Setiri.Witsml .Net SDK
Developer Guide

Setiri LLC

Last Updated: 11/11/2012

Contents

Introduction

The Setiri.Witsml .Net SDK is a Microsoft .Net library designed to simplify the task of communicating with a WITSML based server. The SDK accomplishes this by encapsulating the XML based query and data object structures utilized by the WITSML standard into a series of .Net classes. The developer can then work with these strongly-typed and validated queries, interact with a remote WITSML server, and WITSML data objects via .Net entity objects instead of raw XML.

The SDK enables full Read/Write/Update/Delete access to any server supporting the WITSML 1.3.1.x standard STORE interface.

The quickest way to get started with the SDK is to view the sample project included with the software. This project demonstrates the most common interactions with a remote WITSML server:

- Create a new Well and Wellbore hierarchy
- Retrieve list of Wells and Wellbores
- Create a Log object based on depth or time. Append data to this Log
- Query for a portion of a Log object, based on depth or time
- Delete objects based on query parameters

Install and Setup

Installation

The SDK is distributed as a simple DLL file that should be referenced by your .Net project. To install, simply extract the zip file into a project folder on your development PC.

Demo Project

The demo/sample project included with the sdk can be found in <install path>/Setiri.Witsml.Samples. A visual studio solution file in this folder will launch the demo project within visual studio.

The demo project is a console based application written in C#. The SDK also works with VB.Net but currently no VB.Net sample project is included.

Include in Project

In a new or existing .Net project, the SDK should be included by:

- Click “Add Reference”
- Browse to the <install path>/Bin
- Click on “Setiri.Witsml.dll” file to add the reference to your project.

You may then include the namespace in your source file(s) using:

```
using Setiri.Witsml;
using Setiri.Witsml.v1311;
```

Witsml Overview

The core of WITSML is the standardized drilling-related data that it defines. This data exists as XML data, and are defined using XML schemas.

The data objects fall into a few categories

- Core data objects
- Recurring structure objects
- Reporting objects

Core Objects and Hierarchy

Every WITSML object contains a unique identifier, called the UID. This value uniquely defines the object on a server, so that queries or modifications to an object can matched up to the original object on the server.

All WITSML objects are defined under a two-layer hierarchy under two other object types: the Well and the Wellbore. Thus every object (aside from the Well and Wellbore themselves) has a parent of a Wellbore and a grandparent of a Well. Thus a tree view of the objects on a given server will look like:

- Well 1
- Well 2
    - Wellbore X
    - Wellbore Y
        - Log Object 1
        - Log Object 2
        - Trajectory Obj X
        - <other object types>

This means that any non-Well or Wellbore objects will have a single parent Well and Wellbore unique id associated with it.

For applications that need to navigate over a list of objects on a server, the first task will almost always be to query and retrieve the list of Wells. From there, a second query will then retrieve the Wellbores associated with a selected Well, and from there the application can perform queries for any other objects that are children of a selected Wellbore.

Recurring Structures

Many objects are simple collections of values, but there are several which represent objects that have (potentially long) recurring sets of values defined. The three primary “recurring structure” objects are:

- Log
- MudLog
- Trajectory

Reporting Objects

Witsml also includes a set of objects utilized for storing well reporting data. These objects include:

- CementJob
- FluidsReport
- OpsReport

.Net Witsml objects

All of the objects defined by WITSML are represented by .Net classes in the SDK. These objects exist inside the Setiri.Witsml.1311 namespace (the currently supported version of WITSML is 1.3.1.1). There are a large number of classes in this namespace, but each of the "top level" objects has a prefix of "obj_". Thus an example for the Well, Wellbore, and Log objects would be the equivalent obj_well, obj_wellBore, and obj_Log .Net classes. The remaining classes in this namespace are support objects utilized inside these primary objects.

For every obj_ class defined, there is also a associated obj_Query class as well. When querying data from the server, the obj_Query classes (ie, obj_wellQuery) are utilized to create the desired query criteria. The properties on these classes mirror the affiliated obj_ classes, but are enhanced with extra query-specific logic. Further explanation of the query classes is provided in the SDK Concepts section.

Server API

A WITSML server by specification will expose a webservice api over HTTP or HTTPS. The most commonly used interface is the "WITSML STORE" interface, which simply defines methods for Get, Add, Update, and Delete. A simple login/password is provided with every request, and it is up to the server vendor to decide how to implement user security – some vendors may only provide full access for every login, while others may define a fully granular security model defining Read/Write permissions down to the individual object level. There is no standard WITSML interface to check security settings, so one must simply try to execute a query type and see if it succeeds or fails.

The login and password sent to the server is effectively clear text, so a high security server should use SSL (HTTPS) to prevent the login information from being viewed over the wire.

Server API Interaction

In addition to the WITSML objects represented by XML, the specification also defines how to interact with a remote WITSML server in order to store and retrieve data. The specification defines a method of querying the server by utilizing the same data objects as "query templates". This means that a query to retrieve a "Well" objects from a server will in fact send a specially formatted Well object to the server as part of the query request. The object will have fields populated as a way to tell the server what results to include and how to filter them. For example, to request a "Well" object with the name "Well-01", a new Well object would be created and the name populated with "Well-01", and then issued to the server as a query. The server will filter the well objects on the server and only return the one(s) that have a matching name of "Well-01".

A couple concepts that are important to grasp for these query templates-

- If a field is present in the query but does not define a value (ie blank), this indicates that the field should be included in the return results.
- If a field is present in the query and does define a value (ie, name="Well-01"), this means that the results should only include objects with fields that match this value- all wells with a name of "Well-01"
- Normally, any values not included in the query template will be excluded in the returned results, thus to get "all values" on an object, the full list of fields should be included in the query (with blanked values).
- However, most servers support an option called "returnElements=all" which tells the server to fully populate the requested object. Thus if we only added a name="Well-01" field to the query, the returned result would be a Well object(s) with every field populated. (see the SDK Concepts section for details on this feature)
- Case Sensitivity- any value specified as a filter in a query template will be matched on the server in a case-insensitive manner. Note that the original casing of strings is maintained on the server; just that matching is performed ignoring case.

The query templates do not support such features as "like" or wildcards, so any query with filters defined need to be an exact match.

SDK Quick Start

Initialization

If your code needs to access a server, create a WitsmlStoreClient:

```
var client = new Setiri.Witsml.Client.WitsmlStoreClient(url);
client.UserName = username;
client.Password = password;
```

```
Some servers handle basic auth incorrectly and require a workaround to force basic
authentication headers to be included on the first request to the server. Set this to
true if needed (applies to Java Axis based servers specificall)
```

```
client.ForceBasicAuth = true;
```

```
Set the base Units as a convenience method to automate populating units in some queries:
```

```
Configuration.SetUnitBase(
        UnitsDefaults.UnitsBase.US,
        UnitsDefaults.DegRads.Degrees);
```

Retrieve Objects

The simplest possible request is to query for a list of all Well objects on a server:

```
var wells = client.GetFromStore<obj_well>(
        new obj_wellQuery() { SelectAll = true }, true);
```

This simple query returns a Document<obj_well> object. The Document class is a container utilized to wrap one or more WITSML objects for both server queries and for data results. This object will iterate over the WITSML objects it contains, as shown:

```
foreach(var well in wells)
{
        Console.WriteLine(well.name + ": " + well.uid);
}
```

The query sent to the server is a obj_wellQuery. This is a query class used to help with querying obj_well's from the server. Each WITSML base object has an accompanying obj_query helper class defined.

Note that the second parameter for GetFromStore is set to "true". This value tells the server to return a fully populated object from the server, even though the query doesn't specify each field to return. This is supported by most servers, but some do not. This effectively sets the "returnElements" to "all".

This same example also sets "SelectAll=true" on the obj_wellQuery object. This is useful for servers that do not support the returnElements=all parameter. This method instructs the query object to include every field so that it will be sent to the server as part of the request, thus accomplishing the same thing

but through brute force. Without this method, the developer would need to manually set each query field to “select” manually to perform a full object query.

A second example will query wellbores from the server, but this time specifying only to return wellbores that are children of a parent Well with a UID of “Well-01”;

```
var bores = client.GetFromStore<obj_wellbore>(
       new obj_wellboreQuery() { uidWell = "Well-01"  }, true);
```

The next example demonstrates retrieving a Log object, with comments following:

```
var olq = new obj_logQuery()
{

          uidWell = “Well-01”,
          uidWellbore = “Wellbore-01”,
          uid = “MyLog-123”,
          description = Query.Select,

          startDateTimeIndex = new DateTime(2000, 1, 1),
          endDateTimeIndex = new DateTime(2020, 1, 1),
          logData = Query.Select,  //include this to get the actual data rows.
                                   //exclude to retrieve just the headers.

          logCurveInfo = new List<cs_logCurveInfoQuery>{
          {new cs_logCurveInfoQuery{
               uid=Query.Select, mnemonic=Query.Select,
                      columnIndex=Query.Select,
                      typeLogData=Query.Select}}}
};

var logs = client.GetFromStore<obj_log>(olq, true);
```

This example shows a few more concepts for querying WITML servers:

- uidWell and uidWellbore are specified in the query, along with the Log uid. This is good practice as some servers require the parent UIDs even when selecting a specific object by its own unique id.
- This is time based log, so we are querying data between the times of 1/1/2000 and 1/1/2020
- We set several fields to Query.Select. This instructs the query object to include that field in the query results. This is useful when creating queries that we only want to return particular fields, and not all of them.

Create, Update, Delete

Example- How to create a Well:

```
var well = new obj_well()
{
       name = “Well 01”,
       uid = “WELL-01”,
       dTimSpud = new DateTime(2005, 5, 5),
       field = "TSTFIELD",
```

```
        statusWell = WellStatus.active,
        timeZone = "-06:00";
};
var doc = new Setiri.Witsml.Document<obj_well>();
doc.Add(well);

short retval = client.AddToStore<obj_well>(doc);
```

Any other WITSML object is required to include the parent uid's during creation.
Data modification queries are similar to this example in that they send actual WITSML objects to the server instead of a query object. When creating a new object, the UID must be unique or else an error will be returned. The error number will be returned in the "retval" variable in this example. A successful operation will return a "1" value for this. Otherwise, the code will indicate an error. To view a message sent from the server, look at the LastMessage property of the client.

To update the same example well, simply create the same obj_well object, populate the same UID, and set any values you wish to update. The UID must exist on the server or an error will be returned. Then call client.UpdateInStore() instead of AddToStore().

Deletion is similar with these variances:

- Call the DeleteFromStore method
- Must provide the UID and all parent UIDs for the object to be deleted. These are the only values that will be observed during deletion.
- The object cannot have any children – ie, you cannot delete a Well object that has existing Wellbore objects
- Only one object can be deleted per server call – cannot send multiples in the Document.

SDK Concepts

Partial Object Population

When data is queried from a WITSML server, it is possible to specify only a subset of fields be returned in the query. The resulting object will have some fields that have null or empty values, but this is because the values were not queried, not because they do not exist. This can cause problems when validating objects (required fields missing) and for updating objects. It would be easy to issue an update query which sets all the fields to blank simply because the original query only selected on field to return!

State Tracking

For these partial population reasons, state tracking is included with each of the .Net WITSML objects. When an object is queries from the server, all fields returned are set to an Active state.

Similarly, when queried or newly created WITSML objects have the value of a field changed, the state of that field is set to "Dirty" to track that it has changed. Then when a create or update query is performed, just the Dirty state fields can be extracted to send to the server.

The field status is normally tracked internally, but can be accessed by the developer via the GetMember method call. Thus to check the Dirty or Active state on a field:

```
isDirty = myWell.GetMember("name").Dirty;  // or .Active
```

The full dictionary of members with their status is available via the Members property

```
var memberlist = myWell.Members;
```

Extended Data

In addition to the state tracking performed on every field, the WITSML objects in the sdk also include extended data for every field. This includes data type data such (such as data type and validation rules) as well as "helper" info such as friendly names and descriptions for each field.

This code demonstrates viewing some extended properties of a particular field:

```
var fldsch = myWell.GetMember("name").Schema;
req = fldsch.IsRequired;  //bool – is this a required field
ml = fldsch.MaxLength;  //int – max length (for string types)
val = fldsch.Validation; //rule for extended validation
```

In addition, each field has a "pretty" name and a description. These are pulled from the WITSML specification, and so are the "official" descriptive names for each field. These are useful for building dynamic applications to work with any WITSML object type, as the field names can by dynamically loaded from the schema and displayed within a user interface:

```
fname =fldsch.PrettyName;
desc = fldsch.Description;
```

These would then allow displaying a string like “Well Name” instead of just “name” in the UI.

Object Validation

The WITSML specification defines data requirements on the data contained within each object types. These include:

- Required fields
- Max data length (for string types)
- Min and Max range (for numeric types)

The SDK has codified all these requirements such that a Validate() method can be called which will check all these constraints and report any invalid values. Call this method on any base WITSML object:

```
var errs = well.Validate(true, false);
```

The Boolean parameters control whether to validate only the Active elements or only the Dirty elements only, instead of validating every field (the default).

The return value is a Dictionary<string,string> object which contains a name-value pairing of field names and their respective data validation error description.

Units of Measure

The WITSML specification includes a library of units of measure, with translations among compatible types. The UOM’s in this spec are the ones that a valid WITSML document will support. A server will usually have a default UOM type- i.e., a server will usually prefer either metric or US standard units, and is allowed to store values in its preferred units set. Friendlier servers will convert to your preferred units before returning your data, but are not required to. Thus, whenever an object is added to a server, the units may be convert to something else, AND when querying data from a server the results may be returned in units you did not request. The SDK includes helper libs for a. enforcing the use of only units that are supported by WITSML and b. enabling unit conversion for the client software.

The Units Manager is exposed as a singleton class via:

```
var unitsman = Units.Manager.Instance;
```

Units can be looked up by their string names as follows:

```
var psi = unitsman.GetUnit(“psi”);
```

After more than one unit has been instantiated, a conversion can be performed between them:

```
var newval = unitsman.Convert(psi, kpsi, 1234.5);
```

The manager will attempt to find a conversion path between the two units types, and if one exists, will convert the value to the new unit of measure value.

In addition to the Units Manager (used specifically for conversions), many unit names are defined in enhanced enumerations so they can be strongly referenced by WITSML object fields. An example is the groundElevattion field on the Well object:

```
well.groundElevation = new wellElevationCoord() {
        uom = WellVerticalCoordinateUom.ft, Value = 123 };
```

This groundelevation value is strongly typed to wellElevationCoord, which in turn has the uom and Value fields. The uom field is an enumeration of the acceptable units of measure for the wellElevationCoord, and the assigned value from the enum in this instance is “ft” (feet).

XML and String Processing

As part of WITSML is an XML based data standard, not all WITSML software needs to interact with servers. WITSML can also be saved to and read from system files, or even just strings within an application. The SDK can convert the objects to valid WITSML XML strings to be utilized in these scenarios.

String based WITSML can be extracted from the same Document object used in querying the server, but instead of passing the document to the client object, extract the XML string via:

```
var xml =doc.XmlString(PrettyFormat:true)
```

The PrettyFormat parameter tells the parser to produce indented XML text that is easier to read. This is only useful if the output needs to be viewed by humans.

By default, the XMLString method will include all the “Dirty” values, and ignore the “Active” values. This is most useful for objects that were created locally and not retrieved from a server. The Active, Dirty, and All parameters can be set to tune the output according the needs of the developer. Any object queried from a server would benefit from turning on Active:true as well so that the values returned from the server (set to Active, but not set to Dirty) will be serialized in the string as well.

To load a WITSML object from an XML string, use:

```
doc.XmlDeserializeString(xmlstring);
```

Recurring Object Queries

Most object server queries are relatively straightforward, but WITSML objects with recurring elements can be trickier to correctly query. For instance, a Log object has multiple “channels” with usually multiple (and often many) rows of data. It is not too difficult to select this data based on time or depth

range, but more advanced operations require attention. Such scenarios include adding a channel to an existing log, appending data to an existing log, and updating existing data in a log.

The three objects to be concerned with for these operations are the Log, the Mudlog, and the Trajectory objects. The special attention required for each is covered below.

Systematically Growing

The Log object is considered "systematically growing" as the data it contains is expected to be contiguous and in a series. This is distinguished from a randomly growing object which defines start and end values for each interval in the object, and is not necessarily in any particular sequence. Systematically growing objects consider the index value – i.e., the depth or the timestamp – to be the key value for the row of data, whereas randomly growing objects must assign a unique id to each interval to allow accurate referencing.

Server query details

The Log object contains an index range at the header level of the object. If these values – startIndex an endIndex – have values assigned during a select query, this sets a global range for the query and for every channel included in the query. The returned object from the server will have these same values set to the actual start and end index of the data returned – in such cases where the start or end index values fell some distance within the initial range, the new range will reflect the new true range of the data.

dataRowCount is another property at the header level – Set this to a maximum row count to return in order to limit the size of the returned data.

When the logData parameter is included in the query, the actual data in the log will be returned. The logData field will contain a List<string> containing a comma-delimited string of the data values. If a value is missing, it will just be blank. However, missing values will not be blank if a "nullValue" has been assigned to the logCurveInfo, such as "-9999" or similar. The order of the values in the delimited data matches with the columnIndex value defined on each logCurveInfo defined on the log.

Update Queries

When updating a Log object, the startIndex and endIndex values take on added importance: If a start and end index is included in the update query, the server will replace any rows on the server in that range with the new ones provided. Effectively this deletes any values between the start and end, and then adds the new data to the log.

If the start and end indexes are left empty, then the update query will simply append the new values to the existing log without overwriting or deleting any existing values.

If a new channel is defined in an update query (that doesn't already exist on the server log), the server will add the new channel to the log and merge any values for it with the existing values, keyed off the index value.

Delete Queries

Normal delete queries only concern themselves with the UID of the object and then remove it from the server. In a Log object query, one can specify the start and end index values in the query, and the server will delete any data that falls within this range. If logCurveInfos are also added to the query and the channel mnemonics added to them, then the delete query will only delete values within the range *and* for the specified channels.

If the logCurveInfo is defined but no start/end Index is specified, the server will delete the entire log curve from the log.

Randomly Growing

Randomly growing objects include the MudLog and the Trajectory. The repeating sections of these objects are defined by intervals with starting and ending values instead of single index values (as a Log does). These objects also include a range in the header (startMd and endMd for the MudLog, and mdMn and mdMx for the Trajectory), but for these objects this range is the encompassment of the smallest starting interval and the largest end interval.

Since every interval in a randomly growing object must have a UID defined, data modification queries are simpler to perform than with the systematically growing (Log) object, as each interval is uniquely defined by the UID.

Custom Data

WITSML objects define many fields, but it is impossible to accommodate every piece of data that might need to be stored with a particular data object. The specification allows for appending custom data to any object and store it as valid data within the XML and for storage on a server.

The custom data should be valid XML, and can be set on an object in this manner:

```
well.customData = new Setiri.Witsml.XmlList().FromXml(
       "<myData>this is my own data</myData>");
```

If the customData field is included in a query to the server, this same custom xml will be returned in the query results and available for parsing out of the customData field. Some vendors may allow sending a selection query for the contents of the customData, but it is up to the vendor to specify this and communicate the special handling to the customer.

Server Variations

The WITSML specification has some undefined areas, and so server vendors are left to implement as they interpret them or to simply guess at the best implementation route. In other cases, some vendors simply do not comply with the specification, whether through error or just partial implementation.

We've listed a few of the observed variations among vendors here as a helper for what to look for when developing against a server, and perhaps assist in debugging encountered issues.

- Required values: Some servers seem to require fully populated parent UID's for any data modification operation, while others are ok with just the UID for the object itself. Best practice is to always include the well and wellbore id in any query.
- Too large result: Some servers will return an error if your query attempts to retrieve too many records in one request. You will need to restrict your query to give fewer results and try querying multiple times.
- Unicode support: The specification does not require support for international character sets or Unicode. Some servers correctly support Unicode and multi-byte characters (like Chinese) while others only support latin/western 8-bit character sets.
- UOM required in query: Some servers will return the server default UOM if none is defined in a query, while others will error and require a specific uom be defined.
- returnElements – Setting the ReturnAll parameter in the GetFromClient() method creates a server option returnElements=All, which tells the server to return all the fields of the object even when the query only specifies a few fields. Most vendors implement this correctly, but some simply ignore it. In these instances, a backup option on the obj_XXXQuery objects is SelectAll, which instead sets all fields in the query to be included in the query to the server. This effectively forces the server to return all elements.

Note that a server is not required to support every part of the specification, but it should publish which parts it does support. The supported methods and classes can be discovered for a server by using the GetCapabilities method:

```
var caps =Client.GetCapabilities();
```

The results of this query is an XML string. The current SDK version does not encapsulate the results of this object in an object, so the XML must be parsed. A future release of the SDK will add capabilities for further processing the server capabilities response.