XML-Webservices

Frank Ruthardt

Dezember 2007

Inhaltsverzeichnis

1 Einleitung und Übersicht 3
2 Motivation und Einsatz von Webservices 3
2.1 Serviceorientierte Architektur (SOA) 3
2.2 Enterprise Application Integration (EAI) 3
3 Einführung in XML 3
3.1 XML-Datenmodelierung 3
3.2 XML-Dokumenttypdefinition (DTD) 4
3.3 XML-Schema (XSD) 5
3.4 Verweise 6
3.5 XML-Serialisierung 7
4 XML-Web-Services 9
4.1 Simple Object Access Protokoll (SOAP) 9
4.2 Webservice-Definition-Language (WSDL) 11
4.3 Proxy-Objekte 12
4.4 Cookies und Sessions 12
4.5 Universal Description, Discovery and Integration (UDDI) 13
5 Implementierung in Microsoft .NET 13
5.1 Bearbeiten von XML-Dateien 14
5.2 Bearbeiten von XML-Schemata 14
5.3 Erstellen eines neuen Webservices 14
5.4 Integrierter Webserver von Visual Studio 15
5.5 Attribute 16
5.6 Attribut WebMethod 16
5.7 Attribut Serializable 17
5.8 Debuggen von Webservices 17
5.9 Attribut DebuggerStepThrough 17
5.10 Session-gestützte Webservices 17
6 Technologie-Empfehlung für VC³ 19
7 Literaturverzeichnis 19
8 Verzeichnis der Codebeispiele und Abbildungen 20

1 Einleitung und Übersicht

Dieses Dokument soll den anderen Projektteilnehmern des Studienprojekts VC³ (Virtual Construction Company Competition) den Einstieg in die Arbeit mit XML-Webservices erleichtern.

Dieses Dokument ist ausdrücklich keine ausführliche Referenz zum Thema XML-Webservices, da dies den Rahmen des Dokuments sprengen würde. Die wesentlichen Technologien sind jedoch aufgeführt und mit anschaulichen Beispielen angereichert.

Das Dokument ist in zwei große Blöcke geteilt. Einen eher theoretischen Block mit den Grundlagen der Webservices und den Grundlagen von XML und einem großen Kapitel über die Implementierung von Webservices durch Microsoft im .NET-Framework.

2 Motivation und Einsatz von Webservices

In diesem Kapitel werden zwei mögliche Einsatzgebiete für Webservices skizziert. Natürlich ließe sich dieses Kapitel beliebig erweitern. Webservices sind eine wichtige Technologie für verteilte Systeme.

2.1 Serviceorientierte Architektur (SOA)

Um möglichst schnell auf Veränderungen in Geschäftsanwendungen reagieren zu können, ist in den letzten Jahren die Idee der serviceorientierten Architektur entstanden. Wie bei der komponentenbasierten Entwicklung besteht die Grundidee in der Aufteilung des Gesamtsystems in kleine organisatorische Einheiten, mit dem Unterschied, dass nun Einheiten nicht nur getrennt entwickelt, sondern auch getrennt betrieben werden. Die Konsequenz ist, dass jede Einheit (jeder Service) vollständig autonom arbeiten muss. Das heißt er ist insbesondere für seine Laufzeitumgebung bzw. Persistenzschicht selbst zuständig.

Ein Dienst/Service ist per Definition:

- abgeschlossen und eigenständig
- durch eine öffentliche Schnittstelle in einem Netzwerk für Anwendungen oder andere Dienste erreichbar
- plattformunabhängig verwendbar

Aus dieser Definition folgt, dass die Kommunikation zwischen den einzelnen Diensten, zum Beispiel über die hier beschriebenen XML-Webservices, über wohldefinierte Schnittstellen erfolgen muss.

2.2 Enterprise Application Integration (EAI)

In Unternehmen trifft man häufig auf eine relativ große Menge von Softwaresystemen, die zwar einerseits vollständig autonom arbeiten, aber dennoch untereinander Daten austauschen müssen. Häufig benötigt man dazu plattformübergreifende Kommunikationstechniken und gerade XML-Webservices bieten dazu einen hervorragenden Ansatz. Im Vergleich zu SOA, bei der ein einzelnes Softwaresystem aufgeteilt wird, werden bei EAI die Interaktionen zwischen verschiedenen Softwaresystemen für ein Gesamtunternehmen betrachtet.

3 Einführung in XML

3.1 XML-Datenmodelierung

XML (eXtensible Markup Language) ist eine vom W3C (World Wide Web Consortium) standardisierte Sprache, mit der Daten auf einheitliche Weise zwischen verschiedenen Softwaresystemen ausgetauscht werden können. Daten können dabei sowohl eine flache als auch eine hierarchische Struktur aufweisen.

Die Daten werden in Tags gegliedert, die im Gegensatz zur reinen Formatierungssprache HTML, beliebig erweiterbar sind ein Tag wird durch `<Tagname>` eingeleitet und durch `</Tagname>` beendet.

Listing 1 (XML-Dokument)

```
<Spiel>
    <Spieler>Müller</Spieler>
    <Spieler>Mayer</Spieler>
</Spiel>
```

Enthält ein Tag keine Unterelemente so kann er auch direkt als eine Zeichenkette `<Tagname/>` notiert werden. Außer dem Inhalt kann jeder Tag noch Attribute enthalten. Durch diese Attribute können weitere Informationen zu einem XML-Element hinzugefügt werden. Die Attribute werden direkt als Schlüssel-Wert-Paar im einleitenden Tag notiert.

Listing 2 (XML- Dokument mit Attributen)

```
<Spiel Startdatum="01.12.2007">
    <Spieler Id="1">Müller</Spieler>
    <Spieler Id="2">Mayer</Spieler>
</Spiel>
```

Kommentare werden in XML wie in HTML durch `<!--` eingeleitet und durch `-->` abgeschlossen.

3.2 XML-Dokumenttypdefinition (DTD)

Eine besondere Eigenschaft von XML-formatierten Daten ist die Eigenschaft, dass ein XML-Dokument gewisse Dokumentregeln enthalten kann, nachdem die Daten aufgebaut sind. Das heißt, es ist jederzeit möglich zu verifizieren, ob die Daten sich noch in einem konsistenten Zustand befinden. Das Regelwerk kann sich dabei sowohl direkt im Dokument befinden als auch per Link verknüpft sein.

Ein XML-Dokument, dass kein Dokumentschema enthält, jedoch den XML-Grundregeln entspricht (es enthält genau ein Wurzelelement und jedes Datum ist korrekt in Start- und End-Tags eingeschlossen) bezeichnet man als wohlgeformt.

Ein XML-Dokument mit Dokumentschema bezeichnet man als gültiges XML-Dokument falls alle Einschränkungen des Dokumentschemas erfüllt sind.

Listing 3 (XML-Dokument mit Dokumenttypdefinition)

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Spiel[
    <!ELEMENT Spiel (Spieler+)>
    <!ATTLIST Spiel StartZeit CDATA #REQUIRED>
    <!ELEMENT Spieler (#PCDATA)>
    <!ATTLIST Spieler Vorname CDATA #REQUIRED>
    <!ATTLIST Spieler Name CDATA #REQUIRED>
]>

<Spiel StartZeit="01.12.2007">
    <Spieler Vorname="Hans" Name ="Mayer" />
    <Spieler Vorname="Martin" Name="Müller" />
</Spiel>
```

Damit die Dokumentschema-Tags nicht mit den Tags für die Daten verwechselt werden, werden sie durch `<!` eingeleitet. Im vorherigen Beispiel ist ein Dokumentschema für das weiter oben angeführt Beispiel des Spiels zu sehen. Betrachten wir nun die einzelnen Elemente des Listing 3 (XML-Dokument mit Dokumenttypdefinition).

DOCTYPE

Im DOCTYPE-Element wird die Wurzel des XML-Dokuments angegeben.

ELEMENT

Durch das Element-Tag werden alle Elemente, die im Dokument verwendet werden können, definiert. Es wird dabei jeweils in Klammern angegeben welche Unterelemente vorhanden sein dürfen bzw. müssen. Ein + bedeutet dabei, dass beliebig viele aber mindestens ein Element erforderlich ist. Ein * verhält sich wie das +, es ist jedoch auch kein Element möglich.

ATTLIST

Durch das Attlist-Element werden alle Attribute eines Elements definiert. Es ist möglich ein Attribut als verpflichtend (#REQUIRED) oder als optional (#IMPLIED) zu definieren.

CDATA steht für den einzigen skalaren Datentyp String; PCDATA für einen komplexen Datentyp (also weitere XML-Elemente).

Das oben gezeigte DTD-Beispiel enthält also ein Element vom Typ Spiel, das beliebig viele untergeordnete Spieler und ein Attribut, wann das Spiel gestartet wurde, enthält. Ein Spieler muss einen Vornamen und einen Nachnamen enthalten.

3.3 XML-Schema (XSD)

Neben der relativ einfachen Möglichkeit durch DTD ein XML-Dokument einzugrenzen, existiert eine weitere weitaus mächtigere Variante, die als XML-Schema bezeichnet wird.

Entscheidende Vorteile sind, dass XSD-Dokumente im Vergleich zu DTD-Dokumenten selbst gültige XML-Dokumente darstellen und dass XSD nicht nur einen Datentyp kennt (wie dies bei DTD der Fall ist) sondern eine ganze Menge (xs:string, xs:decimal, xs:integer, xs:float, xs:boolean, xs:date, xs:time, u.a.) unterstützt. Außerdem lassen sich auch Referenzen in einem XML-Schema abbilden.

XSD hat für die Arbeit mit Microsoft Visual Studio .NET eine besondere Bedeutung, da es umfassend verwendet und unterstützt wird. Beispiele zum Erstellen und Verwenden eines XML-Schemas folgen im Kapitel 5 „Implementierung in Microsoft .NET“

Alle XSD-Tags werden durch `xs:` einem sogenannten Namensraum zugeordnet und sind damit eindeutig von anderen Elementen zu unterscheiden.

Betrachten wir nun wieder unser einfaches, schon im Kapitel 3.2 verwendetes Dokumentschema, dieses mal in XSD:

Listing 4 (XML-Schema)

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
        targetNamespace="http://tempuri.org/XMLSchema.xsd"
        xmlns="http://tempuri.org/XMLSchema.xsd"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- Wurzel-Element -->
    <xs:element name="Spiel" type="SpielType" />

    <!-- Type-Definitions -->
    <xs:complexType name="SpielerType">
        <xs:attribute name="Vorname" type="xs:string" />
        <xs:attribute name="Name" type="xs:string" />
    </xs:complexType>
    <xs:complexType name="SpielType">
```

```
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element type="SpielerType" name="Spieler" />
        </xs:sequence>
        <xs:attribute name="StartZeit" type="xs:dateTime" />
    </xs:complexType>
</xs:schema>
```

Als zentraler Einstiegspunkt in das Schema wird ein Wurzel-Element vom Typ SpielType definiert. Dieser Typ wird weiter unten, ähnlich einer Klasse in einer objektorientierten Programmiersprache, deklariert. Ein SpielType-Element besteht also aus einer Liste (Sequence) von mindestens einem Element des Typs SpielerType und aus der Startzeit.

Das gültige XML-Datendokument muss nun auch geringfügig angepasst werden. (Insbesondere muss das Datum nun W3C konform angegeben werden)

Listing 5 (XML-Dokument mit XML-Schema)

```
<?xml version="1.0" encoding="utf-8"?>
<Spiel xmlns="http://tempuri.org/XMLSchema.xsd"
        StartZeit="2007-12-01T12:37:43+01:00">
    <Spieler Vorname="Hans" Name="Mayer" />
    <Spieler Vorname="Martin" Name="Müller" />
</Spiel>
```

Sowohl DTD als auch XSD lassen sich mit Visual Studio leicht überprüfen. Insbesondere stellt Visual Studio bei einem verfügbaren Schema automatisch nur die richtigen Elemente in der automatischen Vervollständigung zur Verfügung. Beide lassen sich auch in eigenen Anwendungen durch das .NET-Framework einfach validieren. (Siehe Kapitel 5.1 „Bearbeiten von XML-Dateien“)

3.4 Verweise

Da sich nicht immer alle Daten auf hierarchische Strukturen abbilden lassen, sondern häufig zum Beispiel ein Element zwei übergeordnete Elemente hat, reicht die bis dahin vorgestellte hierarchische Struktur nicht immer aus. Um auch solche relationale Strukturen abbilden zu können, können in XML-Dokumenten Verweise auf beliebige Elemente gesetzt werden. Üblicherweise wird dazu bei jedem Element eine eindeutige ID als Attribut gesetzt, die von anderen Elementen referenziert werden kann. Im folgenden Abschnitt fügen wir nun in unsere Datenstruktur ein, welcher Spieler welche Geräte besitzt. Weiter unten im Dokument definieren wir außerdem die verfügbaren Geräte.

Listing 6 (XML-Dokument mit Verweisen)

```
<?xml version="1.0" encoding="utf-8"?>
<Spiel xmlns="http://tempuri.org/XMLSchemaEx.xsd"
        StartZeit="2007-12-01T12:37:43+01:00">
    <Spieler Vorname="Hans" Name="Mayer">
        <SpielerGerät GerätType="Bag" Wert="72000" />
        <SpielerGerät GerätType="Bag" Wert="75000" />
        <SpielerGerät GerätType="Kra" Wert="160000" />
    </Spieler>
    <Spieler Vorname="Martin" Name="Müller">
        <SpielerGerät GerätType="Kra" Wert="140000"/>
    </Spieler>

    <Gerät ID="Bag" Name="Bagger" />
    <Gerät ID="Kra" Name="Kran"/>
</Spiel>
```

Die beiden Spieler besitzen jeweils Geräte, die sich auf die verfügbaren Geräte beziehen. Es kann so aus dem XML-Dokument schnell herausgelesen werden, welche Geräte es überhaupt gibt, ohne die komplette evtl. sehr lange Liste der Spieler zu iterieren. Die Vorgehensweise ähnelt nun der Speicherung in einer relationalen Datenbank.

Betrachten wir nun zum Abschluss das vollständige Schema:

Listing 7 (XML-Schema mit Verweisen)

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
        targetNamespace="http://tempuri.org/XMLSchemaEx.xsd"
        xmlns="http://tempuri.org/XMLSchemaEx.xsd"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <!-- Root-Element -->
    <xs:element name="Spiel" type="SpielType" />
    <!-- Type-Definitions -->
    <xs:complexType name="SpielType">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element type="SpielerType" name="Spieler" />
            <xs:element name="Gerät" type="GerätType" minOccurs="0"
                maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="StartZeit" type="xs:dateTime" />
    </xs:complexType>

    <xs:complexType name="GerätType">
        <xs:attribute name="ID" type="xs:ID" />
        <xs:attribute name="Name" type="xs:string" />
    </xs:complexType>
    <xs:complexType name="SpielerType">
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="SpielerGerät" type="SpielerGerätType" />
        </xs:sequence>
        <xs:attribute name="Vorname" type="xs:string" />
        <xs:attribute name="Name" type="xs:string" />
    </xs:complexType>
    <xs:complexType name="SpielerGerätType">
        <xs:attribute name="GerätType" type="xs:IDREF" />
        <xs:attribute name="Wert" type="xs:decimal" />
    </xs:complexType>
</xs:schema>
```

Zusätzlich zum Schema im vorherigen Abschnitt sind die Typen GerätType und SpielerGerätType hinzugekommen. Alle Geräte die im Spiel vorkommen können, werden damit (wie oben gezeigt) unter den Spielern aufgeführt. Jeder Gerätetyp muss dabei eine eindeutige Bezeichnung im Attribut ID haben.

Zu den Spielern können nun als Unterelemente beliebig viele, aber auch keine Spielergeräte (also Geräte die der Spieler besitzt) hinzugefügt werden. Jedem Spielergerät muss ein Gerätetyp zugeordnet werden, der im Gesamtspiel verfügbar ist. Zusätzlich bekommt jedes Spielergerät einen aktuellen Wert.

3.5 XML-Serialisierung

Unter Serialisierung versteht man die Vorgehensweise ein Objekt aus einer objektorientierten Programmierumgebung in eine sequenzielle Zeichenkette umzuwandeln.

Um dies zu veranschaulichen verwenden wir wieder unser Beispiel mit Spiel und Spieler und betrachten dazu zunächst den dazugehörigen C#-Quellcode.

Listing 8 (C#-Quellcode für serialisierbares Objekt)

```
[Serializable()]
public class Spiel
{
    private List<Spieler> m_Spieler = new List<Spieler>();
```

```
    private DateTime m_StartZeit = DateTime.Now;

    public List<Spieler> Spieler
    {
        get { return m_Spieler; }
    }

    public DateTime StartZeit
    {
        get { return m_StartZeit; }
    }
}

[Serializable()]
public class Spieler
{
    private String m_Vorname = "";
    private String m_Name = "";

    public Spieler(String vorname, String name)
    {
        m_Vorname = vorname;
        m_Name = name;
    }

    public String Vorname
    {
        get { return m_Vorname; }
        set { m_Vorname = value; }
    }
    public String Name
    {
        get { return m_Name; }
        set { m_Name = value; }
    }
}
```

Wir verwenden dazu jeweils eine Klasse für Spiel und eine für Spieler. Die Spielklasse enthält eine Variable StartZeit und eine Liste mit Spielern, die an diesem Spiel teilnehmen. Die Klasse Spieler enthält je eine Variable für Vorname und Nachname.

Auf die Beschreibung der Properties (Getter und Setter) wird an dieser Stelle verzichtet. Sie werden lediglich für die Serialisierung benötigt, da der Standardserialisierer alle öffentlichen Properties speichert und wiederherstellt. Die Properties müssen nicht zwingend als public deklariert sein. Für private Properties wird bei der Serialisierung ein Zugriff über Reflection verwendet.

Wird zur Laufzeit nun ein Objekt von der Klasse Spiel instanziiert, so wird automatisch im Speicher eine neue Liste mit Spielern dazu angelegt. Nehmen wir nun an, wir möchten eine Instanz eines Spielers mit zwei Mitspielern erstellen. Hierfür verwenden wir den folgenden Quellcode zum Instanziieren unserer Objekte.

Listing 9 (C#-Quellcode für Initialisierung eines Objekts)

```
Spiel mySpiel = new Spiel();
mySpiel.Spieler.Add(new Spieler("Hans", "Mayer"));
mySpiel.Spieler.Add(new Spieler("Martin", "Müller"));
```

Wir wollen nun diese Liste serialisieren. Dafür werden alle Objekte und Eigenschaften in XML-Elemente umgewandel. Das Ergebnis sieht folgendermaßen aus:

Listing 10 (XML-serialisiertes Objekt)

```
<Spiel StartZeit="01.12.2007">
    <Spieler Vorname="Hans" Name ="Mayer" />
    <Spieler Vorname="Martin" Name="Müller" />
</Spiel>
```

Diese sequenzielle Zeichenfolge kann nun zum Beispiel einfach über ein Netzwerk übertragen und beim Empfänger deserialisiert, also wieder zu einem Objekt im Speicher rekonstruiert werden. Dabei entsteht unvermeidbar eine Kopie der Objekte. Manipulationen werden nicht automatisch an das Ursprungsobjekt zurückgegeben.

Mithilfe der XML-Serialisierung können beliebige Kombinationen von skalaren Objekten serialisiert, und über ein Übertragungsmedium übermittelt werden.

4 XML-Web-Services

Schon seit einiger Zeit gibt es im Internet Anwendungen, die menschlichen Benutzern Zugriff auf servergestützte Informationssysteme ermöglichen. Dies können zum Beispiel Online-Shops oder andere datenbankbasierte Webseiten sein.

Um wirtschaftlich effizienter arbeiten zu können, ist es für Geschäftsanwendungen jedoch immer wichtiger auch automatisch auf solche Systeme zugreifen zu können. Diesen automatischen Zugriff über HTML-Formulare durchzuführen, ist nahezu unmöglich. Der Grund dafür liegt darin , dass die eigentlichen Rückgabe-Daten stets über Screen-Scrapping-Techniken erst wieder aus dem Ergebnis-HTML-Code herausgefiltert werden müssten. XML-Webservices sind also maschinenorientierte Webseiten, die die benötigten Informationen ohne Formatierung in einem geeigneten Austauschformat (auf XML-Dokument) zur Verfügung stellen.

XML-Webservices basieren generell auf dem Prinzip eines Remoteprozeduraufrufs (RPC). Es wird eine wohl definierte Funktion auf dem Server mit einer Menge von Parametern ausgeführt und das Ergebnis zurück geliefert. Sowohl die Parameter, als auch der Rückgabewert werden dabei XML-serialisiert und können so über beliebige Netzwerkkanäle (in der Regel über das HTTP-Protokoll) übertragen werden.

4.1 Simple Object Access Protokoll (SOAP)

SOAP stellt Regeln auf wie die Kommunikation zwischen Webservice und Client ablaufen muss. Die Grundidee besteht im Versenden von XML-Nachrichten, die einem bestimmten XML-Schema (http://www.w3.org/2001/12/soap-envelope) folgen. Wir verzichten an dieser Stelle darauf, das komplette Schema abzudrucken und bis ins Detail zu diskutieren. Im Folgenden werden wir wieder einfache Beispiele genauer ansehen und die notwendige Theorie davon ableiten. Weitere Informationen sind im Buch Programming Microsoft .NET XML-Webservices zu finden (1).

Listing 11 (SOAP-Nachricht)

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
    <soap:Header>
    </soap:Header>
    <soap:Body>
    </soap:Body>
</soap:Envelope>
```

Generell besteht eine SOAP-Nachricht also aus einem Envelope-Tag das optional einen Header-Tag, und zwingend einen Body-Tag enthält. Beispiele für die Headerdaten sind Benutzernamen, RequestId oder ähnliches. Der Body enthält die anwendungsspezifischen Daten. Üblicherweise ist dies der Funktionsname, der aufgerufen werden soll, sowie die serialisierten Parameterobjekte.

Eine typische Anfrage für VC³ könnte wie folgt aussehen:

Listing 12 (SOAP-Anfrage)

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
    <soap:Header>
        <ServiceAuthHeader xmlns="http://www.ibl.de/vc3/services">
            <Username>frank</Username>
            <Password>*****</Password>
        </ServiceAuthHeader>
    </soap:Header>
    <soap:Body>
        <GetSpieler xmlns="http://www.ibl.de/vc3/services">
            <LanguageCode>de-de</LanguageCode>
            <input>
                <GameId>25</GameId>
            </input>
        </GetSpieler>
    </soap:Body>
</soap:Envelope>
```

Das oberste Element im Rumpf definiert also die Funktion, die aufgerufen wird. In der Funktion wird die Sprache übermittelt, da Client und Server nicht unbedingt die gleiche Sprache verwenden müssen. Der letzte und in der Regel größte Block innerhalb der Funktion enthält die XML-serialisierten Parameterdaten.

Der Webserver schickt die Antwort ebenfalls als SOAP-Nachricht zurück. In diesem Fall die beiden Spielerobjekte.

Listing 13 (SOAP-Antwort)

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
    <soap:Body>
        <GetSpielerResponse xmlns="http://www.ibl.de/vc3/services">
            <SpielerListe>
                <Spieler>
                    <Vorname>Hans</Vorname>
                    <Name>Mayer</Name>
                </Spieler>
                <Spieler>
                    <Vorname>Martin</Vorname>
                    <Name>Müller</Name>
                </Spieler>
            </SpielerListe>
        </GetSpielerResponse >
    </soap:Body>
</soap:Envelope>
```

Wir erhalten also als Ergebnis genau die beiden Spielerobjekte aus Listing 9 (C#-Quellcode für Initialisierung eines Objekts) zurück.

Datenübertragung

SOAP-Nachrichten können generell über beliebige Netzwerkprotokolle übertragen werden. Die üblichste Kombination ist SOAP über HTTP bzw. HTTPS. Im Studienprojekt ist die Kombination über HTTPS zu empfehlen, da als Backend-Server vermutlich der Microsoft IIS 6.0 eingesetzt wird und die Implementierung in Microsoft .NET über HTTPS bereits seit der Version 1.0 mit geringem Aufwand möglich ist. Die SSL-Verschlüsselung ist zwingend notwendig, da die Daten sonst ungesichert durch das Internet geschickt werden.

4.2 Webservice-Definition-Language (WSDL)

Durch die Webservice-Definition-Language wird ein Webservice programmiersprachen- und plattformunabhängig beschrieben. Ein WSDL-Dokument muss alle Meta-Daten zu einem XML-Webservice enthalten. Dazu gehören:

- Zugriffsprotokoll (HTTP/HTTPS/TCP)
- Eindeutige Zugriffsadresse (URL)
- Alle möglichen Befehle
- Alle möglichen SOAP-Nachrichten

Die Beschreibung erfolgt dabei ebenfalls in XML und muss einem vorgegebenen XML-Schema (http://schemas.xmlsoap.org/wsdl/) folgen. Dieses Schema ist im Vergleich zu SOAP noch wesentlich komplexer. Wir verzichten wieder auf das ausführliche Besprechen des Schemas und werfen einen Blick auf unser Beispiel und das daraus resultierende WSDL-Dokument.

Zunächst muss das WSDL-Dokument also die grundlegenden Informationen bereitstellen. In diesem Beispiel lautet der Servername www.idle.de und die Verbindung erfolgt über HTTPS.

Listing 14 (WSDL Teil 1)

```
<service name="vc3">
    <port name="vc3Port" binding="tns:vc3Binding">
        <soap:address xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
            http://www.ibl.de/vc3/services</soap:address>
    </port>
</service>
```

Im nächsten Abschnitt muss definiert werden, wie das Binding für den Webservice aussieht. Es werden die externen Aufrufe an interne Operationen und SOAP-Nachrichten gebunden:

Listing 15 (WSDL Teil 2)

```
<binding name="vc3Binding">
    <operation name="GetSpieler">
        <input>
            <body use="literal" />
        </input>
        <output>
            <body use="literal" />
        </output>
    </operation>
</binding>
```

Im folgenden Abschnitt werden die verfügbaren Operationen definiert:

Listing 16 (WSDL Teil 3)

```
<operation name="GetSpieler">
    <input message="GetSpielerRequest"/>
    <output message="GetSpielerResponse"/>
</operation>
```

Schließlich werden die SOAP-Nachrichten im WSDL-Dokument vordefiniert:

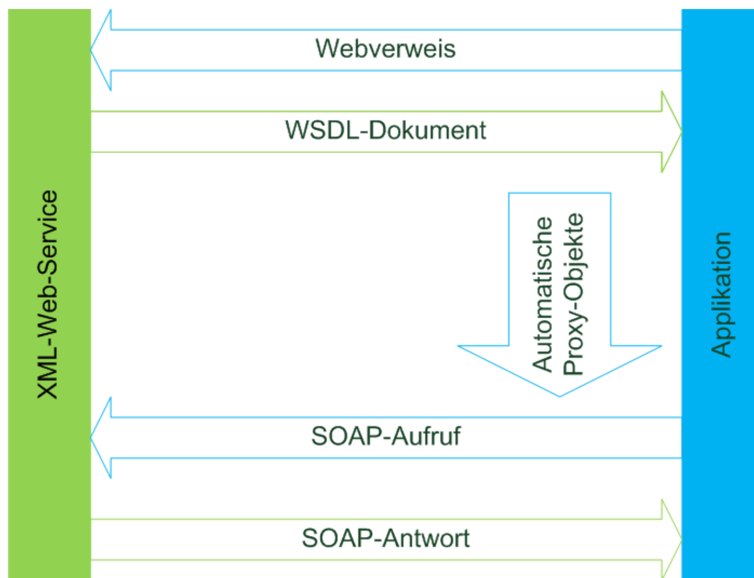
Listing 17 (WSDL Teil 4)

```
<message name="GetSpielerRequest">
    <part name="GameId" type="xs:integer"/>
</message>
```

Im folgenden Abschnitt wird das Zusammenspiel beim Verbinden zu einem Webservices erläutert.

4.3 Proxy-Objekte

Abbildung 1 (Zusammenwirkung von WSDL, Proxyobjekten und SOAP-Nachrichten)



Nach außen zur Applikation werden Webservices als Proxy-Objekte bereitgestellt. Die Aufrufe werden jedoch nicht direkt von diesen Objekten ausgeführt, sondern lediglich die Parameter serialisiert und an den XML-Webservice über SOAP-Nachrichten weitergeleitet. Die Ergebnisse des Webservices werden wieder deserialisiert und als Objekt an die Anwendung zurückgegeben. Der komplette Quellcode für die Datenübertragung wird in den Proxy-Objekten gekapselt.

Für den Anwendungsentwickler ist nicht am Code ersichtlich, dass es sich hierbei um einen Aufruf eines Remote-Servers handelt. (Im folgenden Beispiel wurde ein Webverweis auf einen Webservice gesetzt)

Listing 18 (C#-Quellcode für Initialisierung eines Webservices)

```
Service serv = new Service();
serv.GetSpieler();
```

Die Proxy-Objekte werden in der Regel automatisch aus dem WSDL-Dokument generiert. Da das WSDL-Dokument plattformunabhängig ist, können auch leicht .NET-Proxy-Objekte generiert werden, um auf einen Webservice zuzugreifen, der in einer anderen Sprache geschrieben wurde.

Der Zugriff auf Webmethoden erfolgt standardmäßig synchron, d.h. das aufrufende Programm wartet bis die Rückmeldung des Servers erfolgt.

Es werden von .NET zusätzlich automatisch Funktionen für den asynchronen Zugriff erstellt.

Zum Beispiel:

```
GetSpielerAsync(delegate CallBack)
```

4.4 Cookies und Sessions

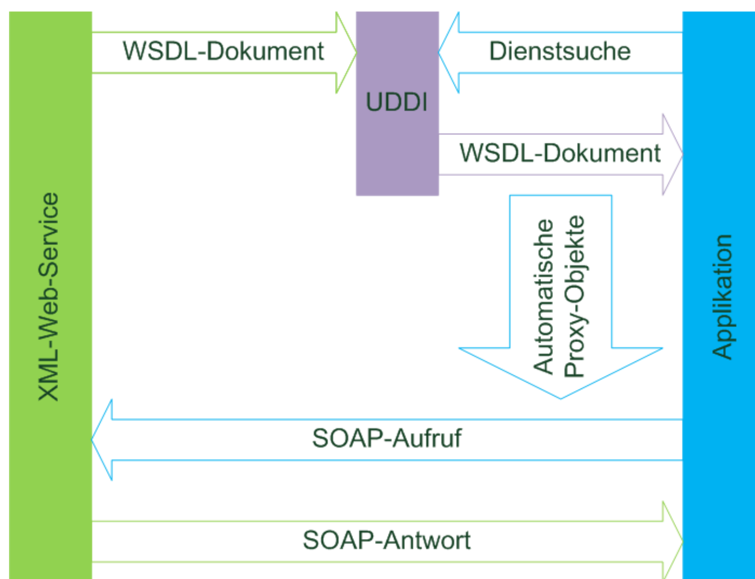
Ein großes Problem von Webservices ist die Tatsache, dass jeder Aufruf einer Funktion zunächst unabhängig von allen vorherigen Aufrufen ist. Wird zuerst eine Funktion namens Connect aufgerufen mit Benutzernamen und Passwort, so ist beim nächsten Funktionsaufruf zum Beispiel der GetSpieler-Funktion diese Information nicht mehr vorhanden.

Dieselbe Einschränkung findet man bei Technologien für servergestützte Webseiten. Diese Einschränkung kann jedoch durch Sessions umgangen werden. Dazu muss bei jedem Aufruf einer Funktion auf eine beliebige Art und Weise die SessionId mitgeliefert werden.

Man könnte dies realisieren, indem man für jede Funktion einen zusätzlichen Parameter einfügt, auf der Serverseite eine Liste mit Sessions manuell verwaltet und die Session-Informationen speichert. Um dies jedoch nicht bei jedem Aufruf im Quellcode machen zu müssen, werden dafür Cookies verwendet. Die genaue Implementierung von Session-gestützten Webservices in Microsoft .NET wird im Kapitel 5.10 „Session-gestützte Webservices“ beschrieben.

4.5 Universal Description, Discovery and Integration (UDDI)

Abbildung 2 (Zusammenwirkung von WSDL, Proxyobjekten, SOAP-Nachrichten und UDDI)



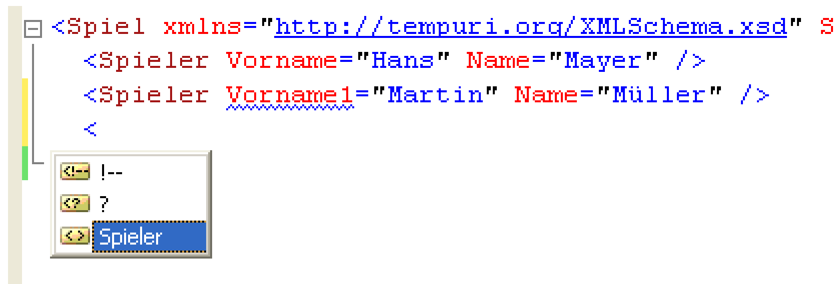
Um XML-Webservices in der öffentlichen Schnittstellenwelt bekannt zu machen, ist der Verzeichnisdienst Universal Description, Discovery und Integration (UDDI) entwickelt worden, der das Auffinden von Webservices ermöglicht. Dazu wird ein Webservice (genauer gesagt, dessen WSDL-Dokument) in diesem globalen Verzeichnisdienst (UDDI) veröffentlicht. Es ist nun möglich in diesem Verzeichnisdienst ohne Kenntnis des Webservices zu suchen. Ist ein geeigneter Webservice gefunden, wird das WSDL-Dokument heruntergeladen und die Proxy-Objekte generiert (aus diesem Grund muss im WSDL auch die vollständige URL zum Webservice enthalten sein). Über die Proxy-Objekte ist anschließend der Zugriff auf den Webservice ohne Probleme möglich.

5 Implementierung in Microsoft .NET

Da es sich bei den XML-Webservices und der damit zugrundeliegenden XML-Technologie um eine ganz zentrale Technologie in der Microsoft .NET-Strategie handelt, ist auch die Toolunterstützung sehr ausgeprägt. In diesem Kapiteln wird der Umgang mit XML und Webservices erläutert.

5.1 Bearbeiten von XML-Dateien

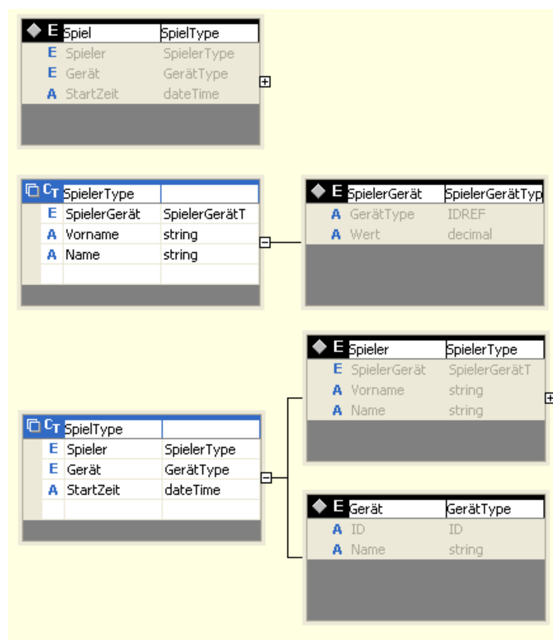
Abbildung 3 (Bearbeiten von XML-Dateien)



Die Grundlage für die XML-Unterstützung ist der Quellcodeeditor des Visual Studios. Der XML-Code wird dabei automatisch farbig unterlegt und korrekt eingerückt. Wird ein Schema verwendet, kann das Visual Studio darüber hinaus den Anwender durch IntelliSense unterstützen die korrekten Tags zu verwenden. Nicht gültige Tags oder Attribute werden blau unterstrichen.

5.2 Bearbeiten von XML-Schemata

Abbildung 4 (Darstellung eines XML-Schemas)

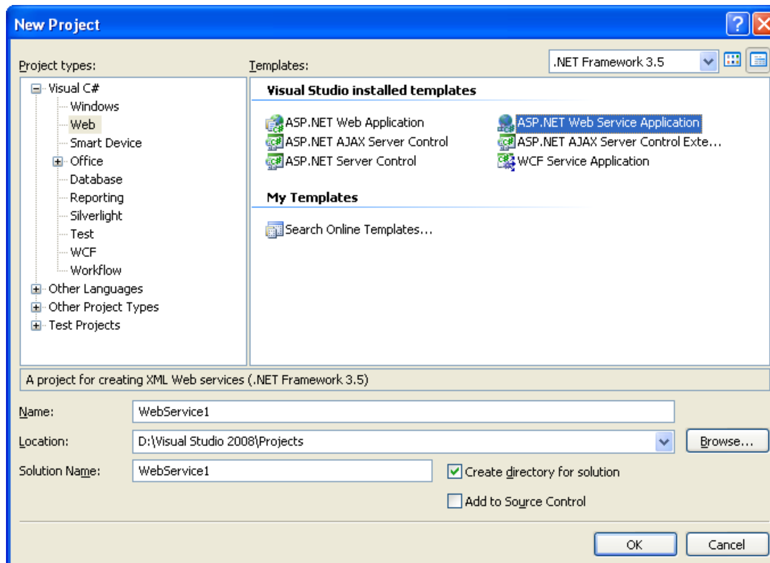


Das Erstellen der relativ schnell komplex werdenden XML-Schemata wird durch das Visual Studio grafisch unterstützt. Typen werden hier blau dargestellt, Elemente jeweils schwarz. Die Abbildung zeigt das bekannte Schema mit dem Wurzel-Element Spiel vom Typ SpielType, das die Sequenzen von Spielern und Geräten enthält. Das Schema kann leicht per Drag & Drop und direkter Manipulation geändert werden.

5.3 Erstellen eines neuen Webservices

Das Erstellen eines Web-Services ist im Visual Studio sehr leicht möglich, indem ein neues Webseiten-Projekt angelegt und im entsprechenden Dialog Web-Dienst als Vorlage ausgewählt wird.

Abbildung 5 (Erstellen eines neuen Webservices)



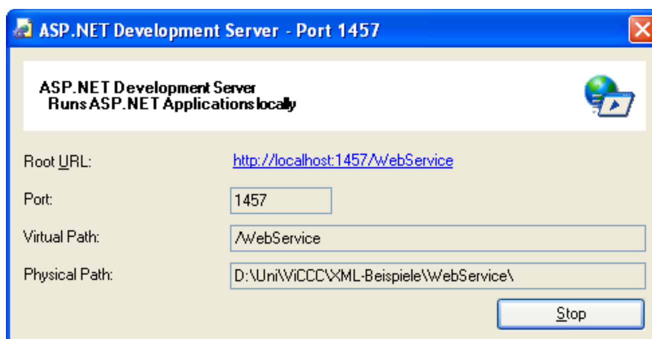
Dadurch wird automatisch ein korrektes Projekt angelegt, das sofort eine Web-Methode enthält und von anderen Projekten referenziert werden kann.

Auf den Web-Service kann von jedem Projekt durch hinzufügen eines Webverweises zugegriffen werden. Dabei werden die Proxy-Objekte automatisch generiert.

5.4 Integrierter Webserver von Visual Studio

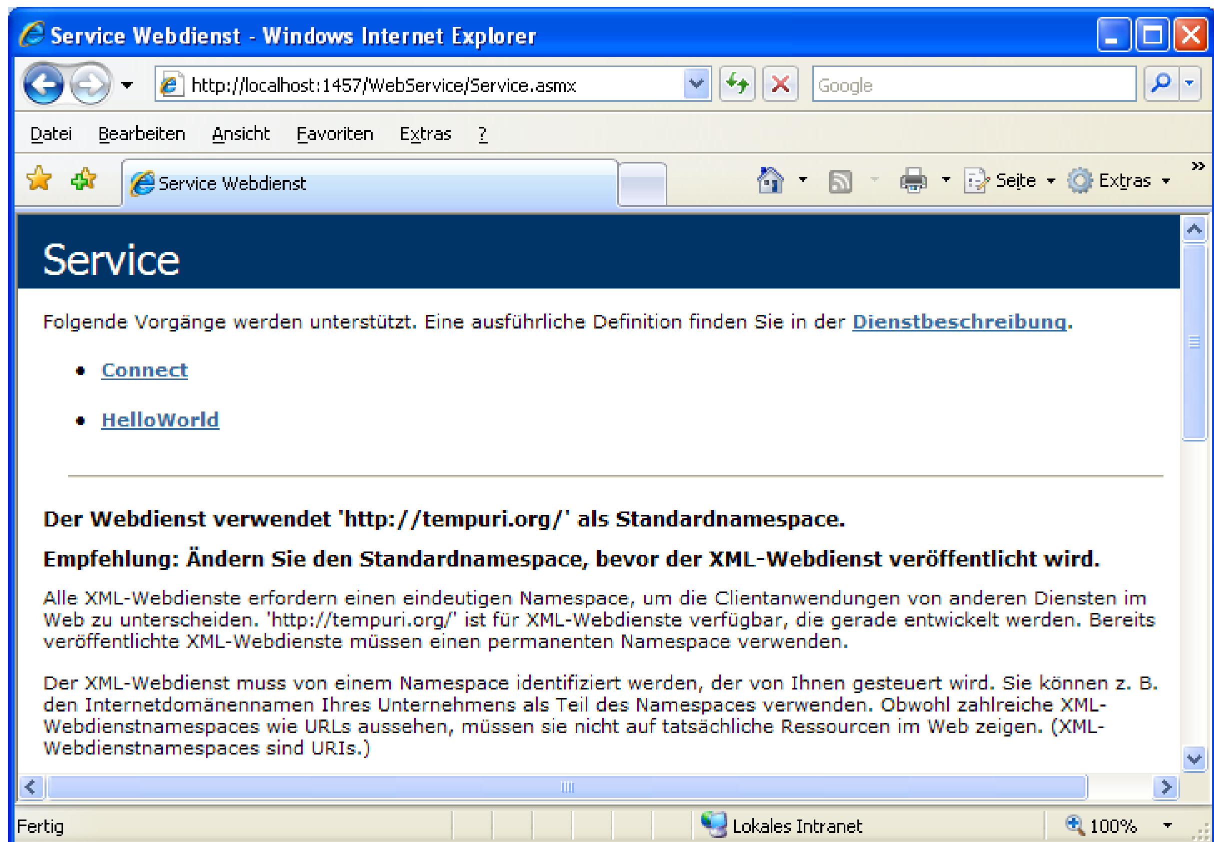
Visual Studio verwendet zum Ausführen eines Web-Projektes automatisch einen integrierten Webserver, der einen temporären Port verwendet, um die Webseite über HTTP zu veröffentlichen.

Abbildung 6 (Visual Studio integrierter Webserver)



Durch die Verwendung dieses Webservers kann zum Testen des Webservices ein Browseraufruf auf den temporären Port gerichtet werden.

Abbildung 7 (Ergebnis eines Browseraufrufs eines Webservices)



Alle Webmethoden kann man im Browser testen. Außerdem kann man das WSDL-Dokument, des Webservices einsehen.

5.5 Attribute

In manchen Fällen ist es sinnvoll, zusätzlich zum Namen einer Methode oder Klasse im Quellcode weitere beschreibende Meta-Daten anzugeben. In Microsoft .NET werden dazu Attribute verwendet, durch die man Programmiersprachenobjekten spezielle Eigenschaften zugeordnen kann. Es können sowohl vordefinierte, als auch benutzerdefinierte Attribute verwendet werden.

Attribute sind in eckigen Klammern zu notieren und entsprechen den Annotations in JAVA. Im Folgenden werden zwei Beispiele vorgestellt die durch Attribute gelöst sind.

5.6 Attribut WebMethod

Innerhalb einer Klasse eines Webservices unterscheidet man zwischen öffentlichen Methoden, die nur bei direktem Objektverweis erreichbar sind und den eigentlichen Web-Methoden. Diese Web-Methoden werden im WSDL veröffentlicht. (Diese Methoden sind also im Sinne dieses Dokumentes die öffentlichen Methoden eines Webservices).

Um die Web-Methoden von den normalen öffentlichen Methoden zu unterscheiden, werden sie mit dem Attribut `WebMethod()` markiert.

Listing 19 (C#-Quellcode einer Webmethode)

```
[WebMethod()]
public Spieler[] GetSpieler()
{
    …
}
```

5.7 Attribut Serializable

Um Objekte in Webservices nutzen zu können, müssen diese in ein XML-Dokument, in der Regel durch einen Standardserialisierer serialisiert werden.

Um eine Klasse serialisierbar zu markieren, wurde das Attribut `Serializable()` eingeführt. Ohne Parameterangaben bedeutet dies, dass die Klasse mit dem Standardserialisierer des .NET-Frameworks serialisiert wird. Es ist jedoch auch möglich explizit in den Klammern einen eigenen Serialisierer anzugeben.

5.8 Debuggen von Webservices

Ein Webservice kann in .NET genauso einfach debuggt werden wie eine normale Windows-Anwendung. Es ist jederzeit möglich Haltepunkte zu setzen und von diesen Punkten an schrittweise weiter auszuführen.

Abbildung 8 (Debuggen einer Webmethode)

Die Infrastruktur des .NET-Frameworks verbirgt dabei den Quellcode der Proxy-Objekte und springt direkt an die richtige Stelle innerhalb des Webservices (also in der Regel direkt zur Web-Methode). Der Rücksprung in die Clientanwendung erfolgt analog.

5.9 Attribut DebuggerStepThrough

Um (wie im vorherigen Abschnitt beschrieben) bestimmten Quellcode beim Debuggen zu überspringen, wird ebenfalls ein Attribut eingesetzt. Eine Methode die durch das Attribut `DebuggerStepThrough()` markiert wird, wird beim Debuggen automatisch als ein Schritt ausgeführt.

Dieses Attribut wird insbesondere bei automatisch generiertem Quellcode (zum Beispiel bei den Proxy-Objekten oder beim Windows-Forms-Designer) eingesetzt.

5.10 Session-gestützte Webservices

Um die Beispiele abzurunden, insbesondere für die Nutzung im Studienprojekt, wird nun ein Session-gestützter Webservice erklärt. Die Session wird durch Cookies auf der Anwendungsseite ermöglicht. Das heißt der erste Aufruf generiert und speichert eine Session-ID, welche automatisch bei jedem weiteren Aufruf dem Server wieder zur Verfügung gestellt wird.

Der Webservice ist so ausgelegt, dass der Client sich beim ersten Zugriff authentifizieren muss. Andernfalls wird eine NotConnected-Exception ausgelöst. Ist der Client authentifiziert, können alle weiteren Web-Methoden aufgerufen werden.

Der Quellcode der Clientanwendung muss um einen Cookie-Container ergänzt werden, damit die Session-ID lokal gespeichert werden kann.

Listing 20 (C#-Quellcode für Initialisierung eines Session-gestützten Webservices)

```
Service serv = new Service();
serv.CookieContainer = new System.Net.CookieContainer();
serv.Connect("frank", "*****");
MessageBox.Show(serv.HelloWorld());
```

In der dritten Zeile wird die Connect-Methode aufgerufen, um den Benutzernamen und das Passwort in die Session einzutragen. Wäre diese Zeile nicht vorhanden, würde die folgende HelloWorld-Methode mit einer NotConnected-Exception fehlschlagen.

Werfen wir nun einen Blick auf den Quellcode der Serverseite.

Der wichtigste Teil passiert im Konstruktor, in dem eine Exception geworfen wird, falls nicht die Methode Connect aufgerufen wird und in der aktuellen Session noch kein Benutzer angemeldet ist.

Listing 21 (C#-Quellcode eines Konstruktors eines Session-gestützten Webservices)

```
public Service()
{
    HttpRequest request = this.Context.Request;
    if (request.PathInfo != "/Connect" && request.Params["HTTP_SOAPACTION"]
        != "\"http://tempuri.org/Connect\"")
    {
        if (Session["userName"] == null
            || Session["userName"].ToString() == "")
        {
            throw new NotConnectedException();
        }
    }
}
```

Zum Schluss betrachten wir die Methode Connect. Diese Methode prüft den angegebenen Benutzernamen und das Passwort auf deren Gültigkeit (Diese Methode wird in der Realität sicher etwas komplexer ausfallen als ein einfacher Textvergleich). Ist die Authentifizierung erfolgreich, wird in die Session der Benutzername gespeichert, sodass im Konstruktor für diese Session ab sofort keine Exceptions mehr geworfen wird.

Listing 22 (C#-Quellcode einer Webmethode Connect)

```
[WebMethod(EnableSession = true)]
public Boolean Connect(string userName, string password)
{
    if (userName == "frank" && password == "*****")
    {
        Session["userName"] = userName;
        return true;
    }
    else
    {
        return false;
    }
}
```

Die eigentlichen Nutz-Methoden kommen ohne zusätzliche Parameter aus. Sie müssen lediglich wie die Methode Connect durch das WebMethod-Attribut als Session-gestützt markiert werden.

Listing 23 (C#-Quellcode einer Webmethode HelloWorld)

```
[WebMethod(EnableSession = true)]
public string HelloWorld()
{
    return "Hello " + Session["userName"];
}
```

6 Technologie-Empfehlung für VC³

Für das Studienprojekt Virtual Construction Company Competition sind XML-Webservices von zentraler Bedeutung. Die Applikation soll öffentlich zugänglich über das Internet betrieben werden, es sollte jedoch aus Sicherheitsgründen keine Möglichkeit geben direkt auf die Datenbank zuzugreifen. Auch technisch wäre dies durch Firewalls vermutlich problematisch bis unmöglich.

Da der Kunde eine Windows-Serverumgebung nutzt, ist der Microsoft SQL-Server 2005 als Datenbankserver sinnvoll. Um den Zugriff auf die Datenbank zu ermöglichen, muss eine Schnittstellenschicht mit einem XML-Webservice geschrieben werden.

Microsoft Visual C# stellt sowohl für den Zugriff auf die SQL-Server-Datenbank, als auch zur Programmierung eines XML-Webservices für den Zugriff der Client-Anwendungen hervorragende Möglichkeiten bereit. Die Realisierung könnte zum Beispiel in Form eines Session-gestützten Webservices erfolgen.

Die Client-Applikationen können so über einen SSL-gesicherten Webservice auf die Datenbank zugreifen. Es ist sinnvoll alle Einstellungen ausschließlich auf dem Server zu speichern. Damit ist der Betrieb des Client in einer Partial-Trust-Umgebung möglich (Die Anwendung läuft mit eingeschränkten Sicherheitsberechtigungen, es ist kein Zugriff auf die lokale Festplatte möglich).

7 Literaturverzeichnis

1. **Foggon, D., et al.** *Programming Microsoft .NET XML-Webservices.* s.l. : Microsoft Press, 2004.

2. **Kemper, A. und Eickler, A.** Datenbanksysteme - Eine Einführung. s.l. : Oldenburg Verlag, 2006, S. Kapitel 19.

3. Wikipedia - Die freie Enzyklopädie. [Online] 22. 12 2007. [Zitat vom: 22. 12 2007.] http://de.wikipedia.org/wiki/Serviceorientierte_Architektur.

4. **Microsoft.** *Visual Studio 2008 Online-Dokumentation.* 2007.

8 Verzeichnis der Codebeispiele und Abbildungen

Listing 1 (XML-Dokument) ... 4
Listing 2 (XML- Dokument mit Attributen) ... 4
Listing 3 (XML-Dokument mit Dokumenttypdefinition) ... 4
Listing 4 (XML-Schema) ... 5
Listing 5 (XML-Dokument mit XML-Schema) ... 6
Listing 6 (XML-Dokument mit Verweisen) ... 6
Listing 7 (XML-Schema mit Verweisen) ... 7
Listing 8 (C#-Quellcode für serialisierbares Objekt) ... 7
Listing 9 (C#-Quellcode für Initialisierung eines Objekts) ... 8
Listing 10 (XML-serialisiertes Objekt) ... 9
Listing 11 (SOAP-Nachricht) ... 9
Listing 12 (SOAP-Anfrage) ... 10
Listing 13 (SOAP-Antwort) ... 10
Listing 14 (WSDL Teil 1) ... 11
Listing 15 (WSDL Teil 2) ... 11
Listing 16 (WSDL Teil 3) ... 11
Listing 17 (WSDL Teil 4) ... 11
Listing 18 (C#-Quellcode für Initialisierung eines Webservices) ... 12
Listing 19 (C#-Quellcode einer Webmethode) ... 16
Listing 20 (C#-Quellcode für Initialisierung eines Session-gestützten Webservices) ... 18
Listing 21 (C#-Quellcode eines Konstruktors eines Session-gestützten Webservices) ... 18
Listing 22 (C#-Quellcode einer Webmethode Connect) ... 18
Listing 23 (C#-Quellcode einer Webmethode HelloWorld) ... 19

Abbildung 1 (Zusammenwirkung von WSDL, Proxyobjekten und SOAP-Nachrichten) ... 12
Abbildung 2 (Zusammenwirkung von WSDL, Proxyobjekten, SOAP-Nachrichten und UDDI) ... 13
Abbildung 3 (Bearbeiten von XML-Dateien) ... 14
Abbildung 4 (Darstellung eines XML-Schemas) ... 14
Abbildung 5 (Erstellen eines neuen Webservices) ... 15
Abbildung 6 (Visual Studio integrierter Webserver) ... 15
Abbildung 7 (Ergebnis eines Browseraufrufs eines Webservices) ... 16
Abbildung 8 (Debuggen einer Webmethode) ... 17