JMT Libraries & Web Services
It's all about types

Web Services[1] are a vital component in today's business landscape, particularly when one considers the growth of mobile devices in B2C scenarios. Even in B2B, particularly where the "service consuming" side needs to blend together data streams from a variety of providers, the use of Web Services is central to the delivery of applications or web content. Often Web Services are referred to as APIs (Application Programming Interface) giving a revised, modern spin to terminology that has been around for quite some time. In this TechNote we will use the terms API to mean concretely a Web Service interface as it would be described, for example, in a Web Services Definition Language (WSDL)[2] file.

Web Services (APIs)

In the WSDL description of the Services, the natural type landscape expected is that defined by Schema. If types in our API were indeed those from a W3C compliant Schema[3], what would be the outcome for the development team?

- consistent type landscape across both the application and API design
- no need to define the "meaning" of the types over and above the strictly business meaning
- valid data range of the types is already established and enforced
- focus can switch away from message-level implementation and more fully to the business logic and workflow, blending the API calls to get the desired data composition and business outcome

However, using traditional semi-manual approaches it can prove impossible to embrace the necessary type landscape.
So how can this situation be resolved in the real world?

JMT Types

All JMT types start life as definitions in a Schema. Large international communities like ISO[4], OpenTravel[5], IATA[6], HL7[7] and so on, spend considerable efforts bringing their respective Schemas into being to describe domain-specific business data items. Traditionally, working with these Schema sets has been difficult for the implementer/developer. Significant business involvement is required to get a clear understanding of the data definitions. In addition significant Schema knowledge is needed by the developer when using the current semi-manual process of class construction.
On the other hand, JMT do away completely with these defective implementation processes. In particular we can highlight:

- the type landscape matches fully the definitions in the associated Schema set
- the developer simply uses JMT types as any other type in their development environment

[1] http://www.w3schools.com/Webservices/default.asp
[2] http://www.w3.org/TR/wsdl
[3] http://www.w3.org/XML/Schema
[4] http://www.iso20022.org/
[5] http://www.opentravel.org/
[6] http://www.iata.org/Pages/default.aspx
[7] http://www.hl7.org/

- full documentation of the types and related Schema entities is provided
- validation of data, whether it appears programmatically or via a UI, is in-the-box for all the JMT types – data quality is assured.

Schemas and Web Services in the Real World

Schemas to be used in a web application scenario are often accompanied by a set of WSDL[8] files. While a Schema is a description of the data within the business domain (describes what the core entities are in the business domain), the WSDL provides a specification of the operations that can be performed on that data (what actions can be done using the data from the business domain). In a real business case both files are provided by a standardization institution or a 3rd party provider and cannot be subject to change. In such cases binding the data (schemas) with the operations (WSDL) from an application perspective may be a challenging task and require the consumer to adopt a specific development technique called Web Service Contract First (WSCF).

The problem that a development team or business entity faces, which requires them to adopt a WSCF technique, is how to communicate with a 3rd party service provider using the specific schema set and the WSDL file. The real challenge is how to create an application that is reliable and (most importantly) easy to maintain. Because of the sheer size of the schemas and WSDL files as well as their complexity, a solution that is based on *generated code* may not meet these requirements.

The Microsoft .NET Framework, as well as the community, provides some out-of-the-box solutions to tackle this problem. Unfortunately, none of them is satisfactory. The two most widely used solutions are Service Model Metadata Utility Tool (Svcutil.exe)[9] provided with the Microsoft Windows SDK, and a community-driven tool from Thinktecture, WSCF.blue[10]. We will now look at the advantages and limitation of both tools.

Limitations of the SvcUtil application stems from its relation to the Xsd.exe tool, also in the Microsoft Windows SDK, over which it forms a wrapper. All the issues encountered when using Xsd.exe are still present when using SvcUtil. By adopting this tool to generate code from a schema set, one ends up with an application that contains large amount of generated code *which doesn't actually match the original schema data definitions at all*. More importantly, code reusability in such a solution is really low or even non-existent. In purely technical terms – in some cases it's not possible to reuse an existing type library. In the .Net world, however, Xsd.exe has the main advantage of simply being the "standard" tool in the SDK.

WSCF.blue which is provided by the community solves some of the problems described in the previous paragraph. Yet in a complex scenario (using a complicated schema set provided by one of the standardization organizations) one would be, again, left with hundreds of lines of generated code which leads to a maintenance nightmare. The advantage, if it can indeed be described as one in this case, is that the tool codebase is open to adaption. Such a step is not particularly attractive though for projects who do not want to be diverted away from their core business development concerns.

Jmt.Wscf to the Rescue

To solve the problems outlined above and overcome limitations of the existing tools; Jet Messaging Technologies proposes a solution that delivers an elegant, complete yet maintainable solution. Along with an existing range of type libraries (see jet-messaging.com/products for a comprehensive list) that cover schemas from various providers, we propose a simple and robust extension for the Microsoft Visual Studio development environment[11]. The aim of extension is to provide a mechanism to create Web Service client code, bringing together the data and operational description part, XSD and WSDL, with a generated type landscape, the JMT dll, that is both *easy to maintain* and covers the *complete landscape of a particular schema set*. The User Interface (UI) of the extension is presented in below Figure 1.

[8] WSDL – Web Services Definition Language, http://www.w3.org/TR/wsdl
[9] http://msdn.microsoft.com/en-us/library/aa347733(v=vs.110).aspx
[10] http://wscfblue.codeplex.com
[11] http://msdn.microsoft.com/en-us/library/bb165336.aspx

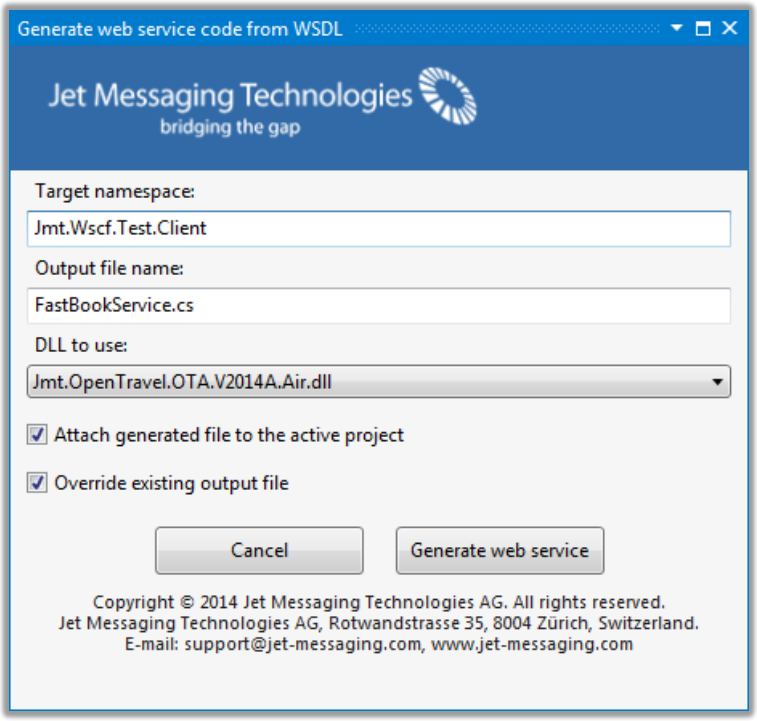

Figure 1 The User Interface of the JMT WSCF extension

The Visual Studio extension shown above works in the context of a WSDL file attached to a Visual Studio project (the next section will describe this in more detail). Please note that it discovers any JMT Library that is already referenced by the project and invites the user to select one to be used in its operation. The only responsibility of the user is to provide the desired namespace as well as the name of the output file.

The "Generate web service" button creates a set of classes that use types from the referenced JMT Library. These classes represent a fully functional web service client that leverages the Microsoft WCF[12] technology. *No further modifications are needed to implement the WS client*. The only task for the developer is to implement the business logic that operates on the data provided by the server and responds to it accordingly.

In the following section we will go through an example that shows how the extension along with the JMT Type Library can be used to provide a solution for a typical business scenario.

An Example Generation

An example to illustrate the benefits of using the JMT Type Library in conjunction with the Jmt.Wscf Visual Studio extension, is now presented. The (mythical) functional requirement is to implement client code that could allow communication via a web service using SOAP messages that adhere to data definitions in the schemas. This will form the basis of the discussion in this section.

This example will use the Schema `OTA_AirAvailRQ.xsd` file from the OpenTravel public delivery. We also use a simplified WSDL file for the purposes of demonstrating how to use the JMT type libraries in a Web Service scenario.

What is required is to bind the type landscape in the JMT Library with the WSDL and generate appropriate client-side starter code for invoking supported services. As was mentioned above, this cannot be achieved by using SvcUtil.exe or alternatives such as WCSF.Blue, so we use Jmt.Wscf to provide the solution. Our example solution looks as follows in Figure 2:

[12] WCF - http://msdn.microsoft.com/en-us/library/vstudio/ms735119(v=vs.90).aspx

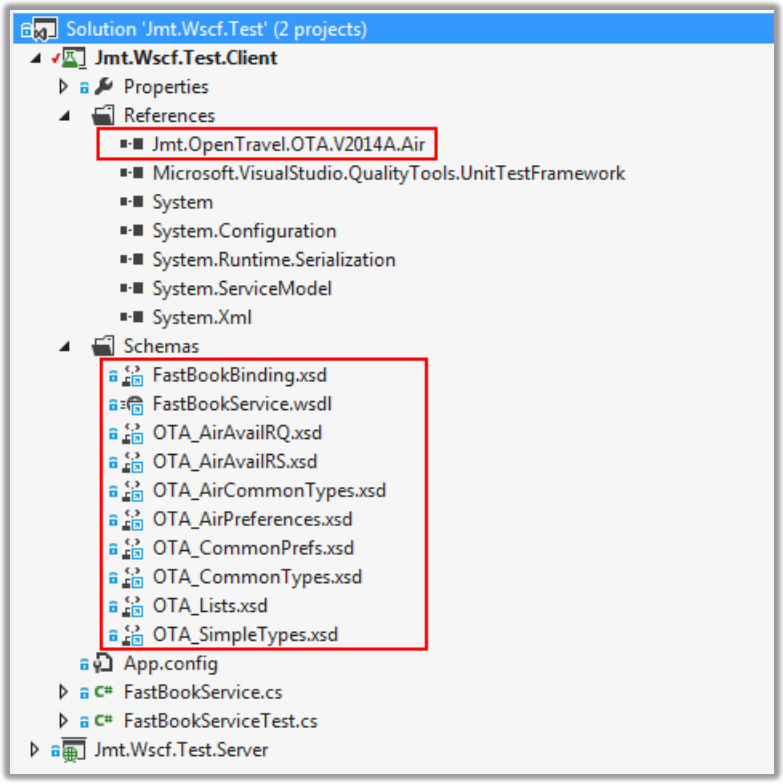

Figure 2 OpenTravel Schemas and JMT Library

Elements of the Example

The following parts comprise our example solution:

- Visual Studio extension
- A WSDL file (FastBookService.wsdl) – that describes the WS operations and their parameters. In our example one is described in more detail below (see "The WSDL file").
- A set of XSD files – describes data that is used by the WS operations. In our example we have a hierarchy of schema files which start with the FastBookBinding.xsd schema which in turn references the other schemas.
- JMT Library – a library of classes that reflect the data definitions in the schemas as strongly-typed classes. In this particular example a library from the OpenTravel OTA V2014A specification, the Air dictionary, is used.

We will describe each of the elements in detail in the following sections.

The WSDL File

The WSDL file used in this example is a simplified version of the *`FastBookService.wsdl`* file that can be found in the OpenTravel specification. In particular, this simplified WSDL declares one operation, `CheckAvailability` that involves two messages, `OTA_AirAvailRQ`, the (input) request and, `OTA_AirAvailRS`, the (output) response.

FastBookService.wsdl

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:ota="http://www.opentravel.org/OTA/2003/05"
3                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
4                    xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.opentravel.org/FastBook/ws/v1">
5    <!-- import the schema definitions here -->
6    <wsdl:types>
7      <xs:schema>
8        <xs:import namespace="http://www.opentravel.org/OTA/2003/05" schemaLocation="FastBookBinding.xsd"/>
9      </xs:schema>
10   </wsdl:types>
11
12   <!-- define the messages here -->
13   <!-- Availability -->
14   <wsdl:message name="CheckAvailabilityRequest">
15     <wsdl:part name="OTA_AirAvailRQ" element="ota:OTA_AirAvailRQ"/>
16   </wsdl:message>
17   <wsdl:message name="CheckAvailabilityResponse">
18     <wsdl:part name="OTA_AirAvailRS" element="ota:OTA_AirAvailRS"/>
19   </wsdl:message>
20
21   <!-- declare the operations and virtual port for FastBook messages -->
22   <wsdl:portType name="FastBookPortType">
23     <wsdl:operation name="CheckAvailability">
24       <wsdl:input message="frz1:CheckAvailabilityRequest"/>
25       <wsdl:output message="frz1:CheckAvailabilityResponse"/>
26     </wsdl:operation>
27   </wsdl:portType>
28   <!-- bind the messages and the port together -->
29   <wsdl:binding name="FastBookBinding" type="frz1:FastBookPortType">
30     <!-- Document Style not RPC interface -->
31     <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
32     <!-- Declare WSDL operations -->
33     <wsdl:operation name="CheckAvailability">
34       <soap:operation soapAction="FastBookAvailability" style="document"/>
35       <!-- I had to add the message explicitly in order to make this working -->
36       <wsdl:input message="frz1:CheckAvailabilityRequest">
37         <soap:body use="literal"/>
38       </wsdl:input>
39       <wsdl:output message="frz1:CheckAvailabilityResponse">
40         <soap:body use="literal"/>
41       </wsdl:output>
42     </wsdl:operation>
43   </wsdl:binding>
44   <!-- declare the physical endpoint for the service -->
45   <wsdl:service name="FastBookService">
46     <wsdl:port name="FastBookPort" binding="frz1:FastBookBinding">
47       <soap:address location="http://fastbookpartner.com"/>
48     </wsdl:port>
49   </wsdl:service>
50 </wsdl:definitions>
```

Figure 3 FastBookService Web Service Definition file (WSDL)

A number of points should be noted about this WSDL:

- line 8: the schema file (FastBookBinding.xsd) that contains the definitions of the data contract messages, is imported to the document
- lines 14-19: two messages (which transform to message contracts in the .NET world) are defined. Note that they use types defined in the Schema by using the "ota:" namespace prefix
- lines 23-26: an operation (CheckAvailabiltyRequest) is defined
- lines 45-50: the FastBookService service is defined

It is important to emphasize here that *all aspects* of the WSDL file are covered either by the JMT Types Library (including all the types imported from the *FastBookBinding.xsd* schema) or the Jmt.Wscf extension (generation of classes that represent the message contract as well as the code of the web service client itself). So here we see the binding of a generated type landscape with a WSDL file that references appropriate schemas and their contained types (by definition).

The Schema

In the previous WSDL file Figure 3 it can be seen that one of the declared messages, named `CheckAvailabilityRequest`, has a type which is defined in the Schema (via the namespace prefix `ota:`) `FastBookBinding.xsd` with a concrete type of `OTA_HotelAvailRQ`.
Currently, a developer would employ the semi-manual process mentioned above, using tools like Xsd.exe, to produce

types such as this from the Schema definition. This approach has a number of *very serious* drawbacks:

- hard to scope the amount of work involved for all the types required
- processing a schema containing message-related data definitions, results in classes that are very poor reflections of the corresponding definition, important aspects such as documentation, type structure, optional/mandatory indication and min/max-Occurs values are all missing
- the validation of data must be coded manually for all types using the Schema definitions as a guide (not forgetting the W3C Primitive types, of course)
- the manual parts need to be maintained should the Schema definitions change
- the work of type development is a serious distraction from the work on the business logic and use cases in the application or service being developed

It should be remarked here that it is a misunderstanding to see the need for types to be restricted just to the top-level message, `CheckAvailabilityRequest` in this example. Schema types operate like icebergs in that to get a *genuine* representation of a single message type, many, many more types *must* be written in the development environment.
If we now look for our type in the Schema OTA_AirAvailRQ`.xsd`, we see as below in Figure 4:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.opentravel.org/OTA/2003/05"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.opentravel.org/OTA/2003/05"
           elementFormDefault="qualified" version="5.000" id="OTA2014A">
    <xs:annotation>
        <xs:documentation source="Description"
            xml:lang="en">ALL SCHEMA FILES IN THE OPENTRAVEL ALLIANCE SPECIFICATION ARE MADE AVAILABLE ACCORDING TO
    </xs:annotation>
    <xs:include schemaLocation="OTA_AirPreferences.xsd"/>
    <xs:element name="OTA_AirAvailRQ">
        <xs:annotation>
            <xs:documentation
                xml:lang="en">The Availability Request message requests Flight Availability for a city pair on a sp
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="POS" type="POS_Type">
                <xs:element name="ProcessingInfo" minOccurs="0">
                <xs:element name="MultimodalOffer" type="MultiModalOfferType" minOccurs="0">
                <xs:element name="OriginDestinationInformation" maxOccurs="99">
                <xs:element name="SpecificFlightInfo" type="SpecificFlightInfoType" minOccurs="0">
                <xs:element name="TravelPreferences" type="AirSearchPrefsType" minOccurs="0">
                <xs:element name="TravelerInfoSummary" type="TravelerInfoSummaryType" minOccurs="0">
                <xs:element name="BookedFlightSegment" type="BookFlightSegmentType" minOccurs="0" maxOccurs="99">
                <xs:element name="Offer" type="AirOfferChoiceType" minOccurs="0">
            </xs:sequence>
            <xs:attributeGroup ref="OTA_PayloadStdAttributes"/>
            <xs:attributeGroup ref="MaxResponsesGroup">
                <xs:attributeGroup ref="DirectAndStopsGroup"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 4 OTA_AirAvailRQ Message Definition

Here we can start to get a sense of the iceberg quality of Schema types.
Within `OTA_AirAvailRQ`, the type we are interested in for our service message, we can encounter types such as:

- POS_Type, OriginDestinationInformation
  these are "complex" types, appearing as xs:element entities, defined in additional Schemas and which contain further type collections. These too contain further types, sometimes in complicated structural forms such as xs:sequence, xs:choice, xs:union etc. as well as extensions and restrictions. These may also contain "extension" types (TPA_Extensions in OpenTravel terms) which allow for custom data to be introduced into messages in a structured way
- StringLength1to128
  this is also defined in an additional Schema and represents a so-called SimpleType. This type contains data validation

as well as restrictions/restriction collections and eventually, in its XML form, will hold an appropriate "value"

- xs:dateTime, xs:language and xs:positiveInteger
  these are types that are defined by the W3C consortium themselves, the so-called XSD Primitive set. In this case they represent a date/time value (in a specific format), a "language" identifier and an Integer whose value must be positive to be valid. For JMT these special types are also generated just as any other, matching their W3C specification.

The JMT Library

For the purpose of this TechNote we will use the OpenTravel V2014A Air dictionary library, which can be downloaded from our website.

Generating the Web Service Client Code

Once the Visual Studio project has been set up (as shown above) and the Jmt.Wscf extension downloaded and installed, it is possible to generate the client code for a web service.
Proceed as follows; right click on the WSDL file and select the *Generate Web Service* menu item, as shown below:

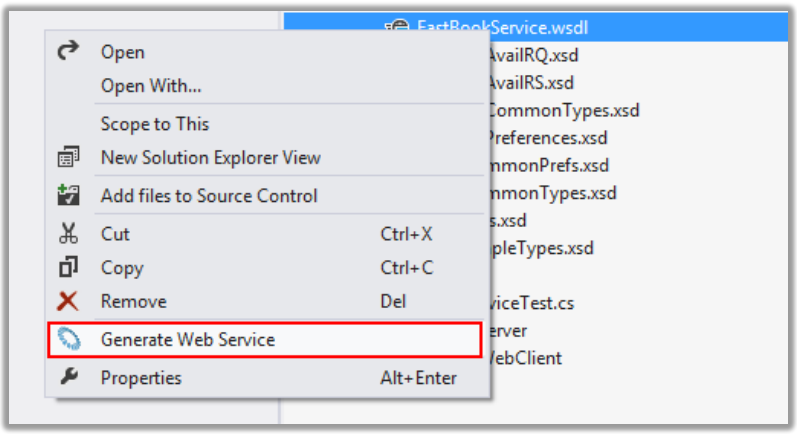

Figure 5 Visual Studio context menu action for running the JMT WSCF extension

Next enter the target namespace and the name of the output file in the *Generate web service code from WSDL* dialog as shown below. It is also possible to choose:

- whether you would like to add a reference to the generated file in the current Visual Studio project
- if any existing file with the same name should be overwritten.

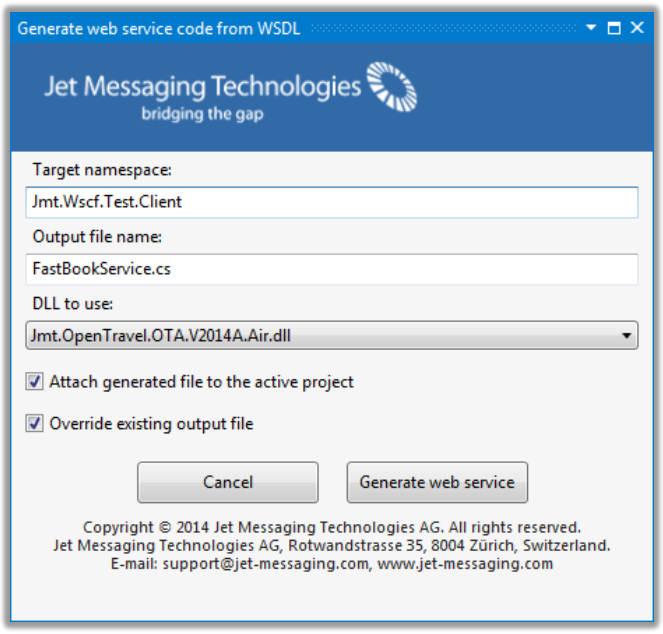

Figure 6 The user interface of the JMT WSCF extension

Click the *Generate web service* button. A confirmation dialog is displayed indicating that the web service client code has been successfully generated.

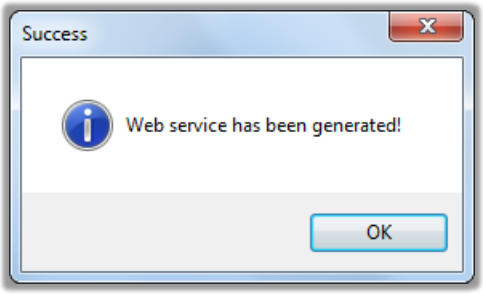

Figure 7 Information window after successful code generation

Before we dive deeper into the technical aspects of the generated code, let's recap the whole process and what was actually done by the extension:

- The Visual Studio extension combines the information about the web service (WSDL file) and the data used by it (a set of XSD files) and generates C# classes that reflect both of these.
- Instead of generating hundreds of lines of data contract classes the extension simply discovers types in the JMT Type Library referenced by the Visual Studio project.
- What is generated is purely infrastructural code that allows an application to communicate with a web service. The only missing part is the actual business logic code that implements the business functional requirements.

The Generated Code

For our example, the file **`FastBookService.cs`** is written and injected into the Visual Studio project. A fragment of this C# file is shown below Figure 8.

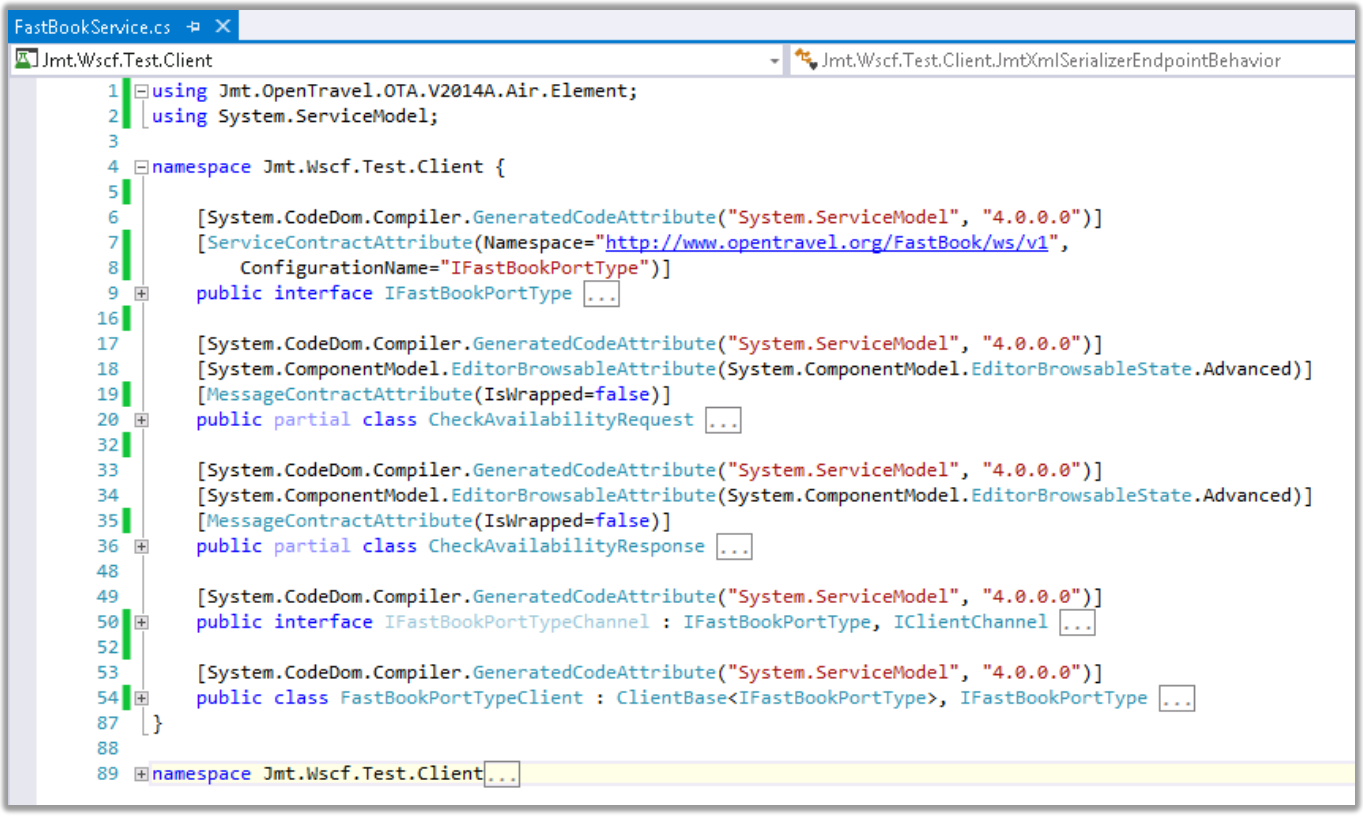

Figure 8 Generated web service client code (fragment)

Note that this represents *complete source code of a web service client* – for our particular case the file is roughly 100 lines of

code. The source code for a similar web service generated by the WSCF.blue or simillar tooling would contain around 80.000 lines of generated code (code which would also have the key deficiencies noted earlier).

The file shown above is (by design) divided into two parts. The first part is the web service code client that contains the following classes:

- IFastBookPortType – the interface (contract definition) of the web service.
- CheckAvailabilityRequest and CheckAvailabilityResponse – message contract classes that represent the messages described in the WSDL file.
- FastBookPortTypeClient – a class that implements the interface and inherits from the WCF's ClientBase infrastructure class. This is the actual web service client code where the logic that covers the business requirement needs to be implemented.

Taking a closer look at one of the *message contract* classes (`CheckAvailabilityRequest`) Figure 9,

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
[System.ComponentModel.EditorBrowsableAttribute(System.ComponentModel.EditorBrowsableState.Advanced)]
[System.ServiceModel.MessageContractAttribute(IsWrapped = false)]
public partial class CheckAvailabilityRequest
{

    [System.ServiceModel.MessageBodyMemberAttribute(Namespace = "http://www.opentravel.org/OTA/2003/05", Order = 0)]
    public OTA_AirAvailRQ OTA_AirAvailRQ;
           class Jmt.OpenTravel.OTA.V2014A.Air.Element.OTA_AirAvailRQ
    public
    {
    }

    public CheckAvailabilityRequest(OTA_AirAvailRQ OTA_AirAvailRQ)
    {
        this.OTA_AirAvailRQ = OTA_AirAvailRQ;
    }
}
```

Figure 9 Generated message contract class (fragment)

it should be noted that the message contract class is the *only one* that is generated. The whole inner structure (the `OTA_AirAvailRQ` property of the `OTA_AirAvailRQ` type) comes from the referenced JMT OpenTravel Types Library (Jmt.OpenTravel.OTA.V2014A.Air). There is no further message-related implementation aspects to be done by the developer who can concentrate on implementing the business logic inside the `FastBookPortTypeClient` web service client class.
The second part of the file is as shown below in Figure 10.

```
/// <summary>
/// Sets the <see cref="JmtXmlSerializerFormatter"/> as the formatter for a client operation.
/// </summary>
public class JmtXmlSerializerFormatterBehavior : IOperationBehavior...

/// <summary>
/// Default formatter to be used with types from Jet Messaging Technologies libraries. The formatter works on message contract classes only
/// when the message is serialized. Each property of such class which is of type that implements the <see cref="IXmlSerializable"/> interface
/// is unwrapped and put directly into the SOAP body without any wrapping XML elements. The deserialization process is left untact.
/// </summary>
internal class JmtXmlSerializerFormatter : IClientMessageFormatter...

/// <summary>
/// An endpoint behavior which applies the <see cref="JmtXmlSerializerFormatterBehavior"/> to all operations
/// of the endpoint.
/// </summary>
internal class JmtXmlSerializerEndpointBehavior : BehaviorExtensionElement, IEndpointBehavior...
```

Figure 10 Generated infrastructure class (fragment)

These three classes are purely infrastructural and their role is not described further.

It's also worth noting that the configuration file of the project was changed so that an endpoint referenced in the WSDL file as well as the previously mentioned infrastructural classes are registered within the WCF runtime. These modifications are shown below in Figure 11.

App.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name="jmtEndpointBehavior">
          <jmtEndpointBehaviorExtension />
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <bindings>
      <basicHttpBinding>
        <binding name="FastBookBinding" />
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost/JmtWscfTestServer/FastBookPortType.svc"
                behaviorConfiguration="jmtEndpointBehavior"
                binding="basicHttpBinding" bindingConfiguration="FastBookBinding"
                contract="IFastBookPortType" name="FastBookPort" />
    </client>
    <extensions>
      <behaviorExtensions>
        <add name="jmtEndpointBehaviorExtension"
             type="Jmt.Wscf.Test.Client.JmtXmlSerializerEndpointBehavior,
                   Jmt.Wscf.Test.Client, version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

Figure 11 Application configuration file with additional settings for the web service

Running the example

To provide an opportunity to see the generated code in action we have included a sample server in the source code of the example. The `Jmt.Wscf.Test.Server` project contains an implementation of a simple server that leverages WCF technology. An important point regarding this implementation is that it doesn't use *any* JMT Library types whatsoever. For simplicity of the example the server code operates on "pure" XML but it could have been generated using one of the technologies available on the market.

The data contract classes in the FastBookService.cs file provide a way to test the functionality. Although they don't reflect the whole type landscape in the schema and are not intended to represent production code, they nonetheless provide a realistic test of the approach. We use it for presentational purpose to show that a web service implementation that uses JMT Type Library is not limited in any way.

Client Code

The `FastBookServiceTest` class located in the `Jmt.Wscf.Test.Client` project contains one test method. The sole responsibility of this is to call the sample server and check the response. The code that can be seen in below Figure 12 instantiates a web service proxy class (`FastBookPortTypeClient`), creates and populates an `OTA_AirAvailRQ` message and finally calls the web service.

```
[TestMethod]
public void CheckAvailabilityRequest_Test()
{
    var request = new OTA_AirAvailRQ();

    request.DirectAndStopsGroup.DirectFlightsOnly = new XsdBoolean() { Text = "true" };
    request.MaxResponsesGroup.MaxResponses = new PositiveInteger() { Text = "10" };
    request.OTA_PayloadStdAttributes.PrimaryLangID = new Language() { Text = "en-us" };
    request.OTA_PayloadStdAttributes.EchoToken = new StringLength1to128Type() { Text = "12345" };
    request.OTA_PayloadStdAttributes.TimeStamp = new XsdDateTime() { Text = DateTime.UtcNow.ToString("o") };
    request.OTA_PayloadStdAttributes.Version = new XsdDecimal() { Text = "2.001" };
    request.OTA_PayloadStdAttributes.SequenceNmbr = new NonNegativeInteger() { Text = "1" };

    var sequence = request.OTA_AirAvailRQSeq.SequenceFirst;
    var pos = sequence.POS.POS_TypeSeq.SequenceFirst;
    var source = request.OTA_AirAvailRQSeq.SequenceFirst.POS.POS_TypeSeq.SequenceFirst.SourceList.SequenceFirst;

    source.AgentSine = new StringLength1to16Type() { Text = "BSIA1234PM" };
    pos.SourceList.SequenceFirst.PseudoCityCode = new StringLength1to16Type() { Text = "24R" };
    pos.SourceList.SequenceFirst.ISOCountry = new ISO3166Type() { Text = "US" };
    pos.SourceList.SequenceFirst.ISOCurrency = new AlphaLength3Type() { Text = "USD" };

    var originDestinationInfo = sequence.OriginDestinationInformationList.SequenceFirst.OriginDestinationInformationTypeSeq.SequenceFirst;
    originDestinationInfo.DestinationLocation.SimpleTypeExtension.LocationGroup.LocationCode = new StringLength1to16Type() { Text = "LAX" };
    originDestinationInfo.OriginLocation.SimpleTypeExtension.LocationGroup.LocationCode = new StringLength1to16Type() { Text = "LHR" };

    var client = new FastBookPortTypeClient();
    var response = client.CheckAvailability(request);
}
```

Figure 12 Client-side implementation of the web service example

Server Code

The web service implementation on the server side creates and populates a response message (`CheckAvailabilityResponse`) and returns it to the client. Please note that the server implementation doesn't use any specific library to generate the SOAP response but rather creates and returns a raw XML message. In a real-world scenario the server code could be implemented using any technology (including .NET and Java framework and libraries).

```
public virtual Message CheckAvailability(Message request)
{
    // To keep the example simple the response message body is constructed from a raw XML content.

    var xml =
        "<OTA_AirAvailRS EchoToken=\"TX-66583\" TimeStamp=\"2014-06-25T12:15:48\" TargetName=\"Cray-X1\" Version=\"6.000\" " +
        "   TransactionIdentifier=\"TX7755-120-00-8876\" SequenceNmbr=\"997529\" RetransmissionIndicator=\"true\" CorrelationID=\"X-66984-B1\" " +
        "   PrimaryLangID=\"en-US\" AltLangID=\"fr-FR\" xmlns=\"http://www.opentravel.org/OTA/2003/05\">"  +
        "   <Success />"  +
        "   <OriginDestinationInformation>"  +
        "       <DepartureDateTime>2003-08-13T10:30:00</DepartureDateTime>"  +
        "       <OriginLocation LocationCode=\"LHR\" />"  +
        "       <DestinationLocation LocationCode=\"LAX\" />"  +
        "       <OriginDestinationOptions>"  +
        "           <OriginDestinationOption>"  +
        "               <FlightSegment Ticket=\"eTicket\" AccumulatedDuration=\"PT2M10S\" Distance=\"2189\" CodeshareInd=\"true\" FlifoInd=\"false\" " +
        "                   DateChangeNbr=\"-5\" FlightNumber=\"837\" FareBasisCode=\"BBX\" DepartureDateTime=\"2003-08-13T10:30:00\" " +
        "                   ArrivalDateTime=\"2003-08-13T11:14:00\">"  +
        "                   <DepartureAirport LocationCode=\"LHR\" Gate=\"X04\" />"  +
        "                   <ArrivalAirport LocationCode=\"LAX\" CodeContext=\"IATA\" Terminal=\"Concourse A\" Gate=\"B12\" />"  +
        "                   <Equipment AirEquipType=\"744\" ChangeofGauge=\"false\" />"  +
        "                   <TrafficRestrictionInfo Code=\"101.EQP\" Language=\"fr-FR\" />"  +
        "                   <Comment />"  +
        "                   <MarketingCabin RPH=\"1\" CabinType=\"First\">"  +
        "                       <Meal MealCode=\"L\" />"  +
        "                       <BaggageAllowance UnitOfMeasureQuantity=\"2\" UnitOfMeasure=\"meters\" UnitOfMeasureCode=\"101.EQP\" />"  +
        "                   </MarketingCabin>"  +
        "                   <MarketingCabin RPH=\"2\" CabinType=\"Business\">"  +
        "                       <Meal MealCode=\"L\" />"  +
        "                       <BaggageAllowance UnitOfMeasureQuantity=\"1.2\" UnitOfMeasure=\"meters\" UnitOfMeasureCode=\"101.EQP\" />"  +
        "                   </MarketingCabin>"  +
        "                   <MarketingCabin RPH=\"3\" CabinType=\"Economy\">"  +
        "                       <Meal MealCode=\"L\" />"  +
        "                       <BaggageAllowance UnitOfMeasureQuantity=\"0.5\" UnitOfMeasure=\"meters\" UnitOfMeasureCode=\"101.EQP\" />"  +
        "                   </MarketingCabin>"  +
        "                   <BookingClassAvail ResBookDesigCode=\"F\" ResBookDesigQuantity=\"5\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"1\" />"  +
        "                   <BookingClassAvail ResBookDesigCode=\"C\" ResBookDesigQuantity=\"8\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"2\" />"  +
        "                   <BookingClassAvail ResBookDesigCode=\"J\" ResBookDesigQuantity=\"3\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"2\" />"  +
        "                   <BookingClassAvail ResBookDesigCode=\"Q\" ResBookDesigQuantity=\"9\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"3\" />"  +
        "                   <BookingClassAvail ResBookDesigCode=\"V\" ResBookDesigQuantity=\"5\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"3\" />"  +
        "                   <BookingClassAvail ResBookDesigCode=\"N\" ResBookDesigQuantity=\"3\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"3\" />"  +
        "                   <BookingClassAvail ResBookDesigCode=\"Y\" ResBookDesigQuantity=\"3\" ResBookDesigStatusCode=\"101.EQP.X\" RPH=\"3\" />"  +
        "                   <StopLocation LocationCode=\"Schippol Intl.\" CodeContext=\"ARC\" />"  +
        "               </FlightSegment>"  +
        "           </OriginDestinationOption>"  +
        "       </OriginDestinationOptions>"  +
        "   </OriginDestinationInformation>"  +
        "</OTA_AirAvailRS>";

    return MessageConverter.FromXml(xml, "CheckAvaiabilityResponse");
}
```

Figure 13 Server-side XML reply message

Running the unit test proves that the communication between the client (which uses JMT Type Library and a proxy class generated by the JMT Extension) and the server (that uses any technology) works. You may see how the response returned by the server was deserialized and mapped to corresponding schema type from JMT Type Library:

```
[TestMethod]
public void CheckAvailabilityRequest_Test()
{
    // Create an object representing the message that we want to send.
    var request = new OTA_AirAvailRQ();

    [Filling the request object with data]

    /* Construct the proxy class object and send the message to the server */
    var client = new FastBookPortTypeClient();
    var response = client.CheckAvailability(request);

    Assert.IsNotNull(response);

    /* The EchoToken attribute defined in the OTA_PayloadStdAttributes attribute group should have the same value as the sent one.
     * The same rule applies to the Version and PrimaryLangID attributes
     */
    Assert.AreEqual(request.OTA_PayloadStdAttributes.EchoToken.Text, response.OTA_PayloadStdAttributes.EchoToken.Text);
                                                    request.OTA_PayloadStdAttributes.EchoToken.Text "12345"

    Assert.AreEqual(request.OTA_PayloadStdAttributes.PrimaryLangID.Text, response.OTA_PayloadStdAttributes.PrimaryLangID.Text);

    Assert.AreEqual(request.OTA_PayloadStdAttributes.Version.Text, response.OTA_PayloadStdAttributes.Version.Text);

    [Other assertions]
}
```

Figure 14 Result of running the web service with a debugger attached

Writing an Application

The code generated by the Jmt.Wscf Visual Studio extension is intended to be a scaffolding code that you can take and use. Although it is fully functional and can be used in a production environment there are a number of aspects you *might* want to change:

- In terms of code readability it would be preferred to split the generated file, moving each class to a separate file.
- In terms of project structure it is advised to separate the contract (the service interface and the message contract classes) from the implementation (the service implementation which derives from the `ClientBase` class).
- In terms of class dependencies remove the inheritance hierarchy in the service implementation in order to use an inversion of control mechanism to resolve dependencies and to automate the process of proxy creation (see Castle Windsor[13] or Microsoft Unity[14]).
- In terms of maintainability move all generated classes which take part in the creation of the SOAP message (`JmtXmlSerializerFormatterBehavior`, `JmtXmlSerializerFormatter` and `JmtXmlSerializerEndpointBehavior`) to a separate infrastructure project.

Benefits of this Solution

To summarize, the main advantages that come with using the Jmt.Wscf extension in conjunction with an appropriate JMT Type Library, are as follows:

- Projects and developers can take fully advantage of the strong type, data validating libraries from JMT.
- The initial creation of a Web service client which supports the *full range of standard operational messages* (as defined in a Schema and WSDL files) is trivial.
- Projects can now process *standard messages* leaving special needs to edge cases where, as is the case for OpenTravel, use can be made of the clearly defined extension points in messages for handling such "one off" partner agreements.
- Developers are empowered to do really fast and robust contract-first development in response to complex business needs.
- The Visual Studio solution has less code which leads to project with a significantly higher maintainability level.
- Projects can use a consistent and complete landscape of types throughout.

The Jet Messaging Technologies team hopes that the solution proposed here will bring major relief to developers faced with the task of producing a Web Service solution in the context of International Schema sets. Liberating developers from writing mundane boilerplate code, particularly when it has major deficiencies, allowing them to concentrate on the business value parts of the project, will really bring a major win.

[13] http://www.castleproject.org/projects/windsor/
[14] http://msdn.microsoft.com/en-us/library/ff647202.aspx

Jet Messaging Technologies AG
Rotwandstrasse 35, 8004 Zurich,
Switzerland
Phone +41 79 303 05 63

Email info@jet-messaging.com
www.jet-messaging.com