software AG

HTTP Client API for JScript

Version 9.5 SP1

November 2013

tamino XML Server

This document applies to HTTP Client API for JScript Version 9.5 SP1.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: HAS-DOC-95SP1-20131030

Table of Contents

HTTP Client API for JScript

This document provides information about the HTTP Client API for JScript. On Microsoft Windows, this API offers a variety of methods and properties which enable applications, written in JScript, to access and manipulate documents in a Tamino database.

This document is intended for software developers who wish to create applications on the basis of JScript that access XML databases stored in Tamino. It is assumed that you are familiar with using the Tamino Manager to create databases, and with using the Tamino Interactive Interface to load schemas and data and perform XML database queries.

This document contains the following sections:

I

1 Introduction

Client applications can communicate with Tamino using APIs that are available for JScript, Java and ActiveX. This is more convenient than using the native HTTP protocol. The APIs then communicate with Tamino at the *HTTP* protocol level. The basic functionality of the three HTTP Client APIs is the same, independent of the language environment. The names of classes and methods may differ.

The three APIs are called *client* APIs because the application or program that communicates with Tamino is a client with respect to the server Tamino. In this context, a servlet would also be a client. Each of the APIs supports the W3C DOM (Document Object Model) Level 1 specification. The APIs are also called *Tamino DOM APIs*. DOM support is implemented in such a way that the API's methods supply a DOM object as a result, or require a DOM object as input. The programmer can then use standardized DOM methods and interfaces to further manipulate the result or navigate through a DOM tree.

The DOM is designed to be used with any programming language. It defines the logical structure of documents and the way a document is accessed and manipulated. With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content.

For more information about the DOM specification, see **http://www.w3.org/DOM/**.

By supporting the DOM specification, the Tamino APIs not only allow their data to be manipulated by other routines, but do so in a way that allows those manipulations to be reused with other DOMs, or to take advantage of solutions already written for those DOMs.

The intention is that any DOM implementation can be plugged together with any DOM-based application. You can use any DOM implementation that supports the W3C specification, and you can change the implementation or parts of it later; your application is independent of the implementation.

General Functionality

The Tamino HTTP Client APIs are object-oriented programming interfaces to Tamino that offer a variety of methods and properties for communicating with the Tamino X-Machine. The implemented APIs provide the same functionality over all supported platforms unless stated otherwise.

The names of methods and properties may differ depending on the language environment.

The main methods that are provided are listed below. A complete list of all methods and properties can be found in the API Reference Section of the respective API documentation. The supported methods can be classified in the following way:

- Methods to manipulate data in Tamino via URLs, such as
  - `query`

  - process
  - insert
  - inodelete
- Methods to manipulate data in Tamino via document names, such as
  - putDocument
  - getDocument
  - deleteDocument
- Methods to control sessions and transactions
  - startSession
  - endSession
  - commit
  - rollback
- Methods to manipulate the Tamino query result, such as
  - getResult
  - getInoId
  - getFirst
  - getPrev

Session Management

If a program needs to open and close database transactions, then there must be a mechanism that allows Tamino to keep user contexts. This can be achieved by invoking the `StartSession` and `EndSession` methods of the APIs. The APIs send the commands `connect` and `disconnect` via HTTP to Tamino. This enables Tamino to keep the user context and deliver a session key and session ID back to the API. Session control is then handled by the API transparently to the user.

If a client program needs to handle session control by itself, there are methods available to get/set these parameters. For example, it could be possible for a client and server program to share the same session. The appropriate methods `setSessionId`, `getSessionId`, `setSessionKey` and `getSessionKey` are not available in JScript.

The `StartSession` method also has the optional parameters `Isolationlevel` and `LockWaitMode`. By providing these parameters instead of using the Tamino defaults a more specific security and/or locking mechanism can be achieved.

For transaction control, the `commit` and `rollback` methods are available. These methods can only be called during an open session.

2 Component Profile and Installation

This chapter provides information about the contents of this Tamino component and the installation.

Component Profile

Here you will find general information about the API and how to use it.

The Tamino HTTP Client API for JScript is installed with the Tamino XML Server installation.

Tamino Component Profile for the HTTP Client API for JScript	
Supported Platforms	All Windows platforms that support Microsoft Internet Explorer.
Location of Installed Component	*<TaminoInstallDir>\SDK\JScriptAPI* (henceforth called *<JScriptAPIDir>*)
Component Files	API JScript file: *<JScriptAPIDir>\TaminoLib.js*
	Examples: *<JScriptAPIDir>\examples\SampleChangeCountry.htm*
Sample Schema	*<TaminoInstallDir>\SDK\JScriptAPI\Documentation\examples\phone\TelephoneSchema.tsd*
Sample Data	*<TaminoInstallDir>\SDK\JScriptAPI\Documentation\examples\phone\Telephone.xml*

To work with the HTTP Client API for JScript

- You need a running Tamino database server to deploy an API. If you want to access an existing Tamino database, you must know its URI (such as `http://localhost/tamino/mydb`). In the client HTML page, make sure that it refers correctly to the file *TaminoLib.js*.

Auto-Detect of MSXML3/4

The controls attempt to auto-detect the latest MSXML parser - first MSXML4 and then MSXML3. This behavior can be overridden. Use the method `UseServerHTTP` for the ActiveX controls. Use the SYSTEM environment variable `SAG_NODELEVELUPDATE_MSXML (MSXML3/MSXML4)` for `NodeLevelUpdate`. It is recommended to use MSXML3 with `NodeLevelUpdate`

3 Using the API

JScript is Microsoft's version of JavaScript. The API described here relies on features of JScript that are not available in JavaScript.

What is Provided

The API provides a DOM (Document Object Model) oriented interface to Tamino for Windows systems running Microsoft's Internet Explorer version 5.x or 6.x (IE). The script library can be used in the IE browser, or in any server-side environment that supports active scripting such as Active Server Pages (ASP) or Software AG's System Management Hub. This section gives an overview of the main objects and their methods that are provided in the API, and it indicates the basic programming techniques that you can use to access and modify a Tamino database. For a full definition of the objects and methods provided by the API, see the section **API Reference Section**.

TaminoLib.js Library

The functionality of this API, including HTTP communication with the Tamino X-Machine, is provided in the *TaminoLib.js* library. This library contains objects and methods for retrieving and modifying data in a Tamino database. The library is located in the directory *<TaminoInstallDir>\SDK\JScriptAPI*.

Instantiating TaminoClient

One of the first steps in your JScript code is generally to create a `TaminoClient` object. The definition of this object is provided in the *TaminoLib.js* library. One of the arguments that you specify when you create the object defines the database and collection that is to be accessed whenever any of the object's methods is invoked. The methods provided by this object perform all of the basic communication operations with Tamino, such as submitting an XML database query or creating, updating or deleting data.

To create a `TaminoClient` object that can be used to communication with a given Tamino database, use statements of the form:

```
var dbname="http://localhost/tamino/mydb/mycollection";
var QueryObj;
QueryObj = new TaminoClient(dbname, ...);
```

where *mydb* and *mycollection* are the names of the database and collection that you wish to access.

Performing a Query

You can use the `query` method of the newly-created object to submit a query to the database, using statements of the form:

```
var QueryResult;
var QueryVal="xqlquery";
QueryResult = QueryObj.query(QueryVal);
```

where "xqlquery" is a database query such as `Telephone[Address/City="Frankfurt"]`. The result of this operation is an object (named "QueryResult" in this example) that contains the data that Tamino returns for the query. You can use the `getResult` method of the `QueryResult` object to access the query result element:

```
var xqlResult=QueryResult.getResult();
```

This creates an object that contains the `xql:result` part of the data that Tamino returns.

To get a nodelist, you can now use the DOM method `childNodes` of the `xqlResult` object:

```
var nodelist=xqlResult.childNodes
```

If many documents in the Tamino database match the query, Tamino restricts the number of matching documents to the page size that you specified when you created the TaminoClient object. The default page size is 5. A value of 0 means 'no limitation'.

To access any given document in the result list, use statements of the form:

```
var itemSelected;
itemSelected = nodeList.item(n);
```

where `n` is the index of the document that you want to access (the indexing starts at 0, not 1).

You can apply various DOM methods to the document thus retrieved. For example, to access the elements of the document, use the DOM method `getElementsByTagName` in statements of the form:

```
var zip;
zip = ↩
itemSelected.getElementsByTagName("elementname").item(0).childNodes.item(0).data;
```

where `elementname` is the name of the element.

Microsoft's Internet Explorer contains an implementation of the DOM for JScript, so any HTML page that contains JScript code has access to the DOM interface when it is viewed in this browser.

Updates

To write a modified node back to the Tamino database, use the `process` method of the `TaminoClient` object, specifying as a parameter the name of the object that contains the modified element, for example:

```
var storeResponse = QueryObj.process(itemSelected);
```

where `QueryObj` is the `TaminoClient` object you created previously.

The `process` method issues a `_process` request to the X-Machine, thus causing the document to be updated in the database. Since the root element of each document has a unique value for the mandatory attribute `ino:id`, the document to be modified is identified uniquely.

The `process` method builds a URL of the form `http://hostname/DatabaseLocation/DatabaseName/CollectionName?_process="UpdatedDocument"`, where `UpdatedDocument` is the contents of the document, and sends this URL to Tamino. See the section *Requests using X-Machine Commands* for more details.

Dynamic HTML

You can use Dynamic HTML to build an interactive application that accesses the database, retrieves data, displays the data in the browser, allows you to change the data, and sends the changed data back to be stored in the Tamino database.

Transaction Processing

The JScript library *TaminoLib.js* also includes functionality for transaction processing. You can use the `startSession`, `endSession`, `commit` and `rollback` methods of `TaminoClient`.

4 Example

This section provides an example of how to use the JScript API to access and modify data in a Tamino database.

The example accesses a database that contains an extract from a fictitious telephone directory. It selects all people in the directory who live in Frankfurt, updates their country, then writes the modified data back into the database. When the HTML page that contains the JScript code is viewed in a browser, a button containing the text "Press here to start" is displayed. When you choose this button, the JScript code runs through to completion, updating all country elements as described. The JScript code does not display the updated database, so to view the changes, either you can use the Tamino Interactive Interface, or you can view the results in the browser using a combined URL and an appropriate database query in the browser's address line.

The following sections provide more detail:

Running the Example

To run the example, perform the following steps:

1. Use the Tamino Manager to create a test database. This example assumes that the database name is *mydb*. The collection name is *Telephone*.
2. Use the Tamino Interactive Interface to define the “Telephone” schema that the example requires. The source file that contains the schema is TelephoneSchema.tsd.

   Then use the Tamino Interactive Interface to load the test data from the file Telephone.xml into the Tamino database.

   Data for each person in the telephone directory is stored in an element named `Telephone`. A typical `Telephone` element is:

```
<Telephone>
  <EntryID>127</EntryID>
  <LoginName>Wehner127</LoginName>
  <PassWord>Wehner127</PassWord>
  <Lastname>Wehner</Lastname>
  <Firstname>Anton</Firstname>
  <Date_of_Birth>05.01.1945</Date_of_Birth>
  <Company_Name>KdH AG</Company_Name>
  <Salutation>Herr</Salutation>
  <Email>Wehner127@web.de</Email>
  <Address>
    <Street>Schulplatz 189</Street>
    <City>Darmstadt</City>
    <ZIP>64287</ZIP>
    <Country>Germany</Country>
    <Telephone>06150-113141387</Telephone>
    <Fax>06150-1649896197</Fax>
```

```
  </Address>
</Telephone>
```

At this point, you can check to see how many people in the test database live in Frankfurt by using the Tamino Interactive Interface to submit the query `Telephone[Address/City="Frankfurt"]` to the database.

3. Open the HTML file *SampleChangeCountry.htm* (see below) in your browser. This page contains the embedded JScript code to access and modify the telephone directory. When you view the HTML page, you will see a button with the text "Press here to start". Choose the button to activate the JScript code. After a few seconds, the code completes and a result message is displayed in the browser.

   The effect of the JScript code is to change the country of all persons living in Frankfurt. If the country is "Germany" it is changed to "Deutschland" and vice versa.

4. Use the query `Telephone[Address/City="Frankfurt"]` to display the new entries for all people living in Frankfurt and check that the country elements have changed.

Implementation

The HTML page that contains the JScript source code is *SampleChangeCountry.htm* (the file is provided with this software package). The contents are as follows (line numbers are prepended to simplify the explanations):

SampleChangeCountry.htm

```
01:  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
02:  <HTML>
03:  <HEAD>
04:  <TITLE>Test Page for Tamino's JavaScript API</TITLE>
05:  <SCRIPT LANGUAGE="JavaScript" SRC="..\TaminoLib.js"></SCRIPT>
06:  <SCRIPT LANGUAGE="JavaScript">
07:  function ChangeCountry ()
08:  {
09:      var dbname="http://localhost/tamino/mydb/Telephone";
10:      var QueryObj;
11:      var QueryResult;
12:      var itemSelected;
13:      var country;
14:      var xqlResult;
15:      var countdg = 0;
16:      var countgd = 0;
17:   var QueryVal='Telephone[Address/City="Frankfurt"]';
18:      var pageSize=12;
19:
20:   document.write("<P>Processing query '" + QueryVal + "' in database/collection ↩
```

```
'" + dbname + "'.");
21:
22:      QueryObj = new TaminoClient(dbname,pageSize);
23:      QueryResult = QueryObj.query(QueryVal);
24:      if (QueryResult.errorNo == "0") {
25:        xqlResult=QueryResult.getResult();
26:        if (xqlResult)
27:          {
28:          var nodelist = xqlResult.childNodes;
29:          QueryObj.startSession();
30:          while (nodelist)
31:            {
32:            for (i=0;i<nodelist.length;i++)
33:             {
34:              itemSelected = nodelist.item(i);
35:              country = ↩
itemSelected.getElementsByTagName("Country").item(0).childNodes.item(0).data;
36:              if (country == "Germany")
37:                {
38:                ↩
itemSelected.getElementsByTagName("Country").item(0).childNodes.item(0).data = ↩
"Deutschland";
39:                countgd++;
40:                };
41:              else
42:                {
43:                ↩
itemSelected.getElementsByTagName("Country").item(0).childNodes.item(0).data = ↩
"Germany";
44:                countdg++;
45:                };
46:              QueryObj.process(itemSelected);
47:              };
48:
49:            QueryResult = QueryResult.getNext();
50:            if (QueryResult)
51:              {
52:              xqlResult= QueryResult.getResult();
53:              nodelist = xqlResult.childNodes;
54:              }
55:            else
56:              {
57:              nodelist = null;
58:              };
59:            };
60:          QueryObj.commit();
61:          QueryObj.endSession();
62:
63:          document.write("<P>Processing ended.");
64:          document.write("<BR>Changed " + countdg + " times 'Germany' to ↩
'Deutschland'.");
65:          document.write("<BR>Changed " + countgd + " times 'Deutschland' to ↩
```

```
'Germany'.");
66:
67:           return(1);
68:           }
69:         else
70:           document.write("<P>No Data Returned");
71:         }
72:       else
73:         document.write("<P>Error = "+QueryResult.errorText);
74:  };
75:  </SCRIPT>
76:  </HEAD>
77:  <BODY>
78:  <P>Please press the button to start processing ...
79:  <form>
80:  <input type="button" name="Button1" value="Press here to start" ↩
onClick="ChangeCountry()">
81:  </form>
82:  <P>... and wait for the result to be displayed.
83:  </BODY>
84:  </HTML>
```

Explanatory Comments

Line 5

`<SCRIPT LANGUAGE="JavaScript" SRC="../TaminoLib.js"></SCRIPT>` causes the browser to read in the standard Tamino JScript library from the file *TaminoLib.js*. This library provides the low-level HTTP communication between the JScript client (i.e. the current HTML page) and the Tamino database. See the **API Reference Section** for a full description.

Lines 7-74

define the function ChangeCountry, which accesses the Tamino database and modifies the country of anyone who lives in Frankfurt.

Line 80

creates the button that is displayed with the text "Press here to start" in the browser window. The `onClick` event handler causes the function `ChangeCountry` to be activated when you choose the button.

Line 9

`var dbname="http://localhost/tamino/mydb/Telephone";` specifies the HTTP address of the Tamino database *mydb* and the collection *Sample*. This example assumes that the database is on your local machine, so "localhost" can be used in the URL of the database. Be aware that if you specify the collection name in the URL you cannot specify this or another collection name in the methods supporting this (optional) parameter. You can use the property `XMLDB` to change the URL or collection name if necessary.

Line 17

defines the query that is used in line 23 to retrieve all `Telephone` elements whose `city` element has the value "Frankfurt", i.e. to retrieve the data for all people living in Frankfurt.

Line 18

specifies the page size, i.e. the maximum number of `Telephone` elements that should be returned per call to the database. The first call will retrieve entries 1 to 12, the next call will retrieve entries 13 to 24 and so on. The API places no upper limit on the page size you can specify, but there may be system limitations, such as the amount of memory available on your computer, which limit the page size.

Line 22

creates the `QueryObj` instance of the `TaminoClient` object.

Line 23

invokes the `query` method of the `QueryObj` object. This causes the query that is stored in the `QueryVal` variable to be sent as an HTTP request to the URL of the Tamino database. The result document is stored in the `QueryResult` object. The number of documents that match the query is limited to the value specified in the pageSize variable. In this example, pageSize was set to 12, so if there are more than 12 documents that match the query, only the first 12 will be returned. To return the remaining documents that match the query, the loop mechanism at line 30 is used.

Line 25

invokes the `getResult` method of the `QueryObj` object. This causes the `xql:result` element of the returned result structure to be delivered and stored in the variable `xqlResult`.

Line 28

extracts the child nodes of the `xql:result` element from the result document and stores them in the variable `nodeList`. The variable `nodeList` now contains a set of documents that matched the query. The `childNodes()` DOM method returns a DOM `nodeList`, so DOM methods can be applied to `nodeList`.

Line 29

opens a session in Tamino. The session key and session ID that are returned by Tamino are handled transparently by the API. Using this method enables Tamino to keep the user context and make transaction control (methods `commit` and `rollback`) possible. In line 60 the updates are committed by the `commit` method of the `QueryObj` object, and in line 61 the session is closed.

Line 30

starts a processing loop which continues as long as there are elements to be processed. During each pass of the loop, the number of documents specified by the variable pageSize is processed.

Line 32

starts the processing loop, in which each document in turn is retrieved, modified and written back to the database.

Line 34

uses the DOM `item()` method to retrieve the current document from the document set stored in `nodeList`.

Line 35

uses the DOM `getElementsByTagName()` method to retrieve the `Country` element of the current document.

Lines 36-45
change "Germany" to "Deutschland" or vice versa.

Line 46
sends the modified document back to Tamino as an HTTP request that includes the data of the current document and specifies the X-Machine `_process` request. This tells Tamino to write the document to the database.

Line 49
retrieves the next set of documents that match the query by using the `getNext()` method. As before, the number of documents that are retrieved cannot exceed the value specified in the variable `pageSize`.

Line 50
tests whether any more documents were retrieved.

Line 57
sets `nodeList` to null if no more documents were returned. This subsequently terminates the loop processing that starts at line 30.

Line 67
sets a return value of 1 for the function `ChangeCountry`. This is not strictly required, but it would allow a JScript statement of the type `if (ChangeCountry) {...}` elsewhere in the HTML page to test whether `ChangeCountry` completed successfully and to take appropriate action.

Line 74
completes the function `ChangeCountry`.

Lines 77-83
cause a button with the text "Press here to start" to appear when the HTML page is displayed in a browser. The `onClick` attribute causes the function `ChangeCountry` to be started when the user chooses the button in the browser window.

II JScript API Reference

This chapter describes the JScript API.

Global Properties

You can set global properties either at runtime or by editing.

Property	Default	Description
`XMLHTTP`	MSXML2.XMLHTTP	Selects ActiveX implementation of IXMLHTTPRequest interface.
`inoDOMSupportsXPath`	true	Should be changed according to XMLDOM setting.
`inoJScriptDebug`	off	Enable or disable debugging.
`inoClientDefaultPageSize`	5	Maximum number of pages in the returned result set.
`inoRealCursoring`	true	Use true cursoring or not.
`inoCursorScrollableOn`	yes	Enable or disable backward cursor scrolling.

5 TaminoClient

This chapter describes the class `TaminoClient` of the JScript API.

Instantiating

This constructor has a variable number of parameters. If any parameter is omitted, the corresponding default value is used. The parameters are:

- Database URL
- Page size (default is 5)
- HTTP user name
- HTTP password

Example

```
var tam = new
TaminoClient(); var tam = new
TaminoClient(http://mypc.mycompany.com/tamino/xml/collection); var tam = new
TaminoClient(http://mypc.mycompany.com/tamino/xml/collection,6,"myname","mypassword");
```

Constants

Transactionality Constants

Name	Value	Usage
`XINOSESSIONID`	X-INO-Sessionid	SessionID returned by Tamino
`XINOSESSIONKEY`	X-INO-Sessionkey	SessionKey returned by Tamino
`YES`	yes	Specifying "LockWaitMode"
`NO`	no	Specifying "LockWaitMode"
`PROTECTED`	protected	Specifying isolation level
`UNPROTECTED`	unprotected	Specifying isolation level
`SHARED`	shared	Specifying isolation level

Microsoft XMLDOM Node Constants

Name	Value	Usage
NODE_ELEMENT	1	when using Microsoft DOM methods
NODE_ATTRIBUTE	2	
NODE_TEXT	3	
NODE_CDATA_SECTION	4	
NODE_ENTITY_REFERENCE	5	
NODE_ENTITY	6	
NODE_PROCESSING_INSTRUCTION	7	
NODE_COMMENT	8	
NODE_DOCUMENT	9	
NODE_DOCUMENT_TYPE	10	
NODE_DOCUMENT_FRAGMENT	11	
NODE_NOTATION	12	

Properties

XMLDB

Name	XMLDB
Example	`tam.XMLDB= "http://mypc.mycompany.com/tamino/xml/collection";`
Description	This is the HTTP address of the database that you want to access. The default is null. You must set up the database path this way if you did not set it up in the constructor. In a browser it may be restricted to the host where the web server resides. A collection name must be included.

user

Name	user
Example	`tam.user="myuserid";`
Description	This is the user ID that the web server recognizes.

password

Name	password
Example	`tam.password="mypassword";`
Description	This is the password that the web server recognizes.

pageSize

Name	pageSize
Example	`tam.pageSize=10;`
Description	This sets the page size for Tamino queries. The default is 5. A value of "0" means there is no limitation, all resulting documents are returned.

xmlHeader

Name	xmlHeader
Example	`tam.xmlHeader='<?xml version="1.0"?>';`
Description	This controls the header that is used when XML documents are sent to Tamino during the Update, Insert and Process methods The default is `'<?xml version="1.0"?>'`. You will probably not have to change this.

xmlbase

Name	xmlbase
Example	
Description	This sets an `xml:base` for query responses to override the base URL of `TaminoClient` or the document URI in case of `getDocument`. This is propagated to each `TaminoResult` object instantiated by the `TaminoClient`.

userAgent

Name	`userAgent`
Example	`tam.userAgent="WAP1.0"`
Description	This controls the value of the HTTP header User-Agent in all HTTP requests. The default value is something similar to HAS/JscriptTaminoAPILegacy_4_1_4_1/Browser IE6.

acceptLanguage

Name	`acceptLanguage`
Example	`tam.acceptLanguage="fr"`
Description	This controls the value of the HTTP header Accept-Language in all HTTP requests. The default is "en".

userDefined Headers

Name	`userDefinedHeaders`
Example	
Description	If set to an object, this instantiates the object's properties and values to additional HTTP headers.

realCursoring

Name	`realCursoring`
Example	`tam.realCursoring="off"`
Description	This controls whether cursoring is used or not. It can take either of two values: "on" (default) or "off". If set to "on", cursoring is selected within sessions.

scrollable

Name	`scrollable`
Example	`tam.scrollable="off"`
Description	This controls whether backward scrolling within a cursor is used or not. It can take either of two values: "on" (default) or "off".

Methods

Configuring the Tamino JScript Object

setUserPassword

Name	`setUserPassword`
Example	`tam.setUserPassword("myID","mypassword");`
Description	This sets the user ID and password to the parameter values specified. These are only required if the Tamino web server is using security. If you leave them blank and security is applied then IE5 will prompt you to supply values. This method is a convenience which allows you to set the user ID and the password in one call.

Query Operations

xquery

Name	`xquery`
Result Type	`TaminoResult`
Example	`var tamResult=tamClient.xquery("for $sq in input()/Square return $sq");`
Description	This causes an XQuery to be performed. Queries deliver Tamino Result instances. If the query succeeds with one or more nodes, then they are included in the Tamino Result object in a page. The maximum size of the page is determined by the client object pageSize attribute. Pages of pageSize will ONLY be returned if a transactional session is in progress. The next and previous pages can be read using methods exposed by the Tamino Result object. The setting of the offset (10 in the example) is optional. It determines the starting offset of the returned nodes.

query

Name	`query`
Result Type	`TaminoResult`
Example	`var tamResult=tam.query('Telephone[Lastname="Schmidt"]',10);`
Description	This causes a X-query to be performed. In addition to User-Agent and Accept-Language Headers, the Cache Control Header is set to "no-cache". Queries deliver Tamino Result instances. If the query succeeds with one or more nodes, then they are included in the Tamino Result object in a *page*. The maximum size of the page is determined by the client object pageSize attribute. The next, last, previous or first pages can be read using methods exposed by the Tamino Result object. The setting of the offset (10 in the example) is optional. It determines the starting number of the returned documents.

Update Operations

process

Name	`process`
Result Type	`TaminoResult`
Example	`tamResult=tam.process(myelementNode);`
Description	The Tamino object attempts to store the element using HTTP/1.1 POST and the _process verb. In addition to User-Agent and Accept-Language Headers, the Content-Type Header is set to "application/x-www-form-urlencoded; charset=utf-8". If the attribute ino:id is set to a numeric value in the domain of ino IDs, the element overwrites the element with this ID if it already exists, otherwise an error occurs. If no ID is present, a unique ID is assigned. A `TaminoResult` object is returned. The document type and the ino:id can be extracted from the Result object.

processNodeList

Name	`processNodeList`
Result Type	`TaminoResult`
Example	
Description	This processes all element nodes in a DOM nodelist.

processString

Name	processString
Result Type	TaminoResult
Example	
Description	This processes an XML document as a string. Tamino checks if it is well-formed.

insert

Name	insert
Result Type	TaminoResult
Example	tamResult=tam.insert(myelementNode);
Description	This is a variation of the process method to be used when the user wishes to ensure that a new instance will be created. Any ino:id attribute in the element is reset. As a consequence, a new ID is assigned by Tamino. A TaminoResult object is returned.

update

Name	update
Result Type	TaminoResult
Example	tamResult=tam.update(myelementNode,someINOId);
Description	This is a variation of the process method to be used when the user wishes to ensure that a specific instance will be overwritten. Any ino:id attribute in the node supplied as a parameter is ignored; the ID specified in the second parameter is used instead. A TaminoResult Object is returned. This method call can be used to replace a document with a known ID with another document. If the ino:id is not assigned, an error occurs.

xqueryUpdate

Name	xqueryUpdate
Result Type	TaminoResult
Example	var tamResult = tamClient.xqueryUpdate("update replace input()/TelephoneBook/entry/nr[.='12345'] with <nr>45678</nr>");
Description	This causes an XQuery update to be performed. Updates deliver Tamino Result instances. If the update succeeds with one or more nodes then they are included in the Tamino Result object.

inodelete

Name	`inodelete`
Result Type	`TaminoResult`
Example	`tamResult=inodelete(someNode);`
Description	The Tamino object attempts to delete the document specified by the node parameter, using HTTP/1.1 POST and the _delete verb. The node must correspond to a document type known to the database, and an ino:id attribute must be set to an ID in the database. This is the case if the node is returned by a query. In addition to User-Agent and Accept-Language Headers, the Content-Type Header is set to "application/x-www-form-urlencoded;charset=utf-8".

querydelete

Name	`querydelete`
Result Type	`TaminoResult`
Example	`tamResult=tam.querydelete[someQuery];`
Description	The Tamino object attempts to delete all documents that are described by the query. In addition to User-Agent and Accept-Language Headers, the Content-Type Header is set to "application/x-www-form-urlencoded; charset=utf-8".

Binary Operations

Note: An initial `putBinary` request should work without a problem. Subsequent `putBinary` requests that are attempting to update an existing document will cause the API to hang. This appears to be a problem with the XMLHTTP interface not handling an HTTP 204 response correctly. The suggested workaround is to delete the document before attempting the insert.

getBinary

Name	`getBinary`
Result Type	`TaminoResult`
Example	`var tamResult = tamClient.getBinary("BIN/BIN/Smiley"); var bytes = tamResult.BIN;`
Description	This causes a GET of the specific binary. The document to retrieve is specified by the first. The byte array of raw data can be retrieved by inspecting the BIN property of Tamino Result.

putBinary

Name	`putBinary`
Result Type	`TaminoResult`
Example	`var tamResult = tamClient.putBinary("BIN/BIN/Smiley", "image/gif", bytes);`
Description	This causes a binary PUT of the specific byte array. The document type (optional) and document name is given by the first parameter. The Content-Type is specified by the second parameter. The third parameter is the byte array of raw data.

Manipulating XML documents by URL

A URL may be relative to the URL value in the XMLDB property, that is to say it may specify the document type name and identifier only; or it may be absolute, specifying the whole URL.

The identifier is either the ino ID with a prefix of "@", or a document name if it is assigned.

Examples:

- dogs/@2536
- dogs/black/labrador.xml
- http://mypc.mycompany/tamino/xml/animals/dogs/@2536
- http://mypc.mycompany/tamino/xml/animals/dogs/labrador.xml

getDocument

Name	`getDocument`
Type	`TaminoResult`
Example	`tamResult=tam.getDocument("someDoctype/@9576");`
Description	The Tamino object attempts to fetch the document specified by the relative URL, specifying the document type and the ino:id. An absolute URL is also accepted, using HTTP/1.1 GET. In addition to User-Agent and Accept-Language Headers, the Cache-Control Header is set to "no-cache". A Tamino result is returned. The Document element is available in via the method getResult.

putDocument

Name	`putDocument`
Type	`TaminoResult`
Example	`tamResult=tam.putDocument("style/nice.xsl",documentNode);`
Description	The Tamino object attempts to store the document specified by the relative URL, using HTTP/1.1 PUT. The User-Agent and Accept-Language Headers are set. A `TaminoResult` Object is returned. GetResult yields null. The HTTP status code can be used to determine if a document is overwritten or not. 200 implies overwritten; otherwise a new document has been created.

head

Name	`head`
Type	`TaminoResult`
Example	`tamResult=tam.head("style/nice.xsl");`
Description	The Tamino Client tests the status of a document with the relative URL or absolute URL, using HTTP/1.1 HEAD. The User-Agent and Accept-Language Headers are set. A Tamino Result object is returned. GetResult yields null. The HTTP status code can be used to determine if a document exists or not. In addition, the HTTP headers yield the date last updated if it exists.

deleteDocument

Name	`deleteDocument`
Type	`TaminoResult`
Example	`tamResult=tam.deleteDocument("style/nice.xsl");`
Description	The Tamino object attempts to delete the document specified by the relative URL or absolute URLs, using HTTP/1.1 DELETE. The User-Agent and Accept-Language Headers are set. A `TaminoResult` object is returned. GetResult yields null. The HTTP status code can be used to determine if a document was deleted or not.

Transactionality Support

startSession

Name	`startSession`
Result Type	`TaminoResult`

Example	`tam=new TaminoClient();`
	`tam.XMLDB="http://mypc.mycompany.com/tamino/xml/collection";`
	`tamResult=tam.startSession(); OR`
	`tamResult=tam.startSession(PROTECTED,YES)`

Description	The Tamino object attempts to start a transaction session using HTTP/1.1 POST and the _connect verb. A `TaminoResult` object is returned. Parameters isolation and lockwait mode can be specified optionally. Possible value are: IsolationTypes: "protected","unprotected","shared". In the example constants are used for LockWaitTypes: "yes" or "no" (constants YES or NO can be used)

endSession

Name	`endSession`
Type	`TaminoResult`
Example	`tamResult=tam.endSession();`
Description	The Tamino object attempts to end a transaction session using HTTP/1.1 POST and the _disconnect verb. A `TaminoResult` object is returned.

commit

Name	`commit`
Type	`TaminoResult`
Example	`tamResult=tam.commit();`
Description	The Tamino object attempts to commit a transaction using HTTP/1.1 POST and the _commit verb. A `TaminoResult` object is returned.

rollback

Name	`rollback`
Type	`TaminoResult`
Example	`tamResult=tam.rollback();`
Description	The Tamino object attempts to roll back a transaction using HTTP/1.1 POST and the _commit verb. A `TaminoResult` object is returned.

inSession

Name	inSession
Result Type	
Example	`if (tam.inSession()) { ... } else { ... }`
Description	Returns null if no transaction is in progress, or a non-null value if a transaction is in progress.

Metadata Support

define

Name	define
Result Type	TaminoResult
Example	`tam=new TaminoClient("http://mypc.mycompany.com/tamino/xml"); tamResult=tam.define(mySchemaDocumentNode);`
Description	Implements the Tamino _define operation. The Tamino URL need only point to a database. An attempt is made to define a Tamino TSD3 schema corresponding to the node instance.

undefine

Name	undefine
Result Type	TaminoResult
Example	`tam=new TaminoClient("http://mypc.mycompany.com/tamino/xml"); // undefine the collection and all its doctypes tamResult=tam.undefine("myCollectionName"); or // undefine the doctype within the collection tamResult=tam.undefine("myCollectionName/myDoctype");`
Description	Implements the Tamino _undefine operation. The Tamino URL need only point to a database. An attempt is made to undefine a collection or doctype within the specified database. All instances within the doctype(s) described are removed. This works for collections and doctypes defined in TSD3 representation.

Diagnosis

diagnose

Name	`diagnose`
Result Type	`TaminoResult`
Example	`tamResult=tam.diagnose("ping"); if (tam.Result.errorNo) ... // error detected`
Description	The `TaminoClient` object sends a diagnose command to Tamino. The various commands are documented elsewhere.

explainQuery

Name	`explainQuery`
Result Type	`TaminoResult`
Example	`tamResult=tam.explainQuery(someQuery); if (tam.Result.errorNo) ... //error detected else { //show explanation panel.innerHTML= tamResult.DOM.documentElement.transformNode(someStylesheet);`
Description	The `TaminoClient` object sends an ino:explain request command to Tamino to retrieve the execution plan of the given query. The parameters are the query expression to be explained and the level of explanation. Please see section *ino:explain* (Tamino XML Server > X-Query Language Reference > Functions > ino:explain) for a detailed description of ino:explain.

Special Methods requiring filter NodeLevelUpdate.dll

For Microsoft's Internet Information Server (IIS) there is a special filter available, which is a pre-requisite for using the following methods of the Tamino API.

Note: The use of the NodeLevelUpdate ISAPI has been set to deprecated.

insertBefore

Name	`insertBefore`
Type	`TaminoResult`
Example	`tamResult=tam.insertBefore("/dogs/@2536","dates",DOMnode);`
Description	Inserts a child node (third parameter) to the left of the document specified in the relative URL (first parameter) and the query path (second parameter).

appendChild

Name	`appendChild`
Type	`TaminoResult`
Example	`tamResult=tam.appendChild("dogs/@2536","dates",DOMnode);`
Description	Appends a child node (third parameter) as the last child of the document specified in the relative URL (first parameter) and the query path (second parameter).

replaceChild

Name	`replaceChild`
Type	`TaminoResult`
Example	`tamResult=tam.replaceChild("dogs/@2536","dates/date_of_birth",DOMnode);`
Description	Replaces a child node specified by the relative URL (first parameter) and the query path (second parameter) by a new one (third parameter).

removeChild

Name	`removeChild`
Type	`TaminoResult`
Example	`tamResult=tam.removeChild("dogs/@2536","dates/date_of_birth");`
Description	Removes a child node specified by the relative URL (first parameter) and the path (second parameter) .

6 TaminoResult

This chapter describes the class `TaminoResult` of the JScript API.

Properties

errorNo

Name	`errorNo`
Example	`var tamResult = tamClient.query("Telephone[Firstname='Bill']"); if (!tamResult.errorNo) { // if no error ...}`
Description	This delivers the error number corresponding to the action that created the `TaminoResult` object. A zero error value implies that there was no error.

errorText

Name	`errorText`
Example	`var tamResult = tamClient.query("Telephone[Firstname='Bill']"); if (tamResult.errorNo) { alert(tamResult.errorText); return(tamResult.errorNO) } // if no error { ...}`
Description	This delivers the error text corresponding to the action that created the `TaminoResult` object. This only can be interpreted if the `errorNo` property is non-zero.

DOM

Name	DOM
Example	`var dom = tamResult.DOM; var newPar=dom.createElement("P"); newPar.appendChild( dom.createTextNode("Mary had a little Lamb"));`
Description	This delivers the complete Tamino response as a DOM object (provided it can successfully be parsed). This has two uses: ■ To provide a DOM object to facilitate method calls like `createElement`, which are only implemented by the DOM Object. ■ For diagnostic purposes. In particular, this object has a property `parseError` which is non-null when a parse error has occurred. This object has in turn properties, two of which are `errorCode` and `reason` which explain the parsing error. This is described in the Microsoft documentation.

REQ

Name	REQ
Example	`var req = tamResult.REQ; var status=req.status;`
Description	This delivers the XML HTTP request object used to generate the `TaminoResult`. The request object is described in the Microsoft Documentation and can be used to obtain further information about the HTTP request. The following properties are particularly useful: 1. `status` The numerical HTTP error status if any. This would be interrogated if an error 8400 was reported by the Tamino Result. The status value is present in the error message, but it might be more convenient to obtain it this way. Alternatively, it can be used to determine if a PUT caused a document to be overwritten or not. 2. `statusText` The text corresponding to the numerical status. The object has the method `getResponseHeader`, which can be used to pick out specific HTTP headers in the Tamino response. The most useful is the header which describes when the document read was last modified. This is returned when `getDocument` or `head` methods are called. It is invoked as follows: `var dateText=testResult.REQ.getResponseHeader("Last-Modified");`

lastQuery

Name	`lastQuery`
Example	`var tamResult = tam.getDocument(somerelativeurl); if (tamResult.errorNo) alert(tamResult.lastQuery + " gave " + tamResult.errorText);`
Description	This property records the URL used to retrieve the `TaminoResult`.

xmlbase

Name	`xmlbase`
Example	
Description	If not null, it overrides `lastQuery` as document base.

BIN

Name	`BIN`
Example	`var tamResult = tamClient.getBinary("Smiley");` `var bytes = tamResult.BIN;`
Description	Delivers the Tamino response as an array of bytes. Only set for a `getBinary` method call.

Methods

- extractURI

closeCursor

Name	`closeCursor`
Result Type	`void`
Example	`res.closeCursor()`
Description	If a cursor is open, it will be closed. No error is returned if a cursor is not open or Tamino returns an error.

nodes

Name	`nodes`
Result Type	DOM `NodeList`
Example	`var nodeList = tam.nodes(..); if (nodeList) then .. else ..`
Description	If the query delivers at least one result then the result is a node list of elements, otherwise the result is null. It is an abbreviation for `getResult().childNodes`.

getResult

Name	`getResult`
Result Type	DOM `Node`
Example	`var XQLResult = tamResult.getResult(); if (XQLResult) then { var nodeList = XQLResult.childNodes; var nodeCount = nodeList.length; ... } else ...`
Description	If the operation that created a Tamino result object delivers a query result, then this method delivers the `result wrapper` element. Otherwise the method delivers null.

getBooleanResult

Name	`getBooleanResult`
Result Type	Boolean
Example	
Description	Decodes `getResult()` as a boolean.

getNumericResult

Name	`getNumericResult`
Result Type	Number
Example	
Description	Decodes `getResult()` as a number.

getInoId

Name	`getInoId`
Example	`var inoId = tamResult.getInoId();`
Description	This delivers the ino:id from the first ino:object element in the Tamino response document. It is used to obtain the ino:id of the document when a successful store, insert or update is performed. In the case of other operations the value returned is null.

getDocType

Name	`getDocType`
Example	`var docType = tamResult.getDocType();`
Description	This delivers the document type from the first ino:object element in the Tamino response document. It is used to obtain the document type of the document when a successful store, insert or update is performed. In the case of other operations the value returned is null.

getCollection

Name	`getCollection`
Example	`var col = tamResult.getCollection();`
Description	This delivers the collection name from the first ino:object element in the Tamino response document. It is used to obtain the collection name of the document when a successful store, insert or update is performed. In the case of other operations the value returned is null.

getXMLBase

Name	`getXMLBase(elementNode)`
Result Type	`String`
Example	
Description	Calculates the `xml:base` value for this node in the result document.

extractURI

Name	`extractURI(elementNode, attribute)`
Result Type	`String`
Example	
Description	Extracts an absolute URI from the attribute within the element node taking the `xml:base` information into account. The document base is the value of the `lastQuery` property, which may be overriden by the property `xmlbase`. This is useful in the XLink context.

Paging Backward and Forward

- getNext
- getPrev
- getFirst
- getLast
- refresh

getNext

Name	`getNext`
Result Type	`TaminoResult`
Example	`tamResult=tamResult.getNext();`
Description	This causes a query for the next page of results to be performed. The transaction context is that of the original object where the query was performed. The result is a `TaminoResult` object

getPrev

Name	`getPrev`
Result Type	`TaminoResult`
Example	`tamResult=tamResult.getPrev();`
Description	This causes a query for the previous page of results to be performed. The transaction context is that of the original object where the query was performed. The result is a `TaminoResult` object.

getFirst

Name	`getFirst`
Result Type	`TaminoResult`
Example	`tamResult=tamResult.getFirst();`
Description	This causes a query for the first page of results to be performed. The transaction context is that of the original object where the query was performed. The result is a `TaminoResult` object.

getLast

Name	`getLast`
Type	`TaminoResult`
Example	`tamResult=tamResult.getLast();`
Description	This causes a query for the last page of results to be performed. The transaction context is that of the original object where the query was performed. The result is a Tamino result object. This feature delivers null in Tamino versions 2.x.

refresh

Name	`refresh`
Type	`TaminoResult`
Example	`tamResult=tamResult.refresh();`
Description	This causes a page of query results to be regenerated. The transaction context is that of the original object where the query was performed. The result is a `TaminoResult` object. The user might call this after invoking `inodelete` to remove an item from a page, to show the page again.

Getting Server Information

- getTaminoVersion

- getServer

getTaminoVersion

Name	`getTaminoVersion`
Type	`String`
Example	`tamResult=tam.diagnose("ping"); if (tamResult.errorNo) { ... // error detected } else { var taminoVersion=tamResult.getTaminoVersion(); }`
Description	The `TaminoResult` object extracts the Tamino version from the Tamino response. This value is available from every Tamino response.

getServer

Name	`getServer`
Type	`Date`
Example	`tamResult=tam.diagnose("ping"); if (tamResult.errorNo) { ... // error detected } else { var taminoServer=tamResult.getServer(); }`
Description	The `TaminoResult` object extracts the server name from the Tamino response. This value is available from every Tamino response.

Getting Document-Related Information

- getLastModified
- getContentType

getLastModified

Name	`getLastModified`
Type	`Date`
Example	`tamResult=tam.getDocument(someURL); if (tamResult.errorNo) { ... // error detected } else { var docDateTime = new Date(); docDateTime=tamResult.getLastModified(); }`
Description	The `TaminoResult` object extracts the date and time of the last modification from the Tamino response. For `getDocument()` and `head()` it corresponds to the date-time when the specified document was most recently modified.

getContentType

Name	`getContentType`
Type	`String`
Example	`tamResult=tam.head(someURL); if (tamResult.errorNo) { ... // error detected } else { var contentType=tamResult.getContentType(); }`
Description	The `TaminoResult` object extracts the content type from the Tamino response. This value is available from every Tamino response. Normally this is "text/xml" for XML documents. However, for non-XML documents it is the document's respective content type (for instance, "image/gif").

7 URI

This chapter describes the class `URI` of the JScript API. It implements some auxiliary functions that are useful for processing URIs.

Instantiating

This constructor has one parameter: the URI as a string.

Example

```
var myURI = new
URI("http://mypc.mycompany.com/tamino/myDB"); var myURI = new
URI("mydoctype/head.xml");
```

Methods

toString

Name	`toString()`
Result Type	`String`
Example	
Description	Returns the URI argument as a string.

isAbsolute

Name	`isAbsolute()`
Result Type	`String`
Example	
Description	Returns true if the URI argument is an absolute URI, i.e., it contains the name of a protocol, otherwise false.

rebase

Name	`rebase(URI)`
Result Type	`String`
Example	
Description	Takes the URI and a base parameter URI and returns a rebased URI. This method conforms to the URI rebasing algorithm documented in IETF RFC 2396.

extractURI

Name	`extractURI(elementNode, attribute)`
Result Type	`String`
Example	
Description	Extracts an absolute URI from the attribute within the element node, taking the `xml:base` information into account. The document `xml:base` is the URI.

getXMLBase

Name	`getXMLBase(elementNode)`
Result Type	`String`
Example	
Description	Calculates the `xml:base` value for this node in its document context. The document `xml:base` is the URI.

truncateAtHash

Name	`truncateAtHash()`
Result Type	`String`
Example	
Description	Returns the URI argument truncated before the first occurrence of "#", if any.

Index