DATAWORKS ENTERPRISE

CUSTOMER USER GUIDE

WEB SERVICE VERSION 1.0

Document Version 0.1.0
Date of issue: May 14, 2010

THOMSON REUTERS

CONTENTS

1 Introduction 3
1.1 Intended Audience 3
1.2 What are web services? 3
1.2.1 Why use web services? 3
1.2.2 How might web services be used? 3
1.3 Web service developer skills 3
1.4 Useful web services references 4
1.5 Web Services Requests in the Context of Dataworks Enterprise 4
1.6 Contents of this document 4
2 The Dataworks Enterprise Web Service Interfaces 5
2.1 Request Parameters 5
2.1.1 User Data Parameter 5
2.1.2 Request Data Parameter 5
2.1.3 Request Flags Parameter 6
2.2 Advice on Choosing the Most Appropriate interface 7
2.3 Version() 7
2.4 The RequestRecord() and RequestRecords() methods 7
2.4.1 Record Type 8
2.4.2 StatusType type 9
2.4.3 StatusCode type 9
2.4.4 Field Type 10
2.4.5 Handling Errors and Exceptions 10
2.5 The RequestRecordAsXml() and RequestRecordsAsXml() methods 11
3 Example Client Code 14
3.1 Windows .NET client 14
3.1.1 Calling the GetRecord() and GetRecords() methods 14
3.1.2 Calling the GetRecordAsXml and GetRecordsAsXml methods 15
3.1.3 Using XSD.exe in combination with the Request…AsXml() methods 17
3.1.4 Writing Asynchronous .NET code 19
3.2 Java clients 19
3.2.1 Apache Axis based 19
3.2.2 Sun ONE 22
3.2.3 IBM Websphere 22
3.3 Microsoft Office 22
3.4 Microsoft C++ client 23
3.5 Other client types 23
4 General System Guidelines 24
4.1 Coding Guidelines 24
4.1.1 Reducing Dependencies 24
4.1.2 Client design 24
4.2 Responsible Use 25
4.2.1 During Development 25
5 Service Levels 26
5.1 Service Availability 26
5.2 Dataworks Enterprise Under the Hood – Caching 26
6 Appendix 27
6.1 The schema for the RequestRecordAsXml() and RequestRecordsAsXml() return value 27

Document Version 0.1.0
Date of issue: 2010-05-14

THOMSON REUTERS

1 INTRODUCTION

1.1 INTENDED READERSHIP

This document is intended for use by developers who wish to code against the Thomson Dataworks Enterprise (DE) Web Service. Certain sections of this document may be of interest to a secondary audience wishing a higher level, less technical overview of service elements.

1.2 WHAT ARE WEB SERVICES?

Before examining the technical details of the DE web service this section takes some time-out to explain the web services paradigm.

Web services allow the user to invoke services remotely using SOAP or HTTP-GET and HTTP-POST protocols. Requests and responses to web services are based upon a standardized XML format.

1.2.1 Why use web services?

Web services have all the advantages of a stand alone software component plus many more. The most significant benefits include:

- **Language and platform independence:** Web services can be consumed on any operating system just as long as that operating system supports an HTTP network connection.

- **Automatic upgrade:** Unlike components, if a web service requires an update, that update is propagated to all applications consuming that web service immediately. This is because the actual methods and properties for the web service are invoked from the web server remotely, meaning that each function contained within a web service appears as a "black box" to a client: they aren't concerned with the way the function does its job, just as long as it returns the expected result.

1.2.2 How might web services be used?

Web services may be used in any situation where you would consider deploying a data access component. This may be:

- As part of a rich client application running on a workstation.
- In a control downloaded to a user workstation upon accessing a web page.
- As part of a server application.

It would be sensible to use web services to access software services in any environment where there is access to the Internet (or other network where web services may be accessed) and tool or developer support facilitates easy usage.

1.3 WEB SERVICE DEVELOPER SKILLS

The level of developer skill required to use the DE web service is a function of the level of tool support available. A secondary consideration is the particular web method invoked (there is a choice). If the user is using the .NET development environment or .NET scripting application the level of tool support is very high. Consequently the need to understand the underlying technical principles of the web service is not required. The user may simply select an “Add Web Reference” command within their programming environment and start to use the web service as any other software component or class might be used. Some of the web services exposed return XML documents and so in this case the ability to use XML manipulation technologies such as XML DOM is an advantage.

If the user is using a development environment without tool support it may help the developer to understand rather more of the underlying mechanics of SOAP web services. Useful web service references are provided below.

1.4 USEFUL WEB SERVICES REFERENCES

Web sites:
World Wide Web consortium
http://www.w3.org

Microsoft's web services home page
http://msdn.microsoft.com/webservices/

Books:
Programming .NET web Services by Alex Ferrara and Matthew MacDonald. O'Reilly 2002.

1.5 WEB SERVICES REQUESTS IN THE CONTEXT OF DATAWORKS

ENTERPRISE

The full DE programmatic API is not only capable handling simple request/response pull transactions, but also subscription requests. The latter type of request being for data that changes frequently over time – for example ticking exchange data. Simple web services such as the DE web service always have a simple request/response model. Because of this, all data returned from frequently updating DE data sources will be a snapshot of the source at the time the request was made.

1.6 CONTENTS OF THIS DOCUMENT

The DE web service exposes several web methods. Some of these calls overlap – exposing the same functionality but with a different spin. The second chapter reveals the DE web service interface and helps to explain the merits of each type of method call. When this information has been digested the reader will be able to make a choice as to the most appropriate calls to make based on their circumstances.

The third chapter explains set-up of different client development environments and presents some example code for each of the detailed environments so that the user can get up and running quickly.

The fourth and fifth chapters deal with some housekeeping rules on responsible usage and other service parameters such as availability.

2 THE DATAWORKS ENTERPRISE WEB SERVICE

2.1 REQUEST PARAMETERS

Before getting in to the details of the interfaces themselves, there are a few general-purpose parameters that are applicable to all of the interfaces in the collection. These are explained here. The structures/classes explained here are in C# syntax. The conversion to other languages should be self-explanatory.

2.1.1 User Data Parameter

The *UserData* structure holds the DE user credentials. This parameter is compulsory – a request will fail if this structure is not filled in correctly. Your account manager or Thomson Reuters representative will provide this information.

```
public class UserData
{
      public string Username;
      public string Password;
      public string Realm;
}
```

Member Name	Optional (O) or Mandatory (M)	Meaning
Username	M	The DE user name
Password	M	The DE password
Realm	O	The DE Realm. The Realm is analogous to a domain. This member is normally blank.

2.1.2 Request Data Parameter

The *RequestData* structure holds all the information about what data is being requested.

```
public class RequestData
{
      public string Source;
      public string Instrument;
      public string SymbolSet;
      public string Options;
      public string[] Fields;
      public string Tag;
}
```

Member Name	Optional (O) or Mandatory (M)	Meaning
Source	M	This is the name of the data source. For example "SEC" or "History".
Instrument	M	This is the request string understood by the data source. This may be something like a simple instrument name for example "ICI" or a more complicated string understood by the data source. Refer to the documentation associated with the data

THOMSON REUTERS

		source in question for the allowed syntax.
SymbolSet	O	This is the symbol set used inside the instrument. DE can automatically map the identifier of the instrument supplied in the request from the symbology specified here to another type of symbology understood by the data source.
Options	O	This is the options string understood by the data source. Many data sources do not make use of the options string. Refer to the documentation associated with the data source in question for allowed syntax.
Fields	O	Some sources may return a huge collection of fields. When the requester does not want the full field collection returned, the set may be restricted. This is achieved by specifying the name of the fields required. Note that if this member is filled in, the order of the fields in the response will be the same as specified here. If a field specified here does not exist in the full field collection, then a placeholder field is returned in the response. This allows indexing in to the result set.
Tag	O	This is a user-defined cookie that can be used to match up request and response. The contents will be bounced back in the response to this method call. This string should not be longer than 256 characters in length.

2.1.3 Request Flags Parameter

The *RequestFlag* parameter is a 32-bit value that controls the behavior of the web service. Specific control parameters are bitwise –OR'd together in order to create the 32 bit value. Since the individual flags are enum values, often the bitwise OR'd fields must be cast to an "int", for example: (int) IncludeFieldType|IncludeFieldDisplayValue. Note that not all of the flags apply to all of the methods in the interface and the web service will throw an exception if a flag is used incorrectly.

```
public enum RequestFlagTypes
{
	Empty = 0,
	IncludeFieldType = 1,
	IncludeFieldDisplayValue = 2,
	UseSourceFormat = 4,
};
```

Member Name	Meaning
Empty	Accept the method defaults.
IncludeFieldType	Return type information regarding the field. For example it may be useful to know that a particular value is an integer or a date type etc.
IncludeFieldDisplayValue	Return a textual representation of each field.
UseSourceFormat	Without this flag set all data is returned in a flat structure. Turn this flag on to return the data in a value-added structured format that is specific to the data source. Not all sources support this flag. An error will be returned if support from the source is not available.

2.2 ADVICE ON CHOOSING THE MOST APPROPRIATE INTERFACE.

The Dataworks Enterprise Web Service interface consists of two *logical* types of interface: Those that return XML, and those that return objects. The same data content may be retrieved by using either style of interface. The explanation below should help in choosing the most appropriate type of interface.

The *RequestRecord()* and *RequestRecords()* method calls should be employed when the user wishes the data returned inside well-structured objects that are easy to manipulate from a programming language. Where available, the definition of these objects will be autogenerated by web service tool support. Often development environments with those tools will have “intellisense” support for the auto-generated classes, making these methods very easy to use. These methods are particularly well suited for use by a .NET client development environment and other development environments with advanced support for web services. No special knowledge of the underlying XML transport mechanisms is required in order to use these methods.

The *RequestRecord()* and *RequestRecords()* methods may not be the most appropriate choice for a development environment with less advanced support for web services or if there is a desire to directly manipulate the data returned – for example using an XSL style-sheet. In these cases use of the “*RequestRecordAsXml()*” and “*RequestRecordsAsXml()*” methods is recommended.

2.3 VERSION()

```
Int[] Version();
```

This interface returns the version information for the web service. The return value consists of four integers inside an array that constitute a “dotted” version number e.g. “1.0.0.0”. When a new build of the web service is deployed the version number will change. The first two parts indicate the major and minor version of the DE Web Service, the third part the patch number and the fourth the build revision number. The client may use this call to confirm functionality available in the DE Web Service should patches or other releases be made available.

2.4 THE REQUESTRECORD() AND REQUESTRECORDS() METHODS

The *RequestRecord()* and *RequestRecords()* methods have the following prototype:

```
Public Record     RequestRecord   (UserData User,
                                   RequestData Request,
                                   int RequestFlags);

public Record[] RequestRecords(UserData User,
                               RequestData[] Requests,
                               int RequestFlags);
```

The parameters *UserData*, *RequestData* and *RequestFlags* have been explained previously in this document. The *RequestRecord* method makes a single call to exactly one DE data source, and thus returns a single DE record. The *RequestRecords* method does exactly the same except that it allows multiple requests to be made in a single method call. Responses are returned in the same order as they are requested. That is to say the response to a request at index "n" in the *RequestData[]* array will be found at index "n" in the *Record[]* array.

The structure of the *Record* return type is detailed below.

2.4.1 Record Type

```
public class Record
{
        public string Source;
        public string Instrument;
        public string SymbolSet;
        public string Options;
        public string Tag;
        public StatusType StatusType;
        public int StatusCode;
        public string StatusMessage;
        public Field[] Fields;
}
```

Member Name	Meaning
Source	This is the name of the DE source to which the request was made.
Instrument	Normally this will be the same as the request string made to the DE source. However, it is possible that a source will change the instrument string to be different to that requested. This might happen where the supplied instrument has been mapped on to another from a different symbol set.
SymbolSet	This is the symbol set to which the instrument belongs. This value will be different to the symbol set in the request, if the instrument has been mapped from one type of symbology to a different symbology set.
Options	Normally this will be the same as the options string made to the DE source. However, it is possible that a source will change the instrument string to be different to that requested.
Tag	This is a user-defined cookie that can be used to match up request and response. The value of Tag will be the same as the value of Tag passed in to the request for this result set.
StatusType	These constants convey the action that client applications should take when they receive a status notification. See expansion below for possible values.
StatusCode	The status of the record object. This value gives hints as to the reason why a request might have gone wrong. Only check this value if the StatusType is “Stale” or “Failure”.
StatusMessage	A message giving extra information about the status of the record. Only check this string if the StatusType is not “Connected”.
Fields	Array of Field objects. Each field object represents a DE Field. See below for an expanded explanation of Field.

2.4.2 StatusType type

```
public enum StatusType
{
	Pending = 0,
	Connected = 1,
	Stale = 2,
	Failure = 4
}
```

Member Name	Meaning
Pending	This status type is for Thomson Financial internal use only and should not be seen by external clients.
Connected	The data is fine. This is the status to indicate complete success.
Stale	This indicates that the source is unavailable, for example because of an infrastructure failure or a failure in the datafeed communications. It may be worth-while retrying the request after a short interval.
Failure	The data could not be obtained, for example because the instrument was incorrect, or the user is not permissioned for the data.

2.4.3 StatusCode type

You will notice that *StatusCode* has type “int” and is not an enum. This is because data sources are free to return any valid integer value for the status code. However there are certain well-known values for status code that have been defined. When testing the status code applications may cast the "int" *StatusCode* value to the enum “*StatusCode*” in order to make comparisons. Only check this value if StatusType has value “*Stale*” or “*Failure*”.

For example code similar to *if ((StatusCode)Record.StatusCode == StatusCode.StatusAccessDenied) then*…may be written.

```
public enum StatusCode
{
	StatusNoError = 0,
	StatusDisconnected = 1,
	StatusSourceFault = 2,
	StatusNetworkFault = 3,
	StatusAccessDenied = 4,
	StatusNoSuchItem = 5,
	StatusBlockingTimeout = 11,
	StatusInternal = 12,
}
```

Member Name	Meaning
StatusNoError	No reason has been given for the *Failure* or *Stale* status.
StatusDisconnected	The notifier has been disconnected from its data supply.
StatusSourceFault	There is a problem with the source. If DE detects a problem with a source, it will issue a source fault status change.
StatusNetworkFault	There is a problem with the data supply. If the *StatusType* value is *Stale* it may be worthwhile re-trying the request.
StatusAccessDenied	The user does not have permission to obtain the data.
StatusNoSuchItem	There is no item with that name.
StatusBlockingTimeout	The underlying DE source did not respond in the timeout period allowed.
StatusInternal	An internal or other unspecified error occurred processing the request.

2.4.4 Field Type

The field object represents a name-value pair. The name is simply a textual string. The value may be single-valued or be multi-valued (an array of values) but not both. If the value is multi-valued, all values in the array will have the same type. Valid types for *Value* and *ArrayValue* are 32-bit integer, string, 64-bit floating-point number (double), 32-bit floating-point number (single) and date time. If it is known at development time that a particular field with a certain name has a certain type, then the *Value*, or *ArrayValue* field may be cast to the appropriate type. Failing this the type may be queried at runtime.

```
public class Field
{
	public string Name;
	public object Value;
	public object[] ArrayValue;
	public string DisplayValue;
}
```

Member Name	Meaning
Name	The name of the field.
Value	The single valued value of the field
ArrayValue	The multiple-valued value of the field.
DisplayValue	This property is optional and contains suggested text that should be used when displaying the data contained in the value. This value will only be set if the user has requested it using the RequestFlags parameter and if the DE source has supplied the necessary information.

2.4.5 Handling Errors and Exceptions

Before checking the data values in each record, the programmer should check the *StatusType* field. The request has succeeded if the status is "*Connected*". If the *StatusType* is "*Failure*" or "*Stale*" the request has failed. In this case further code is required to work out if this request failed due to some temporary condition and therefore whether if it is worth retrying this request at some point during the future.

If the *StatusType* is "*Failure*" and the *StatusCode* is "*StatusAccessDenied*", this request is never going to succeed unless the user's permissions are updated. It is only worth retrying this request once the user permissions have been updated and the new permissions have been permeated down to the data source.

If the *StatusType* is "*Failure*" and the *StatusCode* is "*NoSuchItem*", there are a number of possibilities. A request may have been made with invalid syntax, a request may have been made for a new instrument that DE does not know about yet or a request may have been made for an instrument that is no longer valid.

A *StatusType* of "*Stale*" could indicate a problem with the network infrastructure behind the Dataworks Enterprise web servers or one of the data sources. In such cases it may be worth employing a sensible strategy to retry the request. A suggested strategy is to leave a time period of 60 seconds before retrying a request and only retrying a maximum of 5 times.

In all cases the *StatusMessage* should give extra information as to the cause of the error. Where possible errors should be logged to aid diagnosis of problems.

The web service will throw a SOAP exception in abnormal circumstances. This is not expected to be a regular event since this situation will usually indicate some kind of catastrophic failure on the server. Should an exception occur client code should adopt some kind of sensible retry policy as it does with an error.

For .NET clients the “SoapException” class is defined in the “System.Web.Services.Protocols“ namespace should be caught and handled.

Also, for .NET clients the auto-generated proxy code may throw a standard “Exception” class if, for example, it cannot contact the web service from the client server.

2.5 THE REQUESTRECORDASXML() AND REQUESTRECORDSASXML() METHODS

```
public System.Xml.XmlNode RequestRecordAsXml(UserData User,
                                              RequestData Request,
                                              int RequestFlags);

public System.Xml.XmlNode RequestRecordsAsXml(UserData User,
                                               RequestData[] Requests,
                                               int RequestFlags);
```

The parameters *UserData*, *RequestData* and *RequestFlags* have been explained previously in this document. The *RequestRecordAsXml*() method makes a single call to exactly one DE data source, and thus returns an XML representation of a single DE record. The *RequestRecordsAsXml*() method does exactly the same except that it allows multiple requests to be made in a single method call. Responses are returned in the response XML in the same order as they were requested.

Example XML documents showing the structure of the XML returned are detailed below. The w3c xsd schema to formally describe the XML documents returned by these two methods can be found in the appendix.

Results of call to method *RequestRecordAsXml()* from source “Datastream” for “BAY~1D”,

RequestFlags = Empty:

```
<?xml version="1.0" encoding="UTF-8"?>

<Record xmlns="http://xml.thomson.com/financial/v1/tools/distribution/dataworks/enterprise/2003-07/">
    <Source>History</Source>
    <Instrument>ICI~-1D</Instrument>
    <Tag/>
    <StatusType>Connected</StatusType>
    <StatusCode>0</StatusCode>
    <StatusMessage/>
    <Fields>
        <CCY>£</CCY>
        <CLOSE>
            <Item>194</Item>
        </CLOSE>
        <DATE>
            <Item>2003-08-29T00:00:00</Item>
        </DATE>
        <DISPNAME>IMP.CHM.INDS.</DISPNAME>
        <SYMBOL>ICI</SYMBOL>
    </Fields>
</Record>
```


Results of call to method *RequestRecordAsXml()* from source “Datastream” for “BAY~1D”,RequestFlags = IncludeFieldType:


```
<?xml version="1.0" encoding="UTF-8"?>

<Record xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xml.thomson.com/financial/v1/tools/distribution/dataworks/enterprise/2003-07/">
    <Source>History</Source>
    <Instrument>ICI~-1D</Instrument>
    <Tag/>
    <StatusType>Connected</StatusType>
    <StatusCode>0</StatusCode>
    <StatusMessage/>
    <Fields>
        <CCY xsi:type="xsd:string">£</CCY>
        <CLOSE>
            <Item>194</Item>
        </CLOSE>
        <DATE>
            <Item>2003-08-29T00:00:00</Item>
        </DATE>
        <DISPNAME xsi:type="xsd:string">IMP.CHM.INDS.</DISPNAME>
        <SYMBOL xsi:type="xsd:string">ICI</SYMBOL>
    </Fields>
</Record>
```


The same request with *RequestFlags = IncludeFieldDisplayValue*, will yield the same as for *RequestFlags = Empty* because this option is only supported for the *RequestRecord()* and *RequestRecords()* methods.

Results of call to method *RequestRecordAsXml()* from source “Datastream” for “BAY~1D”,*RequestFlags = UseSourceFormat*:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<Record xmlns="http://xml.thomson.com/financial/v1/tools/distribution/dataworks/enterprise/2003-07/">
    <Source>History</Source>
    <Instrument>ICI~-1D</Instrument>
    <Tag/>
    <StatusType>Failure</StatusType>
    <StatusCode>12</StatusCode>
    <StatusMessage>Record does not have source format</StatusMessage>
    <Fields/>
</Record>
```

The only difference for the method that returns multiple records (RequestRecordsAsXml()) is that the root element of the returned document is <Records/>. A <Records/> element can contain one or more <Record> elements:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<Records xmlns="http://xml.thomson.com/financial/v1/tools/distribution/dataworks/enterprise/2003-07/">
    <Record>…</Record>
    :
    :

</Records>
```

As was the case with the *RequestRecord()* and *RequestRecords()*, when you specify the fields to be returned in the request, if the field is missing you get a place holder field in the response. The order of the returned fields is the same as specified in the request. This allows the developer to use the technique of expecting particular data values to appear at a specific "index" in the result.

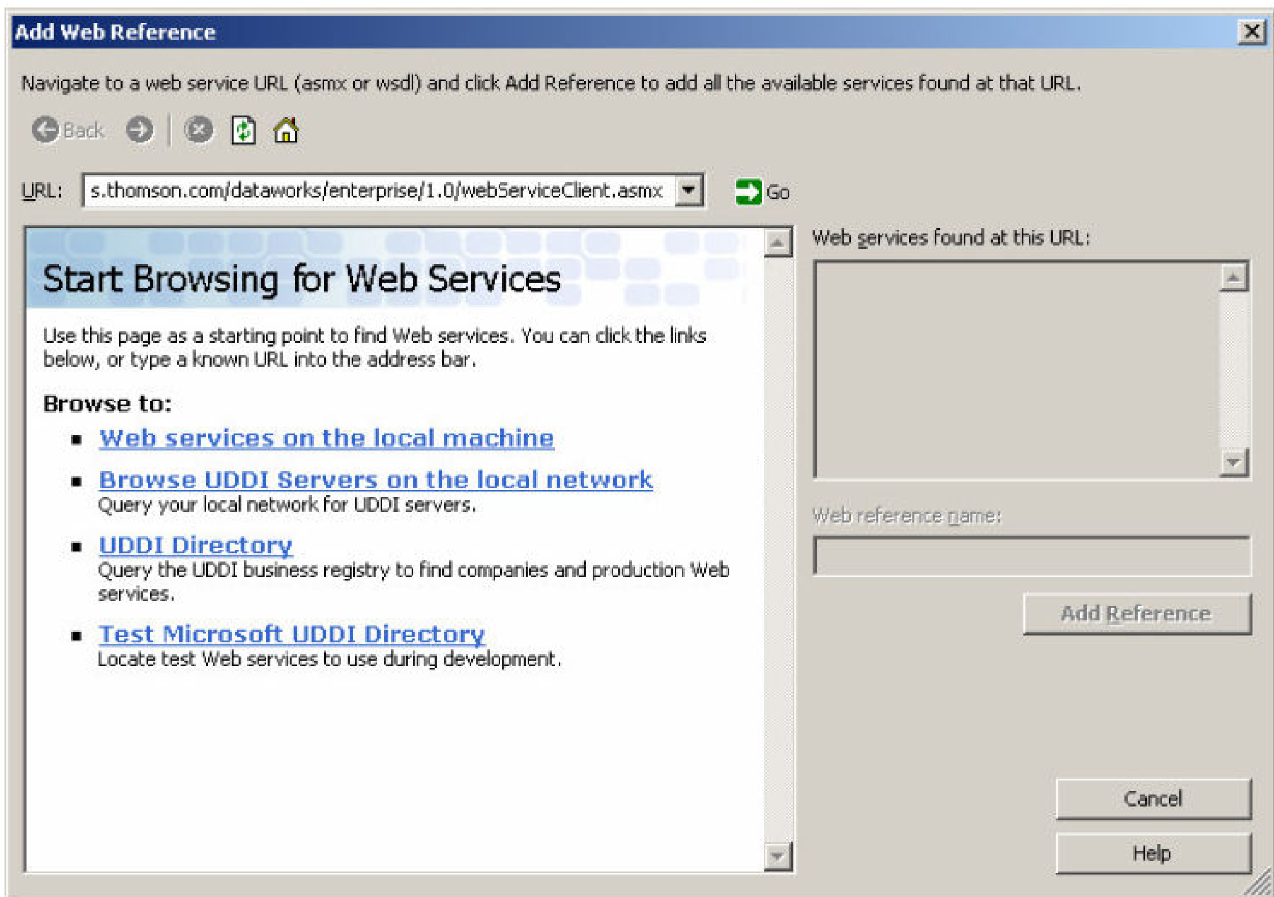
Document Version 0.1.0
Date of issue: 2010-05-14

THOMSON REUTERS

3 EXAMPLE CLIENT CODE

3.1 WINDOWS .NET CLIENT

In order to generate the client side proxy objects that help in accessing the web service it is necessary to add a "web reference" to your visual studio project. There is usually a special web references folder in the solution view where the command to "add a web reference" may be selected. Once this is done a dialog box similar to that below will pop-up. In the URL for the web reference enter the string:
http://dataworks.thomson.com/dataworks/enterprise/1.0/webServiceClient.asmx
or
http://europe.dataworks.thomson.com/dataworks/enterprise/1.0/webServiceClient.asmx
The dialog box may give you the opportunity to choose a sensible name for the web reference. Finally hit the "Add Reference" button.



This action should have generated the proxy objects required to make easy use of the web service.

3.1.1 Calling the GetRecord() and GetRecords() methods

The following example examines calling the *GetRecord()* method on the web service using C#, after the "add web reference" step has been carried out.

Firstly you must add a reference to the namespace where the auto-generated proxies reside. If your solution is called “MySolution” and you named the web reference “DeWeb” then import the namespace “MySolution.DeWeb”. Once this is done you may create a “WebServiceClient” object and execute the methods.

```
using System.Web.Services.Protocols;
using MySolution.DeWeb;

    :
    :
    :

WebServiceClient clt = new WebServiceClient();
UserData         userdata = new UserData();
RequestData      request = new RequestData();

//Fill in the user credentials
userdata.Username = "myusername";
userdata.Password = "mypassword";
request.Instrument = "ICI";
request.Source = "History";

try
{
  Record rec = clt.RequestRecord(userdata, request, 0);
  if (rec.StatusType == StatusType.Connected)
 {
    //Cycle through the fields collection
    for (int i=0; i < rec.Fields.Length; i++)
    {
      Console.WriteLine(rec.Fields[i].Name);
      if (rec.Fields[i].Value != null)
      {
        Console.WriteLine(rec.Fields[i].Value.ToString());
      }
      else
      {
        for (int j=0; j < rec.Fields[i].ArrayValue.Length; j++)
        {
          Console.WriteLine(rec.Fields[i].ArrayValue[j].ToString());
        }
      }
    }
  }
  else
  {
    //Handle error
    Console.WriteLine(rec.StatusMessage);
  }
}
catch (SoapException exception)
{
  Console.WriteLine(exception.ToString());
}
```

The extension to call the *GetRecords()* method is simple and is left as an exercise for the reader. The code above will request a time series record for the stock “BAY” from the “Datastream” source. Once this is complete, the code will iterate through the field collection printing the values out to the command line. Note that some values are array based (multi) and some values are singular.

3.1.2 Calling the GetRecordAsXml and GetRecordsAsXml methods

The following example examines calling the GetRecordAsXml()method on the web service using C#, after the “add web reference” step has been carried out.

Firstly you must add a reference to the namespace where the auto-generated proxies reside. If your solution is called “MySolution” and you named the web reference “DeWeb” then import the namespace “MySolution.DeWeb”. Once this is done you may create a “WebServiceClient” object and execute the methods.

```
WebServiceClient clt = new WebServiceClient();
UserData          userdata = new UserData();
RequestData       request = new RequestData();
XmlNode          responseDoc;
XmlNode          statusTypeNode;
XmlNodeList       itemNodes;
XmlNodeList       fieldNodes;

//Fill in the user credentials
userdata.Username = "myusername";
userdata.Password = "mypassword";
request.Instrument = "ICI";
request.Source = "History";

try
{
  responseDoc = clt.RequestRecordAsXml(userdata, request, 0);
  XmlNamespaceManager nsMgr = new
XmlNamespaceManager(responseDoc.OwnerDocument.NameTable);

  nsMgr.AddNamespace("xx", responseDoc.NamespaceURI);

  statusTypeNode = responseDoc.SelectSingleNode("//xx:StatusType/child::text()",
nsMgr);

if (statusTypeNode != null && statusTypeNode.Value == "Connected")
{
  fieldNodes = responseDoc.SelectNodes("//xx:Fields/child::*", nsMgr);

  foreach (XmlNode fieldNode in fieldNodes)
  {
    if (fieldNode.ChildNodes.Count == 1)
    {
      //Single valued
      Console.WriteLine("Name {0}", fieldNode.Name);
      Console.WriteLine("Value {0}", fieldNode.ChildNodes[0].Value);
    }
    else
    {
      //multi valued
      Console.WriteLine("Name {0}", fieldNode.Name);
      foreach (XmlNode itemNode in fieldNode.ChildNodes)
      {
        Console.WriteLine("Value {0}", itemNode.ChildNodes[0].Value);
      }
    }
  }
}
}
catch (SoapException exception)
{
  Console.WriteLine(exception.ToString());
}
```

The extension to call the GetRecordsAsXml() method is simple and is left as an exercise for the reader. The code above will request a time series record for the stock “BAY” from the “Datastream” source. Once this is complete, the code will iterate through the field collection printing the values out to the command line. Note that some values are array based (multi) and some values are singular.

3.1.3 Using XSD.exe in combination with the Request…AsXml() methods

This section may be of benefit if you choose to use the *RequestRecordAsXml*() or *RequestRecordsAsXml*() methods specifying "*UseSourceFormat*" for the request flags parameter.

There is a tool supplied with visual studio.NET called "xsd.exe" capable of generating XSD schema from XML document instances and C# object definitions from XSD schema. The generated object definitions also include code to initialize themselves by deserializing XML input. This means that the output from the *"…AsXml()"* methods may be deserialized directly in to auto-generated objects negating the need for the user to parse the document tree. This technique will enable the programmer to use the results to the method call in an "early bound" sense. This means that the field names are known at compile time. Because xsd.exe will use actual source specific document instances, the objects that it generates will be business domain specific, unlike the normal generic DE interface.

If this method is used we would recommend that the "Fields" member of the "RequestData" structure is filled-in on request. This is because new fields may be added to the output at any time and this could break the xsd.exe-based implementation. If the "Fields" member is filled in the exact output from the web service is completely deterministic thus avoiding problems.

To use the "xsd.exe" tool select the "visual studio.NET command prompt" from the "tools" menu of your 'studio installation.

3.1.3.1 Step 1 – Generate the XSD schema file

Use the DE web service test page: http://xml.thomson.com/dataworks/enterprise/1.0/sample/default.aspx to generate a document instance. Save the document instance to a file ending in the suffix ".xml". Generate XSD schema for this document instance using the xsd.exe tool. A sample document instance for the "SEC" source is shown below:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Record
xmlns="http://xml.thomson.com/financial/tools/distribution/dataworks/enterprise/2003-
07/">
  <Source>MarketPlace</Source>
  <Instrument>STATE=MD/IM</Instrument>
  <SymbolSet />
  <Options />
  <Tag />
  <StatusType>Connected</StatusType>
  <StatusCode>0</StatusCode>
  <Fields>
    <ArrayOfObject0>
      <Object0>
        <ENTITY_ID>10391</ENTITY_ID>
        <NAME_FULL>Adams Express Co.</NAME_FULL>
        <MANAGED_AMT>1024.8</MANAGED_AMT>
        <ADDR_CITY>Baltimore</ADDR_CITY>
        <ADDR_STATE>MD</ADDR_STATE>
      </Object0>
      <Object0>
        <ENTITY_ID>309383</ENTITY_ID>
        <NAME_FULL>Alex. Brown Investment Management, Ltd</NAME_FULL>
        <MANAGED_AMT>7069.4</MANAGED_AMT>
        <ADDR_CITY>Baltimore</ADDR_CITY>
        <ADDR_STATE>MD</ADDR_STATE>
      </Object0>
    </ArrayOfObject0>
    <REF_COUNT>46</REF_COUNT>
  </Fields>
</Record>
```


3.1.3.2 Step 2 – Generate C# class files from the XSD file

Use the xsd.exe tool again on the generated ".xsd" file to generate a C# class file. The result should be something like this:

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.3705.288
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

//
// This source code was auto-generated by xsd, Version=1.0.3705.288.
//
using System.Xml.Serialization;

/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="http://xml.thomson.com/financial
/tools/distribution/dataworks/enterprise/2003-07/")]
[System.Xml.Serialization.XmlRootAttribute(Namespace="http://xml.thomson.com/financial
/tools/distribution/dataworks/enterprise/2003-07/", IsNullable=false)]
public class Record {

    /// <remarks/>
    public string Source;
    /// <remarks/>
    public string Instrument;
    /// <remarks/>
    public string SymbolSet;
    /// <remarks/>
    public string Options;
    /// <remarks/>
    public string Tag;
    /// <remarks/>
    public string StatusType;
    /// <remarks/>
    public string StatusCode;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("Fields")]
    public RecordFields[] Fields;
}
/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="http://xml.thomson.com/financial
/tools/distribution/dataworks/enterprise/2003-07/")]
public class RecordFields {

    /// <remarks/>
    public string REF_COUNT;

    /// <remarks/>
    [System.Xml.Serialization.XmlArrayItemAttribute("Object0",
typeof(RecordFieldsArrayOfObject0Object0), IsNullable=false)]
    public RecordFieldsArrayOfObject0Object0[][] ArrayOfObject0;
}

/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="http://xml.thomson.com/financial
/tools/distribution/dataworks/enterprise/2003-07/")]
public class RecordFieldsArrayOfObject0Object0 {

    /// <remarks/>
    public string ENTITY_ID;
    /// <remarks/>
```

```
    public string NAME_FULL;
    /// <remarks/>
    public string MANAGED_AMT;
    /// <remarks/>
    public string ADDR_CITY;
    /// <remarks/>
    public string ADDR_STATE;
}

/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="http://xml.thomson.com/financial
/tools/distribution/dataworks/enterprise/2003-07/")]
[System.Xml.Serialization.XmlRootAttribute("NewDataSet",
Namespace="http://xml.thomson.com/financial/tools/distribution/dataworks/enterprise/20
03-07/" +
"", IsNullable=false)]
public class NewDataSet {

      /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("Record")]
    public Record[] Items;
}
```

3.1.3.3 Step 3 – Write your code

Use the autogenerated object to deserialize and manipulate the output of the web service.

```
WebServiceClient ws = new WebServiceClient();
XmlDocument doc = ws.RequestRecordAsXml(user, request, format);

Record record = (Record)convertXmlToObject(typeof(Record),
                                           doc.DocumentElement);

object convertXmlToObject(System.Type type, XmlNode xmlNode)
{
  XmlSerializer serializer = new XmlSerializer(type);
  XmlReader reader = new XmlNodeReader(xmlNode);
  return serializer.Deserialize(reader);
}
```

3.1.4 Writing Asynchronous .NET code

All of the .NET code examples have used the synchronous invocation model. It is easy to invoke web services asynchronously using the built in support for asynchronous invocation. This is achieved by calling the methods *BeginRequestRecord(), BeginRequestRecordAsXml()* on the auto-generated proxy, and supplying a callback delegate that is triggered when the web method returns results.

3.2 JAVA CLIENTS

3.2.1 Apache Axis based

The following sections detail how to setup for and write a client using Java In combination with the Axis.

3.2.1.1 Prerequisites prior to writing client

1. Download the latest JDK from Sun - Java Standard Edition (http://java.sun.com/j2se). In writing this document version 1.4.1 was installed. Run the install.

2. Download axis from apache (http://ws.apache.org/axis/index.html). In writing this document the 1.0 release was downloaded. Unzip the redistributable and copy the files where you want them. Axis is essentially a SOAP engine -- a framework for constructing SOAP processors such as clients, servers, gateways, etc. The current version of Axis is written in Java, but a C++ implementation of the client side of Axis is being developed. Axis isn't just a SOAP engine -- it also includes:

- A simple stand-alone server.
- A server that plugs into servlet engines such as Tomcat.
- Extensive support for the Web Service Description Language (WSDL).
- Emitter tooling that generates Java classes from WSDL..
- Some sample programs.
- A tool for monitoring TCP/IP packets.

3. Put location of java.exe, javaw.exe, javac.exe etc. in to your path in your environment variables.

4. Download and install Tomcat (http://jakarta.apache.org/tomcat/). Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and Java Server Pages technologies. The Java Servlet and Java Server Pages specifications are developed by Sun under the Java Community Process. In writing this document version 4.1 was installed.

5. Install the eclipse IDE (http://www.eclipse.org). This is a nice free IDE that works well with the Sun JDK.

3.2.1.2 Writing the client

1. Generate the stub proxies to access the remote web service. Do this by running the class generator supplied by running the following command line utility:

```
java org.apache.axis.wsdl.WSDL2Java http://<<url>>/myservice.asmx?wsdl
```

You may find that you need to add the following java archive files to your CLASSPATH environment variable before this will work.

```
C:\<<mypath>>\xml-axis-10\lib\axis.jar;C:\<<mypath>>\xml-axis-
10\lib\wsdl4j.jar;C:\ <<mypath>>\xml-axis-10\lib\commons-
discovery.jar;C:\<<mypath>>\xml-axis-10\lib\commons-
logging.jar;C:\<<mypath>>\xml-axis-10\lib\jaxrpc.jar;C:\<<mypath>>\xml-axis-
10\lib\log4j-1.2.4.jar;C:\<<mypath>>\xml-axis-
10\lib\saaj.jar;C:\<<mypath>>\xml-axis-10\lib\axis-ant.jar
```

(where <<mypath>> indicates the location on your hard drive.

You must specify if there is an HTTP proxy server between you and the web service, since Java will probably not be using your usual HTTP stack. Forget setting any environment variable or control panel settings to do this. It only seems to work if you specify on the command line using the "-D" option as follows:

```
java -Dhttp.proxyHost=dsproxy -Dhttp.proxyPort=80
org.apache.axis.wsdl.WSDL2Java http://<<url>>/myservice.asmx?wsdl
```

This command will generate a set of java source files sitting in a set of directories that mirror the namespace of the web service. If the web service is in the “xml.thomson.com” namespace the classes generated will be in the directory “\com\thomson\xml”.

2. Create a new project in the eclipse IDE and IMPORT the auto-generated classes to the project. In doing this action the classes should be copied to the location of your project by the IDE. Add the axis and tomcat .jar files to the referenced libraries (project->properties->libraries tab). The names of the .jar files that you actually need are listed in the bitmap labeled *figure 2*.

An example of the type of code that you need to write in order to access the web service is contained in figure 1. Firstly create a “Locator” object and request a “ClientSoap” object. Use the “ClientSoap” object to invoke the web service. This code will operate in a synchronous manner.

```
 */
package com.thomson.xml;

/**
 * @author barkert
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comment
 */
public class TestClient {

    public static void main(String[] args)
    {
        try
        {
            WebServiceClientLocator cl = new WebServiceClientLocator();
            WebServiceClientSoap sc = cl.getWebServiceClientSoap();

            RequestData rd = new RequestData();
            rd.setSource("History");
            rd.setInstrument("ICI");

            Record rec = sc.requestRecord(null, rd);
            if (rec != null)
            {
                ArrayOfField af = rec.getFields();
                if (af != null)
                {
                    Field f = af.getField(0);
                }
            }
        }
        catch (Exception e)
        {
            System.out.println(e.toString());
        }
    }
}
```

Figure 1: Code and project layout for a basic java web service client

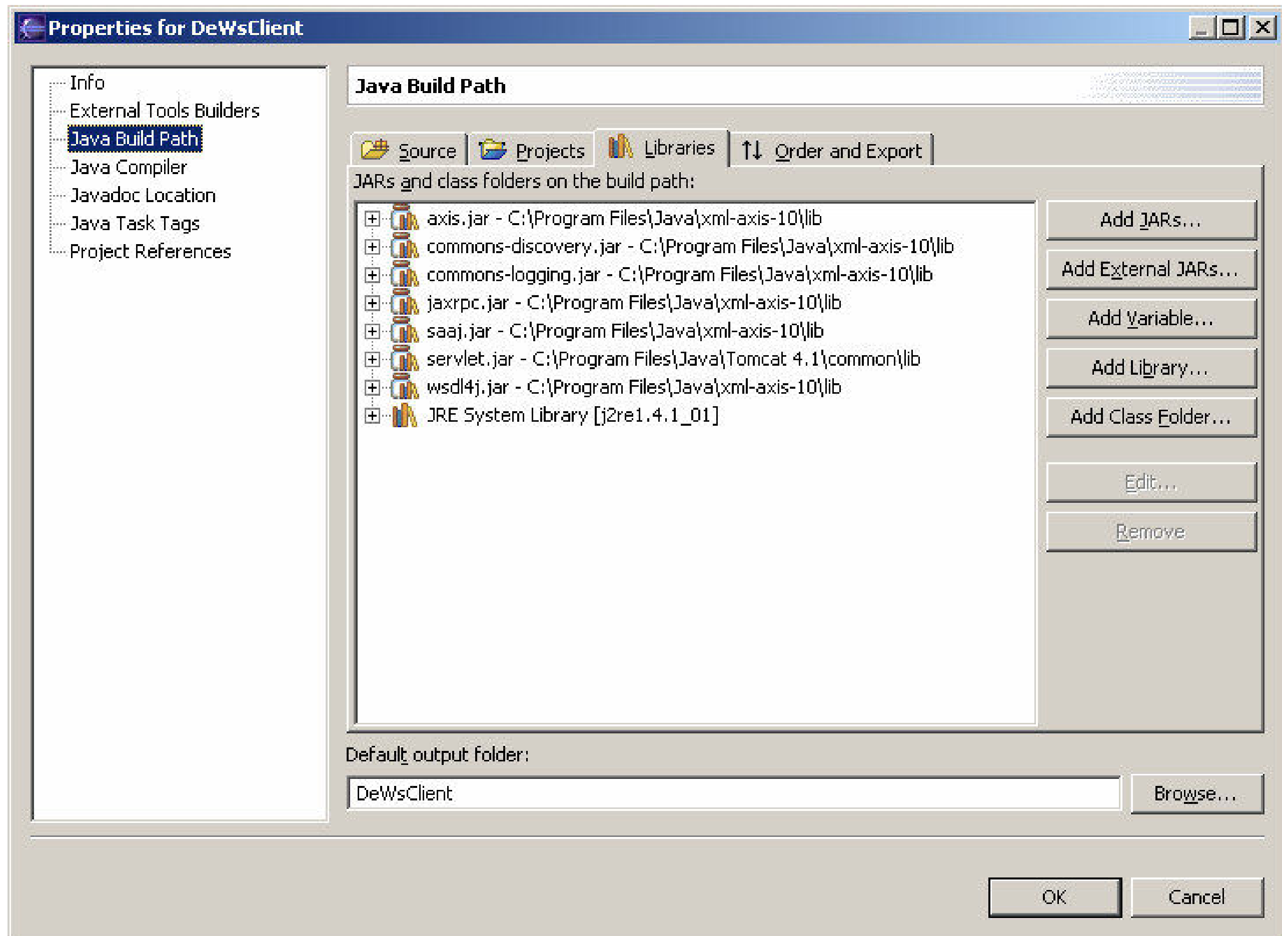


Figure 2: Required java archive references

3.2.1.3 Using Castor

With the help of castor (http://castor.exolab.org), you can also deserialize an XML Document into a well-defined object, just as shown in the previous .NET section. You also need to create a (Java) class file for your well-defined object. Castor has a utility class (org.exolab.castor.builder.SourceGenerator) to convert XML schema into a Java class file. The same class can also help you to deserialize (custom deserialization) the SOAP message in to a Java object. Please refer Axis and castor documents for details.

3.2.2 Sun ONE

For the latest information regarding web service compatibility of the Sun ONE studio browse to: http://www.sun.com/sunone/.

3.2.3 IBM Websphere

For the latest information regarding web service compatibility of IBM webshere browse to: http://www.ibm.com/websphere.

3.3 MICROSOFT OFFICE

At the time of writing Microsoft are only just starting to incorporate web services technology in to their products. Office XP has a web services toolkit that offers proxy generation for very simple web services (simple types in and simple types out). Unfortunately the DE web service, while not being particularly complex exceeds the limits of complexity that can be

handled by the XP web services toolkit.

It is however fairly easy to develop VBA code that can use the DE web service without the need for the auto-generated proxies. There are a number of articles on how to accomplish this on the Microsoft developer network. In addition two sample Excel spreadsheets have been developed that demonstrate usage of the web service. One of the spreadsheets uses the Microsoft SOAP toolkit v3.0 and MSXML and the other uses just MSXML. The code for the version using the SOAP toolkit is marginally easier, but this of course introduces an additional dependency. In both cases the code is structured to wrap the client and record objects for easy reuse. Note that VBA code is provided on an “as-is” basis and will not be supported. Note that when driving web services from an environment where tool support is less rich the developer must remember to take care of details such as escaping invalid characters in the data such as "&".

MSDN articles:
· http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnxpwst/html/odc_wsrtct.asp
· http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnofftalk/html/office09062001.asp

There is also a toolkit called pocket SOAP (http://www.pocketsoap.com/), and a VB6.0 proxy object generator (from the WSDL) that may be of assistance (http://www.pocketsoap.com/wsdl/).

3.4 MICROSOFT C++ CLIENT

A web service client can be written in C++ by using the same COM calls against the MSXML and Microsoft SOAP toolkit libraries as detailed in the Microsoft Office section.

3.5 OTHER CLIENT TYPES

Other client architectures may be employed. Unless web service tool support is available, the developers will need to use lower level communications libraries. These might include HTTP or even TCP/IP communication libraries. During development, it can be advantageous to use tools such as network monitors to spy on communications traffic.

4 GENERAL SYSTEM GUIDELINES

4.1 CODING GUIDELINES

4.1.1 Reducing Dependencies

There are some guidelines that should be followed to make maintenance of client applications easier.

4.1.1.1 Don't Hardcode the SOAP End-Point URL

Make sure that the web service URL can be changed easily. Ideally do not hard code the URL – use a configuration file instead. If using .NET set the "dynamic" property of the web service proxy to "true" and load the web service URL from a configuration file. The reason behind this is that at some point in the future you may want to check your application's compatibility with a new version of the web service hosted on a different server. Using this approach avoids the need to recompile.

4.1.1.2 Be Careful when using "XPath"

When using XPath to navigate around a returned XML document, be as specific as possible when addressing a node. It is not a good idea to refer to a field by expecting it to be in position "n" inside the document. This is because new fields may get added to a source, and this will break your program. A better technique is to refer to field elements by using the field name. Alternatively you could restrict the fields returned by specifying this in the "RequestData" structure. In this case it is safe to expect a field to be at a particular index in the response.

4.1.2 Client design

Please recognize that DE is a powerful data retrieval mechanism. Requests that clients send in can cause significant CPU and memory load on the servers. While the underlying server architecture can easily be scaled to meet demand, and to keep response times within a reasonable window, we ask customers to adopt sensible coding strategies.

4.1.2.1 Limit the number of threads/processes in your application capable of making Requests Concurrently

The design of the client should limit the number of concurrent requests that can be made asynchronously. If calling the web service asynchronously only a finite number of responses should be outstanding at any one time.

4.1.2.2 Be aware of how much data you are requesting from each source

Not every data source from the DE data catalogue is well suited to bulk download. For example with the "Datastream" source, there is a batch download product that is better suited to bulk downloads.

4.1.2.3 Optimize Requests

Try to optimize requests. Very often it is quicker to make one bigger request than several smaller requests because of the relatively high transport overhead with web services. To optimize requests for each data source, refer to data catalogue documentation on http://dtn.tfn.com/.

4.1.2.4 Coding policy request timeout and retrying failed requests

During busy periods response times may climb. Please adopt a sensible strategy to handle failed or slow requests. Where possible leave at least one minute between retries (or perhaps double time interval between retries after a failure). Only retry a failed request up to 5 times. Particularly aggressive strategy, especially at busy times can make server performance worse.

4.2 RESPONSIBLE USE

4.2.1 During Development

In the process of developing an application, if there is a need to stress test please contact the technical support team to give prior notice.

5 SERVICE LEVELS

5.1 SERVICE AVAILABILITY

The availability of the DE web service is a function of the underlying data source. Most data sources are available 24 hours a day, 7 days a week.

5.2 DATAWORKS ENTERPRISE UNDER THE HOOD – CACHING

The web service makes use of a DE feature called lazy caching. Once a request has been made to the web service, the result will be held in a cache store on the server for 30 seconds. If a request for the same data is made during that 30-second window, the cached copy of the data will be served. Where data is served from the cache a very fast response should be expected.

6 APPENDIX

6.1 THE SCHEMA FOR THE REQUESTRECORDASXML() AND REQUESTRECORDSASXML() RETURN VALUE

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="Record" type="RecordType"/>
    <xs:element name="Records" type="RecordsType"/>
    <xs:complexType name="RecordsType">
        <xs:sequence>
            <xs:element name="Record" type="RecordType" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="RecordType">
        <xs:sequence>
            <xs:element name="Source" type="xs:string"/>
            <xs:element name="Instrument" type="xs:string"/>
            <xs:element name="Tag" type="xs:string" minOccurs="0"/>
            <xs:element name="StatusType" type="StatusTypeType"/>
            <xs:element name="StatusCode" type="xs:int"/>
            <xs:element name="StatusMessage" type="xs:string"/>
            <xs:element name="Fields" type="FieldsType"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="FieldsType">
        <xs:sequence>
            <xs:any namespace="##any" processContents="lax"/>
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="StatusTypeType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Pending"/>
            <xs:enumeration value="Connected"/>
            <xs:enumeration value="Stale"/>
            <xs:enumeration value="Failure"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

© 2010 Thomson Reuters. All rights reserved.
Republication or redistribution of Thomson Reuters content, including by framing or similar means, is prohibited without the prior written consent of Thomson Reuters. 'Thomson Reuters' and the Thomson Reuters logo are registered trademarks and trademarks of Thomson Reuters and its affiliated companies.

For more information
Send us a sales enquiry at
http://thomsonreuters.com/about/contact_us
Read more about our products at
http://thomsonreuters.com/products_services
Find out how to contact your local office
http://thomsonreuters.com/about/locations

Document Version 0.1.0
Date of issue: 2010-05-14

THOMSON REUTERS