Tamino

X-Machine Programming

Version 10.1

April 2018

WEBMETHODS

This document applies to Tamino Version 10.1 and all subsequent releases.

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

Document ID: INS-XPROG-101-20180413

Table of Contents

X-Machine Programming

This document gives a summary of the mechanisms available for performing low-level, HTTP-based client communication with the X-Machine. The X-Machine is a central component of the Tamino XML Server architecture. It provides many Tamino core services such as highly efficient storage and retrieval of XML and non-XML documents as well as standard query language support.

You should be familiar with the concepts of Tamino collections and schemas. For information on these subjects, refer to the document Tamino XML Schema User Guide.

This document is intended for use by application programmers who wish to develop client applications that communicate with the X-Machine using HTTP requests. Note that client applications can also use the Tamino APIs (see the appropriate product documentation) instead of HTTP requests for accessing Tamino via application programs.

This information is structured into the following sections:

1 Introduction

This introduction summarizes the methods available for programming against the X-Machine using HTTP requests.

The information in this introduction is organized under the following headings:

Accessing Documents in the X-Machine

Tamino's X-Machine can store and retrieve XML documents as well as non-XML documents. In X-Machine terms, XML documents are well-formed XML documents.

X-Machine provides two methods of accessing documents:

- Using plain URL addressing: Individual documents can be addressed using a URL in an HTTP GET, PUT, DELETE or HEAD request. The URL reflects a certain *directory path* structure that addresses the document uniquely. If you want to access a non-XML document in Tamino, you must use this method. With plain URL addressing, X-Machine behaves like an HTTP server, meaning that the response is returned only via HTTP mechanisms and HTTP-related return codes. If a document is requested using the GET method, it is returned in the HTTP response body.

  This access method is described in the section ***Requests using Plain URL Addressing***.

- Using X-Machine commands: This access method allows special commands to be sent to the X-Machine either in URLs that are sent using the HTTP GET method or as HTML form data that is sent using the HTTP POST method.

  If the GET method is used, the commands are provided in the URL as keyword/value pairs in the `search part` of the URL (as defined in section *3.3, HTTP* of the IETF document *RFC1738*, **http://www.ietf.org/rfc/rfc1738.txt**). The URL path is separated from the commands by a question mark ("?"). If there are two or more commands in a single URL, they are separated by an ampersand character ("&").

  This access method is described in the section ***Requests using X-Machine Commands***.

  X-Machine commands support all of the X-Machine functionality provided by plain URL addressing and support a wide range of additional operations. The X-Machine responds to each request with an appropriate X-Machine response document inside the HTTP response body rather than returning a plain HTTP response. If, for example, an application requests a set of documents to be returned from the X-Machine, the returned documents are enclosed by default in the HTTP response body inside an XML wrapper. This wrapper consists of elements and attributes defined in the Tamino namespace *http://namespaces.softwareag.com/tamino/response2* using the prefix "ino", such as Tamino return codes and cursor information. The wrapper can be suppressed; see the section ***Suppressing the Tamino Response Wrapper*** for further information.

Storage and Retrieval Format of XML and non-XML Documents

When storing and retrieving documents, the following rules apply:

- The X-Machine converts XML documents to Unicode before storing them in Tamino.
- By default, the X-Machine converts non-XML text documents to Unicode before storing them in Tamino. However, there is an option to suppress this conversion, i.e. to store and retrieve the documents unconverted. This option is activated by specifying the element `tsd:noConversion` as a child of the element `tsd:nonXML` in the schema for the non-XML doctype. See the description of the element `tsd:nonXML` in the Schema Reference Guide for more information.

  When `tsd:noConversion` is specified, documents are processed as follows:

  - When a document is stored the character encoding which was given in the HTTP request in the `Content-Type` field is stored.
  - Indexing can be performed on a non-XML document only if the character coding is given in the HTTP header.
  - When a document is retrieved, the data is returned without any conversion being performed. This applies even if the `Accept-Charset` HTTP header field is set. If a character encoding was specified in the HTTP header when the data was stored, this character encoding is returned in the header of the HTTP response.
- Binary documents are stored and retrieved unchanged byte for byte in Tamino.

See the section ***Media Type Requirements*** for a description of how Tamino distinguishes between XML and non-XML documents.

Security

Security for Tamino databases can be implemented using Tamino's built-in security mechanism that is based on entries in a special collection called *ino:security*. This is described in detail in the Security section of the documentation for Tamino Manager.

In addition, some limited security mechanisms are offered by the web server if a web server is used as the interface to Tamino. By configuring the web server, protection is possible at the database level. This controls access to databases, using the same mechanism as for ordinary directory structures under the control of the web server.

Note however that security mechanisms based on the web server are limited in use. It is for example not possible to set security at the collection level because XQuery can be used to access data across collection boundaries.

The Special Collection ino:etc

If a request is made to store an XML or non-XML document, but the request contains no collection name, the document will be stored in the collection *ino:etc*. In such cases, XML documents and non-XML documents consisting of text are stored with text retrieval indexing applied to the whole document, and non-XML documents (regardless of whether they consist of text or not) are stored in the special doctype *ino:nonXML* within *ino:etc*.

The collection ino:etc is always present in every defined database.

2 Session Handling

This section introduces the following aspects of Tamino's session handling mechanism.

Session Context

Requests sent to Tamino can be executed either inside or outside a session context. A session context is established via the `_connect` command and the usage of the session is restricted to the user who established it. A session allows you to group several commands issued in multiple requests as a transaction which is then either committed or rolled back using the command `_commit` or `_rollback`. A session can contain multiple transactions. A session context is also required if you use a query cursor that spans several requests.

If a request is executed outside a session context it is executed in a separate transaction. If execution ends successfully, the transaction is committed. Otherwise, the transaction is rolled back.

Session ID and Session Key

The way in which requests are associated with sessions depends on the request. Tamino supports the following types of request (these are described in more detail in subsequent sections):

- ***Requests using Plain URL Addressing***
- ***Requests using X-Machine Commands***

Each request belonging to a session causes Tamino to return two values, namely the session ID and the session key. A unique session ID is returned in the response to the `_connect` command and remains unchanged for all requests belonging to that session. Tamino generates a new session key in response to each request sent to Tamino.

There are two ways in which these values are returned to the client:

- Using the HTTP extension headers `X-INO-Sessionid` and `X-INO-Sessionkey`, which are always returned.
- Using the attributes `ino:sessionid` and `ino:sessionkey`. These attributes are returned only for requests that use X-Machine commands for which the response is embedded in an `<ino:response>` wrapper element. See also the section *Example of a response to the _connect command* later in this document.

The values for the session ID and session key attributes returned in a request's response must be sent to Tamino in the follow-up request for the same session. Again, there are two ways of passing the session ID and session key to Tamino:

- Using the HTTP extension headers `X-INO-Sessionid` and `X-INO-Sessionkey` in requests that use either plain URL addressing or X-Machine commands.

- Using the `_sessionid` and `_sessionkey` parameters in requests that use X-Machine commands.

For more information, see the section ***The HTTP header fields X-INO-Sessionid and X-INO-Sessionkey*** later in this document.

Queueing a Follow-Up Request

When using a multi-threaded streaming application, it is possible to issue a follow-up request before the response from the previous request has been fully received (once the new session key is known). The follow-up request should be issued in a separate thread so that the previous request's reply can reach completion. Tamino will queue the follow-up request until the previous one is completed in the following cases:

- The server XML parameter `queue next request` is set to "yes" (the default value) and the session connection parameter does not override it.
- The parameter `_QueueNextRequest` on the session's `_connect` command is set to "yes".

In addition to a normal completion of a previous request, a previous request is considered complete when any of the following conditions is met:

- The stream is closed before reading all of the previous request's reply.
- The stream is read to completion (i.e. end of file is reached).
- A timeout for sending the complete reply occurs.

3 Requests using Plain URL Addressing

This section describes the format of the URL used for plain URL addressing, and the corresponding HTTP request and response structure.

Plain URL addressing access is done using the HTTP methods PUT, GET, DELETE, and HEAD.

The methods PUT, DELETE and HEAD do not return an HTTP body. For GET, the HTTP body contains only the required document, i.e. there is no additional XML wrapping.

Plain URL addressing is described further under the following headings:

URL format for Plain URL addressing

An X-Machine document can be addressed directly via HTTP using the following URL structure:

`http://HostName:PortNumber/tamino/DatabaseName/CollectionName/DoctypeName/DocumentName`

with the following meaning:

HostName	Host address for the web werver.
PortNumber	Port number for the web server.
DatabaseName	Name of the Tamino database.
CollectionName	The name of the X-Machine collection that contains the existing document or will contain the new document.
DoctypeName	The name of the X-Machine doctype that contains the existing document or will contain the new document.
DocumentName	When storing a document, Tamino assigns this name to the document. The name could be a string such as a file name, e.g. *mypicture01.jpg*. Use the same name to retrieve the document in subsequent operations. The document name is returned in the pseudo-attribute `ino:docname` when the document is retrieved via X-Query. **Note:** The document name is unique within a given doctype. When retrieving a document, it is possible to specify either the name of the required document or the document's ID. The document ID is assigned automatically by Tamino when the document is originally stored, and the ID value is returned in the pseudo-attribute `ino:id` when the document is retrieved via X-Query. If the document ID is used instead of the document name, it must be prefixed by "@".

Since the names *HostName*, *PortNumber*, *DatabaseName*, *CollectionName*, *DoctypeName* and *DocumentName* are used in a URL as shown above, any characters in their names that are not allowed in a URL must be URL-encoded. For example, the characters “#”, “?”, “&” and “=” have a special meaning in URLs, so if a document name contains any of these characters (specified by the `ino:docname` attribute), it is necessary to use the escaped versions of these characters when addressing the document in a URL (namely “%23” for “#”, “%3F” for “?”, “%26” for “&” and “%3D” for “=”).

This form of addressing may be used for XML and non-XML documents.

Even though individual documents are accessible via plain URL addressing, the collections and doctypes used to store the documents must be created not by plain HTTP requests but by the X-Machine command `_define`.

The explicit creation of a doctype by passing a schema document in a `_define` command is not required (or in some cases prohibited) if the corresponding collection has been configured for schemaless storage (for example, the collection *ino:etc*).

Addressing Existing Documents via Document ID

When addressing an existing X-Machine document, the related URL may contain the document's ID instead of the document's name (as specified by `ino:docname`). This reference is indicated using an "at" ("@") character.

For example, the following URL specifies the Tamino document with document ID 4711635 in the collection "FDSdemo_e" and doctype "Construction", in the database "MyTestDb01", where the web server is at port "80" of the machine "myhost":

```
http://myhost:80/tamino/MyTestDb01/FDSdemo_e/Construction/@4711635
```

When a new document is stored, the X-Machine assigns an document ID automatically to the document. The document ID is returned as the value of the `ino:id` attribute in the `ino:object` element in the Tamino response document.

It is not possible for a client to assign an explicit ID when a new document is being stored. A 31-bit integer is used to represent the document ID, so the total number of document IDs per doctype is approximately 2,000,000,000. A document ID is unique within a given doctype, but does not need to be unique across doctypes, i.e. a document ID in one doctype can be the same as a document ID in another doctype. In the Tamino namespace, the name `ino:id` is reserved for the document ID.

Note: When a document is deleted, the document ID can be reused when a new document is created. This behaviour is controlled by the schema element `tsd:systemGeneratedIdentity`.

Criteria for Inserting or Replacing a Document

When a document is sent to Tamino, Tamino uses the values of the document ID and/or document name that can optionally be supplied in the input request to decide whether to insert a new document or replace an existing document.

The document ID of an existing document can either be supplied in the URL (as described above) or as the value of the `ino:id` attribute of the document's root element (if the document is an XML document), or both. If both are supplied, the values must be identical, otherwise an error will be returned.

The document name of an existing document can only be supplied in the URL.

If a document name and a document ID are both supplied, a document with this name and ID must already exist, otherwise an error will be returned.

The rules determining whether a new document will be inserted or an existing document will be replaced are the same as for the X-Machine `_process` command. See the section ***Criteria for inserting or replacing a document*** within the description of the `_process` command for details.

If you supply neither a document name nor a document ID when using plain URL addressing as described here, an error will be returned. Note however that the `_process` command allows you to omit both the document name and the document ID when inserting a new document. See the `_process` command for details.

HTTP Header and Body Content

When issuing HTTP GET, DELETE and HEAD requests to Tamino, no HTTP body is supplied. For an HTTP PUT request, the body must contain the document (XML or non-XML). For XML documents, wrapping is neither required nor allowed.

The URL describes the location for the document inside Tamino.

After a successful response from Tamino, the HTTP body will contain data if a GET was issued. This data is the requested document without any XML wrapping. The HTTP header will contain standard HTTP status codes.

If a PUT was issued, the HTTP response header field `X-INO-id` will contain the document ID of the document that was processed.

If a PUT, GET or DELETE was issued, the HTTP response header field `X-INO-Docname` will contain the name of the document that was processed.

The version number of the Tamino server being used is returned in the HTTP response header for every HTTP request. The value is returned in the field `X-INO-Version`.

HTTP Status Codes

The following table lists a selection of the standard HTTP status codes that can result from a plain URL addressing request to the X-Machine. For the full list of HTTP status codes, see the HTTP specification at **http://www.ietf.org/rfc/rfc2616.txt**.

HTTP status code	HTTP Method	Meaning
200	GET	OK, document retrieved
200	HEAD	OK, document found
201	PUT	OK, document created (did not already exist)
204	DELETE	OK, document deleted
204	PUT	OK, existing document replaced
400	GET, DELETE, HEAD, PUT	Request cannot be processed (e.g. unknown `ino:id` value or document not well formed)
401	GET, DELETE, HEAD, PUT	Access denied (the specified user ID and/or password are not valid)
404	GET, DELETE, HEAD	Document not found
500	GET, DELETE, HEAD, PUT	Internal error in the Tamino server, document not processed (i.e. not retrieved, not deleted, not stored)
502	GET, DELETE, HEAD, PUT	Error in a communications component such as a web server interface
503	GET, DELETE, HEAD, PUT	Service unavailable temporarily (e.g. due to a locking conflict)

When accessing WebDAV resources, various other response codes are possible. Please refer to **RFC2518**, **RFC3253**, **RFC3744** and the WebDAV SEARCH specification at **http://www.ietf.org/** for a complete list.

Authentication Aspects

This section covers the following topics:

- Passing a user ID and password to Tamino

- Authentication for client requests

Passing a user ID and password to Tamino

In order to pass a user ID and password to Tamino, the following methods are available:

- Basic authentication field in HTTP header
- Special Tamino authentication field in HTTP header

Basic authentication field in HTTP header

Using the standard mechanism for the basic authentication scheme of the HTTP protocol, a user ID and password can be passed to Tamino. The field `Authorization` of the HTTP header can be used for this purpose.

Note: The field `X-INO-Authorization`, if present in the HTTP header, takes precedence over the field `Authorization`. See the section ***Special Tamino authentication field in HTTP header*** for details.

The format of the field `Authorization` is as follows:

```
Authorization: Basic ID:Password
```

where *`ID:Password`* is the user ID and password in UTF-8 encoding separated by a colon, then converted to base64 representation. For a description of base64 encoding, which defines a mapping of any binary data to printable characters in 7-Bit US-ASCII, see **http://www.ietf.org/rfc/rfc2045.txt**.

When using a user domain, the user ID should be preceded by the domain name and a backslash, for example, "MyDomain\MyUserID". Note that Tamino domains and user IDs are case-sensitive, also when they are mapped to a Windows domain.

Special Tamino authentication field in HTTP header

The Tamino-specific field `X-INO-Authorization` can be used in the HTTP header to pass a user ID and password.

Note: When the fields `Authorization` and `X-INO-Authorization` are both present in the HTTP header, the field `X-INO-Authorization` takes precedence over the field `Authorization`.

The format of the field `X-INO-Authorization` is as follows:

```
X-INO-Authorization: Basic ID:Password
```

where *`ID:Password`* is the user ID and password in UTF-8 encoding separated by a colon, then converted to base64 representation. For a description of base64 encoding, which defines a mapping of any binary data to printable characters in 7-Bit US-ASCII, see **http://www.ietf.org/rfc/rfc2045.txt**.

When using a user domain, the user ID should be preceded by the domain name and a backslash, for example, "MyDomain\MyUserID". Note that Tamino domains and user IDs are case-sensitive, also when they are mapped to a Windows domain.

Authentication for client requests

Tamino supports the following types of authentication for client requests:

- Non-authenticated access
- Authentication using web server authentication
- Tamino authentication

Non-authenticated access

This method can be used when the Tamino XML property `Authentication` is set to "none". In this case, the user ID found in the header fields will be used and the password will be ignored. This is not recommended if access restrictions are defined using Tamino Security (see the section *Tamino Security* in the documentation of the Tamino Manager for information).

Authentication using web server authentication

This method can be used when the Tamino XML property `Authentication` is set to "web server". The user ID and password, which must be known to the authenticating web server, must be provided in the HTTP basic authentication header field.

If the property is set to "web server" and a non-authenticating web server is used, no user ID will be passed to Tamino, so all requests will be treated as if they originate from the default user group; here, only the basic authentication scheme is supported.

For this type of authentication to work, the web server must be configured to use basic authentication and must be able to authenticate the users who will communicate with Tamino using this method. Please refer to the documentation of your web server for information on how to do this configuration.

Tamino authentication

This method can be used when the Tamino XML property `Authentication` is set to "tamino". The user ID and password can be passed in either the HTTP basic authentication header field or the special Tamino authentication header field. Tamino authenticates the user against the users known to Tamino (users stored directly in Tamino or users known to Tamino via an LDAP server or the local operating system). If the authentication fails, an HTTP response 401 is returned.

Transaction Aspects

In the same way as for X-Machine commands, requests using plain URL addressing can be executed in the context of a transaction that is part of an X-Machine session established via the `_connect` command. As there is no way to pass request parameters using plain URL addressing, the values of various session parameters can be passed in HTTP request header fields.

Setting session ID and session key

The values of `_sessionid` and `_sessionkey` can be passed in the HTTP request header fields `X-INO-Sessionid` and `X-INO-Sessionkey`. For details of session ID and session key see the topic *Transaction-Related Commands*.

Overriding session parameters

It is possible to override certain session parameter defaults (typically set on the X-Machine `_connect` command) in an HTTP request by supplying values for the following HTTP request header fields:

HTTP request header field	Corresponding X-Machine parameter
X-INO-isolation	_ISOLATION
X-INO-isolationLevel	_ISOLATIONLEVEL
X-INO-lockMode	_LOCKMODE
X-INO-lockWait	_LOCKWAIT

When the request completes, the values of the session parameters revert to the default values.

For more information on these parameters, refer to the section *Session Parameters*.

4 Requests using X-Machine Commands

This section describes how to communicate with the X-Machine using the X-Machine's own processing commands. It also describes the corresponding HTTP request and response structure.

The X-Machine offers a set of commands for storing, retrieving and deleting X-Machine documents, creating or erasing collections or schemas, performing transaction processing and diagnostic testing. These commands are sent to the X-Machine either as parameters that are appended to a URL in an HTTP GET request (parameterized URL addressing), or as HTML form data in an HTTP POST request.

The commands are:

Command	Meaning
`_admin`	Perform an administration function
`_commit`	Commit a transaction
`_connect`	Start a database session
`_cursor`	Perform a cursor-related command
`_define`	Create a collection, schema or doctype; modify an existing schema or doctype
`_delete`	Delete one or more documents
`_destroy`	Remove a prepared query
`_diagnose`	Perform a diagnostic test
`_disconnect`	Terminate a database session
`_execute`	Execute a prepared query
`_htmlreq`	Create and query Tamino documents (used only in the context of HTML forms)
`_prepare`	Prepare (precompile) a query for later execution
`_process`	Store one or more documents into a collection; or modify an existing XML document
`_rollback`	Roll back a transaction
`_undefine`	Delete a collection, schema or doctype
`_xql`	Retrieve one or more documents using the Software AG's XPath-based X-Query query language
`_xquery`	Specify a query based on the W3C XQuery query language

X-Machine commands are described under the following headings:

X-Machine Command Format

Each X-Machine command described in this section can be sent either via an HTTP GET request or via a multipart form HTTP POST request.

X-Machine commands that are sent via HTTP GET requests are contained as keyword/value pairs in URLs according to the syntax given below (parameterized URL addressing). For some commands, additional information can be supplied in the form of keyword/value pairs.

For HTTP POST requests with HTML form data, only the URL prefix (the part specifying the path of the database to be accessed) is supplied as the HTTP address, and the keyword/value pairs are supplied in the body of the HTML form data.

In general, the examples in the following sections for the individual commands use parameterized URL addressing.

Parameterized URL addressing via HTTP GET

The syntax of parameterized URL addressing is as follows:

```
URLprefix/CollectionName{?Command[/Stylesheet]=Data[&Keyword=Value]}...
```

The syntax has the following meaning:

URLprefix	The URL of the database to be accessed, for example: *http://myhost:80/tamino/mydatabase* The host name and port number must point to the computer and port where your web server is running.
CollectionName	The name of the collection to be accessed. This is optional for some commands.
Command	The X-Machine command, which is one of the following verbs: `_define`, `_undefine`, `_process`, `_delete`, `_xql`, `_xquery`, `_connect`, `_commit`, `_rollback`, `_disconnect`, `_admin`, `_cursor`, `_htmlreq` or the special verb `_diagnose`, `_destroy`, `_prepare`, `_execute`. Command names are case-insensitive.
Stylesheet	A URL pointing to an XSL stylesheet. When this is specified, the response to the request will contain an XSL processing instruction of the form: `<?xsl:stylesheet href='Stylesheet'?>` Browsers capable of interpreting such processing instructions (such as Internet Explorer) will format the response document according to the formatting specified in the stylesheet. Refer also to the section ***Syntax of XML Responses*** for information about the content of the response document.

	The stylesheet may be specified either as an absolute URL starting with `http://` or as a relative URL which is evaluated by the client application (web browser) according to the standard rules. Example: If "_xql/http://aaa/bbb.xslt=patient" is specified, a processing instruction `<? xsl:stylesheet href='http://aaa/bbb.xslt' ?>` will be added to the response document.
Data	This specifies the data to be processed by the command. This can be XML documents or query expressions depending on the preceding verb, or parameters for commands such as `_diagnose` and `_admin`. The transaction and session-related command verbs `_connect`, `_commit`, `_rollback` and `_disconnect` require an asterisk ("*") here, for example `_commit=*`, and the remaining data must be supplied in the form of keyword/value pairs (see the description of *Keyword* and *Value* below).
Keyword	The name of a keyword that further qualifies the processing to be done by the command. There can be multiple keyword/value pairs. Each keyword/value pair must be preceded by an ampersand ("&"). Keywords are case-insensitive. Unknown keywords are ignored.
Value	Value of the keyword. See the description of the commands below for information on the values allowed.

If the web server-based security mechanism is implemented at your site, the collection must be entered to the left of the question mark.

Restrictions when using parameterized URL addressing

If you use parameterized URL addressing rather than HTML form data, please be aware of the following restrictions:

- Some web servers restrict the length of the URLs that they can process. In some cases, a URL that you want to send might be longer than the maximum length allowed by the web server on the client or server side, so it might be necessary to use HTTP POST with HTML form data instead.
- The X-Machine accepts IRIs (Internationalized Resource Identifiers). In an IRI, all Unicode characters can be used, provided that they are properly encoded. Each byte of the hexadecimal representation of the UTF-8 code point of such a character must be prefixed by "%".

  Example: the German character "ä" (the character "a" with an umlaut) has the Unicode code point U+E4. The equivalent UTF-8 code point represented as a hexadecimal value is "C3A4", so this must be represented as "%C3%A4" in the IRI.

HTML multipart form data via HTTP POST

For X-Machine commands sent in HTTP POST requests with HTML form data, the information sent is equivalent to that sent by parameterized URL addressing (see the section ***Parameterized URL addressing via HTTP GET*** above), but the keyword/value pairs are supplied in the body of the HTML form data.

Here is an example of the HTTP request body using the `_define` command to define a schema cluster consisting of 2 schemas *S1* and *S2*:

```
POST /tamino/myDB HTTP/1.1
User-Agent: ….
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 1250
Host: localhost:80
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

--xYzZY
Content-Disposition: form-data; name="_define"
Content-Type: text/plain
Content-Length: 7

$S1,$S2
--xYzZY
Content-Disposition: form-data; name="$S1"

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
   <xs:annotation>
      <xs:appinfo>
         <tsd:schemaInfo name="S1">
            <tsd:collection name="cluster"/>
            <tsd:doctype name="root"/>
         </tsd:schemaInfo>
      </xs:appinfo>
   </xs:annotation>

   <xs:element name="root">
     <xs:complexType>
       <xs:sequence>
         <xs:element ref="child"/>
       </xs:sequence>
     </xs:complexType>
   </xs:element>

   <xs:element name="child"/>

</xs:schema>
```

```
--xYzZY
Content-Disposition: form-data; name="$S2"

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
   <xs:annotation>
      <xs:appinfo>
         <tsd:schemaInfo name="S2">
            <tsd:collection name="cluster"/>
         </tsd:schemaInfo>
      </xs:appinfo>
   </xs:annotation>

</xs:schema>

--xYzZY--
```

Description of X-Machine Commands

This section describes the individual X-Machine commands. The list of available commands is as follows:

- The _admin command
- The _commit command
- The _connect command
- The _cursor command
- The _define command
- The _delete command
- The _destroy command
- The _diagnose command
- The _disconnect command
- The _execute command
- The _htmlreq command
- The _prepare command
- The _process command
- The _rollback command
- The _undefine command
- The _xql command
- The _xquery command

Caution: Web servers that log requests log the URL but not the body of the request. So if a user issues a request using a GET, the data can be obtained by reading the web server log, which could lead to potential security or privacy problems. If PUT or POST are used, the data cannot be seen in the web server log.

The _admin command

The `_admin` command offers several administration functions for Tamino. The format of this command is:

```
_admin=Function(Parameter, ...)
```

After execution of the `_admin` command, the response document indicates either a successful execution or an error. In the case of success, a response such as the following will be delivered:

```
<ino:message ino:returnvalue="0">
<ino:messageline>
  starting admin command AdminCommand
</ino:messageline>
</ino:message>
<ino:message ino:returnvalue="0">
<ino:messageline>
  admin command AdminCommand completed
</ino:messageline>
</ino:message>
```

Otherwise, in the case of an error, the response will contain entries like this:

```
<ino:message ino:returnvalue="Value">
<ino:messagetext ino:code="ErrorCode">
   MessageText
</ino:messagetext>
</ino:message>
```

The following administration functions are available.

- The ino:Accessor function
- The ino:CancelMassLoad function
- The ino:ChangeUserPassword function
- The ino:DisplayIndex function
- The ino:Index function
- The ino:RecreateIndex function
- The ino:RecreateTextIndex function
- The ino:RepairIndex function

- The ino:Request function

The ino:Accessor function

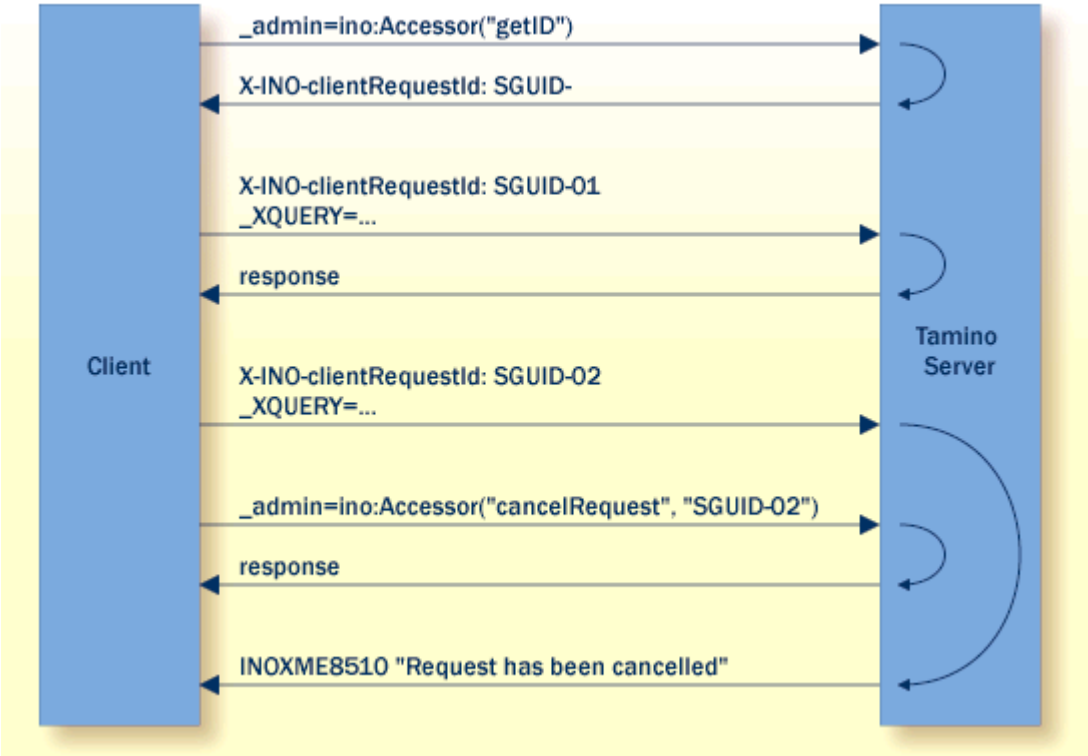
This administration function allows a client to cancel a request that it submitted previously. This could be used, for example, to cancel a request that is taking a long time to execute.

In order to label a request so that it can be uniquely identified for cancellation at a later stage, the client assigns a unique request ID to the request when the request is sent to the X-Machine. This ID can be used in a later call of `ino:Accessor("cancelRequest", "requestID")` to cancel the execution of the request. The request ID consists of a unique part generated by the X-Machine, obtained by a call of `ino:Accessor("getId")`, plus a client-generated part.

The function can be called with the following parameters:

`ino:Accessor("getId")`	When this command is sent to the X-Machine, the X-Machine returns a server-generated unique ID in the HTTP response header field `X-INO-clientRequestId`. In any subsequent request from the client to the X-Machine, the client can pass a request ID, constructed from the server-generated unique ID, in the HTTP request header field `X-INO-clientRequestId` to label the request uniquely, so that the request can be cancelled by a subsequent call of `ino:Accessor("cancelRequest", "requestID")`. The request ID is composed of the server-generated unique ID plus a client-generated part. The client generated part can be freely chosen, as long as the resulting request ID for a given request is unique. The request ID must be UTF-8 encoded.
`ino:Accessor("cancelRequest", "requestID")`	This command cancels the request identified by the *requestID*. The meaning of *requestID* is described above.

A typical flow of events is shown in the following diagram. In this example, the initial call of `_admin=ino:Accessor("getId")` returns the server-generated unique ID "SGUID" in the HTTP header field `X-INO-clientRequestId` in the response. The request ID generated by the client for each subsequent request has the format "SGUID-*nn*", where *nn* is a numeric counter starting from "01". The example shows two requests, with the request IDs "SGUID-01" and "SGUID-02", with each request sending an `_XQUERY` command. To cancel the request that has the request ID "SGUID-02", the client issues the command `ino:Accessor("cancelRequest", "SGUID-02")`.



Related information

The ino:Accessor function is used by a client to cancel a request that it previously submitted itself. A user with administration rights can also cancel any active Tamino request, regardless of which client originally submitted it, by using the `ino:Request` function of the `_admin` command.

The ino:CancelMassLoad function

This administration function allows you to cancel a halted session of the Tamino data loader utility on the Tamino server machine. This situation can arise if the client machine from which the mass load was started has become unavailable. The call has one parameter, namely a string specifying either (a) the desired doctype and the respective collection or (b) the session ID:

```
_admin=ino:CancelMassLoad("CollectionName/DoctypeName" | SessionID)
```

Note: This function can only cancel a session that is inactive, i.e., you cannot stop a running session.

If the function completes successfully, the reply of this function will contain entries like these:

```
<ino:message ino:returnvalue="0">
   <ino:messageline>session 12345 ended</ino:messageline>
</ino:message>
```

If no active session was found you will receive a result document containing entries like these:

```
<ino:message ino:returnvalue="8555">
   <ino:messageline>Invalid session ID</ino:messageline>
</ino:message>
```

In the case of an active session but unsuccessful execution, the result document will contain entries like these:

```
<ino:message ino:returnvalue="8285">
   <ino:messageline>Invalid session ID</ino:messageline>
</ino:message>
```

If the function is issued against a doctype for which no data loader session is halted, an error response 8310, "Invalid parameter detected" will be returned.

The ino:ChangeUserPassword function

This function allows a user to change his or her password.

The function has one parameter, which is the new password in plain text:

```
_admin=ino:ChangeUserPassword("NewPassword")
```

For further information about this command, refer to the section *Tamino Security* in the documentation of the Tamino Manager.

The ino:DisplayIndex function

The function `ino:DisplayIndex` displays the contents of standard indices or text indices. The description of this function consists of the following sections:

- Syntax
- Standard Index and Collation
- Standard Index With Truncated Values
- Index options
- Compound Index
- Unique Keys
- Multipath Index
- Computed Index

- Reference Index

Syntax

The function call has the following syntax:

```
_admin=ino:DisplayIndex("CollectionName", "ElementPath", "StartValue", "Size", ↩
"IndexType")
```

where *CollectionName* is the name of the collection, *ElementPath* is the absolute path of the indexed element, *StartValue* is the first index value that you want to display, *Size* is the number of values to display for the index (must be a positive integer), and *IndexType* specifies the index type, which can be one of "standard", "text", "multipath-standard", "multipath-text" or "computed-standard".

Example

```
_admin=ino:DisplayIndex("Customers","Customer/Name","B","10","standard")
```

This will display the first 10 values that exist for the standard-indexed element `Name` in the doctype *Customer* in the collection *Customers*. The start value "B" indicates that the first value returned for the index should be equal to or greater than "B". The result document could, for example, contain the following lines:

```
<ino:index ino:indexcoll="Customers" ino:indexpath="Customer/Name" ↩
ino:indextype="standard">
  <ino:indexvalue ino:indexcount="28">Baker</ino:indexvalue>
  <ino:indexvalue ino:indexcount="33">Barclay</ino:indexvalue>
  <ino:indexvalue ino:indexcount="14">Bayliss</ino:indexvalue>
  <ino:indexvalue ino:indexcount="1">Bean</ino:indexvalue>
  <ino:indexvalue ino:indexcount="28">Bedford</ino:indexvalue>
  <ino:indexvalue ino:indexcount="23">Bingham</ino:indexvalue>
  <ino:indexvalue ino:indexcount="676">Black</ino:indexvalue>
  <ino:indexvalue ino:indexcount="22">Bolton</ino:indexvalue>
  <ino:indexvalue ino:indexcount="563">Brown</ino:indexvalue>
  <ino:indexvalue ino:indexcount="47">Butler</ino:indexvalue>
</ino:index>
```

The collating sequence used for the `ino:DisplayIndex` function is the standard sequence of Unicode scalar values (“codepoints”).

If there are several indexes of the specified index type at the given path, then all indexes will be displayed, and each index will have its own `ino:index` element, as described below. If there are several indexes, the given size (number of values to display) will apply to each index. If there are several indexes, the given start value will apply to each index (see also below the special case for a compound index). The index type specifies whether standard or text indexes are to be displayed. There is no further possibility to select an index of a specific kind (e.g., display only multipath indexes), or to select a particular index (e.g., the third out of five compound indexes).

Indexes are not available while they are built/rebuilt during a `_define` command or a `ino:RecreateIndex`/`ino:RecreateTextIndex` administration command. In this case the corresponding `ino:index` element will have its attribute `ino:status` set to "not-available". As the index values are being built at that point in time, no `ino:indexvalue` elements will be displayed. For example:

```
<ino:index ino:indexcoll="myColl"
           ino:indexpath="myDoc/field"
           ino:indextype="standard"
           ino:status="not-available">
</ino:index>
```

Standard Index and Collation

If a non-composite standard index has a collation, then the index value will be displayed in hexadecimal format, as there is currently no possibility of converting it back to a readable value. The attribute `ino:value` of the `ino:indexvalue` elements will have the value "collation-encoded". For example:

```
<ino:index ino:indexcoll="myColl"
           ino:indexpath="myDoc/field"
           ino:indextype="standard">
  <ino:indexvalue ino:indexcount="1" ino:value="collation-encoded">value1
  </ino:indexvalue>
  <ino:indexvalue ino:indexcount="1" ino:value="collation-encoded">value2
  </ino:indexvalue>
</ino:index>
```

Standard Index With Truncated Values

Tamino sets an upper limit on the length of an index, and indexes that exceed this limit are truncated. Information about the size of this limit in the current Tamino release is provided in the section *Definition of Unique Keys* in the *Tamino XML Schema User Guide*.

If a standard index contains a truncated value, the corresponding `ino:index` element will have the attribute `ino:status` set to the value "has-truncated-values". The corresponding index values that are truncated will be marked similarly. For compound indexes this was already described above. For simple standard indexes the `ino:indexvalue` element will have the attribute `ino:value` set to "truncated". As far as possible the truncated value will appear as the contents of `ino:indexvalue`. For example:

```
<ino:index ino:indexcoll="myColl"
           ino:indexpath="myDoc/field"
           ino:indextype="standard"
           ino:status="has-truncated-values">
  <ino:indexvalue ino:indexcount="1" ino:value="truncated">value1
  </ino:indexvalue>
  <ino:indexvalue ino:indexcount="1">value2</ino:indexvalue>
</ino:index>
```

For more information on truncated values, refer to the section

Index options

Depending on the index type (standard or text), the options `compound` and `multipath` can be available. These options may be combined: a compound index (or more precisely, a standard index with the compound option) may be part of a multipath index. Moreover, several indexes of the same kind may occur at a particular path. For example, an element may have several compound indexes.

The compound and multipath options are represented as an attribute of the `ino:index` element. If an index has several options, then all the corresponding attributes will appear with the `ino:index` element.

Compound Index

A compound index is represented by an `ino:fields` attribute, the value of which is the concatenation of the index components as given in the schema (separated by blanks). The following example shows a schema with a compound index and the corresponding `ino:index` element in the index display:

```
<xs:element ...>
  ...
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:native>
        <tsd:index>
          <tsd:standard>
            <tsd:field xpath="C"/>
            <tsd:field xpath="B/@b" />
          </tsd:standard >
          ...
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
<xs:element>
```

Schema with a compound index

```
<ino:index ino:indexcoll="myCollection"
           ino:indexpath="myDocument/myElement"
           ino:indextype="standard"
           ino:fields="C B/@b">
```

ino:index element in the index display

The following general rules apply for the display of compound indexes:

- The start value specified in the `ino:DisplayIndex` call will apply to the first component of the compound index.
- If there are several compound indexes defined for the requested path then the respective first components may have different datatypes. Then `ino:DisplayIndex` will try to convert the start value to the required datatype. If that fails, the execution will abort with an error message.
- The index values of a compound index will be split into their component parts. Each part will be displayed in an own `ino:field` element within the `ino:indexvalue` element. For example, if there is a compound index with three components, the `ino:indexvalue` element will look like this:

```
<ino:indexvalue ino:indexcount="1">
  <ino:field>value of 1st component</ino:field>
  <ino:field>value of 2nd component</ino:field>
  <ino:field>value of 3rd component</ino:field>
</ino:indexvalue>
```

- If a component does not have a value (the content of the component does not exist), the contents of `ino:field` will be empty, for example:

```
<ino:indexvalue ino:indexcount="1">
  <ino:field>value of 1st component</ino:field>
  <ino:field/>                                  <!-- empty component value -->
  <ino:field>value of 3rd component</ino:field>
</ino:indexvalue>
```

- If a component does not exist, the contents of `ino:field` will be empty, and the attribute "ino:value="non-existing"" will indicate the missing component, for example:

```
<ino:indexvalue ino:indexcount="1">
  <ino:field>value of 1st component</ino:field>
  <ino:field ino:value="non-existing"/>          <!-- non-existing component -->
  <ino:field>value of 3rd component</ino:field>
</ino:indexvalue>
```

- If a component value was truncated (because the whole compound value was longer than 1004 bytes), the attribute `ino:value` will indicate this. If the component's datatype allows the truncated prefix to be converted to a meaningful value, this value will appear as contents of the `ino:field` element, otherwise (e.g. for float components), `ino:field` will be empty. For example:

```
<ino:indexvalue ino:indexcount="1">
  <ino:field>value of 1st component</ino:field>
  <ino:field ino:value="truncated">value of 2nd c</ino:field>
  <ino:field ino:value="truncated"/>
</ino:indexvalue>
```

- If a component has a collation, the attribute `ino:value` will have the value "collation-encoded". The value itself will be displayed in hexadecimal format, as there is currently no possibility of converting the index value back to a readable value. Note that a collation-encoded value may as well be truncated. For example:

```
<ino:indexvalue ino:indexcount="1">
  <ino:field ino:value="collation-encoded">0x112233aabb</ino:field>
  <ino:field ino:value="collation-encoded truncated">0x1122</ino:field>
  <ino:field ino:value="truncated"/>
</ino:indexvalue>
```

Unique Keys

There is no explicit way to display unique keys. As they are implemented as standard/compound indexes, they are displayed implicitly when ino:DisplayIndex() is called with a path at which such an index exists. Consider for example the following schema:

```
<tsd:doctype name="A">
  ...
  <tsd:unique name="CB-key">
    <tsd:field xpath="C"/>
    <tsd:field xpath="B/@b" />
    </tsd:unique>
  <tsd:unique name="D-key">
    <tsd:field xpath="D"/>
  </tsd:unique>
  ...
</tsd:doctype>
```

Schema with unique key

This example means that if ino:DisplayIndex() is called with "path="A/D"", then the unique key "D-key" will be displayed, and if ino:DisplayIndex() is called with "path="A"", then the unique key "CB-key" will be displayed. In both cases, the attribute `ino:unique="true"` in the `ino:index` element will identify the indexes as unique keys:

```
<ino:index ino:indexcoll="myCollection"
           ino:indexpath="A/D"
           ino:indextype="standard"
           ino:unique="true">

<ino:index ino:indexcoll="myCollection"
           ino:indexpath="A "
           ino:indextype="standard"
           ino:fields="C B/@b"
           ino:unique="true">
```

Multipath Index

For a multipath index, the label as given in `tsd:multiPath` will be displayed in the `ino:multiPath` attribute. Consider the following sample schema:

```
<xs:element name = "Title" type = "xs:string">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:physical>
          <tsd:native
            <tsd:index>
              <tsd:text>
                <tsd:multiPath>MultiPathIndex</tsd:multiPath>
              </tsd:text>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

The `ino:index` element shown for this schema is:

```
<ino:index ino:indexcoll="myCollection"
           ino:indexpath="myDocument/Title"
           ino:indextype="text"
           ino:multiPath="MultiPathIndex">
```

As a multipath index in general is not based on a specific path in the doctype, it is also possible to display the entries of a specific multipath index by using the following extended syntax:

```
_admin=ino:DisplayIndex("Collection", "Doctype/IndexName", "StartValue", "Size", ↩
"IndexType")
```

where *IndexName* is the name specified in the `tsd:multipath` element, and *IndexType* is either "multiPath-standard" or "multiPath-text".

Computed Index

For an example schema defining a computed index see e.g. Appendix 5: Example Schemas for Indexing in the *XML Schema User Guide*. It is possible to display the entries of a computed index by using the following extended syntax:

```
_admin=ino:DisplayIndex("Collection", "Doctype/IndexName", "StartValue", "Size", ↩
"IndexType")
```

where *IndexName* is the value of the `name` attribute of the `tsd:computed` element, and *IndexType* must be "computed-standard".

Reference Index

For a reference index, the label as given in `tsd:refers` will be displayed in the `ino:refers` attribute. Consider the following sample schema:

```
<xs:element name = "D" minOccurs = "0" type="xs:string">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:physical>
          <tsd:native>
            <tsd:index>
              <tsd:standard>
                <tsd:refers>/A/B</tsd:refers>
              </tsd:standard>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

The `ino:index` element shown for this schema is:

```
<ino:index ino:indexcoll="myCollection"
           ino:indexpath="A/B/D"
           ino:indextype="standard"
           ino:refer="/A/B">
```

The ino:Index function

This function is intended for performing maintenance tasks on indexes. The syntax is

```
_admin=ino:Index("Action"[","CollectionName"[","DoctypeName"]])
```

The parameter `Action` determines the action to be performed for the given doctype within the given collection. If the doctype name is omitted, all doctypes will be processed. If the collection name is omitted, all doctypes in all collections will be processed.

Currently, the only value supported for `Action` is "optimize". It causes the following steps to be executed for each doctype being processed:

1. Optimize the internal storage used for the maintenance of long index values. This refers to index entries whose length exceeds 1000 bytes. Note that the length of an index entry depends strongly on the underlying datatype. For strings, the length of the UTF-8 representation is relevant.

   The `ino:DisplayIndex` function can be used to check whether such long index values have occurred. The result may look as follows:

   ```
   <ino:index ino:indexcoll="myCollection"
              ino:indexpath="myDocument/myElement"
              ino:indextype="standard"
              ino:status="has-truncated-values">
   ```

   The function `ino:Index("optimize",...)` should be used when a considerable amount of data has been deleted. This may speed up query execution considerably.
2. Shrink the size of the structure index if possible. The structure index may contain paths which have been added when storing documents which have been deleted in the meantime. Again, using the function `ino:Index("optimize",...)` may improve the performance of query execution.

The ino:RecreateIndex function

The function `ino:RecreateIndex` causes the regeneration of all index information related to a given doctype. This includes all types of indexes:

- standard indexes (simple and compound)
- text indexes
- structure index
- index for ino:docname

The intended use of this function is to rebuild the indexes of a doctype in cases where indexes have been corrupted. If you wish to recreate the text indexes only, use the command `ino:RecreateTextIndex` instead.

Caution: During normal operation, indexes are kept consistent by Tamino, and there is no reason to use this function. If, however, you think that a certain index should be recreated, the recommended method is to use the Tamino Schema Editor as described in the section *Maintaining Tamino Indexes*.

Function call

The function call has two parameters, namely the desired doctype and the collection it resides in. The syntax is:

```
_admin=ino:RecreateIndex("CollectionName", "DoctypeName")
```

The ino:RecreateTextIndex function

The function `ino:RecreateTextIndex` causes the regeneration of all text indexes for a given doctype within the database. A example of the usage of this function is to recreate text indices after the language tokenizer has been changed. Its call has two parameters, namely the desired doctype and the collection it resides in. The syntax is:

```
_admin=ino:RecreateTextIndex("CollectionName", "DoctypeName")
```

Caution: Do not change the setting of the language tokenizer while this function is in progress.

If you change the language tokenizer (for example, from "white space-separated" to "japanese", on platforms where this is supported), you must run the function `ino:RecreateTextIndex` to re-create the text indices, unless the doctype was empty.

The ino:RepairIndex function

The function `ino:RepairIndex` is used to fix indexes that are incomplete. It is called with 3 parameters:

```
_admin=ino:RepairIndex("CollectionName", "DoctypeName", "action")
```

The use of this function is necessary under various conditions such as:

- An `ino:RecreateIndex` or `ino:RecreateTextIndex` function that ran in a non-session context was terminated prematurely due to an error or server shutdown.
- An implicit construction of indices that was triggered by a schema update was terminated prematurely due to an error or server shutdown.

In all cases, none of the affected indexes can be used for query optimization, which may result in decreased performance of query execution and a higher server load.

There are two conceptually different ways to fix the problem, as indicated by the parameter `action`, which can take one of the following values:

- "continue": The index regeneration is started or resumed, starting at that point where the error occurred. In most cases, this is the recommended way of invocation.
- "drop": The index regeneration is not performed. Instead, the situation is resolved by removing all affected indices from the database and from the schema. Thus, ensure you have backed up your schema before using this feature. Also be aware that further queries may slow down if they rely on the indexes for optimization. This way of invocation is a kind of emergency exit to quickly recover from an interrupted index-creating operation. For example, this could be necessary if you issued a schema update that added new indices, but there is not enough disk space available to hold the additional index information. In this scenario, the invocation with "continue" would repeatedly fail, whereas "drop" would return the collection into a consistent state.

During server startup, Tamino checks whether one or more incomplete indices exist, and writes appropriate messages to the job log. Therefore, after an unexpected interruption of operation such as a power failure, check the job log for such messages. If necessary, issue `ino:RepairIndex` for the reported doctypes to prevent performance degradation.

The job log could, in such a case, contain messages such as the following:

```
Info: (INOAAI0574) Starting database 'mydb'
Info: (INODSA1002) Tamino server 4.2.1 on Windows ...
Info: (INOXHI8265) Default tokenizer 'white space-separated'
Info: (INOXRI8801) Request to create indexes for doctype Play
                   in collection Shakespeare is pending
Info: (INODSI1452) Server session 3 started
Info: (INODSI1636) Tamino server successfully started
```

The ino:Request function

The function `ino:Request` allows the administrator to cancel any currently active Tamino client request, regardless of which client originally issued the request. It is called with 2 parameters:

```
_admin=ino:Request("cancel", "requestID")
```

The request ID is a unique identifier by which the X-Machine identifies currently active requests.

To determine the IDs of all active requests, issue the following XQuery command:

```
declare namespace tf='http://namespaces.softwareag.com/tamino/TaminoFunction'
tf:current-requests()
```

In the `xq:result` section of the response document of this XQuery command, details for each active Tamino request are provided in an `ino:request` element. The required request ID is given by the value of the attribute `id`. Here is a sample `ino:request` element, in which the request ID has the value "00000002":

```
<ino:request bytes_returned="0" collection="ino:collection" duration="0"
    http_method="POST" id="00000002" memory_usage="1535992"
    started="2008-08-06T07:03:16+02:00" status="active">
  <ino:from
    remote_address="nnn.nnn.nnn.nnn" server_host="MyHost"
    server_software="Apache/2.2.9 (Win32)"
    user_agent="Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.16) ↩
Gecko/20080702 Firefox/2.0.0.16"/>
  <ino:command value="declare namespace tf='http://namespaces.softwareag" ↩
verb="_XQUERY"/>
</ino:request>
```

If the client that originally issued the request assigned a request ID (see the description of the `ino:Accessor` administration function for details), the value of the `id` attribute will be the value of the client-assigned request ID, otherwise the value of the `id` attribute will be a system-generated value.

You can prevent unauthorized usage of the administration function `ino:Request` and the XQuery command `tf:current-requests()` by defining appropriate access control elements (ACEs) using the Tamino Security Manager, as well as the required access control lists (ACLs) and user groups. See the description of ACEs, ACLs and user groups in the Security Manager section of the Tamino Manager documentation for details. Note that when you add an ACE, the Security Manager dialog asks you to specify a node name for which the ACE applies. Instead of entering a node name in the ACE dialog, enter the function name `ino:Request()` or `tf:current-requests()`. If you specify `tf:current-requests()`, a further dialog will ask you to define the namespace *tf*; in this case, use the namespace declaration shown above.

Related information

The ino:Request function is used by the administrator to cancel any active Tamino request, regardless of which client originally submitted it. Clients without administrator rights can cancel their own requests using the `ino:Accessor` function of the `_admin` command.

The _commit command

The `_commit` command is used to commit the database changes made so far during the current transaction. A new transaction is started automatically when the next command in a session is received that causes an update or acquires a lock.

Note: For a summary of restrictions, see the section ***Transaction-Related Commands***.

Example:

```
URLprefix?_commit=*&_sessionid=479356&_sessionkey=220815
```

The _connect command

The `_connect` command is used to establish a session. A session ID and a session key are returned in the X-Machine response. The first subsequent command to the X-Machine must specify this session ID and session key using the `_sessionid` and `_sessionkey` parameters. The session ID must be supplied unchanged in all subsequent commands for the current session. The session key is a unique identifier for the current command. Each time the X-Machine receives a command that contains the session ID and session key from the previous response, it processes the current command and returns a new session key to be used in the next command. The X-Machine uses an algorithm to generate each new session key; it does not simply increment the old session key.

The first transaction in the session is started implicitly as soon as the X-Machine receives a command that changes the database, such as the `_process` command, or as soon as a command is issued that acquires a lock.

The command can take the parameter `_QueueNextRequest` with the value of "yes" or "no". This parameter specifies whether or not it is possible to send a new request in the session before the response of the previous request in the session has been fully received. The value you specify for this parameter overrides the value of the server XML parameter `queue next request` for the duration of the session. See the section ***Queuing a Follow-Up Request*** for further information.

Example:

To establish a session, specify the following.

```
URLprefix?_connect=*
```

When you start a session, you can specify the access rights that other users or applications can have to the data that is processed during the session. See the section ***Session Parameters*** for details.

The _cursor command

The cursor command is used for processing the result of a query that uses XQuery (via the `_xquery` command) or X-Query (via the `_xql` command). The result is a sequence of items for XQuery or a node set for X-Query. The cursor allows you to position within that sequence. The items of the sequence are arbitrary nodes and values.

Any number of cursors may be defined and used within the same session. To identify any given cursor, a *handle* is used. This is an identifier that is returned by Tamino in the response document of the initial "_cursor=open" request that defines the cursor. Subsequent requests that involve the cursor must specify the option "_handle=*Handle*".

A cursor is opened using the option "_cursor=open". In subsequent requests using the option "_cursor=fetch", a number of entries starting at a given position in the result set can be retrieved by specifying "_position=*Position*" and "_quantity=*Quantity*". The subset of the result set fetched when "_position" and "_quantity" are specified is termed the *fetch set*.

If the option "_scroll=yes" is specified when the cursor is opened, the value specified for *Position* in subsequent fetches for the same cursor can be anywhere in the range from 1 to the maximum number of entries in the result set.

If the option "_scroll=no" is specified when the cursor is opened, the value specified for *Position* in subsequent fetches for the same cursor must be higher than the last entry returned from the previous fetch. Thus, if a fetch returns the entries 101-110, the lowest entry that can be returned from the next fetch is 111. In this example, a subsequent fetch could return entries 111-115 or 125-140, but not 105-115.

The default value for scroll is "_scroll=no".

The parameter `_sensitive` is required when opening a cursor with `_xquery`. Legal values are "no" and "vague". If you specify "_sensitive=no", an *insensitive* cursor is opened. This means that the query is calculated on a fixed input when the cursor is opened, and thus the result sequence remains unchanged as long as the cursor is active. If you specify "_sensitive=vague", a *vague* cursor is opened. The query is calculated on an input that takes modification operations of parallel transactions into account. Thus, the result sequence can vary during the lifetime of the cursor if documents that match the original query criteria are inserted, updated or deleted in the meantime.

Note: A transaction that has an open vague cursor cannot perform document modifications (update, insert, delete). A transaction that has an open insensitive cursor can do XQuery updates.

With cursors in the context of an `_xql` command, the behaviour always implicitly corresponds to "_sensitive=vague". You can specify the `_sensitive` parameter explicitly, but the only valid value is "vague".

The parameter `_count` allows you to specify whether the response document should contain a count of the number of documents that satisfy the query. If you specify "_count=no", no count is

returned. If you specify "_count=cheap", the count is returned if Tamino can calculate this without major additional effort. The default value is "_count=no". If you specify any other value for the `_count` parameter, the default value is assumed. If a count is returned, it is delivered as the value of the attribute `ino:count` of the element `ino:cursor`. Tamino delivers no `ino:count` attribute if "_count=no" was specified or if Tamino cannot deliver the count without major additional effort.

Opening a cursor

To open a cursor, use a command of the following form. The commands are shown split across several lines for typographical reasons.

```
_cursor=open&_xquery=Query
[&_scroll={yes|no}]
&_sensitive={no|vague}
[&_count={no|cheap}]
```

or

```
_cursor=open&_xql=Query
[&_scroll={yes|no}]
[&_count={no|cheap}]
```

This identifies a set of entries that match the given query. All subsequent operations using this cursor will operate on this result set.

Get result documents

To retrieve entries from the result set, use a command of the following form:

```
_cursor=fetch&_handle=Handle&_position=Position[&_quantity=Quantity]
```

Here, *Handle* is the handle returned from the "_cursor=open" request, *Position* is the required starting position within the result set and *Quantity* is the number of entries to be returned. If you do not specify a quantity, the default value is 10.

Specifying "_quantity=0" is supported. It can be used for checking if a particular position, as given by the `_position` keyword, exists within the cursor. If a response document contains no error, the position exists in the cursor, otherwise the position does not exist in the cursor.

The position of the first cursor result is 1.

The result document contains

```
<ino:cursor ino:handle="Handle">
  <ino:current ino:position="p" ino:quantity="q"/>
  <ino:next ino:position="p+q"/>
  <ino:prev ino:position="p-q"/>
</ino:cursor>
```

where:

Handle is the handle of the cursor (generated by Tamino and returned in the result of the open cursor call).

p is the position as specified in the original request,

q is the quantity delivered (which usually is the quantity requested except at the end of the cursor),

The attribute `ino:position` in the elements `ino:next` and `ino:prev` gives the cursor position that would be required for a subsequent fetch request that uses the same cursor.

The element `ino:prev` is not returned in the case of a non-scrollable cursor, or if *p* is 1. If *p-q* is less than 1, `ino:position` will be set to 1. The element `ino:next` is not returned if there are no further results for the cursor.

Combining open and fetch

The open and fetch operations can be combined by using a request of the following form:

```
_cursor=open&_xquery=Query[&_scroll={yes|no}]&_sensitive={no|vague}
&_position=Position[&_quantity=Quantity]
```

```
or
```

```
_cursor=open&_xql=Query[&_scroll={yes|no}]
&_position=Position[&_quantity=Quantity]
```

Note: If you are not working within the context of a Tamino session, this combination of open and fetch is the only cursor command that makes sense. You can fetch a subset of the query result set with this kind of command outside a Tamino session, but subsequent calls to fetch data from the cursor are not possible. Note that in XQuery this functionality should not be used as it is offered in a more efficient way as part of the query language by applying a position filter as the outermost expression ((queryexpression)[position() ge start and position() le start + quantity])

Close a cursor

To close a cursor, specify a command of the following form:

```
_cursor=close&_handle=Handle
```

Further aspects

In a session context, a cursor is implicitly closed at the end of a transaction.

If a cursor is opened outside the context of a session, it is closed immediately after the completion of the command in which it was opened.

Associated error messages

The error numbers are 8305 for an invalid handle, 8306 for an invalid position value, 8307 when trying to position backwards on a non-scrollable cursor.

If the query result when opening a cursor is empty, the cursor will still be created. Any attempt to fetch result documents from such a cursor will result in response 8306.

When trying to close a non-existing cursor, response 8305 will be returned.

Example

The following example illustrates how the `cursor` command is used.

1. Open a session.

   ```
   URLprefix?_connect=* ↩
   ```

   Following this command, session *SessionID* with session key *SessionKey* is established.
2. To open the cursor, use a command of the following form:

   ```
   URLprefix/Collection?_cursor=open&_sessionid=SessionID
   &_sessionkey=SessionKey&_xquery=Query
   ```

   Note that *SessionID* and *SessionKey* have to be specified without quotes. The *Query* represents your query statement. A handle is returned for the cursor (in this example the handle is assumed to have the value "1", as used in the following step). A new session key *NewSessionKey* is returned.
3. Use the cursor to retrieve items from the result sequence. The new session key is required also.

   ```
   URLprefix/Collection?_cursor=fetch&_handle=1&_position=2
   &_quantity=2&_sessionid=SessionID&_sessionkey=NewSessionKey
   ```

This statement will return two items matching your query, starting from the second item. Here again, the values for `handle`, `position`, `quantity`, `sessionid` and `sessionkey` have to be given without quotes.

4. Close the cursor, using a command of the form:

```
URLprefix/Collection?_cursor=close&_handle=1&_sessionid=SessionID&_sessionkey=...
```

The _define command

The `_define` command is used to create a new schema or a collection, or to modify an existing schema or collection. The collection and schema can be specified together in a schema definition file. In this case, the collection and / or schema(s) to be defined or updated have to be expressed in terms of the Tamino schema language. For more information on the Tamino schema language see the *Tamino XML Schema User Guide*.

In addition, the collection can be defined separately by using a collection definition document.

The `_define` command can also be used to define schema clusters, i.e. several schemas in a single command.

- Defining a schema
- Defining a schema cluster
- Defining a collection
- Restriction for non-XML doctypes

Defining a schema

The syntax for using a schema definition file is as follows:

```
URLprefix?_define=SchemaDefinition[&_mode=validate]
```

The input schema definition document defines both the schema and the collection to which it belongs. If the collection does not already exist, it is created automatically.

The option `_mode=validate` can be used for cases of schema evolution, i.e. where an existing schema is modified. For more information, refer to the section *Schema Modifications* in the *Tamino XML Schema User Guide*.

Defining a schema cluster

It is possible to define a schema cluster, i.e. two or more schemas with a single _define command. This can be useful if, for example, you wish to define several schemas that reference each other (cyclic schema definition). Tamino checks the schemas for completeness only after all of the schemas in the _define command have been processed. This means also that a schema that imports or includes other schemas can be defined together with the required import or include files with a single _define command; in this case, the order in which the schema definition and its import or include files are specified in the _define command is irrelevant. In addition, defining multiple related schemas in a single command is often more efficient, since many schema checks will only be performed once.

The syntax of this variant of the _define command is:

```
URLprefix?_define=$S1,$S2,...&$S1=SchemaDef1&$S2=SchemaDef2...
```

where *$S1*, *$S2* etc. are placeholders for schema documents that follow later in the command, and *SchemaDef1*, *SchemaDef2* etc. represent the actual schema definitions. The placeholders *$S1*, *$S2* etc. can be any names consisting of 7-bit printable ASCII characters and must begin with a dollar character. Note that the placeholder names are not the names assigned to the schemas that will be created; the schema names and the names of any doctypes defined for the schemas are defined in the schema definitions *SchemaDef1*, *SchemaDef2* etc.

There is no restriction to the number of schemas you can define in this way with a single _define command.

Defining a collection

The syntax for using a collection definition file is as follows:

```
URLprefix?_define=CollectionDefinition[&_mode=validate]
```

The structure of *CollectionDefinition* is as follows:

```
<tsd:collection name="CollectionName"
     xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
    <tsd:schema use ="Option"/>
</tsd:collection>
```

where *CollectionName* is the name of the collection and *Option* can have one of the following values:

Option	Meaning
required (this is the default value)	Before a document can be stored in the collection, a schema describing the corresponding doctype must be defined explicitly.
optional	A schema is not necessary. If a schema describing a matching doctype is already defined, the document will be stored there. If no such schema exists, Tamino will store documents without a user-defined doctype.
prohibited	The explicit creation of a schema is not permitted. Tamino will store the documents without schema information.

For more information on how Tamino stores documents according to the option specified, see the description of the `_process` command below.

The value of the option can only be modified from "required" or "prohibited" to "optional".

The collection *ino:etc* uses the setting "optional".

Restriction for non-XML doctypes

If an existing schema for a non-XML doctype is being updated, the `tsd:noConversion` child element of the `tsd:nonXML` element of the doctype may only be added or removed if the doctype currently contains no documents.

The _delete command

The `_delete` command is used to delete database documents. You can use X-Query language expressions to select the document(s) to be deleted.

Example:

To delete the XML document of doctype *patient* and with document ID "4711" in the collection *Hospital*:

```
URLprefix/Hospital?_delete=patient[@ino:id="4711"]
```

Note: You can also delete documents by using the `update delete` statement in an `_xquery` command.

The _destroy command

The `_destroy` command removes a query that was prepared using the `_prepare` command. The prepared query is specified by its handle. The following request shows an example that destroys the prepared query that has the handle 42:

```
URLprefix?_destroy=42&_sessionid=479356&_sessionkey=729661
```

See also the section ***Prepared Queries*** for more information about using prepared queries.

The _diagnose command

The `_diagnose` command is used to test the functionality of the outer X-Machine layers concerned with the HTTP handling (the HTTP connection layers).

The X-Machine command `_diagnose` works with HTTP GET and POST requests.

echo

```
URLprefix?_diagnose=echo
```

delivers the HTTP headers seen in the Tamino Server if X-Machine is reachable by the web server, otherwise an error message is generated by the web server.

ping

```
URLprefix?_diagnose=ping
```

tries to establish an HTTP connection and returns a positive answer in the case of success.

time

```
URLprefix?_diagnose=time
```

returns the total amount of CPU time that the server has been active in user mode (i.e. executing user requests) since the database server was started, and the total amount of CPU time that the server has been active in system (kernel) mode (i.e. executing system calls that result from user requests) since the database server was started. The values returned are the total times for all users together, not the individual times for each user.

version

```
URLprefix?_diagnose=version
```

returns the version of the Tamino Server.

The _disconnect command

The `_disconnect` command is used to finish the current session that was opened using a `_connect` command. If a transaction is still open, it is committed (i.e. the changes made during this transaction are stored in the database) before the session is closed.

Note: For a summary of restrictions, see the section ***Transaction-Related Commands***.

Example:

```
URLprefix?_disconnect=*&_sessionid=479356&_sessionkey=934482
```

The _execute command

The `_execute` command executes a prepared query, i.e. an XQuery query that has been precompiled using the `_prepare` command. A prepared query can be executed multiple times in the same session.

The syntax of the command is as follows.

```
URLprefix?_execute=Type&_handle=Handle
```

where *Type* is a text string that describes the type of operation to be executed, and *Handle* represents the handle that was returned when the query was prepared using the `_prepare` command.

Currently, *Type* can only take the value "prepared-xquery".

Example

```
URLprefix?_execute=prepared-xquery&_handle=42&_sessionid=479356&_sessionkey=358290
```

See also the section ***Prepared Queries*** for more information on prepared queries.

The _htmlreq command

The `_htmlreq` command is a special command for processing HTML forms. Refer to the document *Tamino Forms Handler*.

The _prepare command

The `_prepare` command precompiles an XQuery query. Such prepared queries can be executed multiple times at a later stage in the same session by using the `_execute` command. You can define any number of prepared queries in a session. Each prepared query is identified by a handle that is returned by Tamino when the `_prepare` command is issued. The prepared query exists for the duration of the current session; when a session terminates, all prepared queries of that session are deleted.

The `_prepare` command gets the XQuery string as argument. It returns a handle that identifies the prepared query within a session.

The syntax of the command is as follows.

```
URLprefix?_prepare=QueryStatement&_sessionid=SessionID&_sessionkey=SessionKey
```

where *QueryStatement* is the XQuery statement, *SessionID* is the current session ID and *SessionKey* is the session key returned by the previous command in the session.

An example request looks like:

```
URLprefix?_prepare=for $a in collection("bib") return ↩
$a&_sessionid=479356&_sessionkey=220815
```

Tamino returns a response that specifies the handle of the prepared query (in this example, the value of the handle returned is "42"):

```
<ino:query ino:handle="42"/>
```

See also the section ***Prepared Queries*** for more information on prepared queries.

The _process command

The `_process` command is used to insert one or more new documents into a Tamino database or to replace one or more existing documents. If more than one document is being inserted or replaced, the documents must be wrapped in `ino:request` and `ino:object` elements, using the format described in the section *Input and Output File Formats* of the Data Loader documentation. If a single document is being inserted or replaced, the use of the wrappers is optional.

The X-Machine converts XML documents to Unicode before storing them in Tamino.

Tamino does not always preserve the literal representation of XML documents. Also, entities are resolved.

When an XML document is retrieved from Tamino, the returned document is equivalent to the canonical form of the original document. For example, an empty element such as `<br/>` in a document stored into Tamino is returned in the canonically equivalent form `<br></br>`. Also, the order of multiple attributes given in an element's start tag may be altered.

Criteria for inserting or replacing a document

Tamino uses the values of the document ID and/or document name that can optionally be supplied in the input request to decide whether to insert a new document or replace an existing document. The rules governing this decision are as follows:

- If a document ID but no document name is supplied, a document with this ID must already exist and will be replaced. If such a document does not already exist, an error will be returned.

- If a document name but no document ID is supplied, and a document with this name already exists, it will be replaced. If such a document does not already exist, the new document will be inserted with the given document name.
- If a document ID and document name are both supplied, a document with this ID and name must already exist and will be replaced. If such a document does not already exist, an error will be returned.
- If neither the document name nor the document ID is supplied, the new document will be inserted without a document name. Tamino will assign a document ID automatically to the new document.

  Note that this behaviour is different from the processing of requests that use plain URL addressing, as shown in the table below.

This can be summarized as follows:

Document name supplied?	Document ID supplied?	Document with this ID and/or name already exists?	Resulting action
no	no	-	The document is inserted with no name. Tamino assigns an ID automatically. This behaviour is different from the behaviour when using plain URL addressing, for which either the document name or the ID or both are required. See the section ***Criteria for inserting or replacing a document*** in the chapter ***Requests using Plain URL Addressing*** **for details.**
yes	no	yes	The document with the given name is replaced.
		no	The document is inserted with the given name.
no	yes	yes	The document with the given ID is replaced.
		no	Not permitted: if an ID is supplied, a document with this ID must already exist. An error is returned.
yes	yes	yes	The document with the given name and ID is replaced.
		no	Not permitted: if a name and an ID are supplied in the same request, a document with this name and ID must already exist. An error is returned.

Specifying the document name

The document name can be specified in several ways:

- As the value of the `ino:docname` attribute of an `ino:object` element that wraps the document, for example:

```
URLprefix/CollectionName?_process=
<ino:request    xmlns:ino="http://namespaces.softwareag.com/tamino/response2">
  <ino:object ino:docname="DocumentName">
    DocumentContent
  </ino:object>
</ino:request>
```

- In the URL, in the same way as described for plain URL addressing in the section ***URL format for Plain URL addressing***. The syntax is:

```
URLprefix/CollectionName/DoctypeName/DocumentName?_process=DocumentContent
```

If more than one way of supplying the name is used in the same request, the values given for the name must be identical, otherwise an error response will be returned.

Specifying the document ID

The document ID can be specified in several ways:

- As the value of the attribute `ino:id` of an `ino:object` element that wraps the document, for example:

```
URLprefix/CollectionName?_process=
<ino:request    xmlns:ino="http://namespaces.softwareag.com/tamino/response2">
  <ino:object ino:id="IDvalue">
    DocumentContent
  </ino:object>
</ino:request>
```

- As the value of the attribute `ino:id` in the root element of the document. For example:

```
URLprefix/CollectionName?_process=
<DocumentRootElement ino:id="IDvalue">
        ...
</DocumentRootElement>
```

- In the URL, in the same way as described for plain URL addressing in the section ***URL format for Plain URL addressing***. The syntax is:

```
URLprefix/CollectionName/DoctypeName/@IDvalue?_process=DocumentContent
```

If more than one way of supplying the ID is used in the same request, the values given for the ID must be identical, otherwise an error response will be returned.

Storing new documents

You can store XML documents and non-XML documents into a Tamino database. Tamino decides whether a document is an XML document or a non-XML document depending on the MIME media type specified in the HTTP header. See the section ***Media Type Requirements*** for details.

If you do not specify a collection name, regardless of whether the document is an XML document or not, the document will be stored in the collection *ino:etc*.

For any XML document that will be stored in a collection, the root element must have the same expanded QName as the QName of an existing doctype. If there is no such doctype, an error will be returned in the X-Machine response document, unless the schema allows for schemaless storage. See the section ***The _define command*** for information on how to create such a collection. For information about QNames, see the section *General Information on Namespaces* in the document *XML Namespaces in Tamino*.

For a document that will be stored in a schemaless collection other than *ino:etc*, the behaviour is as follows:

- If the document is an XML document, it will be stored in a doctype that is created implicitly by Tamino via an internal hidden schema, and the doctype will have a text index on the root node.
- If the document is a non-XML document, it will be stored in the doctype *ino:nonXML*.

Example

```
URLprefix/Collection?_process=XMLdocument
```

causes the specified XML document to be stored in the specified collection. The doctype is identified by the expanded QName of the XML document's root node.

Replacing existing documents

In general, Tamino offers two ways in which the contents of an existing document can be modified:

- The document can be *replaced*, meaning that the existing document is deleted and a new document is stored. When the document is deleted, all indexing information for the document is also deleted, and when the new document is stored, all appropriate indexing information for the new content is created.
- The document can be *updated*, meaning that the document is not deleted but is modified at its current location. Since any required updates in the index information are limited to the updated parts of the document, and since only the modified data has to be specified, it is in general quicker for Tamino to update a document than it is to replace it.

The `_process` command can be used to replace documents but not to update them. Updating a document is possible using the XQuery `update` command, as indicated in the section ***The _xquery command*** below and also in the section *Performing Update Operations* of the *XQuery User Guide*.

To replace an existing document, you need to address it by the document ID or document name that was assigned to it when the document was created, as described above in the section ***Criteria for inserting or replacing a document***.

Example: replace by specifying the document ID on the root element

Assume that the patient *Charles Dickens* had been originally stored with the following data in the *patient* doctype of the *Hospital* collection:

```
<?xml version="1.0"?>
<patient>
  <name>
    <surname>Dickens</surname>
    <firstname>Charles</firstname>
  </name>
</patient>
```

and the response document contained the following data, specifying that the document ID "15" had been assigned to the new document:

```
<ino:object ino:collection="Hospital" ino:doctype="patient" ino:id="15" />
```

Then, to change the contents of this document, for example to change the contents of the element `firstname` from "Charles" to "Charlie", send a `_process` request with the new data, specifying ino:id="15" on the root element, for example:

```
<?xml version="1.0"?>
<patient ino:id="15">
  <name>
    <surname>Dickens</surname>
    <firstname>Charlie</firstname>
  </name>
</patient>
```

The _rollback command

The `_rollback` command is used to discard all of the database changes that were made during the current transaction. Supply the session ID that was returned at the start of the session, and the session key that was returned from the previous command.

If an active transaction has modified an external database via X-Node and receives a rollback request, a rollback is also issued on the external database.

Note: For a summary of restrictions, see the section *Transaction-Related Commands*.

Example:

```
URLprefix?_rollback=*&_sessionid=479356&_sessionkey=340711
```

The _undefine command

The `_undefine` command is used to delete one or more existing collections, schemas or doctypes.

Caution: When you delete a collection, schema or doctype, all documents and other data belonging to that collection, schema or doctype are also deleted.

Undefining collections, schemas and doctypes

The syntax of the command is as follows. Note that to delete a doctype it is also necessary to supply the name of the schema.

```
URLprefix?_undefine=Collection[/schema[/doctype]], ...
```

The name of the collection must be supplied as a parameter to the `_undefine` command. The name of a collection in the *URLprefix*, if present, is ignored.

In the same way as it is possible to use the `_define` command to define schema clusters, you can use the `_undefine` command to delete schema clusters. Thus, for example, you can use a single `_undefine` command to delete several schemas that reference each other, without leaving the remaining Tamino schema definitions in an inconsistent state (see the following examples).

Collections whose names start with the characters "ino:" cannot be deleted via `_undefine`; these are reserved for internal use in Tamino.

Examples

To delete the collection *Hospital*, specify :

```
URLprefix?_undefine=Hospital
```

To delete the schema *HospitalSchema* and all its defined doctypes from the collection *Hospital*, specify:

```
URLprefix?_undefine=Hospital/HospitalSchema
```

To delete the doctype *patient* in the schema *HospitalSchema* in the collection *Hospital*, specify:

```
URLprefix?_undefine=Hospital/HospitalSchema/patient
```

To delete the collection *Schools*, the schema *HospitalSchema* in collection *Hospital* and the doctype *Cars* in schema *ResourceSchema* in collection *Transport*, specify:

```
URLprefix?_undefine=Schools,Hospital/HospitalSchema,Transport/ResourceSchema/Cars
```

The _xql command

The `_xql` command performs a database query to retrieve XML documents, using the X-Query query language. X-Query is described in detail in the document *X-Query User Guide*.

Note: Certain characteristics of the documents or nodes returned in the query response document can vary from those returned by requests that use **plain URL addressing**. In particular, the following points apply to the query response that do not apply to plain URL addressing: (a) a response wrapper is returned (b) the pseudo-attributes `ino:id` and `ino:docname` (if defined) are returned (c) the XML prolog is not delivered.

Example

To retrieve the list of surnames of all patients in the doctype *patient* of the collection *Hospital*, use the following:

```
URLprefix/Hospital/patient?_xql=patient/name/surname
```

X-Query search modes

By default, locks are set on all required indexes while an X-Query request is processing them. This ensures that if a query needs to scan several indexes, the query runs in an atomic way, i.e. no concurrent X-Machine command can alter any of the indexes while the query is processing them. This behaviour can sometimes lead to a situation that the indexes are locked for a relatively long time, for example:

- If a query uses a word fragment index, the X-Machine first scans the word fragment index, possibly returning many hits, then scans a text index for all words returned from the word fragment index.

- If a query predicate holds for almost all documents.

While the indexes are locked, no new update/insert/delete requests can be processed, because of course they must wait until the index locks are removed. However, if new read requests arrive while update/insert/delete requests are queued, Tamino queues them behind the update/insert/delete requests. This ensures that the update/insert/delete requests are not kept permanently waiting by newly-arriving read requests that might again require index locks. So a situation can arise whereby one or more relatively simple read requests have to wait a long time in a queue due to the index locks.

For performance reasons, an application might not require the index locks to be set in the way described. By removing this restriction, query results can become slightly inaccurate in some cases if concurrent requests are performing update/insert/delete operations, but this might not be critical for the application.

The optional parameter `_querysearchmode` specifies whether an X-Query request will run in an atomic way, i.e. setting index locks until the request completes, or whether it is acceptable for concurrent requests to access and possibly modify the indexes while the X-Query request is running.

The parameter `_querysearchmode` extends the `_xql` command syntax as follows.

```
_xql=QueryString&_querysearchmode=ModeValue
```

where *`QueryString`* is the X-Query query definition and *`ModeValue`* takes one of the values shown in the following table:

Parameter value	Meaning
_querysearchmode=accurate	All indexes required by the X-Query request are locked. This is the default behaviour.
_querysearchmode=approximate	Indexes required by the X-Query request are not locked.
_querysearchmode=nonserialized	Same as approximate, but additional postprocessing is omitted

The use of either `_querysearchmode=approximate` or `_querysearchmode=nonserialized` has advantages and disadvantages:

- Advantages: Since locks are not used, concurrent update/insert/delete requests do not need to wait, which in turn means that queued queries can be processed sooner. This leads to increased throughput of requests.
- Disadvantages: Queries that could otherwise be processed by using only the indexes might now require a postprocessing phase, in order to ensure that the query result, which might have become inaccurate due to concurrent update/insert/delete requests, complies with the original query predicate. Such a postprocessing phase can lead to much increased query processing times. The postprocessing phase is required if whole documents or parts of them are returned by the query; the postprocessing phase is not required if only aggregated values (for example, values returned by functions such as "count()") are returned. Using `_querysearchmode=nonserialized` avoids the postprocessing phase.

The Tamino-specific HTTP header field `X-INO-querySearchMode` can be used in the HTTP header to pass a value for the query search mode. This can only be done for requests that contain an X-Query command (i.e. requests in which the `_xql` command is used). The effect of `X-INO-querySearchMode` is restricted to the single HTTP request that contains it. It cannot be used on a `_connect` command to set a default for the entire user session.

Note: If the parameter `_querysearchmode` and the HTTP header field `X-INO-querySearchMode` are both supplied, `X-INO-querySearchMode` takes precedence over `_querysearchmode`.

The format of the field `X-INO-querySearchMode` is as follows:

```
X-INO-querySearchMode: QueryMode
```

where *`QueryMode`* can be any one of the allowed *`ModeValue`* values of the `_querysearchmode` parameter, for example:

```
X-INO-querySearchMode: approximate
```

The _xquery command

The `_xquery` command performs a database query using the language XQuery, to retrieve XML documents and optionally update or delete them. XQuery is described in detail in the document *XQuery User Guide*

Note: Certain characteristics of the documents or nodes returned in the query response document can vary from those returned by requests that use **plain URL addressing**. In particular, the following points apply to the query response that do not apply to plain URL addressing: (a) a response wrapper is returned (b) the XML prolog is not delivered.

Examples

Example 1

To find all documents in which the patient's name is "Atkins" in the doctype *patient* in the collection *Hospital*, and for each occurrence return a new document containing the surname and firstname in a "result" element, use the following URL:

```
URLprefix/Hospital?_xquery=
for $p in input()/patient
where $p/name/surname="Atkins"
return  <result>{$p/name/surname},{$p/name/firstname}</result>
```

Here, "input()/patient" delivers a list of `patient` root elements. The `return` statement delivers an element such as the following for each document that matches the query:

```
<result>Atkins,Paul</result>
```

Example 2

To delete all documents in which the patient's name is "Atkins" in the doctype *patient* in the collection *Hospital*, use the following URL (shown here split across several lines for better readability):

```
URLprefix/Hospital?_xquery=
update
for $p in input()/patient
where $p/name/surname="Atkins"
delete $p/..
```

Here, "input()/patient" delivers a list of `patient` root elements. However, documents can only be deleted by deleting the document node rather than the root element, therefore "$p/.." must be specified in the `delete` statement to address the document node which is the parent node of the root element.

For `_xquery`, cursor locks are held until the cursor's transaction is committed or aborted.

Serialization of Response Document

The result of every query performed with `_xquery` is a sequence of elements. By default, the returned documents are enclosed in a wrapper `xq:result` element.

If a direct serialization of a returned node type is not possible, an element wrapper will be used, as follows:

- Attributes directly contained in a sequence are wrapped by the special element `xq:attribute`, for example:

  ```
  <xq:attribute anAttribute="anAttributeValue"/>
  ```

  Only one attribute is allowed per `xq:attribute` element.
- Text nodes directly contained in a sequence are wrapped by the special element `xq:textNode`, for example:

  ```
  <xq:textNode>theTextNode</xq:textNode>
  ```

- Values directly contained in a sequence are wrapped by the special element `xq:value`, for example:

```
<xq:value xsi:type="xs:decimal">theValue</xq:value> ↩
```

- Document nodes directly contained in a sequence are wrapped by the special element `<xq:object>`, for example:

```
<xq:object>
<xq:documentprolog>
<![CDATA[
<!DOCTYPE anDoctype [
<!ENTITY anEnt "an Entity Value">
]>
]]>
</xq: documentprolog>
<aRootElement> …. </aRootElement>
</xq:object> ↩
```

Search Modes

Similar to query search modes available for `_xql`, you can change the search modalities for an XQuery request in order to improve speed. This only affects dirty read transactions, i.e. `_isolationLevel` is set to "uncomittedDocument" or `_lockMode` is set to "unprotected". In all other cases the server returns an error. It is also an error to use it for _xquery update commands.

If no `_querysearchmode` parameter is used, `_querysearchmode=accurate` is assumed.

You can use `_querysearchmode=approximate` for XQuery requests executed in streaming mode, i.e. every document access involves at most one and only one index scan. Using `_querysearchmode=approximate` has the same effect as described above for X-Query requests including a possible postprocessing phase. Prepared queries are implicitly recompiled before execution in case the `_querysearchmode` parameter value has changed. If you add the explain directive to the query prolog in order to analyze query processing it is indicated whether this query can be processed in streaming mode. A warning is returned when `_querysearchmode=approximate` is specified but not applicable.

Queries that cannot be processed in streaming mode, e.g. because they need to access multiple indexes and documents, require locks during the entire query execution in order to guarantee a consistent result. An example illustrates what could happen when locks are not in place the whole time:

Assume there is a doctype `Person` with an ID attribute, together with an element `spouse` whose ID attribute stores the value of the married person and elements `children` with an ID attribute as well. Let us further assume that persons have only children with their spouses. Then a query could ask for a (female) person, her children, her husband and the number of her husband's children. While executing this query the woman's first child is returned, but has not yet been added to the father. As a result, the query returns the woman, its child, the father, and the information that he has no children.

Using _querysearchmode=nonserialized it is possible to force query processing without locks, thus improving performance and request throughput, but at the cost of inaccurate query results: Having no locks could lead to a situation that a query accesses a document multiple times, but discovers that the document has changed or even been deleted since the last access. In this case there is no meaningful result for this query and the error INOXQE6312 will be returned instead.

As with _querysearchmode=approximate prepared queries are implicitly recompiled before execution in case the _querysearchmode parameter value has changed. Furthermore, _querysearchmode=nonserialized can also be used for queries in streaming mode.

X-Machine Command Options

Duration Measurement

The _duration command can be used in conjunction with one or more other X-Machine commands and causes timing information about the other commands to be returned in the response document. Currently it takes one value which must be set to "on", otherwise it will be ignored.

The syntax of the command is as follows.

```
URLprefix?OtherCommands?_duration="on"
```

where *OtherCommands* represents one or more X-Machine commands for which the timing information is to be returned.

When _duration="on" is specified, the response document contains an ino:time element that contains the following attributes:

- ino:time: This specifies the time of day when the request started.
- ino:date: This specifies the date when the request started.
- ino:duration: This specifies the duration in milliseconds of the request.

Syntax of XML Responses

Responses to X-Machine commands are delivered as XML documents that combine context information (optional), result information (optional), and message information (optional) in a Tamino-defined wrapper. Normally, this wrapper has to be parsed by the client that issued an XML command to the X-Machine. The attributes and elements that may occur in a response document are preceded with the namespace prefix string "ino:" to indicate XML constructs from the Tamino namespace http://namespaces.softwareag.com/tamino/response2.

Note: The response wrapper cannot be suppressed by using X-Machine command options. However, certain commands have options to suppress the response wrapper. See the section *Suppressing the Tamino Response Wrapper* for further information.

In general, the success or failure of the client request is indicated in the `ino:returnvalue` attribute of the `ino:message` element of the response document. A value of 0 indicates a successful response, whereas a non-zero value indicates either an error or a non-standard response.

If you use the *Tamino Interactive Interface* with an XML-capable browser, the response document from the X-Machine is displayed in a separate frame of the browser. This is a very useful way of becoming acquainted with the typical response documents that the X-Machine returns for the various X-Machine commands.

Tip: The schema for the response document is available for reference in XML Schema format in the file *TaminoResponse.xsd* in the directory *Files/schemas* under the Tamino installation directory.

Below are examples of the X-Machine responses for various commands.

Example of a response to the _connect command

When a session is started using the `_connect` command, the response document has the following structure:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/" ino:sessionid="2" ino:sessionkey="30381">
  <ino:message ino:returnvalue="0">
    <ino:messageline>_CONNECT: session 2 established</ino:messageline>
  </ino:message>
</ino:response>
```

Example of a response to the _define command

When a doctype is defined using the `_define` command, the response document has the following structure (the example assumes the doctype to be in the schema *HospitalSchema* in the collection *Hospital*):

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="0">
    <ino:messageline>_DEFINE: schema HospitalSchema in collection Hospital ↩
defined</ino:messageline>
  </ino:message>
</ino:response>
```

If the collection to which the schema belongs did not already exist, it is created automatically when the schema is defined. There is no additional X-Machine response to confirm this.

Example of a response to an updating _define command

When a schema is updated using the `_define` command, the response document has the following structure (the example assumes that the schema *HospitalSchema* in the collection *Hospital* is being updated):

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="0">
    <ino:messageline>_DEFINE: schema HospitalSchema in collection Hospital updated ↩
successfully
    </ino:messageline>
  </ino:message>
</ino:response>
```

Example of a response to the _delete command

When an XML document is deleted using the `_delete` command, the response document has the following structure:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="0">
    <ino:messageline>_DELETE: document(s) deleted</ino:messageline>
  </ino:message>
</ino:response>
```

If no documents were found that matched the delete request, the following structure is returned:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="8300">
    <ino:messagetext ino:code="INOXIE8300">No matching document ↩
found</ino:messagetext>
  </ino:message>
</ino:response>
```

Examples of responses to the _diagnose command

The command `?_diagnose=ping` returns a response document with the following structure:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:request>
    <ino:diagnose ino:request-type="ping" />
  </ino:request>
  <ino:message>
    <ino:messageline ino:subject="Server">is alive</ino:messageline>
  </ino:message>
</ino:response>
```

The command `?_diagnose=echo` returns a response document with the following structure:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
              xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:request>
    <ino:diagnose ino:request-type="echo"/>
  </ino:request>
  <ino:message>
    <ino:messageline ino:subject="Authenticated User ID"></ino:messageline>
    <ino:messageline ino:subject="Authentication Type"></ino:messageline>
    <ino:messageline ino:subject="Request Method">GET</ino:messageline>
    <ino:messageline ino:subject="Client's IP address">127.0.0.1</ino:messageline>
    <ino:messageline ino:subject="Client's DNS name"></ino:messageline>
    <ino:messageline ino:subject="Webserver's ↩
hostname">mypc.mycompany.com</ino:messageline>
    <ino:messageline ino:subject="Server Software">Apache/2.0.54 ↩
(Win32)</ino:messageline>
   <ino:messageline ino:subject="User-Agent">Mozilla/5.0 (Windows; U; Windows NT ↩
5.1; en-US; rv:1.8.1.16)
                                          Gecko/20080702 ↩
Firefox/2.0.0.16</ino:messageline>
    <ino:messageline ↩
ino:subject="Accept-Charset">ISO-8859-1,utf-8;q=0.7,*;q=0.7</ino:messageline>
   <ino:messageline ino:subject="Accept-Language">en-us,en;q=0.5</ino:messageline>
    <ino:messageline ino:subject="TransportService">XTS</ino:messageline>
  </ino:message>
</ino:response>
```

The command `?_diagnose=version` returns a response document with the following structure:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:request>
    <ino:diagnose ino:request-type="version" />
  </ino:request>
  <ino:message>
    <ino:messageline ino:subject="Version">n.n.n.n</ino:messageline>
  </ino:message>
</ino:response>
```

where "n.n.n.n" is the Tamino version number.

The command `?_diagnose=time` returns a response document with the following structure:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:request>
    <ino:diagnose ino:request-type="time" />
  </ino:request>
  <ino:message>
    <ino:messageline ino:subject="User Time" ↩
ino:unit="100ns">18626784</ino:messageline>
    <ino:messageline ino:subject="Kernel Time" ↩
ino:unit="100ns">14520880</ino:messageline>
  </ino:message>
</ino:response>
```

Example of a response to the _duration command

The following structure is an example of the X-Machine response when timing information is requested for the query "count(*)" on a set of test data:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
              xmlns:xql="http://metalab.unc.edu/xql/">
  <xql:query>count(*)</xql:query>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQL Request processing</ino:messageline>
  </ino:message>
  <xql:result>11</xql:result>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQL Request processed</ino:messageline>
   </ino:message>
  <ino:time ino:date="2004-01-23" ino:time="13:48:09.218" ino:duration="2" />
</ino:response>
```

Example of a response to the _process command

The following structure is an example of the X-Machine response when an document is stored in the user-defined doctype *patient* in the collection *Hospital*:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="0">
    <ino:messageline>document processing started</ino:messageline>
  </ino:message>
  <ino:object ino:collection="Hospital" ino:doctype="patient" ino:id="3" />
  <ino:message ino:returnvalue="0">
    <ino:messageline>document processing ended</ino:messageline>
  </ino:message>
</ino:response>
```

Example of a response to the _undefine command

When a schema is deleted using the `_undefine` command, the response document has the following structure:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="0">
    <ino:messageline>_UNDEFINE: schema deleted</ino:messageline>
  </ino:message>
</ino:response>
```

When a collection is deleted using the `_undefine` command, the response document has the following structure:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
 xmlns:xql="http://metalab.unc.edu/xql/">
  <ino:message ino:returnvalue="0">
    <ino:messageline>_UNDEFINE: collection deleted</ino:messageline>
  </ino:message>
</ino:response>
```

Example of a response to the _xql command

The following response is returned following the query

```
URLprefix/Hospital?_xql=patient/name/surname
```

which retrieves the `surname` children of the `name` elements of the *patient* doctype in the collection *Hospital*. The full list of the surnames has been limited here to 2 surnames for display purposes. Note that the actual result of the query is the content of the `xql:result` element.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
  xmlns:xql="http://metalab.unc.edu/xql/">
  <xql:query>patient/name/surname</xql:query>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQL Request processing</ino:messageline>
  </ino:message>
  <xql:result>
    <surname ino:id="1">Atkins</surname>
    <surname ino:id="2">Bloggs</surname>
  </xql:result>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQL Request processed</ino:messageline>
  </ino:message>
</ino:response>
```

If no document matches the query, the following structure is returned (the example assumes that a search was done for the surname "xxxx", and that this surname does not exist):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
xmlns:xql="http://metalab.unc.edu/xql/">
  <xql:query>patient[name/surname="xxxx"]</xql:query>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQL Request processed, no object returned</ino:messageline>
  </ino:message>
</ino:response>
```

Example of a response to the _xquery command

The following response is returned following the query

```
URLprefix/Hospital?_xquery=for $p in input()/patient
return  <result>{$p/name/surname}</result>
```

which retrieves the `surname` children of the `name` elements of the *patient* doctype in the collection *Hospital*. The full list of the surnames has been limited here to 2 surnames for display purposes. Note that the actual result of the query is the content of the `xql:result` element.

```
<?xml version="1.0" encoding="windows-1252" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
xmlns:xql="http://metalab.unc.edu/xql/">
  <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
    <![CDATA[
    for $p in input()/patient
    return  <result>{$p/name/surname}</result>
    ]]>
  </xq:query>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQuery Request processing</ino:messageline>
  </ino:message>
  <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
    <result>
      <surname>Atkins</surname>
    </result>
    <result>
      <surname>Bloggs</surname>
    </result>
  </xq:result>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQuery Request processed</ino:messageline>
  </ino:message>
</ino:response>
```

If no document matches the query, the following structure is returned (the example assumes that a search was done for the surname "xxxx", and that this surname does not exist):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
  xmlns:xql="http://metalab.unc.edu/xql/">
  <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
    <![CDATA[
    for $p in input()/patient
    where $p/name/surname="xxxx"
    return  <result>{$p/name/surname}</result>
    ]]>
  </xq:query>
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQuery Request processing</ino:messageline>
  </ino:message>
  <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result" />
  <ino:message ino:returnvalue="0">
    <ino:messageline>XQuery Request processed</ino:messageline>
  </ino:message>
</ino:response>
```

Elements and Attributes in Tamino Response Documents

The following table shows commonly occurring elements and attributes in Tamino response documents.

Name	Element (E) or Attribute (A)	Purpose
ino:collection	A	Name of a collection
ino:current	E	Indicates the current cursor position in a result fetch set and the number of documents returned
ino:cursor	E	Cursor segment in a response block, describing parameters for the hit lists from query responses.
ino:database	E	Name of a database
ino:date	A	Contains the date on which the `_duration` command was executed.
ino:diagnose	E	Contains the value of the `_diagnose` keyword that was originally submitted to Tamino
ino:doctype	A	Name of a doctype
ino:duration	A	Contains the duration of the command(s) for which the `_duration` command was specified.
ino:id	A	Document ID
ino:handle	A	The handle number of an open cursor
ino:message	E	Message block
ino:messageline	E	Message line in plain text
ino:messagetext	E	Diagnostic text or error message text
ino:next	E	Indicates the next cursor position in a result fetch set
ino:object	E	Summarizes a document that was successfully stored. Attributes of this element could describe, for example, the collection and doctype into which a new document was stored.
ino:position	A	Indicates the position in the result fetch set
ino:prev	E	Indicates the previous cursor position in a result fetch set
ino:quantity	A	Indicates the number of documents returned from the result fetch set
ino:request	E	Summarizes the original client request to Tamino
ino:response	E	Top-level wrapper element for the response document
ino:returnvalue	A	Return code from XML retrieval
ino:sessionid	A	Session ID of a Tamino XML session, will remain unchanged during a session (which lasts from a `_connect` until a `_disconnect` command)
ino:sessionkey	A	Session key of a command *sub-transaction* inside a Tamino XML session

Name	Element (E) or Attribute (A)	Purpose
ino:time	E, A	Element resulting from a `_duration` command, containing date, time and duration information. Also the attribute within the `ino:time` element that contains the time at which the `_duration` command was executed.
xq:query	E	Contains the XQuery query that was submitted to Tamino
xq:result	E	Wrapper element for the XQuery query result
xql:query	E	Contains the X-Query query that was submitted to Tamino
xql:result	E	Wrapper element for the X-Query query result

Suppressing the Tamino Response Wrapper

In general, the response wrapper cannot be suppressed by using X-Machine Programming commands. However, various possibilities exist for suppressing the response wrapper for queries and server extensions. These possibilities are discussed in this section.

For related information on the response wrapper, see the section ***Syntax of XML Responses***.

- Suppressing the response wrapper in XQuery commands
- Suppressing the response wrapper in X-Query commands
- Suppressing the response wrapper by setting the media type
- Restrictions for suppressing the response wrapper
- Error Handling
- Further information

Suppressing the response wrapper in XQuery commands

There is an option available in the syntax of the `_xquery` command for suppressing the response wrapper for XQuery commands. It is described in the *XQuery User Guide* in the section *Suppressing the Response Wrapper*.

Suppressing the response wrapper in X-Query commands

There is no option available in the syntax of the `_xql` command for suppressing the response wrapper for X-Query queries. However, if the `_xql` command is of the form:

```
_xql=queryFunction(expression)
```

where *queryFunction* is a server extension acting as an output handler, and the code of *queryFunction* includes a server extension callback that sets the media type of the response document, then the response wrapper of the `_xql` command is suppressed implicitly. This is summarized in the section ***Suppressing the response wrapper by setting the media type***.

The query function must be specified in the _xql request on the root level and must not contain a union or any other top-level binary operator. Therefore, the following examples are NOT allowed:

```
_xql=doctype[queryFunction(expression)]
```

```
_xql=queryFunction(expression1)|queryFunction(expression2)
```

Suppressing the response wrapper by setting the media type

It is possible to use the system callback SxsSetProperty with the property SX_PROPERTY_RSP_MEDIATYPE to set the media type of the response document. If a server extension that is used as an output handler for a query (XQuery or X-Query) uses this callback, this implicitly causes the response wrapper to be suppressed.

Restrictions for suppressing the response wrapper

The response wrapper can only be suppressed in the ways indicated above if the following criteria are met:

- A query request sent to the X-Machine must contain only one command, namely a single _xquery or _xql command.

  The query must not be nested within another query (for example, a query that is issued in an XML callback that is contained in a server extension that is being used as an output handler).
- No query cursor is being used.

Error Handling

If an error occurs during the execution of a request whose response wrapper has been suppressed, an error text will be appended to the response written up to that point. The error will be wrapped in the standard error markup (<ino:message>, <ino:messagetext> etc.).

Further information

If a response with a suppressed wrapper is being returned within a session, the session ID and session key are passed in the HTTP response header. See the section ***The HTTP header fields X-INO-Sessionid and X-INO-Sessionkey*** for details of the response header fields.

Transaction-Related Commands

If Tamino accesses Adabas databases via X-Node, then transactions are started automatically on the X-Node database. These are closed with a commit or rollback when the Tamino transaction completes.

The following information is available for transaction-related commands:

- Summary of Commands and Usage
- Restrictions
- The HTTP header fields X-INO-Sessionid and X-INO-Sessionkey
- Session Parameters
- Comparison of Locking Mechanisms
- Default Parameters for Sessions, Transactions and Requests
- Influence of Locking on Query Processing
- Effect of implicitly terminating a transaction

For more information on transaction processing, see the document *Transactions Guide*.

Summary of Commands and Usage

The commands `_connect`, `_commit`, `_rollback` and `_disconnect` are needed to perform transaction processing using X-Machine commands.

A typical transaction sequence is:

1. Start a new session by using the X-Machine `_connect` command. The response from the X-Machine for this command contains two attributes only delivered within a session, namely `ino:sessionid` for a unique session ID for the new session and `ino:sessionkey` for a key describing the current command.
2. Issue some commands against X-Machine, e.g. `_define`, `_process` to define a schema and load data. These "normal" commands must be accompanied by the `_sessionid` and `_sessionkey` information in order to maintain the transaction continuity. This implies a request with at least three keyword/value pairs. The order is not important, therefore we may assume the following syntax for an X-Machine request during a transaction:

   ```
   URLprefix?Command=Data&_sessionid=SessionID&_sessionkey=SessionKey
   ```

3. Finish the transaction by either:

   Discarding the previous changes using the `_rollback` command:

   ```
   URLprefix?_rollback=*&_sessionid=SessionID&_sessionkey=SessionKey
   ```

   Or finishing the transaction by committing the previous changes:

```
URLprefix?_commit=*&_sessionid=SessionID&_sessionkey=SessionKey
```

4. Perform other transactions as required by repeating steps 2 and 3.
5. Close the session with the `_disconnect` command:

```
URLprefix?_disconnect=*&_sessionid=SessionID&_sessionkey=SessionKey
```

Restrictions

On platforms where distributed transactions are supported using two-phase commit, the following commands are not allowed within a distributed transaction:

- `_commit`
- `_disconnect`
- `_rollback`

For the `_disconnect` command, the following additional restriction applies:

- On platforms supporting COM+, this command cannot be used with declarative COM+ transactions.

The HTTP header fields X-INO-Sessionid and X-INO-Sessionkey

Instead of the `_sessionid` and `_sessionkey` parameters, the X-Machine request may also specify the HTTP header fields `X-INO-Sessionid` and `X-INO-Sessionkey` in the request header. This feature is especially important in conjunction with plain URL addressing which does not allow specifying the `_sessionid` and `_sessionkey` parameters.

If an X-Machine request contained a `_connect` command or successfully continued in a previously established session context by passing a session ID and session key, it will return the session ID and the new session key to be used in the subsequent request in the following ways:

- the HTTP header fields `X-INO-Sessionid` and `X-INO-Sessionkey` will always be returned in the HTTP response header
- the `ino:sessionid` and `ino:sessionkey` attributes are embedded in the XML response body.

This method of returning this information will only be used if the response is embedded into an `<ino:response>` wrapper element. See the example in the section ***Example of a response to the _connect command***. On the other hand, when using plain URL addressing, there is no response wrapper and the new session context will only be returned in the HTTP response header.

If a request containing an invalid session context is received by the X-Machine, one of the following will happen:

- if plain URL addressing is being used, HTTP status 400 will be returned;

- otherwise an XML document describing the error will be returned.

If both the HTTP header field `X-INO-Sessionid` and the parameter `_sessionid` are specified in the request, an error will be reported if they are inconsistent. If both the HTTP header field `X-INO-Sessionkey` and the parameter `_sessionkey` are specified in the request, only `X-INO-Sessionkey` will be processed. If `X-INO-Sessionkey` is valid, the `_sessionkey` parameter is ignored.

Session Parameters

There are additional session parameters that can be specified when a session is started with the `_connect` command. They are specified in the `_connect` command as additional keyword/value pairs. The available session parameters are:

The _isolationLevel Parameter

This parameter, together with the `_lockMode` parameter, specifies in what way two or more transactions in a session context can access the same data simultaneously. The `_isolationLevel` parameter can also be specified on requests in a non-session context, since such requests represent self-contained transactions.

For a detailed description of the isolation level and lock mode settings with examples, refer to the *Transactions Guide* document.

The interaction between the `_isolationLevel` and `_lockMode` parameters is described in the section ***The _lockMode Parameter*** below.

The parameter can have the following values:

- uncommittedDocument
- committedCommand
- stableCursor
- stableDocument
- serializable

In a session context, the default is `_isolationLevel=stableDocument`. In a non-session context, the default is `_isolationLevel=uncommittedDocument`, except for XQuery update commands, for which the default is `_isolationLevel=committedCommand`.

The value of `_isolationLevel` cannot be changed within a transaction.

The available isolation levels are described in the following sections.

_isolationLevel=uncommittedDocument

A command within a transaction with this isolation level can read a so-called *dirty* document at any time, which means that a concurrent transaction has changed or stored the document but might abort later on. The document content might be outdated in the sense that a concurrent transaction has changed the content after the current transaction has read it.

A command can also modify a document if no concurrent transaction is modifying the document, or no other transaction need the document in a stable state (isolation level `stableCursor` and stronger).

In this isolation level, the commands `_delete` and `_process` are executed as if they were in isolation level "committedCommand" (see below).

XQuery update commands with this isolation level are not possible. In such a case, the command is rejected with a response code 8552.

_isolationLevel=committedCommand

A command within a transaction in this isolation level can read documents that have been modified, inserted or updated by committed transactions but not documents that have been modified, inserted or updated by concurrent non-committed transactions.

_isolationLevel=stableCursor

A transaction with this isolation level guarantees that a document in the cursor result set will not be changed by a concurrent transaction (i.e. will still match the query predicate) in the following cases:

- In the case of a non-scrollable cursor: until the document has been returned to the requesting application and the document is no longer in the current fetch set of the cursor.
- In the case of a scrollable cursor: as long as the cursor exists.

_isolationLevel=stableDocument

This isolation level guarantees that a document that has been read within the current transaction cannot be changed by a concurrent transaction until the end of the current transaction. This isolation level does not guarantee repeatable query results, i.e. a concurrent transaction can create another document that matches the search criteria of a query in the current transaction, so that if the query is issued twice in the same transaction the results can be different.

_isolationLevel=serializable

This isolation level guarantees that the result set of a query and in consequence the database changes made by the commands `_process`, `_delete` and `update` statement of `_xquery` cannot be influenced by concurrent transactions.

Effect of isolation level on concurrent transaction

The following table shows how the setting of the isolation level affects two concurrent transactions that try to access the same data. A transaction in session S1 is already accessing data that a transaction in session S2 now tries to access. S1 can be at any of three stages: (1) the data has been read but not yet written or committed (2) the data has been written but not yet committed (3) the data has been committed.

S1 current status	S1 isolation level	S2 permitted actions
data read but not yet written	uncommittedDocument	S2 can read and write the data
	committedCommand	S2 can read and write the data
	stableCursor	S2 can read data but cannot write it as long as the data is in the result set of an open cursor
	stableDocument	S2 can read the data
	serializable	S2 can read the data
data written but not yet committed	(any isolation level)	S2 can read the data with isolation level set to uncommittedDocument only
data committed	(any isolation level)	S2 can read the data regardless of the isolation level set for S2

The _lockMode Parameter

This parameter, together with the `_isolationLevel` parameter, specifies in what way two or more transactions in a session context can access the same data simultaneously. The _lockMode parameter can also be specified on requests in a non-session context, since such requests represent self-contained transactions. If one transaction is currently accessing data from the database and a second transaction attempts to access or modify the same data, the setting of the`_lockMode` and `_isolationLevel` parameters for each of the transactions determines whether access is possible for the second transaction.

For a detailed description of the isolation level and lock mode settings with examples, refer to the *Transactions Guide* document.

The locking of documents, doctypes and collections is controlled mainly by the setting of the `_isolationLevel` parameter. The `_lockMode` parameter gives advanced users additional possibilities for controlling the locking behaviour. Using `_lockMode`, it is possible to override the locking behaviour that has been defined by the `_isolationLevel` parameter. If you set the value of `_lockMode` to one of the valid values as described below, you set or remove locks on documents regardless of the setting of the `_isolationLevel` parameter.

The default behaviour for locking is defined by the `_isolationLevel` parameter. For this reason, there is no default value for `_lockMode`.

The `_lockMode` parameter can also be specified in a non-session context on a single command. This controls the behaviour of the command if it accesses data that is currently under the control of a transaction from another session or another command using the parameter in a non-session context.

The parameter can have the following values:

- unprotected
- shared
- protected

Since each of the concurring transactions or non-session commands specifies its own value for the parameter, there is a combined effect that is best described using an example.

The following example assumes that there are two sessions, namely S1 and S2, and a transaction in S1 is already accessing data that a transaction in S2 now tries to access. S1 can be at any of three stages in the transaction: (1) the data has been read but not yet written or committed (2) the data has been written but not yet committed (3) the data has been committed.

S1 current status	S1 current lock mode	S2 permitted actions
data read but not yet written	(no value specified)	S2 can access the data according to the value of the isolation level (see the isolation level table above)
	unprotected	S2 can access the data regardless of the S2 lock mode and isolation level settings
	shared	S2 can read the data regardless of the S2 lock mode and isolation level settings, but cannot write the data with any S2 lock mode setting and cannot read the data if the S2 lock mode is protected
	protected	S2 can only do a dirty read, with the lock mode set to unprotected or the isolation level set to uncommittedDocument
data written but not yet committed	(any lock mode)	S2 can only do a dirty read, with the lock mode set to unprotected or the isolation level set to uncommittedDocument
data committed	(any lock mode)	S2 can access the data regardless of the S2 lock mode and isolation level

The `_lockMode` parameter can be specified on every command.

The _lockWait Parameter

This parameter specifies which action to take if data is not accessible to the current transaction because another transaction has used the `_isolationLevel` or `_lockMode` parameter to restrict access to the data. The `_lockWait` parameter can also be specified on requests in a non-session context, since such requests represent self-contained transactions.

The `_lockWait` parameter can have the following values:

Parameter and value	Meaning
_lockWait=yes	If data is locked by a concurrent transaction, wait until the data is unlocked by the concurrent transaction and then continue processing.
_lockWait=no	If data is locked by a concurrent transaction, terminate the request immediately and return error message INOXYE9155 in the `<ino:message>` section of the response body.

Within a session context, the default value for the `_lockWait` parameter is "yes". In a non-session context, the default value is "no". The `_lockWait` parameter can be specified on every command.

The _maximumTransactionDuration parameter (previously named _transactionTimeout)

This parameter specifies the maximum length of time for which a transaction can be active before the Tamino server rolls back the currently active transaction. It overrides the Tamino server property `maximum transaction duration` for the current session.

This parameter only takes effect when specified in the `_connect` command. It can be specified for other commands but is ignored.

The parameter can have the following value:

Parameter and value	Meaning
_maximumTransactionDuration=*Value*	This specifies the maximum time in seconds for which a transaction can be active. The minimum value is 20 and the maximum value is 2592000.

The _nonActivityTimeout parameter

This parameter specifies the maximum elapsed time (in seconds) that a transaction may be inactive. If this time is exceeded, the Tamino server rolls back the currently active transaction and also terminates the session. The value you specify overrides the value of the Tamino server property `Non-Activity Timeout`.

Note that the figure you specify for this parameter is only approximate. In any particular instance, the actual amount of time can vary from this value by up to 10 seconds.

The minimum value is 20, the default value is 900 and the maximum value is 2592000 (=30 days).

The _isolation Parameter (deprecated)

This parameter is provided for compatibility with Tamino Version 3 and is deprecated in the current version. In the current version it has been renamed to `_lockMode`. The permitted parameter values and their effects are the same as for the parameter `_lockMode`. In all new applications please use only `_lockMode`.

Comparison of Locking Mechanisms

The locking of documents, doctypes and collections is controlled mainly by the setting of the `_isolationLevel` parameter.

The `_lockMode` parameter gives advanced users additional possibilities for controlling the locking behaviour. Using this parameter, it is possible to override the locking behaviour that has been defined by the `_isolationLevel` parameter. If you set the value of `_lockMode` to one of the valid values as described above, you set or remove locks on documents regardless of the setting of the `_isolationLevel` parameter.

The default behaviour for locking is defined by the `_isolationLevel` parameter.

Default Parameters for Sessions, Transactions and Requests

The value given for the parameter `_isolationLevel` in the `connect` command at the beginning of a session is used as the default value of this parameter at the start of every transaction in the session. This value can be changed at the beginning of each transaction in the session by specifying a new value for the parameter, but the value reverts to the original default value at the start of every subsequent transaction. The parameter can be supplied on each request within a transaction, but in this case it must have the same value for every request in the transaction.

The values given for the parameters `_lockMode`, `_isolation` and `_lockWait` on the `connect` command at the beginning of a session are used as the default values of these parameters at the start of every request in every transaction in the session. New values for these parameters can be specified on every request, but the values revert to the original default values at the start of every subsequent request.

If a parameter is specified with no value, it is ignored.

Influence of Locking on Query Processing

During query processing, Tamino uses the defined indexes to select a set of documents that match the query. If no index is defined that can be used to process the query, Tamino temporarily locks all documents in the doctype until it has determined which documents match the query. When the set of documents that match the query has been determined, Tamino releases the locks on the documents that do not match the query.

This mechanism is necessary to ensure that no change can be made to any document in the doctype by a concurrent transaction while the query is being evaluated, since such a change to a document could affect whether or not the document matches the query, thereby causing Tamino to return inconsistent query results.

During query processing, Tamino can only lock all documents in a doctype if none of the documents is already locked by a concurrent transaction (that applies of course only for incompatible locks; if a read-lock is applied to a document, then further read-locks on the same document are allowed). This applies regardless of whether or not the documents locked by the concurrent transaction match the query. In such a case, the transaction that issues the query has to wait (if it has specified `_lockWait=yes`) until the concurrent transaction releases the lock, otherwise the request will be terminated with an error response but the transaction will continue.

If an index has been defined on a numeric element or attribute, it must be specified in the query predicate without quotes, otherwise Tamino will search for a string value instead of a numeric value. For example, the query predicate `[@A=1]` will search for documents in which the attribute `A` has the numeric value 1, whereas `[@A="1"]` will search for documents in which the attribute `A` has the string value "1". Since in this example the index contains only numeric values, Tamino cannot use the index to find documents that match the query, and will therefore proceed according

to the method described above, locking all documents in the doctype while it searches for matching documents.

There is one exception to this rule: the value of the `ino:id` attribute in query predicates can be specified with or without quotes.

In query predicates that involve boolean operators, the presence of an index can also affect the number of documents that are locked by Tamino during query processing. Suppose for example that there is a schema with an element named `A` of type `xs:integer` with a standard index, and there can only be one occurrence of the element `A` in any document based on the schema. Let `p` be a predicate that cannot be evaluated using an index. For three queries with predicates "[A<5] and p", "[A>5] and p", "[A=5] and p", the index processor produces disjunct result sets, thus the queries do not create locking conflicts.

Effect of implicitly terminating a transaction

If a `_disconnect` command is issued while a transaction is pending or open, the transaction is automatically committed.

If a timeout, deadlock or journal overflow occurs while a transaction is pending or active, the transaction is rolled back.

Prepared Queries

Queries expressed in XQuery can be precompiled once and executed many times in the same session. Such a query is called a prepared query. Each prepared query is identified by a handle that is returned by Tamino when the `_prepare` command is issued.

To create a prepared query, use the `_prepare` command.

To execute a prepared query, use the `_execute` command.

To delete a prepared query, use the `_destroy` command.

A prepared query can be defined with one or more so-called external variables (see below), which allows queries to be parameterized.

If a schema or security modification occurs while the session is active, prepared queries are recompiled automatically.

When a session terminates, the prepared query is no longer available.

External variables

In order to pass parameters to a query, XQuery provides external variables. The value of the external variable is determined when the query is executed using a `_execute` command. Thus any given prepared query can be used to execute different queries at different times in a session, depending on the values assigned to the external variables.

To pass a value to an external variable, the `_execute` command gets a key-value pair. The variable name is the key and the value holds the variable value. The variable value is specified by an XQuery expression.

An example is given by the following prepared query, in which the external variable `$y` is used as a placeholder for a value that will be provided in a subsequent `_execute` command:

```
declare variable $y as xs:integer external
input()/bib/book[@year = $y] ↩
```

The request to execute the prepared query looks like the following (assuming the handle of the prepared query to have the value "42"):

```
_execute=prepared-xquery&_handle=42&$y=2000 ↩
```

which causes the following query to be executed:

```
input()/bib/book[@year = 2000]
```

This will for example return a list of all books published in the year 2000.

Currently there is no support for QName variables; this means that variables cannot be bound to a namespace.

All possible instances of the XQuery data model can be bound to an external variable. This means an external variable can be bound to simple type value, a node or a sequence of nodes and simple type values. To pass a sequence with more than a single item, sequence expressions can be used. The following request passes a sequence holding two xs:integers to the external variable `$y` of a prepared query:

```
_execute=prepared-xquery&_handle=42&$y=(xs:integer("2000"),xs:integer("2001"))
```

The XQuery syntax also allows you to pass the integer values in the following way:

```
_execute=prepared-xquery&_handle=42&$y=(2000,2001) ↩
```

The following example show how computed attribute constructors can be used to pass an attribute node to an external variable:

```
_execute=prepared-xquery&_handle=42&$y=attribute year {"2000"}
```

To specify the value of external variables the following XQuery expressions can be used:

- Literals
- Constructor functions with literal and constructor function arguments
- All types of supported node constructors with literal and constructor functions, content and name expression
- Sequences containing literals, constructor functions and node constructors

External variables in ordinary XQuery requests

External variables can be also passed to ordinary XQuery requests. For example:

```
_xquery=declare namespace xs=http://www.w3.org/2001/XMLSchema
declare variable $y as xs:integer external
input()/bib/book[@year = $y]
&$y=2000
```

Prepared queries and cursors

A cursor can use a prepared query by executing a command of the form:

```
_cursor=open&_execute=prepared-xquery&_handle=Handle[&$var1=Value1][&$var2=Value2]...
```

where *Handle* is the handle of the prepared query, as returned by the `_prepare` command, *$var1*, *$var2* etc. are external variables used by the prepared query and *Value1*, *Value2* etc. are the values to be assigned to the external variables.

Example

```
_cursor=open&_execute=prepared-xquery&_handle=42&y=2000
```

If a cursor is opened using the result set of a prepared query in this way, the same prepared query cannot be used in another concurrent cursor or separate `_execute` command until the cursor has been closed.

Order of Execution of Commands

If several X-Machine commands, parameters or options are supplied in a single request, they are processed in a defined internal order, regardless of the order in which they appear in the request. The order is as follows:

1. _sessionkey
2. _sessionid
3. _encoding
4. _duration
5. _isolationLevel
6. _lockMode
7. _isolation
8. _lockWait
9. _maximumTransactionDuration (previously _transactionTimeout)
10. _nonActivityTimeout
11. _scroll
12. _count
13. _handle
14. _position
15. _quantity
16. _sensitive
17. _connect
18. _diagnose
19. _admin
20. _define
21. _process
22. _delete
23. _cursor
24. _xql
25. _xquery
26. _prepare
27. _execute

28. _destroy
29. _htmlreq
30. _undefine
31. _commit
32. _rollback
33. _disconnect

Interactive Environment for sending X-Machine Commands

To send commands interactively to the X-Machine, you can use the Tamino Interactive Interface, which is an HTML form.

Note: If you have an XML-capable browser, the Tamino Interactive Interface displays the response document that the X-Machine sends in reply to each command.

For details, see the documentation for the *Tamino Interactive Interface*.

5 General Requests

This section describes additional requests that can be used to return information about Tamino databases.

Listing Databases served by the Web Server

To receive a list of the Tamino databases served by a web server on a given machine, issue a request via HTTP using the following URL structure:

```
http://HostName:PortNumber/tamino/list=databases
```

where *HostName* is the name of the machine where the web server is running, and *PortNumber* is the used port.

This will return a response document in which the names of the Tamino databases are listed in `ino:database` elements, for example:

```
<?xml version="1.0" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2">
  <ino:message ino:returnvalue="0">
    <ino:messageline>Processing list of databases</ino:messageline>
  </ino:message>
  <ino:database>HospitalDB</ino:database>
  <ino:database>MyDB</ino:database>
  <ino:message ino:returnvalue="0">
    <ino:messageline>List of databases processed.</ino:messageline>
  </ino:message>
</ino:response>
```

The list shows all databases that are known to the web server, namely:

- all local databases (i.e. databases on the same machine as the web server), and
- all active remote databases that use the same XTS directory server as the web server.

6 Using Plain HTML Forms

The form submission methods specified in the HTML specification (see **http://www.w3.org/TR/html4/**) are compatible with the requests described above. So you can use plain HTML without any scripting or programming to access Tamino.

Please refer to the separate documentation for the Tamino Forms Handler for details.

An example of an application that uses plain HTML to communicate with X-Machine is the Tamino Interactive Interface.

7 Media Type Requirements

The X-Machine evaluates the media type settings specified in the `Content-Type` field of an incoming HTTP request. It distinguishes between text and binary documents. A text document can be an XML document or a non-XML document. A binary document can only be a non-XML document.

The distinction between XML document, non-XML text document and binary document is made according to the media type as shown in the following table:

Media type	Type of Tamino Document
*/xml or */*+xml, for example: ■ application/xml ■ text/xml ■ image/svg+xml	text, XML document
■ text/* (other than text/xml) ■ application/rtf	text, non-XML document
nothing or blank	text, XML document
All other media types	binary document

XML documents are handled according to their Tamino schema definition, if it exists (Tamino also allows the storage of XML documents without a matching schema definition). This includes validating against the information stored in Tamino's Data Map and indexing according to the `tsd:physical` settings in the schema.

Notes:

1. The Tamino data loader uses special rules for determining the media type settings. See the section *Known Media Types* in the data loader documentation for details.
2. Tamino schema definitions can be defined for non-XML documents.

8 Character Encoding

This section describes the character encoding mechanisms for HTTP requests to Tamino and responses from Tamino.

The term "encoding" in this section is used with the semantic defined in the W3C XML specification at **http://www.w3.org/TR/REC-xml/**. The terms "charset" and "character set" are used with the semantic defined in the HTTP/1.1 description at **http://www.ietf.org/rfc/rfc2616.txt**.

Character Encoding of Input Documents

Input documents can be supplied for X-Machine commands such as `_process` and `_define`. The encoding of an input document can be specified explicitly in several ways:

- in the `encoding` attribute of the document's XML declaration
- in the `_encoding` parameter passed in the X-Machine command
- in the `charset` value that is defined in conjunction with the document's `Content-Type` parameter in the HTTP request

If the encoding is not specified in one of these ways, the document is assumed to be encoded according to the value of the server XML parameter `XML document default encoding` (for details see the list of server XML properties in the section Database Properties in the documentation for the Tamino Manager).

All input documents with top-level media type "text" are converted to Unicode. Input data is converted from the client's encoding to Unicode. The original encoding of the input is not remembered. X-Machine uses the internet standards for character set names as defined in the document **http://www.iana.org/assignments/character-sets**.

Hint for users of Microsoft Windows: Please note that Microsoft code page 1252 is close to but not identical with ISO-8859-1 (latin1).

Example

Database queries specifying character encoding can be sent to the X-Machine using the X-Machine command `_encoding` followed for example by the X-Machine command `_xql` in a single HTTP request, for example

```
http://myhost:80/tamino/mydb/mycollection?_encoding=utf-8&_xql=patient/name[surname="Bloggs"].
```

The value of the `_encoding` parameter will be applied to the values of all commands that are subsequently executed. See also the section *Order of Execution of Commands*.

Character Encoding of Output Documents

Output documents are converted by the X-Machine to the encoding desired by the client. Character references are used to represent characters that do not exist in the desired encoding. The desired encoding of the output can be specified in the HTTP header "Accept-Charset". If "Accept-Charset" is omitted, X-Machine uses the encoding of the client request.

Supported Character Encodings

The Tamino server supports all standard character encodings and their well known aliases, as shown in the following list.

Note: It is possible that some Tamino product components do not support some of these encodings. Please see the documentation for the individual developer components for a list of their supported encodings.

Encoding Name	Well known aliases
Adobe-Standard-Encoding	csAdobeStandardEncoding
Big5	950, cp950, csBig5, ibm-1370_VSUB_VPUA, x-big5
CESU-8	
cp850	850, csPC850Multilingual, IBM850
cp851	851, csPC851, IBM851
cp856	856, ibm-856
cp857	857, csIBM857
cp858	IBM00858
cp859	
cp860	860, csIBM860, IBM860
cp861	861, cp-is, csIBM861, IBM861
cp862	862, cp867, cspc862latinhebrew
cp863	cp863, csIBM863, IBM863
cp864	csIBM864
cp865	865, csIBM865, IBM865
cp866	866, csIBM866
cp868	868, cp-ar, csIBM868, IBM868
cp869	869, cp-gr, csIBM869
cp921	921
cp922	922

Encoding Name	Well known aliases
EUC-JP	csEUCPkdFmtJapanese, eucjis, Extended_UNIX_Code_Packed_Format_for_Japanese, ibm-33722_VPUA, ibm-eucJP, X-EUC-JP
EUC-KR	csEUCKR, ibm-970_VPUA, ibm-eucKR, X-EUC-KR
gb18030	ibm-1392
GB2312	1383, chinese, cp1383, csGB2312, csISO58GB231280, EUC-CN, gb, gb2312-1980, GB_2312-80, ibm-1383, ibm-1383_VPUA, ibm-eucCN, iso-ir-58, X-EUC-CN
GBK	CP936, ibm-1386_VSUB_VPUA, MS936, zh_cn, windows-936
hp-roman8	csHPRoman8, r8, roman8
HZ-GB-2312	HZ
IBM01140	CCSID01140, CP01140, cpibm1140, ebcdic-us-37+euro
IBM01141	CCSID01141, CP01141, cpibm1141, ebcdic-de-273+euro
IBM01142	CCSID01142, CP01142, cpibm1142, ebcdic-dk-277+euro, ebcdic-no-277+euro
IBM01143	CCSID01143, CP01143, cpibm1143, ebcdic-fi-278+euro, ebcdic-se-278+euro
IBM01144	CCSID01144, CP01144, cpibm1144, ebcdic-it-280+euro
IBM01145	CCSID01145, CP01145, cpibm1145, ebcdic-es-284+euro
IBM01146	CCSID01146, CP01146, cpibm1146, ebcdic-gb-285+euro
IBM01147	CCSID01147, CP01147, cpibm1147, ebcdic-fr-297+euro
IBM01148	CCSID01148, CP01148, cpibm1148, ebcdic-international-500+euro
IBM01149	CCSID01149, CP01149, cpibm1149, ebcdic-is-871+euro
IBM037	cpibm37, ebcdic-cp-us, ebcdic-cp-ca, ebcdic-cp-wt, ebcdic-cp-nl, cp37, cp037, 037
IBM1026	CP1026, csIBM1026, Ibm-1026_STD
IBM273	273, CP273, cpibm273, csIBM273, ebcdic-de
IBM277	277, csIBM277, cpibm277, EBCDIC-CP-DK, EBCDIC-CP-NO, ebcdic-dk
IBM278	278, cp278, cpibm278, csIBM278, ebcdic-cp-fi, ebcdic-cp-se, ebcdic-sv
IBM280	280, CP280, cpibm280, csIBM280, ebcdic-cp-it
IBM284	284, CP284, cpibm284, csIBM284, ebcdic-cp-es
IBM285	285, CP285, cpibm285, csIBM285, ebcdic-cp-gb, ebcdic-gb
IBM290	cp290, csIBM290, EBCDIC-JP-kana
IBM297	297, cp297, cpibm297, csIBM297, ebcdic-cp-fr
IBM367	
IBM420	420, cp420, csIBM420, ebcdic-cp-ar1
IBM424	424, cp424, csIBM424, ebcdic-cp-he
IBM500	500, CP500, cpibm500, csIBM500, ebcdic-cp-be, ebcdic-cp-ch
IBM852	
IBM855	

Encoding Name	Well known aliases
IBM857	
IBM862	
IBM864	
IBM869	
IBM870	CP870, csIBM870, ibm-870, ibm-870_STD, ebcdic-cp-roece, ebcdic-cp-yu
IBM871	871, CP871, cpibm871, csIBM871, ebcdic-cp-is, ebcdic-is
IBM918	CP918, csIBM918, , ebcdic-cp-ar2, ibm-918_STD, ibm-918_VPUA
ISO-2022-CN-EXT	
ISO-2022-CN	
ISO-2022-JP-2	csISO2022JP2
ISO-2022-JP	csISO2022JP
ISO-2022-KR	csISO2022KR
ISO-2022	2022, cp2022
iso-8859-15	
ISO-8859-1	8859-1, cp819, csISOLatin1, IBM819, ISO_8859-1:1987, iso-ir-100, l1, latin1
iso-8859-2	8859-2, 912, cp912, csISOLatin2, ISO_8859-2:1987, iso-ir-101, l2, latin2
iso-8859-3	8859-3, 913, cp913, csISOLatin3, iso-ir-109, l3, latin3
iso-8859-4	8859-4, 914, cp914, csISOLatin4, ISO_8859-4:1988, iso-ir-110, l4, latin4
iso-8859-5	8859-5, 915, cp915, csISOLatinCyrillic, cyrillic, ISO_8859-5:1988, iso-ir-144
iso-8859-6	1089, 8859-6, arabic, asmo-708, cp1089, csISOLatinArabic, ecma-114, ISO_8859-6:1987, iso-ir-127
iso-8859-7	813, 8859-7, cp813, csISOLatinGreek, ecma-118, elot_928, greek, greek8, ISO_8859-7:1987, iso-ir-126
iso-8859-8	916, cp916, csISOLatinHebrew, Hebrew, 8859-8, ISO_8859-8:1988, iso-ir-138
iso-8859-9	8859-9, 920, cp920, latin5, csISOLatin5, ISO_8859-9:1989, iso-ir-148, l5
JIS_Encoding	ISO-2022-JP-1, JIS
KOI8-R	cp878, cskoi8r, koi8
KSC_5601	949, csKSC56011987, ibm949, ibm949_VSUB_VPUA, iso-ir-149, johab, Korean, ksc5601_1992, KS_C_5601-1987, KS_C_5601-1989, ks_x_1001:1992
mac	csMacintosh
SCSU	
Shift_JIS	943, cp943, cp932, csShiftJIS, csWindows31J, MS_Kanji, pck, sjis, windows-31j, x-sjis
TIS-620	874, cp874, cp9066, ms874, windows-874
US-ASCII	ANSI_X3.4-1968, ASCII, ANSI_X3.4-1986, cp367, csASCII, ISO_646.irv:1983, ISO_646.irv:1991, ISO646-US, iso-ir-6, us
UTF-16BE	cp1201, UTF16_BigEndian, x-utf-16be

Encoding Name	Well known aliases
UTF-16LE	cp1200, UTF16_LittleEndian, x-utf-16le
UTF-32BE	UTF32_BigEndian
UTF-32LE	UTF32_LittleEndian
UTF-7	cp65000
UTF-8	cp1208, cp65001
UTF-16	csUnicode, ISO-10646-UCS-2, ucs-2
UTF-32	csUCS4, ISO-10646-UCS-4, ucs-4
windows-1250	cp1250
windows-1251	cp1251
windows-1252	cp1252
windows-1253	cp1253
windows-1254	cp1254
windows-1255	cp1255
windows-1256	cp1256
windows-1257	cp1257
windows-1258	cp1258

9 Maintaining Tamino Indexes

The X-Machine command `_admin` offers various functions for maintaining Tamino indexes. This section provides general guidelines for maintaining Tamino indexes and describes which functions of `_admin` to use for the various possibilities.

General

Indexes are defined for a doctype by adding corresponding definitions to Tamino schema documents. In normal operation, these indexes are maintained when adding, removing or modifying documents in the doctype. However, there are a few scenarios where special actions might be required:

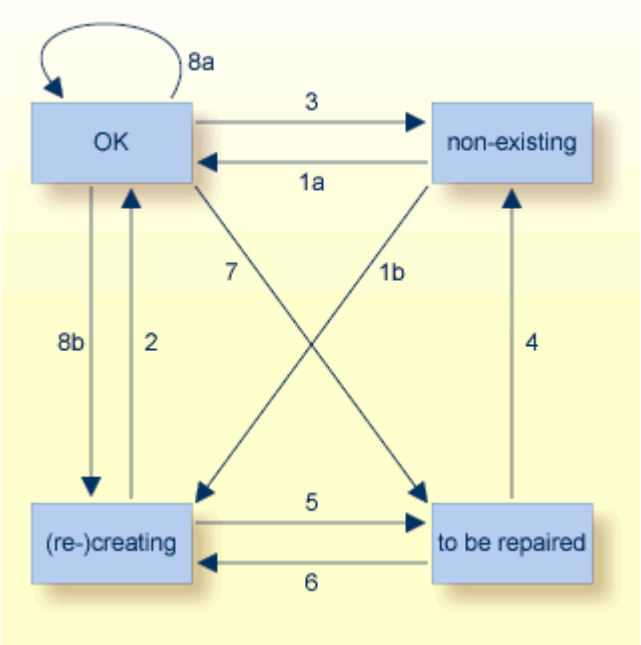
- an index is disabled during an upgrade from a previous version of Tamino
- the Tamino server was aborted during a schema update which requested creation of new indexes
- the Tamino server was aborted while executing a function used to recreate or repair an index was interrupted
- an index has been corrupted

All these scenarios lead to an index which is marked as unusable. An appropriate message will show up in the job log of the Tamino server. Depending on the scenario, an additional message may be returned in a response to the X-Machine request which caused or detected the problem.

The command `_admin = ino:DisplayIndex (...)` may be used to identify the indexes that are disabled. The respective index will be marked as follows:

```
<ino:index ino:indexcoll="myCollection"
           ino:indexpath="myDocument/myElement"
           ino:indextype="standard"
           ino:status="not-available">
```

The following diagram illustrates possible states of an index in a Tamino doctype and the transitions which may occur:



The states are:

State	Meaning
OK	the index is fully operational: it can be used for query processing
non-existing	the index does not exist
(re-)creating	the index is being newly created or regenerated
to be repaired	the index is not usable and must be repaired

Both the "(re-)creating" and the "to be repaired" state are reflected as "not-available" by the `ino:DisplayIndex` function.

Here are the possible reasons for the state transitions (the numbers refer to the arrows in the above diagram):

State Transition	Meaning
1	Update existing schema and add a new index or a new unique constraint (a) in session context (b) without session context
2	An operation which led to "(re-)creating" state has been finished successfully
3	Update an existing schema and remove an existing index or unique constraint
4	Started `_admin=ino:RepairIndex(...,"drop")`. This affects all indexes of a doctype that are in the "to be repaired" state .
5	Tamino server restarted after being aborted while in the "(re-)creating" state

State Transition	Meaning
6	Started `_admin=ino:RepairIndex(...,"continue")` this affects all indexes of a doctype that are in the "to be repaired" state
7	Different possible reasons, as listed at the beginning of this section.
8	Started one of the following commands: ■ `_admin=ino:RecreateIndex(...)`: this affects all indexes of a doctype ■ `_admin=ino:RecreateTextIndex(...)`: this affects all text indexes of the doctype (a) in session context (b) without session context

Most `_admin` functions mentioned above operate on sets of indexes. If you want to recreate a single index, you can use the Tamino schema editor as follows:

To recreate an index using the Tamino Schema Editor

1 Start the Tamino Schema Editor

2 Get the schema defining the index(es) to be recreated, remove the index from the schema definition, then use **Database > Define Schema** to define the schema again. You will be asked whether you want to update the existing schema. Please answer "yes".

3 Use **Edit > Undo Set physical property** to reintroduce the index(es), then use **Database > Define Schema** to define the schema again. You will be asked whether you want to update the existing schema. Please answer "yes". This will cause the index(es) to be recreated.

Note that this operation can run for a considerable length of time.

Special Considerations for Indexes

Special Considerations for Multipath Indexes

Due to the nature of multipath indexes, `_admin=ino:RepairIndex(…,"drop")` will be rejected if there are multipath indexes defined for a doctype.

Special Considerations for Computed Indexes

If a schema defining a computed index for a doctype is updated, all computed indexes will be re-created.

The construction of a computed index relies on the XQuery module where the referenced XQuery function is defined, and potentially also other modules that are imported directly or indirectly by that primary module. If one of these modules has been modified, the index is in general corrupted if the results returned by the indexing function have changed.

However, Tamino does not automatically recreate all computed indexes for potentially affected doctypes. Instead, it is up to the database administrator to determine the set of *potentially* affected doctypes and to invoke the `_admin=ino:RecreateIndex(…)` for all affected doctypes.

The following XQuery can be used to determine the set of potentially affected doctypes if a module with targetNamespace *URI* has been modified:

```
import module namespace si="http://namespaces.softwareag.com/tamino/schemaInfo"

for $dt in si:getDoctypesUsingModule("URI")
return $dt/../*/@name
```

On the other hand, it may happen, that a computed index cannot be recreated at all due to mis-configuration, if for example

- the indexing function's signature has changed
- the indexing function or the enclosing module as a whole have been deleted

In any of these cases it will also no longer be possible to store documents in the affected doctypes.

Note: Queries using the computed index with a modified or broken indexing function may return invalid results.

After such a broken computed index, one of the following can be done depending on the status of the computed indexes:

- update or re-create the module
- update the schema to remove the computed index
- execute `_admin=ino:RepairIndex(…,"drop")`

Dependence on Session Context

All index manipulation commands, namely:

- `_define`
- `_admin=ino:Index("optimize", ...)`
- `_admin=ino:RecreateIndex(...)`
- `_admin=ino:RecreateTextIndex(...)`
- `_admin=ino:RepairIndex(..., "continue")`

may require a long time for execution and perform a lot of changes.

If the command is executed inside a session context, there are potential problems regarding transaction timeouts and journal overflow. In addition, the entire collection is locked exclusively during operation. Hence, it is recommended to use the commands listed above outside a session context (i.e. in autocommit mode).

Performance and Locking Aspects

If an index is being repaired, i.e. it is in the "(re-)creating" or "to be repaired" state, it is disabled. This means that it cannot be used for queries, thus affecting the performance of queries which otherwise could take advantage of that index. When running an index manipulation command, in most cases, except for a short preparation and termination phase of the respective index manipulation command, parallel inserts and updates operating on the respective doctype are possible. If, however, an index used underneath a unique constraint is being disabled, the doctype is locked and no parallel insert or update operations are permitted.

Optimization

In addition, there are scenarios where performance of query execution is also degraded even if the index is not disabled. This may happen if an index is not as selective as it could be, for example:

- a standard or compound index contained long index values which had been truncated to a length of 1000 bytes
- a condensed structure index contains paths with no corresponding documents being stored in the doctype any more

The command `_admin=ino:Index("optimize", ...)` can be used if any of these scenarios might have occurred.

Index

Symbols

A

C

N

O

P

Q

R

S

T

U

W

X