About this document

This document assumes the knowledge of web services concept, SOAP protocol, XML and XML Schema (XSD) formats. It does not describe how to call web service using product X – you can find this information in corresponding product documentation, rather provides information and describes some common techniques related to TeamDesk SOAP API.

Web Service URL

Each application has own web service address. You can find the address specific to your application by entering *Setup* mode, selecting *Setup* tab, and clicking *Integration API/TeamDesk SOAP API* link in *Application* section. The link will navigate you to a page containing formal description of the web service. All calls to web-service are made via the same URL. You can retrieve service definition in WSDL format by adding `?wsdl` to the end of URL.

Authorize the user

```
result = api.Login("youremail@server.com", "password");
```

First thing you need to start working with the API is to authorize the user by performing `Login()` call.

The result returned is a `LoginResult` structure consisting of:

- `SessionId`: globally unique identifier of the session. This identifier is returned in the SOAP header as well and should be present in SOAP headers of all subsequent API calls. Smart SOAP clients such as .NET will capture it from `Login()` call output headers and pass it back for further calls automatically. For simple clients you can capture the id from the structure and pass it back as header by hands.
- `UserInfo`: structure describing various user properties and consisting of:
    - `Id`: integer, the internal id of the user
    - `FirstName`: string, the first name of the user as registered in TeamDesk
    - `LastName`: string, the last name of the user as registered in TeamDesk
    - `Email`: string, the email address (as passed to Login call)
    - `FullName`: string, the combination of `FirstName` and `LastName`
    - `Locale`: string, the culture name in the format "`<languagecode2>-<country/regioncode2>`", where `<languagecode2>` is a lowercase two-letter code derived from ISO 639-1 and `<country/regioncode2>` is an uppercase two-letter code derived from ISO 3166.
    - `TimeZone`: integer, the code of the time zone.

API respects user privileges as defined by user's role. Thus on an attempt to access objects for which the user rights are restricted, an exception will be generated.

Getting the data from the server

Once you have authorized the user, you can start querying the data. There are two methods allowing you to get the data from selected table

Query() method

```
result = api.Query("query string");
```

Query method accepts the query string in an SQL-like syntax. The syntax can be described as:

```
SELECT { TOP n } <column-names> | * FROM <table> { WHERE <condition> } { ORDER BY <column-names> }
```

(curly brackets designate optional clauses)

Column and tables names are always square bracketed. FROM clause accepts single table addressed by its name in a singular form. No JOIN syntax allowed - other tables' data can be queried via lookup/summary columns. Column names and table names are case insensitive. Syntax of `<condition>` is fully identical to view's filter formulas.

Some examples:

```
SELECT TOP 10 [A], [B], [C] FROM [TABLE] WHERE Contains([A], "aaa") ORDER BY [A], [B] DESC
```

```
SELECT [A] FROM [TABLE] WHERE [A] <> [B]
```

```
SELECT [A] FROM [TABLE] ORDER BY [A]
```

```
SELECT * FROM [TABLE]
```

Returned is the data in XML format. This format will be described later in this document.

Retrieve() method

```
result = api.Retrieve("Table", <array-of-column-names>, <array-of-ids>);
```

The record can be also addressed by its internally assigned id. This id is present in URLs TeamDesk generate for the records and also returned from some calls such as `Create()`, `Upsert()`, `GetUpdated()` and `GetDeleted()`. So, if you've an id or the set of ids, you can use Retrieve method to get the data.

The method accepts the name of the table in its singular form, the array of column names and the array of ids. If you need to retrieve single record, pass an array consisting of single element.

Returned is the data in XML format, as in Query call.

XML format of the data

Practically, there is no common interchange format for the data. The format we've chosen provides the definition of the data structure and strict data typing along with the data. Moreover, if you are using .NET

technologies, you can simply transform it from XML to the `DataSet` or `DataTable` class and back simple functions such as:

```
public static DataTable ToDataTable(XmlElement apiResult)
{
	DataSet ds = new DataSet();
	ds.ReadXml(new StringReader(apiResult.OuterXml));
	return ds.Tables[0];
}

public static XmlElement FromDataTable(DataTable table)
{
	StringWriter w = new StringWriter();
	table.DataSet.WriteXml(w, XmlWriteMode.WriteSchema);
	XmlDocument doc = new XmlDocument();
	doc.LoadXml(w.ToString());
	return doc.DocumentElement;
}
```

For those if you how are interested in internal details or using technologies other than .NET, the data returned from `Query()` and `Retrieve()` calls has the following format:

```
<DataSet xmlns="urn:soap.teamdesk.net:dataset">
	<xs:schema>...</xs:schema>
	<Data>...</Data>
</DataSet>
```

The content of `<xs:schema>` element is a standard XML Schema Definition (XSD) and describes names and data types of the data contained in the Data element. The most of schema content is standard and does not change.

```
<xs:element name="DataSet">
	<xs:complexType>
		<xs:choice maxOccurs="unbounded">
			<xs:element name="r">
				<xs:complexType>
					<xs:sequence minOccurs="0" maxOccurs="unbounded">
					... here comes the list of columns...
					</xs:sequence>
					<xs:attribute name="id" type="xs:int"/>
					<xs:attribute name="m" type="xs:boolean"/>
					<xs:attribute name="d" type="xs:boolean"/>
					<xs:attribute name="c" type="xs:string"/>
				</xs:complexType>
			</xs:element>
		</xs:choice>
	</xs:complexType>
</xs:element>
```

The list of columns" is a set of following elements:

```
<xs:element name="<column-name-encoded>" msdata:Caption="<column-name>" type="<column-xml-type>" minOccurs="0"/>
```

As column name in TeamDesk is a combination of any characters and not all of them are allowed in XML

element names, the name of the column in <column-name-encoded> is be encoded according to the following rules:

The escape character is "_". Any XML name character that does not conform to the W3C Extensible Markup Language (XML) 1.0 specification is escaped as _xHHHH_. The HHHH string stands for the four-digit hexadecimal UCS-2 code for the character in most significant bit first order. For example, the name Order Details is encoded as Order_x0020_Details.

The underscore character does not need to be escaped unless it is followed by a character sequence that together with the underscore can be misinterpreted as an escape sequence when decoding the name. For example, Order_Details is not encoded, but Order_x0020_ is encoded as Order_x005f_x0020_. No shortforms are allowed. For example, the forms _x20_ and __ are not generated.

To help with decoding, `msdata:Caption` attribute holds column name decoded.

`column-xml-type` is a column (or formula) type mapped to XSD type according to the following scheme:
Text, Multiline, Autonumber, Phone, Email and Url are mapped to `xs:string` type.
Boolean is mapped to `xs:boolean`
Date is mapped to `xs:date`
Time is mapped to `xs:time`
Number is mapped to `xs:decimal`
Timestamp is mapped to `xs:dateTime`, timestamps are returned/accepted in GMT time zone
Duration is mapped to `xs:duration`
Attachment is mapped to `xs:string`, the name of the recent file version is returned.
User is mapped to `xs:string`, contains the name and e-mail of the user in the form of "Name" <email>

In other words, <Data> element contains the number of <r> elements, each <r> contains number of elements with names equal to column names (encoded). Such elements contain no other elements but textual data in the format according to their data type.

<r> element also holds some system data in its four attributes:
"id" attribute holds internal id of the record, as in Retrieve()
"m" attribute results to "true" if the record can be updated or to "false" otherwise
"d" attribute results to "true" if the record can be deleted or to "false" otherwise
"c" attribute results to color string if row colorization formula is defined on table level.

For example, from the Retrieve call for columns Id and Last Modified you'll get back something like:

```
<DataSet>
<xs:schema>...</xs:schema>
<Data>
<r id="1" m="true" d="true" c="">
<Id>1</Id>
<Last_x0020_Modified>2009-11-12T23:00:00Z</Last_x0020_Modified>
</r>
```

```
</Data>
</DataSet>
```

Modifying the data on the server

There are three methods to modify the data on the server: Create(), Update() and Upsert().

Create() method

```
result = api.Create("Table", xml-data);
```

The method accepts the name of the table in its singular form and the data in XML format as returned from `Query()` or `Retrieve()` calls. The method creates new record or records and returns an array of their internal ids.

If you are using .NET's it might be tempting to create new `DataSet/DataTable`, add some rows in there and pass the data converted to XML to `Create()` call. The common pitfall is that the XML format produced by `DataSet` with its default settings is different from what TeamDesk API expects. Setting up `DataSet` to produce proper format might be a non-trivial task. More easier approach would be to query the table for an empty result set – the XML returned from the server will contain the schema, but no data – convert it to `DataSet`, add some rows and pass it back to `Create()` call. Below are two helper methods you can use in your code. First one query the schema for all columns from the table, second one retrieves the schema for selected set of columns.

```
// Get empty DataTable with all columns
public DataTable GetSchema(string table)
{
	return ToDataTable(api.Query(String.Format("SELECT TOP 0 * FROM [{0}]", table)));
}

// Get empty DataTable with selected columns
public DataTable GetSchema(string table, string[] columns)
{
	return ToDataTable(api.Retrieve(table, columns, new int[] { 0 }));
}
```

Update() method

```
api.Update("Table", xml-data);
```

The method accepts the name of the table in its singular form and the data in XML format as returned from `Query()` or `Retrieve()` calls. The method updates existing record or records based on their internal ids contained in the XML data. If the record is not found by its id, an exception is thrown.

.NET users please note that the method updates all the records that are passed in xml-data parameter. In a database scenario `DataSet` class is tracking the changes and updates only new or modified rows. This information is, however, lost when you convert the `DataSet` to XML. You can obtain 1000 records, convert XML to `DataSet`, modify one or two – but when you convert this back to XML and send it back, all 1000 records will be updated, increasing traffic and server load and processing time. Luckily, `DataTable.GetChanges()` method can help you to obtain a copy of your data containing modifications only.

Upsert() method

```
api.Upsert("Table", xml-data, reserved);
```

This methods combines the functionality of `Create()` and `Update()` methods. First it tries to find and update the record. If the record is not found, new record is created. Third parameter is string, reserved for future use. Pass empty string in it.

Delete() method

```
api.Delete("Table", <array-of-ids>);
```

The method deletes the record or records identified by their internal ids.

Simple Sample

The following piece of code gets the schema from imaginary table, adds new record with some text and numeric data, queries the data back, and updates the text. The code uses `GetSchema()`, `ToDataTable()` and `FromDataTable()` helper methods described above in this document.

```
DataTable dt;
DataRow dr;

dt = GetSchema("Table", new string[] { "Text", "Numeric" });
dr = dt.Rows.Add();
dr["Text"] = "A text";
dr["Numeric"] = 2.5;

// Other types serialized as shown below
// dr["Bool"] = true;                          // true/false
// dr["Date"] = new DateTime(2008, 1, 1);      // use DateTime, time (if any) is ignored
// dr["Duration"] = TimeSpan.FromHours(2.5);   // use TimeSpan
// dr["User"] = "email@server.com";            // user's email
// dr["E-Mail"] = "email@server.com";
// dr["URL"] = "http://www.teamdesk.net";
// dr["Timestamp"] = DateTime.Now; // pass LOCAL time

// will create a row
api.Create("Table", FromDataTable(dt));

dt = ToDataTable(api.Query("SELECT [Text] FROM [Table] WHERE [Text] = \"A text\""));
for(int i = 0; i < dt.Rows.Count; i++)
{
       dr = dt.Rows[i];
       dr["Text"] = "A text 2";
}
// will update rows
api.Update("Table", FromDataTable(dt.GetChanges(DataRowState.Added|DataRowState.Modified)));
```