- **`System.Runtime.Remoting.dll`** wird benötigt
  ➔ eine Referenz auf das entsprechende Assembly hinzufügen
- Um einen Proxy erstellen zu können, werden die Metadaten (z.B. Methodensignaturen) benötigt: im Beispiel die Assembly BankLibrary
- Kanal vom selben Typ wie der des Servers registrieren
- mittels **`Activator.GetObject(...)`** einen Proxy erzeugen
- den Proxy in den entsprechenden Typ umwandeln und verwenden

2 Serveraktivierte Objekte - Client

- Client muss Typ des entfernten Objekts kennen.
  - einfachster Fall: Client hat Zugang zur Implementierung / Assembly (**`RemoteBank.dll`**).
  - Stand-In class:

```
public class RemoteBank : MarshalByRefObject {
    public RemoteBank() {
        throw new System.NotImplementedException();
    }
    ...
```

  - Verwendung eines Interfaces (z.B. **`IRemoteBank`**) oder einer Basisklasse als Remote-Objekt

```
interface IRemoteBank {
    int deposit(int amount);
}
```

  - Metadaten-Assembly: automatische Erzeugung mit **`soapsuds`** (nächste Übung)

2 Serveraktivierte Objekte - Client

- Beispiel

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using BankLibrary;

namespace BankClient {

   class ClientMain {
       public static void Main(String[] args) {
           HttpChannel channel = new HttpChannel();
           ChannelServices.RegisterChannel(channel);

           Object remoteObj = Activator.GetObject(
                  typeof(BankLibrary.RemoteBank),
                  "http://localhost:4711/TestBank.soap");

           RemoteBank bank = (RemoteBank)remoteObj;
           Console.WriteLine("Balance after deposit: {0}",
               bank.deposit(3));
       }
   }
}
```


2 Serveraktivierte Objekte - Client

- Alternative zu **`Activator.GetObject`**: als entfernter Typ registrieren:

```
RemotingConfiguration.RegisterWellKnownClientType(
                  typeof ( BankLibrary.RemoteBank ),
                  "http://localhost:4711/TestBank.soap");

RemoteBank b = new RemoteBank();
```

- Weitere Alternative:

```
RemoteBank b = (RemoteBank) RemotingServices.Connect(
                  typeof( BankLibrary.RemoteBank ),
                  "http://localhost:4711/TestBank.soap");
```


2 Bekanntmachung eines öffentlichen Objekts

- Singleton- und SingleCall-Objekte werden normalerweise dynamisch erzeugt (über Standard-Konstruktor).
- Ein vorher (mit beliebigen Konstruktor) erzeugtes Objekt kann veröffentlicht werden; verhält sich dann wie ein Singleton.
- Objekt erzeugen und mit **`RemotingServices.Marshal`** bekanntgeben

```
namespace BankServer {
   class ServerMain {
       public static void Main (String[] args) {
           HttpChannel channel = new HttpChannel(4711);
           ChannelServices.RegisterChannel(channel);

           RemoteBank bank = new RemoteBank();
           RemotingServices.Marshal(bank,
               "TestBank.soap");

           Console.WriteLine("Server started.");
           Console.ReadLine();
       }
   }
}
```


2 Clientaktivierte Objekte

- Serveraktivierte Objekte stehen mehreren Clients zur Verfügung.
- Clientaktivierte Objekte
  - Für jeden Client wird eine eigene Instanz am Server erzeugt.
  - beliebige Konstruktoren möglich (Erzeugung mit **`new()`**)
  - Nachteil: Client benötigt (Metadaten-)Assembly, Interface oder Basisklasse reicht nicht.
- Die Instanz bleibt über mehrere Aufrufe hinweg aktiv.
  - Damit sind zustandsabhängige Aufrufe möglich.
  - Auch hier bestimmt der Leased-based Distributed Garbage Collector die Lebenszeit.

2 Clientaktivierte Objekte - Server

- Beispiel Server

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace BankServer {

   class ServerMain {

      static void Main (String[] args) {
         HttpChannel channel = new HttpChannel(4711);
         ChannelServices.RegisterChannel(channel);

         RemotingConfiguration.RegisterAcivatedServiceType(
             typeof ( BankLibrary.RemoteBank ) );

         Console.WriteLine("Server started.");
         Console.ReadLine();
      }
   }
}
```


2 Clientaktivierte Objekte - Client

- Beispiel Client

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using BankLibrary;

namespace BankClient {

   class ClientMain {
      public static void Main(String[] args) {
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);

         RemotingConfiguration.RegisterActivatedClientType(
             typeof ( BankLibrary.RemoteBank ),
             "http://localhost:4711");
         // alternativ: Activator.CreateInstance()

         RemoteBank bank = new RemoteBank("Sparkasse");
      }
   }
}
```


2 Clientaktivierte Objekte - Client

- Alternative

```
Object[] attr =
    { new UrlAttribute("http://localhost:4711") };
Object[] args = { "Sparkasse" };

RemoteBank bank = (RemoteBank)Activator.CreateInstance(
    typeof(BankLibrary.RemoteBank), args, attr);
```


2 Zusammenfassung

- Methoden der Klasse **`Runtime.Remoting.RemotingConfiguration`**
  - ◆ Zum Registrieren serveraktivierter Objekte (Singleton oder SingleCall)
    - Client: **`RegisterWellKnownClientType()`**
    - Server: **`RegisterWellKnownServiceType()`**
  - ◆ Zum Registrieren clientaktivierter Objekte
    - Client: **`RegisterActivatedClientType()`**
    - Server: **`RegisterActivatedServiceType()`**
- Klasse **`System.Runtime.Remoting.RemotingServices`** zum Bekanntmachen öffentlicher Objekte (Singleton)
  - Server: **`Marshal()`**, wieder Entfernen mit **`Disconnect()`**, Client: **`Connect()`**
- Klasse **`System.Activator`**
  - ◆ Proxy erzeugen für öffentliche Objekte: **`GetObject()`**
  - ◆ für private Objekte: **`CreateInstance()`**

3 Marshalling

- Wie werden Objekte übertragen? Zwei Möglichkeiten:
  - Objekt fernaufrufbar: Fernverweis wird übertragen
  - Objekt serialisierbar: Objektkopie wird übertragen
  - sonst: Exception
- Standard: Marshal by Value (MBV)
  - Serialisierung: Attribut **`[Serializable]`**
- Marshal by Referenz (MBR)
  - durch Ableiten von **`MarshalByRefObject`**
  - übertragen wird eine Objektreferenz = **`ObjRef`**-Objekt
  - das Referenz-Objekt selbst wird serialisiert (MBV)
- Client muss Typ des entfernten Objekts kennen!

3 Marshalling

- **`ObjRef`**-Objekt enthält Informationen über:
  - Name des Objekts inkl. Name des Assembly
  - Typinformationen über alle Basisklassen des Objekts
  - Typinformationen über alle implementierten Schnittstellen
  - die Adresse (URI = “Uniform Resource Identifier”) des Objekts
  - Informationen über den Kanal des Servers
- Unverhersagbare Aufrufsemantik

```
interface IStack { ... }
class RemoteStack : MarshalByRefObject, IStack { ... }
class Stack       : IStack { ... }

... void op(IStack stack) { ... }
```

Man sieht dem Objekt nicht an, ob es **`MarshalByRefObject`** ist und wenn ja, ob es woanders liegt als der Aufrufer.

4 Objekt-Serialisierung

- Attribut **`[Serializable]`**
- Markierung serialisierbarer Klassen (transitiv)
  - Instanzvariablen werden automatisch serialisiert.
  - Variablen, die mit **`[NonSerialized]`** markiert sind, werden nicht serialisiert.
- Beispiel:

```
[Serializable]
public class Account {
    private int balance = 0;

    [NonSerialized]
    private Bank currentBank;

    public Account (int amount) {
        balance = amount;
    }
}
```


4 Objekt-Serialisierung

- **`IFormatter`** Schnittstelle stellt Methoden bereit, um ein Objekt zu (de)serialisieren:
  - **`void Serialize(Stream outStream, Object graph);`**
  - **`Object Deserialize(Stream inStream);`**
- Implementierungen von **`IFormatter`**:
  - **`Formatter`**: abstrakte Basisklasse für eigene Implementierungen
  - **`SoapFormatter`**: Ausgabe im ASCII-Format (SOAP)
    - Vorteil: Lesbarkeit
    - Nachteil: Konvertierung aufwändig, große Datenmenge
  - **`BinaryFormatter`**: kompakte, binäre Ausgabe

4 Objekt-Serialisierung - Beispiel

- Serialisierung mittels **`Serialize`**:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

Account myAccount = new Account(100);
FileStream myFile = File.Create("Account.txt");

new SoapFormatter().Serialize(myFile, myAccount);

myFile.Close();
```

- Deserialisierung mittels **`Deserialize`**:

```
FileStream myFile = File.Open("Account.txt", FileMode.Open);

Account myAccount =
        (Account) new SoapFormatter().Deserialize(myFile);

myFile.close();
```


4 Objekt-Serialisierung

- Beeinflussen der Serialisierung durch Implementieren der Schnittstelle **`ISerializable`**:
  - Methode um das Objekt zu serialisieren; wird von der CLR aufgerufen
    `public void GetObjectData (SerialisationInfo info, StreamingContext context);`
  - Deserialisierungskonstruktor mit folgenden Parametern:
    `(SerializationInfo info, StreamingContext context);`

4 Objekt-Serialisierung - Beispiel

- Beispiel

```
[Serializable]
public class Account : ISerializable {
   private int amount;
   private Signatur mySig;

   [NonSerializable]
   private Bank currentBank;

   public void GetObjectData (SerializationInfo info,
                              StreamingContext context) {
      info.AddValue("amount", amount);
      info.AddValue("MySignatur", mySig);
      if (ctx.State == StreamingContextStates.CrossProcess)
         ;// Objekt wird an einem anderen Prozess übertragen
   }

   public Account(SerializationInfo info,
                  StreamingContext ctx) {
      amount = info.GetInt32("amount");
      mySig = (Signatur) info.GetValue("MySignatur",
                                       typeof(Signatur));
   }
}
```


5 Channels und Formatter

- Channels sind verantwortlich für den Transport eines Methodenaufrufs und dessen Rückgabewert über das Netzwerk.
- Beispiel:
  - **`TCPChannel`**: jeder Kanal benötigt einen eigenen Port
  - **`HTTPChannel`**: flexibler, mehrere Verbindungen können über Port 80 laufen
- Kanäle müssen bei der Laufzeitumgebung registriert werden.
  - Klasse **`ChannelServices`**
  - Pro *Application Domain* kann immer nur ein Kanal des selben Typs registriert werden (Bsp: 1x TCP, 1x HTTP, aber nicht 2x TCP).
- Kanal ist als Kette von Senken (*Sinks*) organisiert, die der Reihe nach durchlaufen werden.

5 Channels

- Durch Einfügen einer eigenen Senke kann man Einfluss auf die Übertragung nehmen.
  - Erste Senke ist üblicherweise der Formatter.
  - Letzte Senke ist der Transportdienst
  - Beispiele: Verschlüsselungs-Senke oder Logging-Senke
- Transfer Channel
  - **`ProcessMessage()`**: Message, Stream, Headers
  - konvertiert Header (**`ITransportHeaders`**-Objekt als Parameter) in ein Protokoll-abhängiges Format (z.B. HTTP-Header)
  - öffnet Verbindung zum Server und schickt Header und den Inhalt des Streams über diese Verbindung

5 Formatter

- Legt Format eines serialisierten Objekts fest.
- Beispiel:
  - **`BinaryFormatter`**: Standard für TCPChannel
    binäre Übertragung: kompaktes Datenformat
  - **`SOAPFormatter`**: Standard für HTTPChannel
    XML-basiertes Format: größere Datenmenge, aber flexibler
  - Interface **`IFormatter`** und abstrakte Basisklasse **`Formatter`** für eigene Implementierungen (s. Kapitel über Serialisierung)