software AG

Tamino

X-Tension: Tamino Server Extensions

Version 9.5 SP1

November 2013

tamino XML Server

This document applies to Tamino Version 9.5 SP1.

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

Document ID: INS-SERVEXT-95SP1-20131030

Table of Contents

X-Tension: Tamino Server Extensions

This document informs you about Tamino X-Tension, the component of Tamino that provides tools for the development, implementation, administration and execution of Tamino server extensions.

This information is primarily intended for configuration managers (incorporation of legacy data, accessing non-Tamino data sources), application logic experts (document analysis) and Tamino administrators (extending Tamino for business demands, administrating server extensions).

The most important aspects of Tamino X-Tension are described in the following documents:

1 Introduction

Tamino X-Tension enables you to develop, implement, administrate and execute Tamino server extensions. Tamino server extensions can be used to extend the Tamino Server functionality by adding user-defined logic. The fields of extensibility cover query language, trigger and mapping functionality. For a description of the calling contexts in the Tamino Server, see the documentation for *Tamino X-Query*, *Tamino XQuery* and the *Tamino XML Schema User Guide*, as well as for the *Tamino Schema Editor*. For server extension function types related to these contexts, see the section ***Tamino Server Extension Functions***.

Functionality that can be added to the Tamino Server by using Tamino server extensions includes:

- Content-based mapping to/from Tamino and other data sources. For detailed explanations, see the section ***Callbacks***;
- User-defined functions, as documented in the section ***Query Functions***;
- Calling applications such as message forwarding or the EntireX XML Wrapper triggered by document processing independent of mapping.

In order to extend the Tamino functionality, you install Tamino one or more server extension packages in a Tamino database. The extension packages contain, amongst other things, Tamino server extension objects based on the calling infrastructure, which may be Direct or the Java Virtual Machine (JVM).

Server extensions that are implemented in libraries that can be loaded and executed dynamically by Tamino X-Tension are said to use the Direct infrastructure. Server extension objects can be written in Java, C++ or in any programming language that can create a shared library or DLL with the C calling convention. Methods of these objects can then be used to extend the Tamino Server's query, trigger or mapping functionality.

Information to help you develop server extension function code is contained in the documents **Tamino Server Extension Functions** and **Developing Tamino Server Extensions** (the latter treats topics such as constructors, destructors, restrictions for Java-based server extensions, what are

callbacks and how are they used). Knowledge of the chosen programming language and its development environment is assumed throughout this document.

A thorough investigation of the problem to be solved and the schema definitions involved is necessary before developing a Tamino server extension.

Creating and installing Tamino server extension packages is easy, thanks to the Tamino X-Tension tools.

Several infrastructures are supported by the X-Tension technology. The term “infrastructure” here means the media used to link user code into the Tamino Server. One or more programming languages can be used to implement a server extension for a specific infrastructure.

Infrastructure	Language	Operating System	Remarks
Direct	C++, other (must be able to create a share library or DLL with the C calling convention)	Any (without restriction)	Local/Inproc modes possible
Java	Java only	Any that supports Java	Cross-platform development possible

Local mode means that the server extension is executed in a process separated from that of the Tamino server. This mode is recommended for testing purposes. It does not provide the high performance that is available with Inproc mode, but serious errors that might be produced by the server extension do not crash the Tamino server.

Inproc mode means that the server extension is executed in the same process as the Tamino server. It provides better performance than local mode, since the call parameters and any callbacks that might be used do not have to be passed back and forth between two different processes. However, if the server extension produces a severe error during processing, it can crash the Tamino server.

Local mode and inproc mode are available as follows:

Infrastructure/Language	Local mode possible	Inproc mode possible
Direct	yes	yes
Java	no	yes

To help you develop Tamino server extension packages, Tamino X-Tension provides a range of development tools (they are included in the Tamino installation kit). These tools guide you step-by-step through the process, from creating a project into which you can insert your function code to generating a package file ready for installation. These development tools have been chosen to ensure a stable and secure environment for the development of Tamino server extensions and later for their execution in Tamino databases. We strongly recommended you to use these tools.

- **X-Tension Builder**
  The Tamino X-Tension Builder can be used for server extension functions written in Direct/C++ or in Java. For a detailed description, see the section *Using the X-Tension Builder*.
- **Object Analyzer**
  The Tamino Object Analyzer can be used to check whether available files (of type DLL, SO, TLB, CLASS, JAR or EXE) are suitable for use as Tamino server extensions. It can also be used to select some functions from a great number of functions and create a package file ready for installation. For a detailed description, see the section *Analyzing Arbitrary Objects*.
- **Package Viewer**
  The Tamino Package Viewer can be used to view the contents of a server extension package.

The Server Extensions Administration part of the Tamino Manager provides facilities for installing and administrating server extensions in a Tamino database. It is described in the section **Administrating Tamino Server Extensions**.

Server extensions can be traced using the SXS Trace. The SXS Trace and its handling is described in the section **Tracing Tamino Server Extensions**.

See the section **Tamino Server Extension Examples** for information about examples of each type of server extension function in C++ and Java.

In addition, we recommend that you read the document Utilizing Server Extensions.

2 Prerequisites

Administrating Tamino server extensions requires system administrator privileges.

The following sections describe more prerequisites for developing Tamino server extensions.

Developing Tamino Server Extensions Using the X-Tension Builder

To develop Java-based server extensions, a Java compiler (javac) environment is required; this is delivered with Tamino. For full use of the Tamino Builder for Java, the path to the JDK binaries must be included in the system path environment variable. The X-Tension Builder for Direct infrastructure may work correctly without JDK support, but Java is not available then.

3 Tamino Server Extension Functions

The information in this chapter is broken down as follows:

Server Extension Objects

Tamino server extensions follow the object oriented approach. From this point of view, server extensions are objects and server extension functions are their methods.

One server extension object may have an arbitrary number of server extension functions as methods. These can also be functions of different types.

Server extensions have constructors and destructors. In Direct/C++ and Java server extensions, the standard constructors and destructors that are provided with the programming language are used.

Tamino server extensions can have member variables. In programming languages that support object orientation, they are handled in the usual way. More details can be found in the section **Developing Tamino Server Extensions**.

A Tamino server extension object is created when one of its methods is used for the first time. The object lasts for the duration of the associated XML session. In the case of an anonymous session, the object lasts for the duration of the associated transaction or request.

Types of Server Extension Functions

Tamino X-Tension supports the following server extension function types according to the context of execution:

- **Query Functions**

  that are explicitly called from within X-Query queries;
- **Trigger Functions**

  that are used for executing functions when storing or deleting a related document independent of mapping;
- **Shadow Functions**

  that are used to create index values for non-XML documents as a shadow of the original;
- **Mapping Functions**

  that are used for storing, retrieving, or deleting documents and called by Tamino document processing, composing or on delete;
- **Initial Functions**

that allow initializing functions to be executed on the server extension object prior to any query, trigger or mapping function execution;

- **Server Event Functions**

  that ensure transaction consistency.

A server extension can contain multiple mapping, trigger, shadow and query functions, but at most one initial function and one server event function. Usually, mapping functions appear as a complete set (one map in function, one map out function, one delete function and one event function that handles at least the commit and rollback events).

Note: All server extension function types are supported for all X-Tension infrastructures. Implementation-specific details are indicated where appropriate.

Function Naming

A single server extension can contain multiple server extension functions.

The function name must be unique within the Tamino database. If a Tamino Server includes several server extensions, the function name defined in the source code, the “internal name”, may not be unique. Therefore when a server extension is installed, each function is given a unique “external name”. By default, the external name is the dot-separated concatenation of the server extension name and the function's internal name, for example: `MyExtension.FirstFunction`. If a name conflict occurs, the Tamino X-Tension Object Analyzer can be used to change the external function name. For query extension functions, it is even possible to change the external name after installation, using the Tamino Manager. Such functions will often be called directly by Tamino database users, so for the convenience of XQuery users we recommend changing the functions' external names immediately after installation to short, easily remembered names.

Query Functions

Query functions are used to extend Tamino's query languages XQuery and X-Query. You can define shortcuts for complex queries and create custom-defined filters. Multiple query functions can be contained in a single server extension object.

The information about query functions is broken down into the following sections:

- Query Function Parameters and Data Types
- Query Function Call Syntax
- Conversion of Query Function Parameters

- Query Functions in X-Query/XQuery

Query Function Parameters and Data Types

A query function has zero or more “in” parameters and exactly one return value.

Direct and Java

A query function has *n* parameters and one return value.

The XML data types allowed for the parameters and the return value and their equivalents in Direct and Java are shown in the following table:

XML	Direction	Direct/C++	Java
xs:boolean	in	sxdbool	boolean
xs:boolean	return value	sxdbool *	boolean
xs:string	in	sxdstring	String
xs:string	return value	sxdstring *	String
xs:double	in	sxddouble	double
xs:double	return value	sxddouble *	double
xs:float	in	sxdfloat	float
xs:float	return value	sxdfloat *	float
xs:int	in	sxdint	int
xs:int	return value	sxdint *	int
ino:XML-OBJ	in	sxdstring	String
ino:XML-OBJ	return value	sxdstring *	String

Passing Parameters

- Use the ByVal attribute for input parameters.
- Reference parameters (i.e. without the ByVal attribute) are always input/output.
- Retval parameters are real return values, e.g.:

```
Function SXSEvent(ByVal status As Long) As Long
    SXSEvent = 0
End Function
```

- No HRESULT parameter.
- A server extension with a retval parameter is implemented as a Function.
- A server extension without a retval parameter is implemented as a Sub.

Query Function Call Syntax

A query function is called as part of an X-Query/XQuery request using its external name (see the section **Function Naming**). The parameters are passed as a list, separated by commas and enclosed in parentheses. String constants must be enclosed in quotation marks. The calling syntax is independent of the infrastructure and language.

Syntax:

```
<external function name> ( [ <parameter> { [ , <parameter> ] } ] )
```

Examples:

```
MyFunc('Input-String', 13)
MyFunc(/a/b, count(/c/d))
```

Notes:

1. String constants in server extension query functions must be enclosed in single or double quotation marks, otherwise they are interpreted as XPath expressions.
2. A string containing double quotation marks must be enclosed in single quotation marks; for example: query ('ino:name="sample"').

Conversion of Query Function Parameters

As shown in the example in the section **Query Function Call Syntax**, parameters can be passed as constants or expressions. These expressions again can contain query functions: built-in functions or server extension functions.

First, expressions are evaluated, then they are converted into the data type required by the function if possible, and finally they are passed to the function. If it is not possible to convert an expression, a runtime error occurs.

Parameters are converted according to the rules defined by the **W3C specification for the conversion of query function parameters**.

For example, if the result of a query expression, a node set, is passed as a parameter of the data type string, this is done as if the built-in query function `string()` were applied to the query expression. This returns the string content of the node set's first XML node. If the result of the same query expression were passed as a parameter of the data type XML Object, the result would be a completely different value, namely the whole node set in its string representation.

Caution: If the quotation marks are omitted for a string constant, the Tamino Server attempts to evaluate the string as a query expression. This can produce erroneous results that are very difficult to interpret.

Even return values are converted if possible. Special attention must be paid to the data type XML Object. For this data type a syntactically correct string representation of a node set must be returned. There is one exception: a simple string can also be processed if it does not contain XML tags or angle brackets.

Query Functions in X-Query/XQuery

Query functions can appear at different locations within an X-Query expression. The data type of a query function's return value must be valid at the place where the query function is located within the query expression. The different possible locations are discussed in the following:

- **Query Functions at Root Level**
- **Query Functions in Filters**

Query Functions at Root Level

This is as if a query function were entered instead of an XML query.

Examples:

```
MySubString('abcdefg',2,4)
MyFunc('Hello',/a/b,12)
```

At such a location, the function is evaluated once and the return value is inserted into the Tamino response document. The return value can have any of the data types specified in the table in the section **Query Function Parameters and Data Types**.

If a parameter is a query expression, the expression is evaluated and the resulting node set is passed as a parameter (after conversion, if necessary).

In the second example, nodes “b” in the node set of all documents of document type “a” are passed to MyFunc as the second parameter.

Query Functions in Filters

Example:

```
/a/b[name~=MyFunc('Hello', c/d,4)]
```

For such a call, the data type of the function's return value must be convertible into the type required by the filter expression.

The function is called as often as required by the number of nodes found by the query /a/b. The return value is inserted into the filter expression to be evaluated there.

A query expression as parameter is evaluated relative to the context.

Let `/a/b` in the example above result in a node set of three nodes: `n1`, `n2` and `n3`. The function is called three times. As the second parameter, Tamino passes each time a part of the node set `/a/b/c/d`: at the first call the part lying under `n1`, at the second call the part lying under `n2` and at the third call the part under `n3`. After each function call the return value of MyFunc is compared with `name` of the current node, and the result decides whether it will be part of the result document or not.

If a query expression with an absolute path as the parameter specification is evaluated, the whole of the document which contains the current node from whence the function was called is scanned.

Example:

```
/a/b[firstname~=MyFunc('Hello', /a/b/c/d,4)]
```

The MyFunc function is called as often as elements `b` of document type `a` are available. For each single function call, only the current document `a` is considered when evaluating the second function parameter.

Trigger Functions

XML documents or elements contained in XML documents can be associated with a function which is executed using the sub-node's content, but independent of mapping. Moreover, the result of the function can influence the result of the request, but it may not alter the document's structure or content. There are three types of trigger functions:

- For storage or processing: the **Insert Trigger**,

  which is executed after the validation phase during processing;
- For updating: the **Update Trigger**,

  which is executed after the validation phase during the updating of an XML document;
- For deletion: the **Delete Trigger**,

  which is executed when an XML document is to be deleted from the database.

Note: If an X-Machine `_process` command specifying the `ino:id` or `ino:docname` of an existing document is used in order to replace the existing document, Tamino uses the update trigger twice rather than using a delete trigger followed by an insert trigger. This is indicated below for the situation *The document update changes a node "A"* under the ***Execution*** heading of the ***Update Trigger*** section.

Multiple triggers can be contained in a single server extension. Depending on the action to be performed, you can include a complete set of trigger functions (one insert trigger, one update trigger and one delete trigger) in a server extension, but this is not mandatory.

Execution

Triggers can only be executed if they have been associated with schemas after installing the server extension package in a Tamino database. Insert, update and delete triggers that propagate errors by using the exception mechanism (as described in the section *Exceptions*) cause the current request to fail and thus prevent the whole document from being successfully stored, retrieved or deleted. The sequence of execution when multiple triggers are used is undefined and can vary according to the request content, optimization or Tamino version. Nevertheless, the trigger is called only if the request has been successfully executed so far. Subsequent request execution may fail and the server extension's event function is notified. See the section *Tamino Server Extension Examples* for information about sample mapping programs.

Insert Trigger

The Tamino Server calls trigger functions when an XML document with a schema containing a trigger set to “Action/onInsert” is to be stored by the Tamino Server.

The part of a document that is associated with the trigger is passed to the server extension function, along with administrative information.

An insert trigger function has four input parameters whose data types are as follows:

Position	Meaning	XML Type	Direct	Java
1	collection	xs:string	sxdstring	string
2	doctype	xs:string	sxdstring	string
3	ino:id	xs:string	sxdstring	string
4	XML Object	ino:XML-OBJ	sxdstring	string

The `collection`, the `doctype` and the `ino:id` of the document are passed for information to the server extension function. The information may not be persistent, as the subsequent request execution may be erroneous and the document may not be stored.

The `XML Object` represents the (part of the) document to be processed and therefore contains its string representation.

An insert trigger function may be used to verify the respective part of the document or to act upon its content.

Execution

An insert trigger is only executed if the document node contains a value or if there is a default value. If an optional node is not available or the document node is empty and there is no default value, the trigger function is not executed. Insert triggers are executed before any mapping takes place. If a default value is defined, a map-in function is called.

Update Trigger

The Tamino Server calls an update trigger function when an XML document with a schema containing a trigger set to "Action/Update" (update trigger) is to be updated by the Tamino Server. An update trigger function has five input parameters, whose data types are as follows:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	collection	xs:string	sxdstring	string
2 / in	doctype	xs:string	sxdstring	string
3 / in	ino:id	xs:string	sxdstring	string
4 / in	XML Object old state	ino:XML-OBJ	sxdstring	string
5 / in	XML Object new state	ino:XML-OBJ	sxdstring	string

The `collection`, the `doctype` and the `ino:id` of the document are passed for information to the server extension function in every calling context. This is the information that was used to retrieve the part of the document. The `XML Object old state` represents the (part of the) document before the update and contains its string representation. The `XML Object new state` represents the (part of the) document after the update and contains its string representation. Update trigger functions may be used to perform some action based upon the updating of a document.

Execution

There are different update situations, which influence the execution behavior of the trigger and the parameters passed to it. Let there be a node `A` in the schema, which is related to an update trigger. Let this node `A` contain sub-documents.

1. The document is updated in a sub-document of `A` but `A` itself remains unchanged.

   The update trigger is executed for each altered `A`, with "XML Object old state" and "XML Object new state" representing the node `A` and its sub-document before and after the update.

2. The document update removes a node `A`.

   The update trigger is executed once for each removed `A`, with "XML Object old state" representing the node `A` before its removal and "XML Object new state" being NULL.

3. The document update creates a node `A`.

   The update trigger is executed once for each inserted `A`, with "XML Object old state" being NULL and "XML Object new state" representing the node `A` after its insertion.

4. The document update changes a node `A`.

   The update trigger is executed twice for each altered `A`. The first time, "XML Object old state" represents the node `A` before the update and `XML Object new state` is NULL. The second time, "XML Object old state" is NULL and "XML Object new state" represents the node `A` after the update.

This behavior reflects the flexibility of XML update, which allows updating documents in the above mentioned situations for one or more nodes in one request.

Delete Trigger

Delete triggers are called when an XML document is to be deleted from the Tamino Server.

The Tamino Server calls trigger functions when an XML document with a schema containing a trigger set to “Action/onDelete” is to be deleted by the Tamino Server.

The part of a document that is associated with the trigger is passed to the server extension function, along with administrative information.

A delete trigger function has the same four parameters as an insert trigger function. The data types are as follows:

Position	Meaning	XML Type	Direct	Java
1	collection	xs:string	sxdstring	string
2	doctype	xs:string	sxdstring	string
3	ino:id	xs:string	sxdstring	string
4	XML Object	ino:XML-OBJ	sxdstring	string

The `collection`, `doctype` and `ino:id` of the document are passed for information to the server extension function. This information is used to retrieve the part of the document for deletion.

The `XML Object` represents the (part of the) document to be deleted and contains its string representation.

Delete trigger functions may be used to validate the deletion of a document or to act upon the content to be removed.

Execution

Delete triggers are only executed if the document node contains a value. If an optional node is not available or if the document node is empty, the trigger function is not executed.

See the section *Tamino Server Extension Examples* for information about programming examples.

Shadow Functions

A shadow function is used to create a shadow document in Tamino for a non-XML document that can be stored in Tamino or in an external location. If `tsd:storeShadowOnly` is defined in the schema, the shadow document is stored in Tamino but the non-XML document is not copied into Tamino.

A shadow document is an XML document, generated using user-written logic, that can contain information such as metadata, index values and other generated values for the corresponding non-XML document. The purpose of the shadow document is to store information that can be used for queries that would not be possible on the original non-XML document.

The schema that defines the non-XML document must also contain appropriate schema statements to indicate that a shadow function will be used. See the discussion on Using Shadow Functions in the section *Storing Non-XML Objects in Tamino* in the *Tamino XML Schema User Guide* for details. When Tamino processes the non-XML document, the instance is passed as a parameter of binary or textual data type to the shadow function, which builds the shadow document. This shadow document is then passed to the X-Machine for further inserting and index processing. The result is stored as a *shadow* of the original non-XML document.

The general rules covering the behavior of shadow documents and the original non-XML documents are as follows:

- When an XQuery or X-Query query is issued, Tamino examines the shadow documents and not the non-XML documents.
- If the non-XML document is stored in Tamino and a delete command is issued to delete the document, Tamino deletes both the shadow document and the non-XML document.
- When an existing non-XML document is replaced (i.e. old document removed and replaced by a new one), the corresponding shadow document is deleted and a new shadow document is created.
- Shadow documents cannot be updated or deleted by using `xquery update` or `xquery delete` commands.
- If plain URL addressing is used for retrieval, Tamino returns the non-XML document and not the shadow document. If the non-XML document is stored outside Tamino, an HTTP status 404 (file not found) is returned. For details of plain URL addressing, see the section *Requests using Plain URL Addressing* in the *X-Machine Programming* guide).

A shadow function can be implemented in any programming language that is supported by X-Tension, using any supported infrastructure. Like mapping functions and trigger functions, a shadow function has a defined signature and a restricted execution context (here it is limited to insertion of non-XML documents).

A shadow function can be part of any server extension's object. Its execution is part of the request, transaction and session processing, like the execution of a mapping function or trigger function.

The non-XML document that corresponds to the shadow function can contain pure text or "real" binary content, so the shadow function type comes with two kinds of signatures, which are called according to the actual content of the document. Tamino determines whether a non-XML document should be treated as a text document or a binary document solely on the basis of the document's media type (see the section *Media Type Requirements* in the *X-Machine Programming* guide for details). Thus, for example, Tamino stores base64-coded binary objects as text or binary objects depending on the media type setting.

Signature for onTextInsert Functions

Shadow functions related to the node of onTextInsert are defined with the following parameter structure:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	collection	xs:string	sxdstring	string
2 / in	doctype	xs:string	sxdstring	string
3 / in	doctypeURL	xs:string	sxdstring	string
4 / in	ino:id	xs:string	sxdstring	string
5 / in	ino:docname	xs:string	sxdstring	string
6 / in	non-XML-Text	xs:string	sxdstring	string
7 / retval	Shadow-XML	ino:XML-OBJ	sxdstring	string

Signature for onBinaryInsert Functions

Shadow functions related to the node of onBinaryInsert are defined with the following parameter structure:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	collection	xs:string	sxdstring	string
2 / in	doctype	xs:string	sxdstring	string
3 / in	doctypeURL	xs:string	sxdstring	string
4 / in	ino:id	xs:string	sxdstring	string
5 / in	ino:docname	xs:string	sxdstring	string
6 / in	non-XML-Binary	ino:Binary	sxdstring	Byte Array
7 / retval	Shadow-XML	ino:XML-OBJ	sxdstring	string

When Tamino processes a non-XML document, it accepts any type of input (text in any format, binary data of any kind) from the calling context as an input parameter and delivers it without change to the shadow function. The XML schema data types xs:hexBinary and xs:base64Binary as defined in **xmlschema-2** do not serve this purpose because they use encoding (hex or base64) with conversion procedures. Therefore the Tamino data type ino:Binary has been designed. Its values are handed to the shadow function (binary type) in its language dependent representation.

Mapping Functions

XML documents or elements of them can be stored native, i.e. “as-is”, in the Tamino database, or using the X-Node.

In addition, a more dynamic kind of mapping can be performed using Tamino server extensions: the “Mapping to Function”. XML documents or parts of them are passed to or taken from a user-defined mapping function. In this case, mapping functions must be used to handle the sub-document completely.

There are three types of mapping functions:

- One for storage or processing: the **map-in function**,

  which is executed when mapping the verified document;
- one for retrieval or composition: the **map-out function**,

  which is executed when retrieving the document nodes; and
- one for deletion: the **map-delete function**,

  which is executed when an XML document is to be deleted from the database.

Multiple mapping functions can be contained in a single server extension. Usually you should include a complete set of mapping functions (one map-in function, one map-out function and one delete function) in a server extension.

Execution

Mapping functions can only be executed if they have been associated with schemas after installing the server extension package in a Tamino database. Mapping server extension functions which propagate errors cause the current request to fail, and thus prevent the whole document from being successfully stored, retrieved or deleted.

The sequence of execution of mapping functions and server extensions is undefined and can vary according to the request content, optimization or Tamino version. Nevertheless, the mapping function is called only if the request has been successfully executed so far. Subsequent request execution may fail and the server extension's event function is notified.

See the section *Tamino Server Extension Examples* for information about sample mapping programs.

Map-In Functions

Map-in functions are called by the Tamino Server when an XML document is stored by the Tamino Server. The map-in function is called for the part of the document where the associated schema storage type is set to "Map XTension" and the `onProcess` property is defined.

The part of the document that is mapped to a server extension function is passed to the server extension function, along with administrative information.

A map-in function has three or four parameters, whose data types are as follows:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	Object ID	xs:int	sxdint	int
2 / in	Element ID	xs:int	sxdint	int
3 / in	XML Object	ino:XML-OBJ	sxdstring	String
4 (optional) / out	Node Info	xs:string	sxdstring *	StringBuffer

The `Object ID` and `Element ID` are used to uniquely identify the data that is passed to the server extension function in the specific database.

The `XML Object` represents the (part of the) document to be processed and already validated. It contains the string representation of a well-formed XML document or fragment.

The optional fourth parameter can be used by the server extension function to store arbitrary information in the Tamino database, which can only be accessed in a successive map-in or map-delete server extension function.

Map-in functions may be used to store those parts of an XML document that are mapped to a server extension function outside of the Tamino database. The `Object ID` and `Element ID` are database-specific keys, given from the Tamino Server. No semantic is associated with them.

The `Node Info` can be used to conveniently store customer-specific information about where the data is stored, for example the name of a file in which the data is stored, or an SQL command that delivers the data as a result, if it is stored in a third party relational database.

The `Node Info` is transparent to the Tamino Server and is only interpreted by corresponding map-out and delete functions if specified by the Tamino server extension developer. The fourth parameter is optional. However, if the function does not support this parameter, there is no way in which node information can be stored or used.

Note: Remember to define a corresponding map-out function for each map-in function you develop, otherwise you will not be able to retrieve the data stored with your map-in function.

Considerations for key building:

Although `Object ID` and `Element ID` uniquely identify the subdocument within a single Tamino Server, you should take the following into consideration when designing the key for storing, retrieving and deleting the mapped data:

- `Object ID` and `Element ID` may change when the database is unloaded/loaded, for example when the Tamino Data Loader is used for mass loading and unloading of data. The `Node Info` is invariant under massload.
- `Object ID`, `Element ID` and `Node Info` remain unchanged when the database undergoes backup or restore/recovery processes.
- Different databases may use the same internal keys.
- The database name may change as the result of a Tamino Administrator `rename database` command.

You should consider defining a key independent of `Object ID`, `Element ID` and database-name and keeping it in the `Node Info` parameter of the map-in function, where it can be used in all other cases of mapping.

Execution

Map-in functions are only executed if the document node contains a value. If an optional node is not available or the document node is empty, the map-in function is not executed.

See the section *Tamino Server Extension Examples* for information about programming examples.

Map-Out Functions

Map-out functions are called by the Tamino Server when an XML document is to be retrieved from the Tamino Server. The map-out function is called for the part of the document where the associated schema storage type is set to "Map XTension" and the `onCompose` property is defined.

A map-out function has three or four parameters, whose data types are as follows:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	Object ID	xs:int	sxdint	int
2 / in	Element ID	xs:int	sxdint	int
3 / out	XML Object	ino:XML-OBJ	sxdstring *	StringBuffer
4 (optional) / in/out	Node Info	xs:string	sxdstring *	StringBuffer

The `Object ID` and the `Element ID` are used to uniquely identify the data to be retrieved by the server extension function.

The `XML Object` represents the (part of the) document to be retrieved and must contain the string representation of an XML document or its parts that are valid against the schema used. The passed

string must be either the valid string representation of a node set or a character string of printable characters containing neither "<" nor ">".

The optional fourth parameter contains the node information that was passed to the Tamino Server by a previously called map-in function. As an in-out parameter it may be changed by the map-out function. If the previously called map-in function did not pass any node information, this parameter is an empty string (of length zero). The fourth parameter is optional. However, if the function does not support this parameter, there is no way in which node information can be accessed.

A map-out function can be used to retrieve those parts of an XML document stored outside of the Tamino database by a previous map-in function.

Execution

The map-out function is executed only if the corresponding sub-node was previously inserted. Consequently, no optional sub-nodes can be newly generated by means of map-out functions. The map-out output is validated against the schema definition. The document composition may fail if the validation fails.

See the section *Tamino Server Extension Examples* for information about programming examples.

Map-Delete Functions

Map-delete functions are called by the Tamino Server when an XML document is to be deleted from the Tamino database. The map-delete function is called for the part of the document where the associated schema storage type is set to "Map XTension" and the `onDelete` property is defined.

A map-delete function has two or three parameters, whose data types are as follows:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	Object ID	xs:int	in	int
2 / in	Element ID	xs:int	in	int
3 (optional) / in	Node Info	xs:string	in	String

The `Object ID` and the `Element ID` are used to uniquely identify the data to be deleted.

The optional third parameter contains the additional node information that had previously been passed from the map-in or map-out function to the Tamino Server.

Map-delete functions can be used to delete those parts of an XML document stored outside of the Tamino database by a previous map-in function.

See the section *Tamino Server Extension Examples* for information about programming examples.

Execution

The delete function is executed only if the corresponding sub-node was previously inserted.

See the section *Tamino Server Extension Examples* for information about programming examples.

Initial Server Extension Functions

As Tamino server extension objects are created implicitly by the Tamino Server, it is not possible to use constructors that pass parameters when an extension function is called. But it is often useful to initialize member variables with different values. If a server extension function – regardless of type – needs configuration information or the establishing of an initial environment, this can be achieved by executing an initial server extension function. Default values are associated with the parameters of initial functions when they are installed. These default values can be changed by the Tamino administrator using the Tamino Manager.

It is up to the Tamino server extension developer and the execution environment to decide which initialization should be performed, but mostly it will be used to set member variables, establish connections to external data-sources, etc. These initial settings remain valid unless modified by another server extension function until the session in which the server extension is executed ends. We recommend using server extension event functions to do the housekeeping for initialized actions if necessary.

The init function can have any number of input parameters, but no output or return parameter. The data types of the parameters are as follows:

XML Type	Direct/C++	Java
xs:boolean	sxdbool	boolean
xs:string	sxdstring	string
xs:double	sxddouble	double
xs:float	sxdfloat	float
xs:int	sxdint	int
ino:XML-OBJ	sxdstring	string

Each parameter definition must have an associated default value that matches the corresponding type definition, otherwise a runtime error may occur. The X-Tension development tools (X-Tension Builder, the X-Tension Object Analyzer and the X-Tension Administration) allow the definition of these default values but do not enforce them. The values also can be set after the installation of the server extension by using the server extension administration features of the Tamino Manager.

Execution

Each server extension can contain at most one init function, which is implicitly called immediately before the first execution of any of its query, mapping or trigger functions, but after the objects constructor, if any. Subsequently, no parameter value is passed from the calling Tamino context, but all values are taken by default as denoted during development, analysis or modification after installation. In case of an error the init function should throw an error, which is propagated as the result of the current Tamino Server request.

Server Event Functions

If a server extension function is called repeatedly within a command request, transaction or user session, the same instance of the server extension object is used (see the section *Calling Tamino Server Extensions*).

For complex programming, it is therefore necessary to be able to react to Tamino Server events such as commit or rollback, end of request or end of session, for example in order to undo modifications initiated by a function call when a rollback was performed. Server event functions serve this purpose.

Each server extension object can contain at most one server event function.

A server event function has two parameters, whose data types are as follows:

Position / Direction	Meaning	XML Type	Direct	Java
1 / in	State	xs:int	sxdint	int
2 / return value	Return Code	xs:int	sxdint*	int

The first parameter indicates the event that happened, as listed in the following table. The second parameter contains a return code. In general, the return code should be set to zero, indicating success.

The following table shows the Tamino Server events that can and should be handled by a server event function:

Tamino Server Event	Value
XML_REQUEST_END	0
XML_COMMIT	1
XML_ROLLBACK	2
XML_CONNECTION_END	3
XML_SUB_COMMIT	4
XML_SUB_ROLLBACK	5

Direct

These events are defined as enum types in the file sxdinc.h.

Java

These events are defined as constants in the Java-based interface for server extensions, SXSJBase.

Execution

Generally, the Server event function of a server extension object is called if at least one server extension function of type trigger, mapping or query of this object was called at least once during the corresponding period. The same event can be forwarded to several server extensions. In detail:

XML_SUB_COMMIT / XML_SUB_ROLLBACK

These events only occur in user sessions. At the end of a command, they are reported to all server extension objects from which a function was called while processing the command. These commands can occur several times within a request (see the section *Order of Execution of Commands* in the *X-Machine Programming* documentation for related information). They are used in a user session like subtransactions.

XML_COMMIT / XML_SUB_ROLLBACK

These events occur in user sessions when the user sends a commit or rollback command to the Tamino Server. At the end of a transaction they are reported to all server extension objects from which an arbitrary function was called within the same transaction.

In anonymous sessions each command is treated as a transaction of its own and closed with auto-commit or auto-rollback. At the end of each command, an XML_COMMIT or XML_ROLLBACK is reported to all server extension objects from which an arbitrary function was called within the same command.

XML_REQUEST_END

This event is reported to all server extension objects from which an arbitrary function was called within the same request. (Only “usage” requests are considered. A user-issued commit or rollback, which from the Tamino Server's point of view is also a request, does not lead to the XML_REQUEST_END event; it only results in XML_COMMIT or XML_ROLLBACK. The same is true for a session's end.)

XML_CONNECTION_END

This event is reported to all server extension objects from which an arbitrary function was called during the corresponding session, i.e. to all server extension objects that were initialized in the context of this session.

Server event functions are restricted in functionality:

Notes:

1. Server event functions can only call **system callback** or HTTP callback functions.
2. As Server event functions are processed after termination of the XML request which makes the server extension function calls, they cannot call callbacks implying database operations. Therefore XML callbacks and ODBC callbacks cannot be used within Server event functions. If an XML callback is used in a Server event function, a runtime error occurs. If an ODBC callback is used, an error message is output.
3. Exceptions should not be thrown in Server event functions. They are caught, but have no effect.

See the section ***Tamino Server Extension Examples*** for information about programming examples.

4 Administrating Tamino Server Extensions

Tamino server extensions can be administrated from the Tamino Manager. The administration tasks related to a server extension are performed on a running database. This means that a database must have been created and started before you can perform the following tasks:

Installing a Tamino Server Extension

To install a Tamino server extension, you need a package file of type SXP. This file must have been created using one of the following:

- **Tamino X-Tension Builder;**
- **X-Tension Object Analyzer.**

Any number of Tamino server extensions can be installed in a given Tamino database. One Tamino server extension can be installed in several Tamino databases.

During the installation process, entries are made in the Tamino file system and the Tamino database system.

If you want to install a Java-based server extension that calls classes from one or more external JAR or ZIP files, you must modify the public or private Java classpath to include the respective JAR or ZIP files. The public classpath can be set for all Java-based server extensions of all databases on the file system, whereas the private classpath applies to a single Java-based server extension installed in a database only. For more information about administration and the calling hierarchy, see the section *Dialog for Setting the Private Classpath*.

Note: Mapping, trigger and shadow functions can only be executed if you associate them with schemas after installing the server extension package. For a detailed description of schema definition, see the documentation of the Tamino Schema Editor.

To install a Tamino Server extension

1 Start the Tamino Manager.

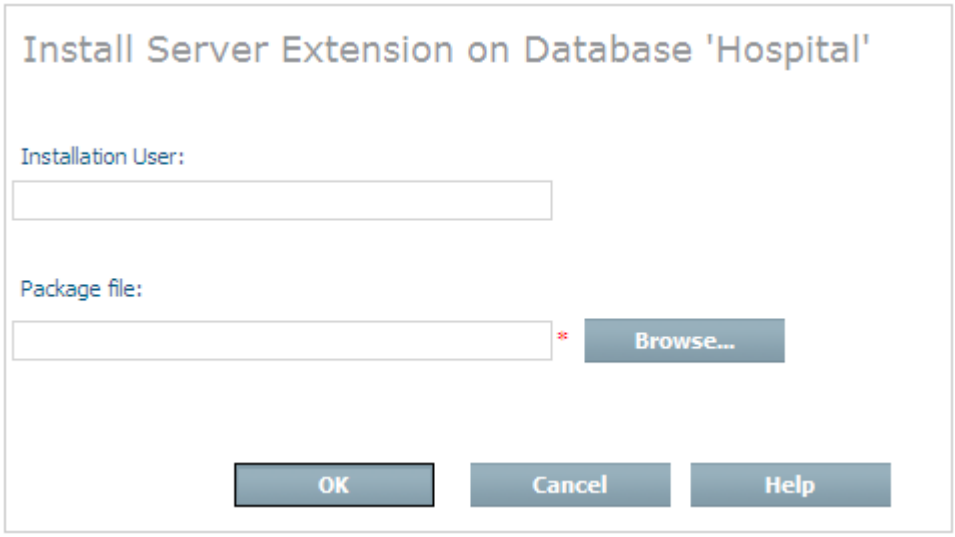
Expand the **Databases** object under the **Tamino** node.

Start and expand the database into which you want to install the server extension package.

2 Select the **Server Extensions** object.

3 From the context menu, choose **Install Extension**.

4 The **Install Server Extension on Database** page appears:



Enter your name in the **Installation User** text box.

Enter the full path name (including the file name) of the server extension package (*your-file.sxp*) to be installed in the **Package file** text field; alternatively, browse to the directory that contains the server extension package to be installed and select the package.

Choose **OK** to install the server extension.

The **Job Monitor** page appears, informing you about the success or failure of the installation.

After installation, the name of the server extension is included under the expanded Server Extensions object in the Tamino Manager. Select the server extension to display a summary of information, or expand it and select individual functions for more details. You can also expand individual functions to view details of the associated parameters.

If you try to install a server extension object that is already installed in the database, you will receive an error message.

If you are installing a Java server extension that needs a private classpath, a page appears, which allows you to define the private classpath. The private classpath can be set for a single server extension in a database. The value initially supplied in this field is the configured classpath, if this value exists.

You can either enter the private classpath directly in the **Private Classpath** field, or choose the **Edit** button to enter a dialog for constructing the private classpath. This is described in the section *Dialog for Setting the Private Classpath* below.

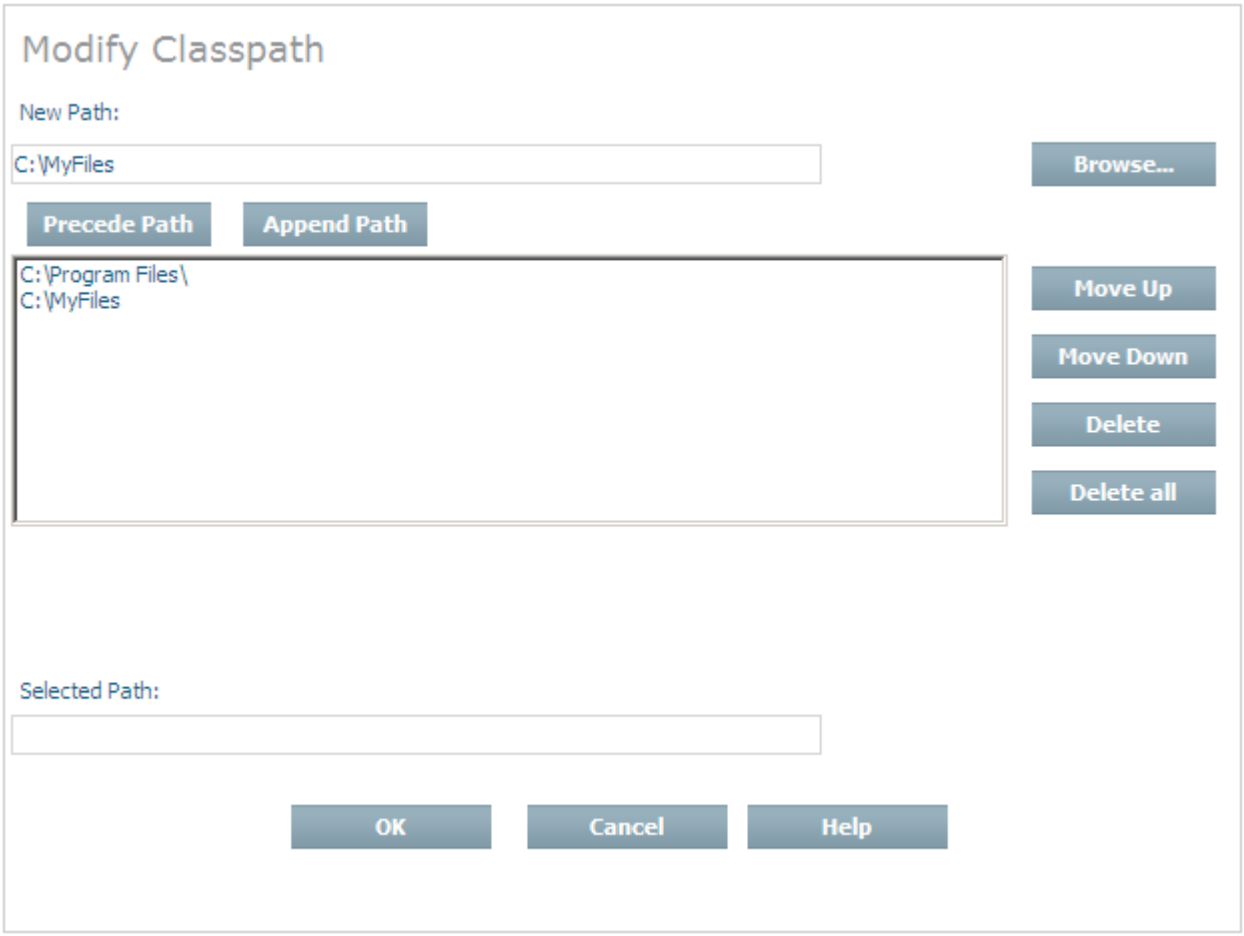
Then choose **OK**.

Private and Public Classpaths for Java

The public classpath can be set for all Java server extensions of all databases on the file system, whereas the private classpath relates to a single Java server extension installed in a database only. During execution of a Java server extension the private classpath precedes the public classpath, which overrules the standard classpath setting of the environment variable `CLASSPATH`. Using the private Java classpath, the public Java classpath and the environment variable `CLASSPATH` leads to the search path of the mentioned hierarchy. As the X-Tension class loader is derived from the standard class loader, classes, JAR files and ZIP files are recognized. The following abbreviation allows you to avoid lengthy path expressions for JAR files contained in the same directory in private or public Java classpaths: to add all JAR files in a directory (e.g. D:\X\Y) to the private or public classpath, you can enter either the directory name alone or the directory name followed by *.jar into the classpath (e.g. D:\X\Y*.jar). Other expressions with wildcard characters are not supported. Please observe that the search hierarchy is not defined when using the "*.jar" expression. Different classes with the same fully qualified class name should therefore be avoided.

Dialog for Setting the Private Classpath

Several dialogs contain an **Edit** button that opens a subdialog that allows you to construct the private classpath. When you choose this button, the following dialog appears (the example assumes that the currently-defined classpath is "C:\Program Files\;C:\MyFiles"):



When the classpath comprises several paths, as in this example, each path is displayed on a separate line. This makes it easier to add and remove paths and to change the order of the existing paths. A path can point to a directory or to a JAR file within a directory.

The existing classpath can be extended by adding one or more paths to the beginning or end of the existing classpath definition.

To add a new path to the existing classpath definition

1 Enter a new path in the **New Path** field. You can either enter the path value directly or use the **Browse...** button to search for the required path;

2 Choose the **Precede Path** button to add the new path to the start of the classpath, or the **Append Path** button to add the new path to the end of the classpath.

The order in which the paths appear in the classpath can be changed as follows:

To change the position of a path within the classpath

1 Select the path to be moved in the list of paths displayed;

2 Choose the **Move Up** button to move the path one position towards the start of the classpath, or choose the **Move Down** button to move the path one position towards the end of the classpath.

The classpath can be shortened by removing paths from the classpath.

To remove a path from the existing classpath definition

1 Select the path to be removed from the list of paths displayed;

2 Choose the **Delete** button.

You can use the **Delete all** button to clear the classpath. If you use this button, it is not necessary to select the paths before deleting them.

The path names in the list of paths are not scrollable, so long path names are truncated in the display. When you select a path in the list of paths, it is also displayed in the **Selected Path** field. This field is horizontally scrollable, so if a path name is long, you can scroll in this field to see its full value.

Specifying Java Options

When starting Tamino with Java X-Tension usage switched on, options can be specified to influence the behavior of the JVM. These user-specified options can be used, for example, to configure the size of the JVM or to debug Java server extensions.

The options are specified as a character string, the first character of which serves as a delimiter for the options that follow. For example, entering the string

```
$-Xms64m$-Xmx64m
```

would result in the two parameters `-Xms64m` and `-Xmx64m` being passed to the JVM at startup.

To modify the Java options

1 Start the Tamino Manager.

Select the database for which you want to add Java options;

2 Select the **Properties** object, then select the **X-Tension** properties group;

3 Choose **Modify**. This opens the dialog for setting or modifying the X-Tension properties;

4 Enter a new value for the X-Tension Java options in the **Configured Value** column. Placing the cursor over the question-mark icon in the **Details** column displays a tool tip that includes a brief description of the property. If you choose the button **All Defaults**, all of the properties are reset to their default values/settings;

5 Choose OK to apply the changes;

6 The new value(s) will take effect when the server is started or restarted.

The user options that have been specified are displayed in the job log when the Tamino server is started. If they are invalid, a warning message displaying the options is issued and the Tamino server starts with the internal default options.

Caution: Certain JVM parameter settings may have an adverse effect on the performance of the Tamino Server and/or Java server extensions.

Setting the Java classpath using this property may not work or may have unexpected side-effects. Please use the appropriate methods documented in the sections *X-Tension Tools – Modifying the Public Java Classpath* and *Administrating Tamino Server Extensions – Dialog for Setting the Private Classpath* to set the appropriate global or private classpath.

Modifying the Properties of a Server Extension

If you choose the **Modify Extension** button, the **Modify Server Extension** page appears, in which you can modify the execution mode of a Direct-based server extension as well as the external names of functions.

To modify the properties of a server extension object

1 Start the Tamino Manager.

Expand the **Databases** object.

Start and expand the database containing the server extension you want to modify;

2 Expand the **Server Extensions** object;

3 Select the server extension that you want to modify;

4 From the context menu choose **Modify Extension** ;

5 The **Modify Server Extension** page appears;

Some of the fields are read-only, others such as **External Name** can be modified. Make your changes as required.

For Java server extensions, the field **Private Classpath** is offered, which allows you to modify the value of the private classpath. See the section *Dialog for Setting the Private Classpath* above for details;

6 Choose **OK** to modify the server extension.

Modifying the Properties of a Server Extension Function

You can change the external name of a server extension query function in the **Modify Function** View.

To modify the server extension function

1 Start the Tamino Manager.

Expand the **Databases** object.

Start and expand the database which contains the server extension with the function you want to modify;

2 Expand the **Server Extensions** object;

3 Expand the server extension which contains the function you want to modify;

4 Select the server extension function you want to modify;

5 From the context menu, choose **Modify Function**;

6 The **Modify Function** page appears.

Enter the new name for the function in the **External Name** text box.

Choose **OK** to modify the name.

The **Job Monitor** page appears, informing you about the success or failure of the modification.

Upgrading a Server Extension

A server extension is upgraded when the **server extension version information** is increased (in Java this is the `sxsVersion` variable), and then the server extension is installed using the normal installation procedure.

The following rules should be taken into account when modifying an existing server extension:

- All changes made to existing interfaces should be done with care, since for example applications or schemas may have dependencies on these interfaces .
- Changing things such as default values should be done using the administration interface and not by creating a new version of a server extension.
- Creating new interface methods or functions is the most appropriate form of modifying a server extension package.

- If a server extension is installed in multiple Tamino databases of differing versions, the use of new functionality or callbacks is not allowed, because the older versions of Tamino do not understand these callbacks.
- Upgrading a server extension that is shared by multiple databases means implicitly upgrading the extension for all databases. Databases that are not active at that time will be upgraded the next time they are started.
- Upgrading may only take place if a server extension is not currently in use (across all databases that share this extension).

Hint: When upgrading a server extension, we strongly advise you not to change the *Install.xml*, class, JAR, DLL or shared library files in the server extension install directory, since the next server restart or recovery from backup may overwrite them. Instead, you should install new versions using the System Management Hub server extension installation procedures following the rules above.

Uninstalling a Server Extension

Uninstalling a Tamino server extension means deleting the database entries and file system entries, including any associated files.

When you start the uninstallation of a server extension, the server extension is locked for all Tamino sessions started thereafter. A server extension that is in use cannot be uninstalled, therefore the installation process waits until all sessions that are using the corresponding server extension have terminated; then the uninstallation is performed. If the sessions have not terminated within 150 seconds, the uninstallation is aborted with an error message.

Prerequisites for Mapping, Trigger and Shadow Functions

A server extension containing server extension functions that are referenced by schemas should only be uninstalled after all schema references to these functions have been removed. If this server extension function is not available but there are still references in the schema, a runtime error occurs.

To uninstall a Tamino server extension

1 Close any applications (e.g. Microsoft Windows Explorer) that are accessing server extension files or their directories, because otherwise inconsistent data may remain;

2 Start the Tamino Manager.

Expand the **Databases** object.

Start and expand the database from which you want to uninstall a server extension;

3 Expand the **Server Extensions** object;

4 Select the server extension that you want to uninstall;

5 From the context menu, choose **Uninstall Extension**;

6 The **Uninstall Server Extension from Database** page appears, showing related schemas if they exist. If you choose to uninstall a server extension with related schemas, the schemas will become invalid;

Choose **OK** to uninstall the server extension.

The **Job Monitor** page appears, informing you about the success or failure of the uninstallation.

Switch for the Tamino Server Extension Trace

To activate or deactivate server extension tracing

1 Start the Tamino Manager.

Start and select the database;

2 Select the Server Extensions object;

3 From the context menu, choose the **X-Tension Settings** button;

4 The **Specify X-Tension Settings** page appears;

To activate tracing, check the box; to deactivate tracing, remove the check mark. Choose **OK** to activate or deactivate tracing. It is deactivated by default whenever the Server is restarted, to prevent unintended tracing. Trace output is written to the collection *ino:SXS-Trace* and can be queried or deleted there. For detailed information, see the section ***Tracing Tamino Server Extensions***.

5 Calling Tamino Server Extensions

When the Tamino Server calls a server extension function for the first time in a given XML session, an instance of the server extension object that implements the server extension function is initialized. If the same server extension function is called again within the same XML session, this same instance of the server extension object is used. When an XML session ends, all server extension objects that were created by that session are destroyed. In the case of sessionless XML requests (running in so-called anonymous sessions), the life cycle of the server extension object is the XML transaction instead of the XML session.

If a second XML session calls the same server extension function, the call is handled in a separate instance of the server extension object. This means that any data held as a member variable of the server extension object is protected from concurrent calls, whereas global variables would be shared by both instances. Data that is held as a member variable of a server extension object is kept for the duration of an XML session (or XML transaction in the case of anonymous sessions) and can be used by a subsequent call of the same server extension function.

6 Failure of Tamino Server Extension Functions

If a server extension fails, the entire XML request of which this is a part fails. The response document generated by Tamino contains information about the nature of the error. If the request was made in an anonymous session, the current transaction is implicitly rolled back. If the function was called from a request that was part of a user's session, the user must evaluate the response document and decide whether or not to perform a rollback.

A server extension function fails:

- when it throws an exception (see the section ***Exceptions*** for more information);
- when programming errors in server extension code lead to exceptions (for example, memory violations for a C++ or Direct server extension or OutOfBoundsExceptions for a Java-based server extension);
- in Direct or Java-specific error situations;
- if the server extension function attempts to write to the Tamino database using an **XML callback** and this attempt fails, the call of the whole server extension function fails. The response document contains either a detailed error message coded by the server extension developer or a general error message referring to the error situation.

7 Building a Tamino Server Extension Package

This document explains how you can build server extension packages based on **Direct** and **Java-based** infrastructures using the Tamino X-Tension Builder. Server extensions can be written in C++, Java or any language that supports dynamic loading of object libraries.

Programming Languages and Development Tools

The following combinations of programming languages and infrastructures are supported by Tamino Tools and/or described in this documentation:

Infrastructures / Programming Languages	Direct	Java
Direct/C++	X-Tension Builder	-
Java	-	X-Tension Builder

Building a Direct Infrastructure-Based Tamino Server Extension Package

Server extensions written in C++ or any programming language that supports the dynamic loading of object libraries can be executed using the Direct infrastructure of Tamino X-Tension. The **X-Tension Builder** supports the development of server extension packages based on this infrastructure.

Building a Java-Based Tamino Server Extension Package

The **Tamino X-Tension Builder** supports the development of Java-based server extension packages on Microsoft Windows and UNIX platforms. The infrastructure is established by means of a JVM attached to the Tamino Server process. The usage of Java is restricted to avoid harmful influences on the Tamino Server.

Using the X-Tension Builder

This section tells you how to build a server extension package. The X-Tension Builder supports developing server extensions using the Direct and Java-based infrastructures. For information about what Tamino server extensions are and how they can be developed, see the ***Introduction*** and the sections ***Tamino Server Extension Functions*** and ***Developing Tamino Server Extensions***. The individual steps related to building a server extension package - which is what you need when you want to install a server extension into a database - are described in the following sections:

- Creating a Server Extension Project

Creating a Server Extension Project

The Tamino X-Tension Builder is used to create a server extension project. The tool automatically generates files containing all of the code required by the infrastructure-specific parts of the program and for communication with the Tamino Server. You are responsible for implementing the functionality. This section explains how to use the tool to set up a new project.

To create a server extension project

1 Create the directory or folder where the source files for your project are to be stored. Use different directories for different projects!

2 Microsoft Windows:

- Start the Tamino X-Tension Builder from the **Tamino** program group under the Windows **Start** menu.

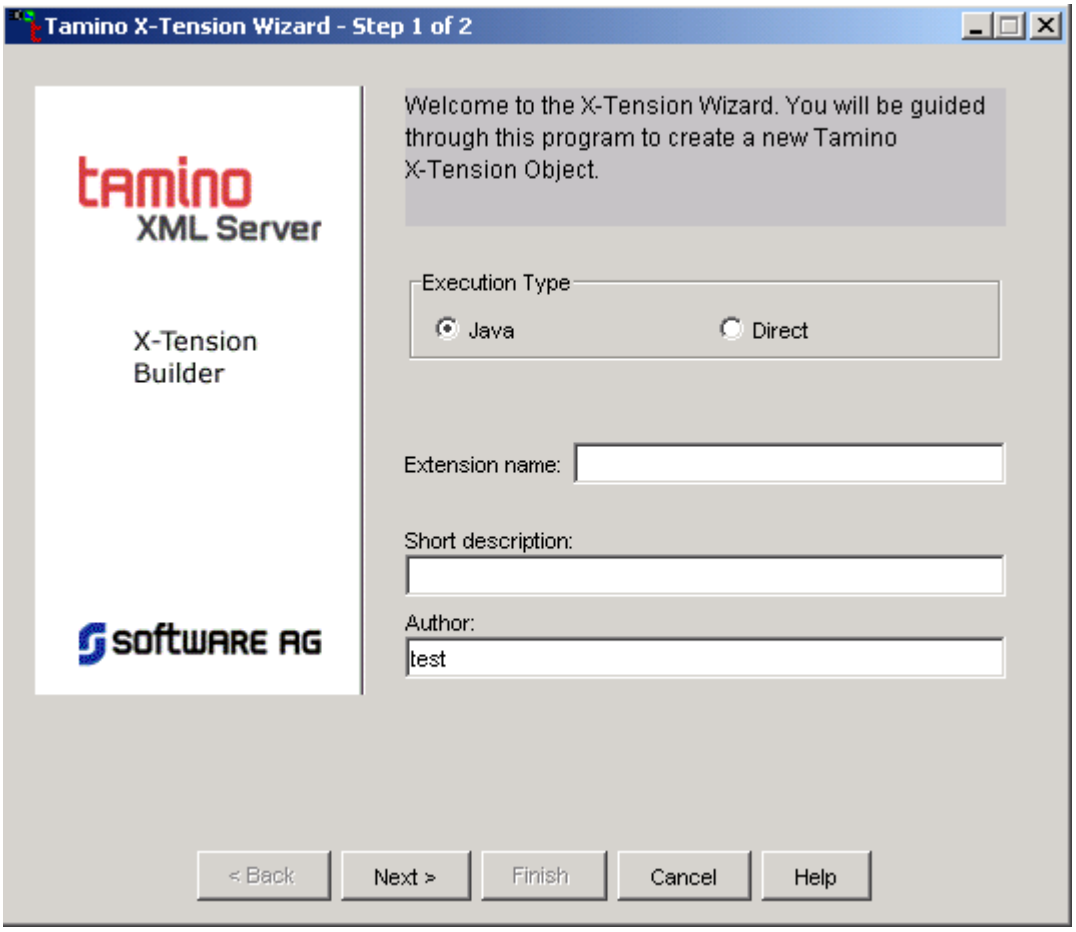
UNIX:

- Call the script *inosxbuilder.sh* from the command line.

3 The Tamino X-Tension Builder appears and displays an empty screen.

Choose **New** from the **File** menu to open a new project.

4 The first of two dialog boxes appears:



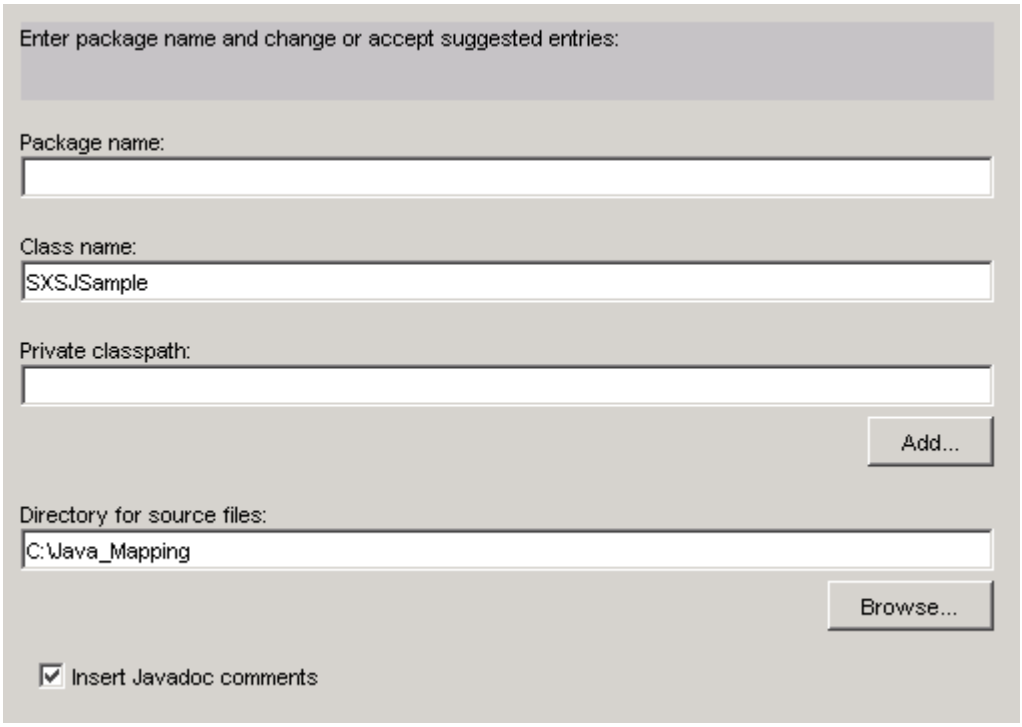
5 You can create either a Direct/C++ server extension or a Java-based server extension. Select the appropriate radio button. The following description shows the results of choosing the Java-based infrastructure for the server extension; the results of choosing the Direct infrastructure are similar.

Enter a name for the server extension. This name is used to reference the object within Tamino.

The text box for the **short description** is filled in automatically. You can edit this text, and also the text in the **Author** box.

Choose **Next**.

6 A second dialog box appears:



Enter a package name for the Java class. To ensure unique naming, Software AG recommends that you use your domain name in the package name.

The class name is filled in automatically. You can edit this text.

Enter the private classpath (if needed).

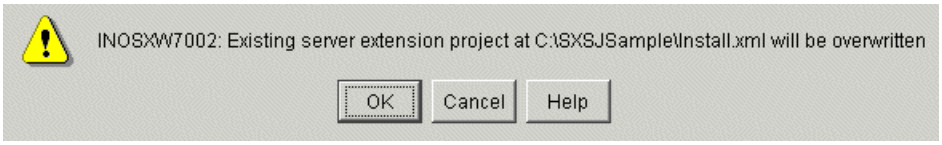
Enter the directory where the source files are to be stored, or choose **Browse...** to select a drive and directory.

If you keep the check mark in the **Insert Javadoc comments** box, you can generate Javadoc comments for your new server extension object and any server extension function that you include in the project after compiling the class file.

When you have made all desired changes, choose **Finish** to create the new project.

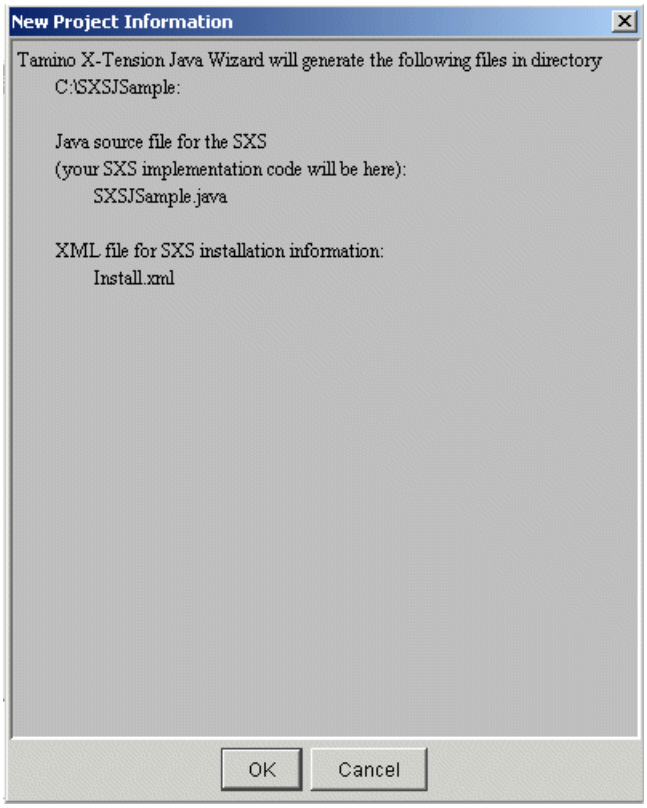
Note:

- If the specified directory had already been used for a previous project, the following message is displayed:



If you select **OK** in the message box, the *Install.xml* file will be overwritten. If you choose the **Cancel** button, you are returned to the previous dialog, where you can choose a different directory.

7 A list of files that will be created is displayed:



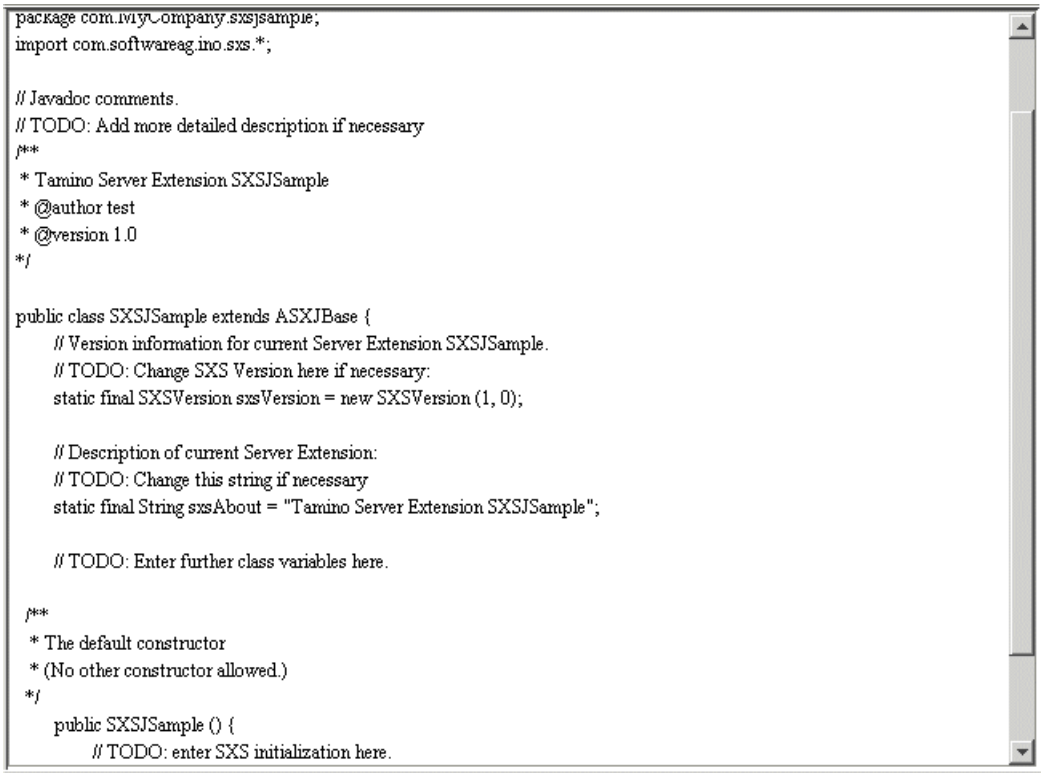
These two skeleton files are of special interest to the server extension developer:

*`your-project`**.java***
: The source file to which the signature of a server extension function is added using the **Add Function** menu item from the **Project** menu and into which the developer's function code has to be inserted;

Install.xml
: The installation file which is required to install the finished server extension package. The XML server uses this file to put information about the server extension object into the repository.

Choose **OK** to create the files.

8 The source file *your-project.java* is opened in the editing window of the Tamino X-Tension Builder:



```
package com.mycompany.sxsjsample;
import com.softwareag.ino.sxs.*;

// Javadoc comments.
// TODO: Add more detailed description if necessary
/**
 * Tamino Server Extension SXSJSample
 * @author test
 * @version 1.0
*/

public class SXSJSample extends ASXJBase {
      // Version information for current Server Extension SXSJSample.
      // TODO: Change SXS Version here if necessary:
      static final SXSVersion sxsVersion = new SXSVersion (1, 0);

      // Description of current Server Extension:
      // TODO: Change this string if necessary
      static final String sxsAbout = "Tamino Server Extension SXSJSample";

      // TODO: Enter further class variables here.

 /**
  * The default constructor
  * (No other constructor allowed.)
 */
      public SXSJSample () {
            // TODO: enter SXS initialization here.
```

At various points in the generated framework code, you will find comments of the form:

```
// TODO: ...
```

Add your code after these comment lines.

Save the file.

9 This file can be edited as described in the section **Adding a Server Extension Function to an Existing Server Extension Project**.

Opening an Existing Server Extension Project

You can open a previously-generated server extension project with the Tamino X-Tension Builder at any time to edit and manipulate the source file (add the signature of a server extension function and the function code, compile and pack the server extension object) as long as the server extension has not yet been installed in one or more databases.

To open an existing server extension project

1 Microsoft Windows:

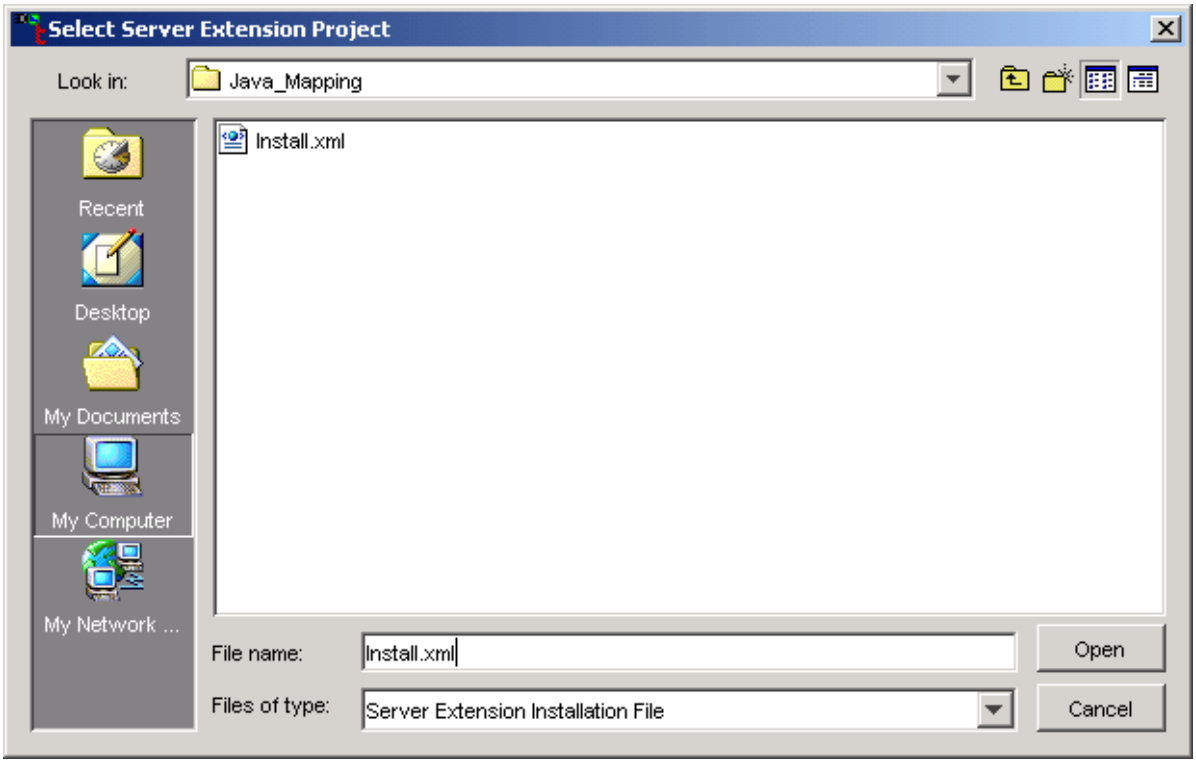
- Start the Tamino X-Tension Builder from the Windows **Start** menu.

UNIX:

- Call the script *inosxbuilder.sh* from the command line.

2 Choose **Open** from the **File** menu of the Tamino X-Tension Builder.

3 The **Select Server Extension Project** dialog box appears:



Browse to the drive and directory that contains the *Install.xml* file for the project.

Select the file and choose **Open**.

4 The *your-project.java* source file for this server extension project appears in the Tamino X-Tension Builder's editing window.

You can view all source files for the project by selecting the filename in the tree displayed in the left-hand frame of the X-Tension Builder.

Note: You should not change the *Install.xml* file. If you do change it, for example if you change parameter names, you must also make these changes in the *your-project.java* source file.

Note: It is not possible to install different versions of a server extension at the same time. If you want to make changes to the function code of a server extension that is installed in a database without uninstalling it, **create a new project** specifying a different name for the server extension, add the modified function code, and compile and pack the project as usual with the Tamino X-Tension Builder.

Adding a Server Extension Function to an Existing Server Extension Project

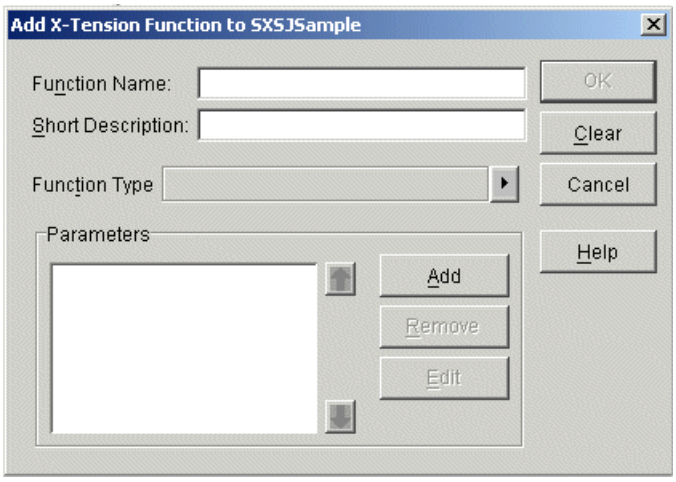
You can add any number of map-in, map-out, map-delete, trigger, query and shadow functions, but only one init or event function to a server extension object.

We recommend including a complete set of mapping functions (one map-in, one map-out and one delete function) or trigger functions in a server extension object. Implement an init function to perform initializing actions prior to first function execution. Usually you will also need an event function to react to commit or rollback events.

To add a server extension function to an existing server extension project

1 With the source file (*your-project.java*) open in the editing window, choose **Add Function...** from the **Project** menu.

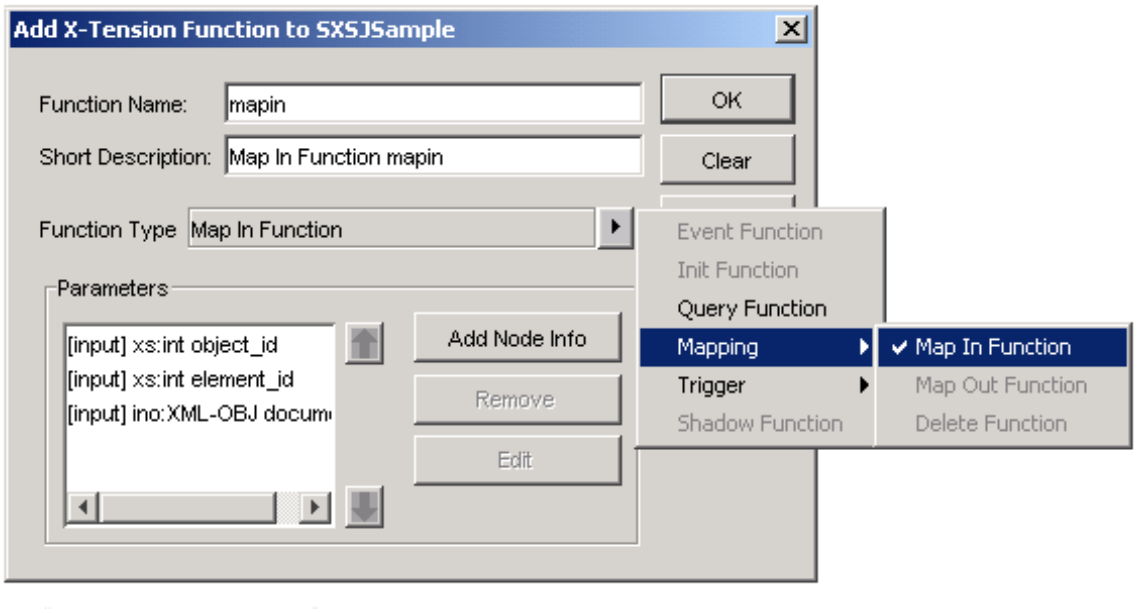
2 The **Add X-Tension Function** dialog box appears:



Enter the name of the server extension function in the **Function Name** text box.

Either:

- Select the **Function Type** drop-down list box for the server extension function.



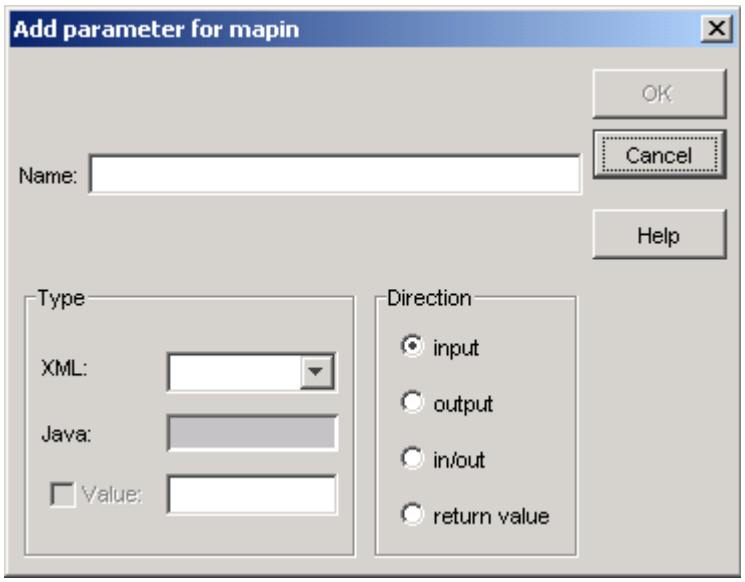
If you choose **Map In Function**, **Map Out function** or **Delete Function**, parameters with default names and types are generated automatically, and the **Add** button changes to **Add Node Info**, which can be used to add the nodeinfo parameter to the function.

Choose **OK** to add the map-in function to your server extension.

Or:

- Choose **Add**.

  The **Add parameter** dialog box appears:



Enter a name for the parameter you want to define and select the XML type and attribute.

Choose **OK**.

The **Add parameter** dialog box appears again, allowing you to add all the parameters that are required for the type of server extension function you want to define.

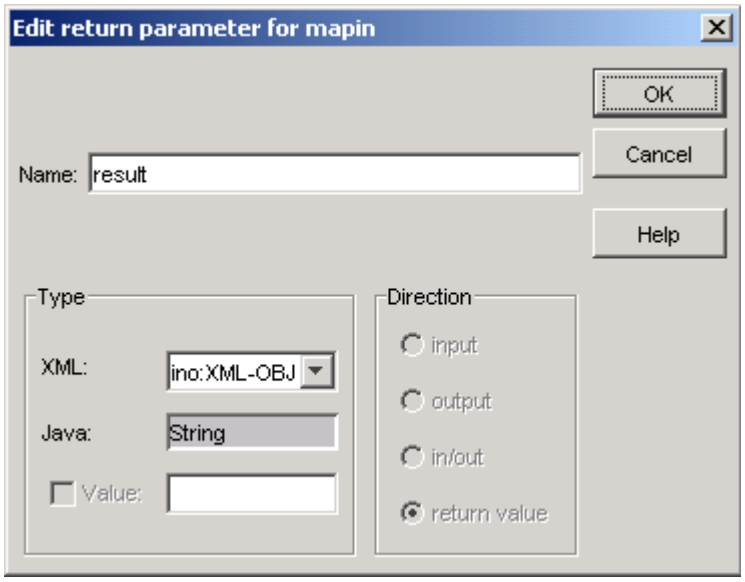
When you have specified all of your parameters, choose the **Function Type** (only function types with suitable parameter lists are enabled). Missing mandatory parameters are added.

Optional parameters can be reordered. Select the parameter you want to move to another position, and the up and down arrows are enabled if reordering is allowed for that parameter.

If you select one of the displayed parameters, the **Remove** and **Edit** buttons are enabled.

Query function selected as Function Type

- If you choose **Query Function** from the **Function Type** drop-down list box, the **Edit return parameter** dialog box appears, where you can define a return value:



Select the appropriate type (XML document or character string) of the value to be returned by your query function from the **XML** drop-down list box. (If you select "XML-OBJ", the XML document processor parses the result. If the result is not a well-formed XML document, the parser fails with the error message INOXPE8702 `Invalid document prolog`.)

Choose **OK**.

The **Add Extension Function** dialog box appears again with the **Add** button changed to **Add Optional**, allowing you to generate an arbitrary number of optional parameters for a query function. All of these parameters must be input parameters.

Optional parameters can be reordered. Select the parameter you want to move to another position, and the up and down arrows are enabled if reordering is allowed for that parameter.

If you select one of the displayed parameters, the **Remove** and **Edit** buttons are enabled.

Choose **OK** to add the function's signature to the source file (*your-project.java*) and the *Install.xml* file.

3 The following predefined code is displayed:

```
//     SXSJSample.java: Implementation of Server Extension SXSJSample:

package com.MyCompany.sxsjsample;
import com.softwareag.ino.sxs.*;

// Javadoc comments.
// TODO: Add more detailed description if necessary
/**
 * Tamino Server Extension SXSJSample
 * @author test
 * @version 1.0
*/

public class SXSJSample extends ASXJBase {
      // Version information for current Server Extension SXSJSample.
      // TODO: Change SXS Version here if necessary:
      static final SXSVersion sxsVersion = new SXSVersion (1, 0);

      // Description of current Server Extension:
      // TODO: Change this string if necessary
      static final String sxsAbout = "Tamino Server Extension SXSJSample";

      // TODO: Enter further class variables here.
```

```
 * The default constructor
 * (No other constructor allowed.)
*/
    public SXSJSample () {
        // TODO: enter SXS initialization here.
    }

    // Description of Server Extension Function mapin
    // TODO: Change description if necessary
    static final String mapinAbout = "Map In Function mapin";

    // Javadoc comments.
    // TODO: enter more detailed parameter descriptions below
    /**
     * Map In Function mapin
     * @param object_id parameter
     * @param element_id parameter
     * @param document parameter
    */
    public void mapin (int object_id, int element_id, String document)
    {
        // TODO: Add your SXF implementation here.
        // To delete mapin, remove it here and from Install.xml.
    }
}
```

4 At various points in the generated framework code, you will find comments of the form:

```
// TODO: ...
```

Add your code after these comment lines.

5 Save the file.

Compiling the Server Extension Class File

When you have made all entries to the source file, the class file for the server extension must be compiled before the server extension can be packed. The class file is compiled using the packaging information that is specified in the second dialog box for creating a new project.

The server extension class file can only be compiled if you have installed a Java compiler (by default this is installed as part of the standard Tamino installation procedure) and set the system environment variable `path` to point to it.

To compile the server extension class file

1 With the source file (*`your-project`.java*) open in the editing window:

Either:

- Choose **Compile** from the **Project** menu.

Or:

- Choose **Copy Compile Command** from the **Project** menu.

  This copies the compile command to the clipboard. You can now paste it at a command prompt (so-called "DOS box"), where it can be executed.

2 The source file appears, showing messages about the compilation process in the Result Frame.

- If errors occurred, correct your function code and recompile the file.

Generating Javadoc Comments for a Server Extension Class File

If the **Insert Javadoc comments** check box was selected in the second dialog box for creating a new project, you can generate Javadoc comments for the server extension class file and include them in the server extension package.

To generate Javadoc comments

1 With the source file (*`your-project`.java*) open in the editing window, choose **Call Javadoc** from the **Project** menu.

2 A list of the created files appears:

```
Calling Javadoc...
Loading source file C:\SxsJSample\SXSJSample.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Building index for all classes...
Generating C:\SxsJSample\allclasses-frame.html...
Generating C:\SxsJSample\index.html...
Generating C:\SxsJSample\packages.html...
Generating C:\SxsJSample\com\MyCompany\sxsjsample\SXSJSample.html...
Generating C:\SxsJSample\serialized-form.html...
Generating C:\SxsJSample\package-list...
Generating C:\SxsJSample\stylesheet.css...
Finished Calling Javadoc...
```

The HTML file that is named after your project and stored in the directory where the class file is located is of special interest to the server extension developer. The other files are stored in the project's main directory.

Close the box.

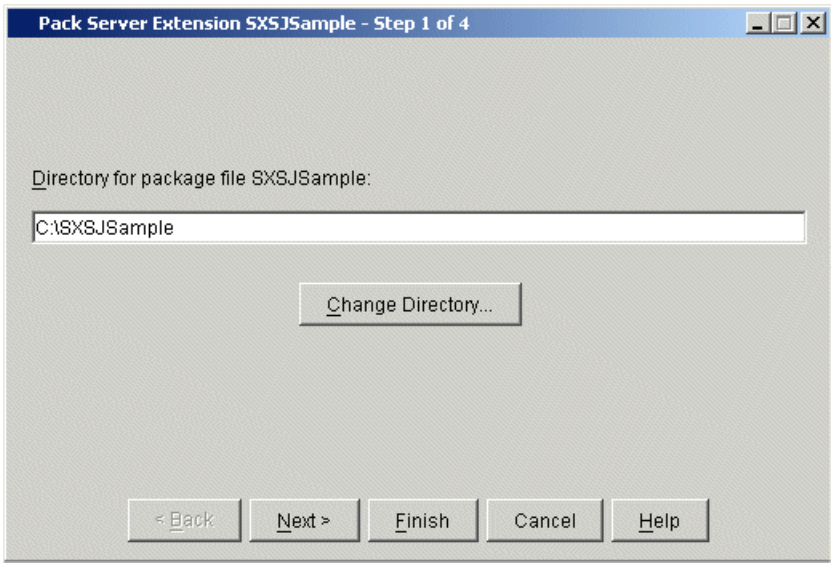
Packaging a Server Extension

When the server extension class file has been successfully compiled and, if desired, the Javadoc comments have been generated, the server extension package can be created.

To create a server extension package

1 With the source file (*your-project.java*) open in the editing window, choose **Pack Server Extension** from the **Project** menu.

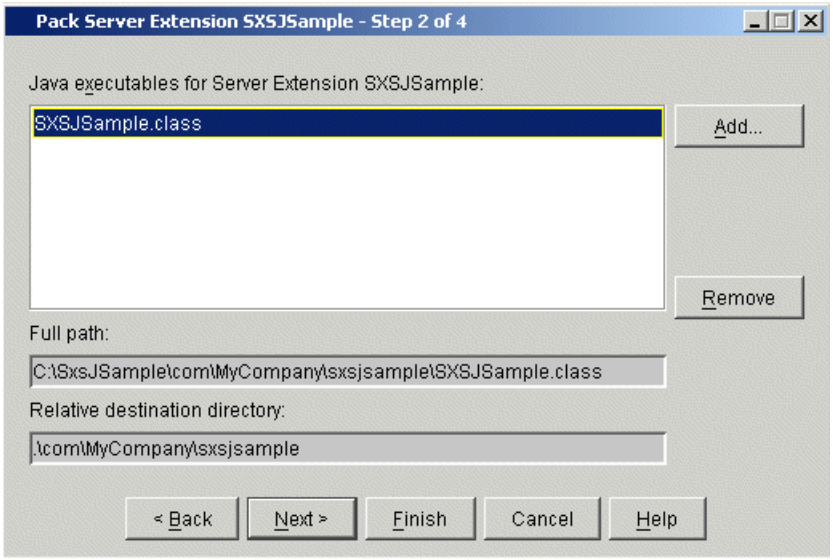
2 The first **Pack Server Extension** dialog box appears:



Choose **Change Directory...** if you want to create the package in a directory other than the default.

Choose **Next** to continue, or **Finish** to complete the wizard using default values.

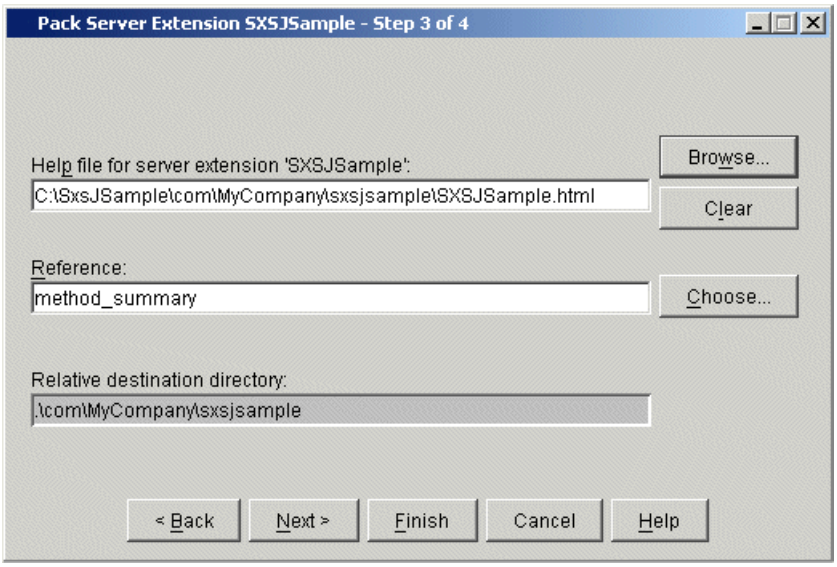
3 The second **Pack Server Extension** dialog box appears, displaying the Java executables available for the specified server extension object:



Select one of the displayed executables and choose **Add...** to include it into the package. If you want to exclude an executable you have specified from the package to be created, select it and choose **Remove**. **Full path** and **Relative destination directory** are displayed automatically. You cannot edit them.

Choose **Next** to continue, or **Finish** to complete the wizard using default values.

4 The third **Pack Server Extension** dialog box appears, displaying a help file (HTM, HTML, XML and TXT) for the server extension object.



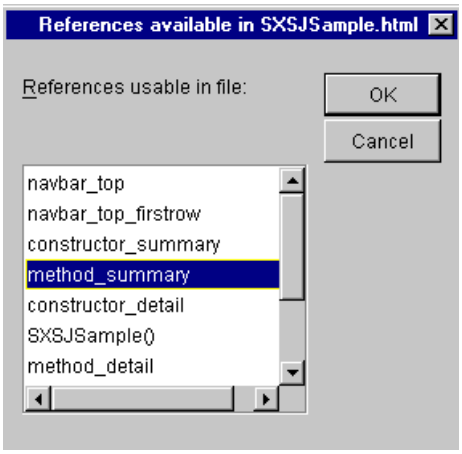
The relative destination directory of the selected help file is displayed.

If you want to exclude a help file you have specified from the package to be created, choose **Clear**.

The **Reference** text field can be used to specify a bookmark in the help file. In this case, the help file will start at the position indicated by the bookmark.

Choose the **Choose...** button.

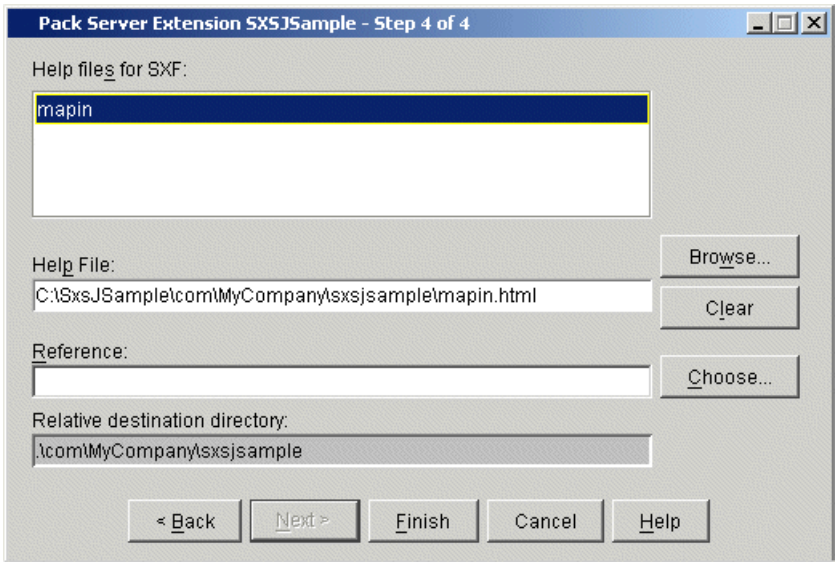
The **References available** list box appears showing the bookmarks available in the current help file.



Select a bookmark.

Choose **OK**.

5 The fourth **Pack Server Extension** dialog box appears:



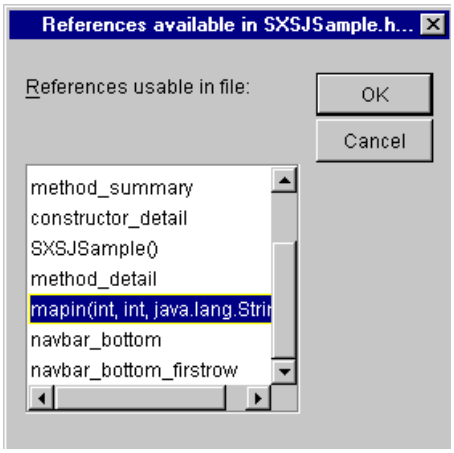
A help file for the individual server extension function is displayed, together with the relative destination directory.

Choose **Browse...** to select a different help file than the one suggested in the dialog box, or choose **Clear** to remove a help file.

The **Reference** text field can be used to specify a bookmark in the help file. In this case, the help file will start at the position indicated by the bookmark.

Choose the **Choose...** button.

The **References available** list box appears, showing the bookmarks available in the current help file:



Select a bookmark.

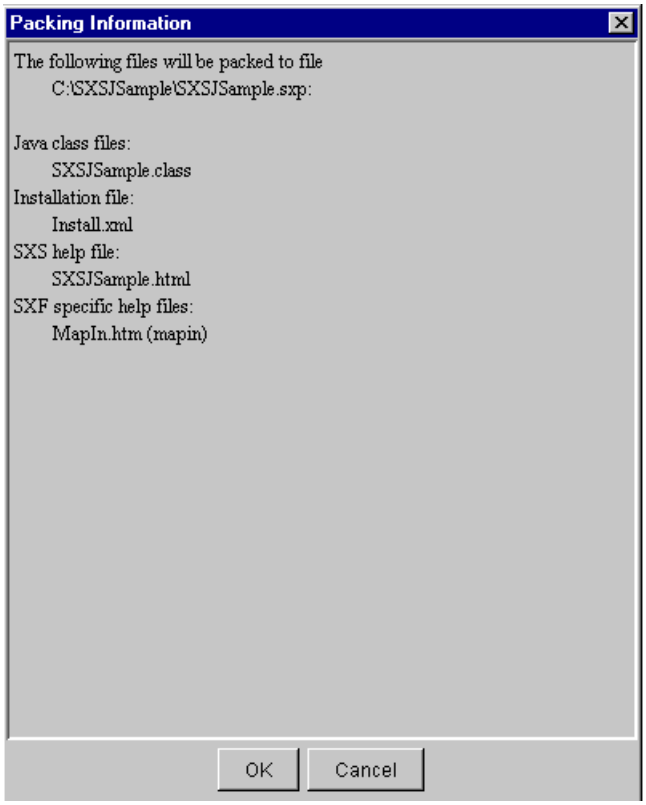
Choose **OK**.

Choose **Finish** to complete the wizard actions.

Note: The complete help information may be contained in one file. This file can be specified repeatedly; for example, for the server extension object (as described in **step 3**) and for any server extension function that you select in the current page.

If you want to include all files generated using the **Call Javadoc** item in the **Project** menu, select all the files, one after the other, for all functions or one of the functions contained in the selected executable.

6 A list of the files to be included in the package is displayed:



Choose **OK** to create the server extension package.

The package file (*your-project.sxp*) can now be installed as described in the section **Installing a Tamino Server Extension**.

Using Direct Infrastructure

Server extensions written in C++ or any other programming language that supports dynamic loading of object libraries can be executed using Tamino X-Tension's Direct infrastructure. The X-Tension Builder supports the development of Direct/C++ server extension packages.

To create a server extension project

- Open a project in the X-Tension Builder. On the first page there is a button **Direct**. Select this button and you will be guided through the whole building procedure using the Direct infrastructure. The procedure is very similar to that described for the **X-Tension Builder**.

8 Developing Tamino Server Extensions

This section describes how server extensions can be implemented. The Direct and Java infrastructures can be used. Direct extensions are written in C++; Java extensions are of course written in Java. First decide on the infrastructure that you want to use before choosing the programming language. The tool to be used depends on this decision. For related information about Tamino server extensions and how they can be developed, see the ***Introduction*** and the section ***Tamino Server Extension Functions***.

As described in the section **Building a Tamino Server Extension Package**, it is easy to create Tamino server extensions using the tools provided with Tamino. Basic knowledge of C++ or Java is sufficient to create simple but effective server extensions. We recommend using these tools whenever you develop a Tamino server extension.

Note: The use of external synchronization mechanisms within a server extension that is not under control of the Tamino Server / X-Machine can lead to problems. Such problems can include deadlocks that cannot be resolved by the X-Machine's deadlock resolution capabilities; this may lead to requests hanging up, either completely or until a transaction timeout occurs. Therefore, for example, the Java code in a server extension should not use Java-based locking.

The following sections, which address some specific programming issues, should help you write valid function code.

Constructors and Destructors

Constructors and destructors that are required for object-oriented programming can also be used in Tamino server extensions. As Tamino server extension objects are created implicitly by the Tamino Server, if one of its extension functions is called, it is not possible to use constructors that pass parameters, with the exception of the callback handle in the Direct infrastructure (see below).

Direct

You can use common C++ constructors. Note that two types of constructors can be called when a Direct server extension is initialized:

- Constructors that can accept one single parameter of type SXDCHandle. This type of constructor must be available if callbacks are to be used in the Direct server extension's code. The SXDCHandle parameter is the callback handle that must be used for all callbacks. We recommend storing this handle in a member variable of the server extension's class:

```
CMyExtension::CMyExtension (SXDCHandle h)
        : m_callback (h)
{
       // more initializations here
}
```

- The default constructor (i.e. a constructor without parameters). In this case no callbacks are available.

Java

You can use the common Java constructors. Note that the default constructor (this is one without any parameters) is called when a Java server extension is initialized. If you do not define any constructors in your server extension class, this default constructor is implicitly provided by Java. As stated above, constructors with parameters cannot be called by the Tamino Server; instead, you should use an initial function. If for some reason your server extension class contains one or more constructors with one or more parameters, you must also define a default constructor without any parameters; otherwise an error occurs when initializing the class.

Initial Functions

The initial function can be generated automatically if the Tamino server extension development tools are used. You can find detailed information in the section **Initial Server Extension Functions**.

String and Memory Handling

Direct

For all kinds of string parameters (including the XML type “ino:XML-OBJ”) and byte arrays, the special data type “sxdstring” must be used. For return and output values, sxdstrings must be allocated by the callback functions:

sxdstring AllocString (wchar_t*)
Input can be a zero-terminated ASCII or wide character string or a string constant.

sxdstring AllocStringByte (void*, sxdulong)
Used for binary data.

Strings allocated by these functions (including input parameters) may be freed by

- FreeString

However, it is not mandatory to use FreeString, as the memory used for a Direct server extension is released when a server extension object is deleted.

For reading operations, you may treat strings of type `sxdstring` like common wide character strings (see the examples).

There are two more string handling functions:

- sxdulong StringLen (sxdstring)
- sxdulong StringByteLen (sxdstring)

to get the length of string variables.

Examples:

Get the callback handle in the constructor (see above):

```
MyExtension::MyExtension (SXDCHandle h)
        :m_callback (h)
{ ... }
```

Have a query function:

```
sxdstring MyExtension:::myQuery(sxdstring input)
{
    // allocate a wide character string myBuffer
    swprintf(myBuffer, L "This was the input: %", input);

    sxdstring retval = m_callback->AllocString(myBuffer);
    free (myBuffer);

    return retval;
}
```

Caution: Do not assign a string constant to an output parameter of type sxdstring*, for example `*nodeinfo=L"abc";` instead, use `*nodeinfo = m_callback -> AllocString (L"abc");`

Java

All Java string objects used as “in” and “out” parameters are initialized by the Tamino Server and can be used in the server extension function. You must write code in the server extension function to initialize any java.lang.String that is used as a return value.

As is usual in Java, it is not necessary to free or delete objects.

Specialities of Different Infrastructure/Language Combinations

The information in this section is broken down as follows:

Restrictions for Java Server Extensions

To ensure the stability of the Tamino Server and the Java Virtual Machine, the following operations are locked for Java-based server extensions. Any attempt to call them leads to a security exception:

- Exit the Java Virtual Machine;
- Modify or stop Java system threads;
- Redefine the standard I/O streams;
- Define your own security manager;
- Modify Java security configuration aspects such as policy, security provider, identifications, and private keys.

As the Java Virtual Machine runs in batch mode, the standard I/O streams cannot be used in server extensions. Output has no effect. To avoid long waits when trying to read from standard input, this stream is set to null, so any attempt to read from it immediately leads to a null pointer exception.

Callbacks

Arbitrary database calls to arbitrary databases, including the Tamino Server, can be implemented in a server extension function via the standard database interfaces. In general, however, it is more desirable to access databases from within the same session and transaction context as the Tamino Server session in which the server extension function is called. Databases can be accessed in this way by using Tamino callbacks. Tamino callbacks are interfaces to the Tamino Server that can be used in a server extension function. They provide access to the various databases that can be attached by the Tamino Server, and also to system information available in the running Tamino Server.

The following categories of callbacks are supported: **XML callbacks**, **ODBC callbacks** and **system callbacks**.

For information about runtime errors that can result from callback calls and the handling of these errors, see the section ***Callback Error Handling***.

XML Callbacks

XML callbacks support X-Machine command requests to the running Tamino database. The callback functions and their corresponding X-Machine commands are listed in the following table:

Callback Function	X-Machine Command
SxsXMLAdmin	_admin
SxsXMLDefine	_define
SxsXMLDelete	_delete
SxsXMLProcess	_process
SxsXMLUndefine	_undefine
SxsXMLXql	_xql
SxsXMLXQuery	_xquery

These X-Machine commands are described in the section *Requests using X-Machine Commands* of the *X-Machine Programming* documentation.

There is an additional callback function, SxsXMLGetMessage, to get messages from the Tamino Server.

The following administration functions are available in the SxsXMLAdmin callback:

- ino:DisplayIndex
- ino:Index

All other administration functions mentioned in the documentation section X-Machine Programming > Requests using X-Machine Commands > The _admin command are unavailable.

Notes:

1. When an XML callback is used, the following database commands cannot be issued: `open database session`, `close database session`, `commit` and `rollback`. These commands are implicitly handled by the Tamino Server.
2. An XML callback cannot be used within an event function.

Direct

The following functions are methods of the callback handle and must be called using `m_callback-> <function name> (<arglist>)`:

```
sxdint SxsXMLXQuery (sxdstring collection, sxdstring query, sxdresult* result);

sxdint SxsXMLXql (sxdstring collection, sxdstring query, sxdstring* response);

sxdint SxsXMLProcess (sxdstring collection, sxdstring xmlDoc, sxdstring* response);

sxdint SxsXMLDefine (sxdstring xmlSchema, sxdstring* response);

sxdint SxsXMLUndefine (sxdstring collection, sxdstring schema, sxdstring doctype, ↩
sxdstring* response);

sxdint SxsXMLDelete(sxdstring collection, sxdstring query, sxdstring* response);

sxdint SxsXMLAdmin (sxdstring command, sxdstring* response);

sxdint SxsGetDocument (sxdstring collection, sxdstring doctype, sxduint inoid, ↩
sxdresult* result);

sxdint SxsGetDocument (sxdstring collection, sxdstring doctype, sxdstring documentId, ↩
sxdresult* result);

sxdint SxsXMLGetMessage(sxdstring* msgContent);
```

Input parameters of type sxdstring must be allocated using `m_callback -> AllocString()` and freed after the call using `m_callback -> FreeString()`. The output parameters `response` and `msgContent` must be valid pointers to an sxdstring, and their contents should be null. The output parameter `result` must be a valid pointer to an sxdresult, and its contents should be valid (the constructor guarantees this).

Java Calls

```
int SxsXMLXQuery (String collection, String query, Result result);

int SxsXMLXql (String collection, String query, StringBuffer response);

int SxsXMLProcess (String collection, String xmlDoc, StringBuffer response);

int SxsXMLDefine (String xmlSchema, StringBuffer response);

int SxsXMLUndefine (String collection, String schema, String doctype, StringBuffer ↩
response);

int SxsXMLDelete (String collection, String query, StringBuffer response);
```

```
int SxsXMLAdmin (String command, StringBuffer response);

int SxsGetDocument (String collection, String doctype, int inoid, Result result);

int SxsGetDocument (String collection, String doctype, String documentID, Result ↩
result);

int SxsXMLGetMessage (StringBuffer msgContent);
```

The output parameters `response` and `msgContent` must be initialized StringBuffer objects, and their content should be the empty string. On return, they contain the response document of the XML request or the Tamino Server message.

Meaning of Parameters (For All Infrastructures)

The parameters used in the syntax descriptions above have the following meanings:

Parameter	Meaning
collection	The name of the collection upon which the callback operates. For the SxsXMLDefine function, this is the name that is assigned to the newly-created collection. For the other requests, it must be the name of a collection that already exists in the database.
xmlDoc	XML document for the command `_process`.
xmlSchema	XML schema document for the command `_define`.

The syntax of each X-Machine command is defined in the section Requests using X-Machine Commands of the section X-Machine Programming.

collection
The name of the collection upon which the callback operates. For the SxsXMLDefine function, this is the name that is assigned to the newly-created collection. For the other requests, it is the name of an existing collection.

reqContent
The valid input for this parameter can be found in the description of the appropriate X-Machine command in the *X-Machine Programming* documentation.

The Meaning of sxdresult (Direct) and Result (Java)

The output parameter of the callback functions SxsXMLXQuery and SxsGetDocument is of type sxdresult (Direct) or Result (Java).

In the case of the Direct infrastructure, `sxdresult` is a class of the form:

```
class sxdresult
{
    public:

    sxdresult();
    virtual ~sxdresult();

    /* is result a string? */
    sxdbool isString() const;

    /* is result a binary? */
    sxdbool isBinary() const;

    /* get media type of result */
    sxdstring getMediaType() const;

    /* get string value if isString is true else SXD_NULL */
    sxdstring getStringValue() const;

    /* get binary value if isBinary is true else SXD_NULL */
    sxdbinary getBinaryValue() const;
}
```

In the case of the Java infrastructure, `Result` is a class:

```
public class Result
{
    public Result();

    // Reset to an empty but valid result set.
    public void reset();

    // Get the media type of the result.
    public String getMediaType();

    // Is this a string value?
    public boolean isString();

    // Get the result value as a string.
    // @returns the string value or an empty string
    public String getStringValue();

    // Is this a binary value?
    public boolean isBinary();
```

```
    // Get the result value as a binary (byte array).
    // @returns the binary value or a zero length byte array
    public byte[] getBinaryValue();
}
```

Return Values and Output Parameters

For all infrastructures, the return values and output parameters are as follows:

A return code of 0 indicates success; anything else indicates that the call failed. For more information about errors, see the section *Callback Error Handling*.

After a successful call, the content of the output parameter `response` is as follows:

Callback function	Content of output parameter `response`
SxsXMLXQuery	The node set resulting from the specified XQuery, enclosed in the tags `<xq:result>` ... `</xq:result>` as from the Tamino response document.
SxsXMLXql	The node set resulting from the specified X-Query, enclosed in the tags `<xql:result>` ... `</xql:result>` as from the Tamino response document.
SxsXMLProcess	A node `<ino:object ino:collection="..." ino:doctype="..." ino:id="...">` containing the collection name, document type and ID of the processed XML document as from the Tamino response document.

If the return value of any callback function is nonzero and SxsGetMsgNo() reports a Tamino Server error (see **Callback Error Handling**), then the function SxsGetMsgText() should be called. The output parameter `msgContent` of this function contains a message as it would have appeared in the node `<ino:messageline>=...</ino:messageline>` of the Tamino response document.

A complete example of the usage of an XML callback including appropriate error handling can be found in the section **Callback Error Handling**.

Note that XML callbacks can be used recursively. For example, an XML document contains an element that is mapped to a server extension function that uses an XML callback to send an SxsXMLProcess() request to the Tamino Server. The resulting XML document provided by the Tamino Server also contains an element that is mapped to the same server extension function, which is called recursively. In such cases the following points should be taken into consideration:

- Any data stored in a member variable of the server extension function object or any other object created in the server extension function context can be overwritten during the recursive server extension function call, and may therefore have been changed when the XML request returns.
- It is the responsibility of the server extension function developer to prevent infinite loops. The SXS infrastructure makes some checks that attempt to prevent excessive recursion.

For SxsXMLXql()

The node set resulting from the passed XML query, enclosed in the tags `<xql:result>...</xql:result>` as known from the Tamino response document.

For SxsXMLProcess()

A node `<ino:object ino:collection="..." ino:doctype="..." ino:id="...">` containing the collection name, document type, and ID of the XML document that was just processed, as known from the Tamino response document.

If the return value of any callback function is non-zero and SxsGetMsgNo reports a Tamino Server error (see ***Callback Error Handling*** below), then the function SxsGetMesText() should be called. The out parameter `msgContent` of this function contains a message as it would have appeared in the node `<ino:messageline>=...</ino:messageline>` of the Tamino response document.

A complete example of the usage of an XML callback including appropriate error handling can be found in the section ***Callback Error Handling*** below.

The callback function SxsXMLXql() can now return namespace-clean results. This means that the results can be passed directly to any standard XML parser. This option is controlled by a parameter that can be accessed using the Tamino Manager: start the Tamino Manager, then in the tree view select **Managed Hosts > localhost > Tamino > Databases >** *database_name* **> Properties > X-Tension**. The option `X-Tension callback namespace clean` is displayed, and by choosing the command `Modify` you can display a screen in which you can change the value of this parameter from "yes" to "no" or vice versa. The default value of the `X-Tension callback namespace clean` parameter for all new databases is "yes". The default value of the `X-Tension callback namespace clean` parameter for a database that has been converted by using the Tamino Manager `Set version` command and that has server extensions installed is "no"; this ensures backward compatibility with old server extensions that add namespace declarations to the results returned by the SxsXMLXql() callback function.

Note:

XML callbacks can be used recursively. For example, an XML document contains an element that is mapped to a server extension function which uses an XML callback to send an SxsXMLProcess() request to the Tamino Server. The resultant XML document provided by the Tamino Server also contains an element that is mapped to the same server extension function, which is called recursively. The following points should be taken into consideration in such cases:

- Any data stored in a member variable of the server extension function object or any other object created in the server extension function context can be overwritten during the recursive server extension function call, and may therefore have been changed when the XML request returns.
- It is the responsibility of the server extension function developer to prevent infinite loops.

Examples

The following section shows some examples in the various infrastructures. Error handling has been omitted for the sake of clarity.

Direct

```
sxdstring collection = m_callback->AllocString(L"MyCollection");
sxdstring query = m_callback->AllocString(L"/MyDoctype/Node_A[anyId='5']");
sxdstring response = SXD_NULL;

int ret = m_callback->SxsXMLXql(collection, query, &response);

if (ret != 0)
{
    // do any error handling, see chapter Callback Error Handling
}

// do any work ...
```

Java

```
String collection = "MyCollection";
String query = "/MyDoctype/Node_A[anyId='5']";
StringBuffer response = new StringBuffer();

int ret = SxsXMLXql(collection, query, response);

if (ret != 0)
{
    // do any error handling, see chapter Callback Error Handling
}

// do any work ...
```

ODBC Callbacks

ODBC callbacks support calls to any relational database that can be accessed via the Microsoft Windows ODBC Manager as a system DSN (Data Source Name).

ODBC callbacks are [currently] only available for Microsoft Windows ODBC Manager.

The parameter types used within an ODBC callback are defined in the *Sqltypes.h* file, which is stored in the directory *...\Microsoft Visual Studio\VC98\Include*. This file can be included in a server extension using an ODBC callback.

Character data is given and received as byte arrays in the encoding of the respective data source, which is commonly ASCII. Since Tamino stores XML data as Unicode strings, conversion may be necessary.

The next sections describe how you can allocate a **handle** and **memory**, and which **ODBC functions** are supported for Direct and Java.

Note: ODBC callbacks cannot be used within event functions. ODBC functions can be used in init functions to establish ODBC connections.

Allocating Handles

Connect Handles

Connect handles are required to access a database. They are allocated as follows:

Direct call:

```
SQLRETURN SxsSQLGetConnect(SQLTCHAR* dbName,
                           SQLTCHAR* userName,
                           SQLTCHAR* password,
                           sxdint* dbHandle);
```

Java call:

```
public int SxsSQLGetConnect(String dbName,
                            String userName,
                            String password,
                            IntRef Handle);
```

All infrastructures:

This call is used instead of the calls of the common ODBC interface that are used to prepare a connection. This call delivers the connect handle if a valid ODBC database name was provided. Note that this handle is not a valid ODBC handle and cannot be used for ODBC calls (its data type is not SQLHDBC but a simple long integer). The server extension function developer can access an ODBC database using ODBC calls independently of the Tamino Server, but he or she must perform all of the ODBC steps required to set up a connection and to close it again later.

The handle is only valid when using the Tamino Server ODBC callback interface. This interface implements a subset of the functions provided with the ODBC core level, version 3. The common ODBC calls are used with the prefix "Sxs", for example SxsSQLExecute(). The usage and the parameters of the functions are the same as for the original ODBC functions, with the exception of handles, which are long integers.

Statement Handles

Statement handles are allocated as follows:

Direct:

All ODBC callbacks are methods of the callback handle.

```
SQLRETURN SxsSQLAllocHandle(short hdType,
                            sxdint dbCHandle,
                            sxdint* dbSHandle);
```

This call does not return an ODBC handle of data type SQLHSTMT; rather, it returns a long integer that can only be used with the Tamino Server ODBC callback interface. This function can only be used with the HandleType SQL_HANDLE_STMT. This also applies to the function SxsSQLGetDiagField.

```
SQLRETURN SxsSQLFreeHandle(short hdType, sxdint dbCHandle);
```

Java:

```
public int SxsSQLAllocHandle(short HandleType,
                             int InputHandle,
                             IntRef OutputHandle);
```

This call does not return an ODBC handle; rather, it returns an integer that can only be used with the Tamino Server ODBC callback interface. SxsSQLGetDiagField and SxsSQLFreeHandle must also be of the HandleType SQL_HANDLE_STMT.

A statement handle allocated with SxsSQLAllocHandle() must be freed using:

```
public int SxsSQLFreeHandle(short HandleType,
                            int Handle);
```

Allocating Memory

Java:

The byte arrays for the result values must be initialized with the length that is also given as a parameter using `new byte [BufferLength]`. `IntRef` is an auxiliary class for output parameters as well as for in and out parameters. It represents a reference to an integer with the following methods:

```
public IntRef(int initVal);

public int getVal();

public void setVal(int newVal);
```

All parameters of type `IntRef` must be initialized.

The Other Supported ODBC Functions

Direct:

```
SQLRETURN SxsSQLPrepare(sxdint StatementHandle,
                        SQLTCHAR* StatementText,
                        SQLINTEGER TextLength);

SQLRETURN SxsSQLBindParameter(sxdint StatementHandle,
                              SQLUSMALLINT ParameterNumber,
                              SQLSMALLINT InputOutputType,
                              SQLSMALLINT ValueType,
                              SQLSMALLINT ParameterType,
                              SQLUINTEGER ColumnSize,
                              SQLSMALLINT DecimalDigits,
                              SQLPOINTER ParameterValuePtr,
                              SQLINTEGER BufferLength,
                              SQLINTEGER * StrLen_or_IndPtr);

SQLRETURN SxsSQLGetData(sxdint StatementHandle,
                        SQLUSMALLINT ColumnNumber,
                        SQLSMALLINT TargetType,
                        SQLPOINTER TargetValue,
                        SQLINTEGER BufferLength,
                        SQLINTEGER *StrLen_or_Ind);

SQLRETURN SxsSQLBindCol(sxdint StatementHandle,
                        SQLUSMALLINT ColumnNumber,
                        SQLSMALLINT TargetType,
                        SQLPOINTER TargetValue,
                        SQLINTEGER BufferLength,
                        SQLINTEGER *StrLen_or_Ind);

SQLRETURN SxsSQLGetDiagField(SQLSMALLINT HandleType,
                             sxdint Handle,
                             SQLSMALLINT RecNumber,
                             SQLSMALLINT DiagIdentifier,
                             SQLPOINTER DiagInfo,
                             SQLSMALLINT BufferLength,
                             SQLSMALLINT *StringLength);

SQLRETURN SxsSQLExecute(sxdint StatementHandle);
```

```
SQLRETURN SxsSQLFetch(sxdint StatementHandle);

SQLRETURN SxsSQLCloseCursor(sxdint StatementHandle);
```

Java:

```
public int SxsSQLPrepare(int StatementHandle,
                         String StatementText);

public int SxsSQLBindParameter(int StatementHandle,
                               short ParameterNumber,
                               short InputOutputType,
                               short ValueType,
                               short ParameterType,
                               int ColumnSize,
                               short DecimalDigits,
                               byte[] ParameterValuePtr,
                               int BufferLength,
                               IntRef StrLen_or_IndPtr);

public int SxsSQLGetData(int StatementHandle,
                         short ColumnNumber,
                         short TargetType,
                         byte[] TargetValue,
                         int BufferLength,
                         IntRef StrLen_or_IndPtr);

public int SxsSQLBindCol(int StatementHandle,
                         short ColumnNumber,
                         short TargetType,
                         byte[] TargetValue,
                         int BufferLength,
                         IntRef StrLen_or_IndPtr);

public int SxsSQLGetDiagField(short HandleType,
                              int Handle,
                              short RecNumber,
                              short DiagIdentifier,
                              byte[] DiagInfoPtr,
                              short BufferLength,
                              IntRef StringLengthPtr);

public int SxsSQLExecute(int StatementHandle);

public int SxsSQLFetch(int StatementHandle);

public int SxsSQLCloseCursor(int StatementHandle);
```

Examples

This section shows some examples in the various infrastructures. Error handling has been omitted for the sake of clarity.

Direct

```
sxdint ConnectHdl = 0;
SQLRETURN sqr = 0;

sqr = m_callback->SxsSQLGetConnect(m_callback->AllocString(L"Meyer"),
                                   m_callback->AllocString(L"secret"),
                                   m_callback->AllocString(L"MySqlDB"),
                                   &Connecthdl);

sxdint StmtHdl = 0;

sqr = m_callback->SxsSQLAllocHandle(SQL_HANDLE_STMT, ConnectHdl, &StmtHdl);
sqr = m_callback->SxsSQLPrepare(StmtHdl,
                                m_callback->AllocString(L"select * from hotel"),
                                  19);
sqr = m_callback->SxsSQLExecute(StmtHdl);
sqr = m_callback->SxsSQLFetch(StmtHdl);

int ExpectedLength = 1000;
char Result[1000];

sqr = m_callback->SxsSQLGetData(StmtHdl, 1, SQL_C_CHAR,
                                  Result, ExpectedLength, NULL);

// evaluate or copy Result anyway, perform some more calls using StmtHdl ...

sqr = m_callback->SxsSQLFreeHandle(SQL_HANDLE_STMT, StmtHdl);

// ODBC disconnect will be done by Tamino Server
```

Java

```
IntRef connectHdl = new IntRef(0);
int sqr = 0;

sqr = SxsSQLGetConnect("MySqlDB", "Meyer", "secret", connectHdl);

IntRef stmtHdl = new IntRef(0);

sqr = SxsSQLAllocHandle(SQL_HANDLE_STMT, connectHdl.getVal(), stmtHdl);

sqr = SxsSQLPrepare(stmtHdl.getVal(), "select * from hotel");
sqr = SxsSQLExecute(stmtHdl.getVal());
sqr = SxsSQLFetch(stmtHdl.getVal());
```

```
int bufferLength = 1000;
byte[] resBuffer = new byte[bufferLength];
IntRef resIndicator = new IntRef(0);

sqr = SxsSQLGetData(stmtHdl.getVal(), 1, SQL_C_CHAR, resBuffer,
                        bufferLength, resIndicator);

// evaluate or copy Result anyway, perform some more calls using stmtHdl ...

sqr = SxsSQLFreeHandle(SQL_HANDLE_STMT, stmtHdl.getVal());

// ODBC disconnect will be done by Tamino Server
```

System Callbacks

System callbacks can be used in all types of server extensions to obtain information about the Tamino Server. The following system callback methods are available:

- Callback Method SxsSystem
- Callback Method SxsGetHttpHeaderField
- Callback Method SxsSetProperty

Callback Method SxsSystem

The following call types are available:

Call Type for Java / Call Type for Direct	SysBuf	Description
IS_REPLICATION / SX_IS_REPLICATION	"YES"/"NO"	Flag indicating whether the database is a replication database
JVM_USEROPTIONS / SX_JVM_USEROPTIONS	string	The server parameter indicating the JVM options in use
USERID / SX_USERID	user ID as string	
SESSIONID / SX_SESSIONID	session ID as string	
SERVER_READ_ONLY / SX_SERVER_READ_ONLY	"YES"/"NO"	The server parameter indicating whether the server is in read-only mode
XML_DEF_ENCODING / SX_XML_DEF_ENCODING	string	The server parameter indicating the XML default encoding (for responses from Tamino)
XML_DEF_TOKENIZER / SX_XML_DEF_TOKENIZER	string	The server parameter indicating the XML default tokenizer (possible values: “white-space separated”, “Japanese”)
XNODE_DEF_ENCODING / SX_XNODE_DEF_ENCODING	string	The server parameter indicating the X-Node default encoding (for external data sources accessed by Tamino)

Call Type for Java / Call Type for Direct	SysBuf	Description
XTENSION_JAVA_USAGE / SX_XTENSION_JAVA_USAGE	"YES"/"NO"	The server parameter indicating whether Java server extensions can be used
XTENSION_DIRECT_USAGE / SX_XTENSION_DIRECT_USAGE	"YES"/"NO"	The server parameter indicating whether Direct server extensions can be used

All call types for the method `SxsSystem()` of the callback classes `CSXSJavaCallback` and `CSXSDirectCallback` are constant enumerations that are inherited from ASXJBase (for Java) or defined in *sxenum.h* (with prefix SX_, for Direct).

Direct

The following function is available:

```
sxdint SxsSystem(sxdint callType, sxdstring *ppXMLBuf);
```

The available call types are defined as constants in the callback interface *sxenum.h*.

Java

The following method is available:

```
public int SxsSystem(int callType,
                     StringBuffer pSysBuf);
```

The call types are defined as constants that are known to the server extension by inheritance from ASXJBase.

Callback Method SxsGetHttpHeaderField

The HTTP header callback function can be used in all types of server extensions to obtain information about the current request's HTTP header. All HTTP header fields specified in the table below can be accessed by this method. String constants, which match the W3C standard HTTP header field name definitions, are defined for these fields. They are available in the language-dependent include files. A non-zero return code is issued if the HTTP header field name cannot be found among the current HTTP header fields. The following string constants are available:

Call String Constants	HTTP Header Field (Software AG)	Meaning
REQUEST_CLIENT_ADDRESS	REMOTE_ADDR	The remote IP address of the client's host
REQUEST_DOCUMENT_MEDIATYPE	Document-Mime-Type	The media type of the document
REQUEST_METHOD	REQUEST_METHOD	The request method
REQUEST_SERVER_HOST	SERVER_HOST	The server host name of the HTTP server
REQUEST_STRING	QUERY_STRING	The request string

Call String Constants	HTTP Header Field (Software AG)	Meaning
REQUEST_URI	URI	The URI of the request

Call String Constants	HTTP Header Field (W3C)
REQUEST_ACCEPTED_CHARSETS	Accept-Charset
REQUEST_MEDIATYPE	Content-Type
REQUEST_PREFERRED_LANGUAGE	Accept-Language

Direct

The following function is available:

```
sxdint SxsGetHttpHeaderField(sxdstring attrName,
                             sxdstring *ppXMLBuf);
```

The `attrName` string is the HTTP header field name. The file *sxdinc.h* defines constant values for fields in the above tables.

Java

The following method is available:

```
public int SxsGetHttpHeaderField(String attrName,
                                 StringBuffer pSysBuf);
```

The `attrName` string is the HTTP header field name. The constant values for fields in the above tables are known to the server extension by inheritance from ASXJBase.

Callback Method SxsSetProperty

The callback `SxsSetProperty` allows a value to be set in the Tamino response document. Currently the only value that can be set is the media type.

To set the media type, specify the following parameter:

Parameter `property`	**Parameter** `mediaType`	**Description**
SX_PROPERTY_RSP_MEDIATYPE	A text string such as "text/xml" representing a media type.	This specifies that the media type defined in the response document will be the value given.

For some combinations of media type and server extension return parameter type, Tamino escapes the XML content in the response document according to the standard XML escaping rules ("<"

becomes "<" etc.). The rules determining when escaping is used are summarized in the following table:

	Return parameter type is ino:XML-OBJ	Return parameter type is xs:string
Media type specifies non-XML text document	XML content is escaped	XML content is not escaped
Media type specifies XML document	XML content is not escaped but the contents must be well-formed XML	XML content is escaped

See the section *Media Type Requirements* in the documentation for *X-Machine Programming* for a summary of the media type rules used by Tamino.

Notes:

1. If you set the media type using this callback method, the Tamino response wrapper is suppressed. See also the section *Suppressing the Tamino Response Wrapper* in the *X-Machine Programming* Guide for related information.
2. One of the examples provided with the Tamino distribution kit shows how to construct a document with an arbitrary media type specified in a stylesheet. It is in the directory *Examples/Server Extension/Java_Query_XSLT* under the Tamino installation directory.

Direct

The following function is available:

```
sxdint SxsSetProperty(sxdint property,
                      sxdstring value);
```

Java

The following method is available:

```
public int SxsSetProperty(int property,
                          String value);
```

Callback Error Handling

Runtime errors because of callbacks called from server extensions can occur for many reasons. They can result from the communication mechanism between Tamino Server and server extension (a Java error), from the Tamino mechanism for calling server extensions, from the Tamino Server itself (in the case of an **XML callback** or a **system callback**), from a database other than a Tamino database, or from the ODBC mechanism (in the case of an **ODBC callback**). Therefore special error handling is provided for server extension callback functions. Errors produced by callbacks are reported to the calling server extension and can only be evaluated there.

If the callback call is successful, the callback function returns the value 0.

If the return value is non-zero, the following functions can be used to obtain more information:

Direct

```
sxdint SxsGetMsgNo();

sxdstring SxsGetMsgText();

sxdint SxsGetSqlMsgNo();

sxdint SxsGetInoMsgNo();
```

Java

```
int SxsGetMsgNo();

String SxsGetMsgText();

int SxsGetSQLMsgNo();

int SxsGetInoMsgNo();
```

All Infrastructures

SxsGetMsgNo() always returns a special message code for a server extension callback.

SxsGetMsgText() always returns the corresponding message text.

The messages and codes that can occur are listed in the section **Server Extension Messages**.

Depending upon the situation, the other callback error functions produce the following results:

Tamino Server Errors (All Infrastructures)

If SxsGetMsgNo() reports a Tamino Server error, an error occurred while processing the XML request in the Tamino Server. SxsGetInoMsgNo() can be used to obtain the Tamino message code. Then the Tamino documentation can be used to find out the reason for the error. Such an error can occur for an XML callback or the system callback. If errors occur in case of an XML callback, SxsXMLGetMessage() should always be called. For the syntax, see the section ***XML Callbacks***. This function returns the content of the "ino:messageline" as it appears in the response document of the Tamino Server.

SQL Errors in ODBC (All Infrastructures)

If SxsGetMsgNo() reports a general SQL error, SxsGetSqlMsgNo() can be used to obtain the return value of the original ODBC function (SQLRETURN). (This value is usually also returned for an ODBC callback.) The ODBC callback function SxsSQLGetDiagField() can be used to retrieve more information from the ODBC driver or foreign database involved. Then the ODBC documentation can be used to find out the reason for the error.

Other Errors (All Infrastructures)

All other messages refer to errors occurring while processing the callback functionality. They have to be interpreted using the messages listed in the section **Server Extension Messages**.

XML Write Errors (All Infrastructures)

XML callbacks that write to a database [SxsXMLDefine(), SxsXMLProcess(), SxsXMLDelete(), SxsXMLXQuery()] can produce special situations that have to be considered.

If such a callback returns a non-zero value and a subsequent SxsGetInoMsgNo() returns a non-zero error code, then a Tamino Server error has occurred. To ensure database consistency in such cases, the entire server extension call must fail (return non-zero). This causes the enclosing transaction to fail and actions to be rolled back.

In this case, the server extension developer should ensure that the necessary clearing up is done. (It does not make sense to issue more XML callback calls that initiate write operations, because they would be rolled back at the end of the transaction.) Then a server extension exception should be thrown. Its error code and text are returned to the Tamino user in the response document of the current request (see the section ***Exceptions***). If no exception is implemented by the server extension developer, an exception is initiated at the return of the server extension function causing the server extension function and the request to fail containing MessageNo and Text of the original SXGetInoMsgNo(), SXSXMLGetMessage(). The message text of the exception is included in the response document to inform the Tamino user.

Failure of a function and, as a consequence, the failure of the request from which the function was called automatically cause a rollback in the case of an anonymous session. In a user session the user must perform a rollback command explicitly.

Server Extension Messages

The following table shows the error codes [available by calling SxsGetMsgNo()], the error texts [available by calling SxsGetMsgText()] and some explanations:

Error code Direct	Error code Java	Error text	Explanation
SXS_CALLB_NO_MEM_NO	-	Not enough memory	Memory was exhausted during callback call.
SXS_CALLB_INO_ERROR_NO	INO_ERROR	Tamino Server error when calling a callback	A Tamino Server error occurred. Use SxsGetInoMsgNo() to retrieve the message number, then refer to the manual for more information.
SXS_CALLB_SQL_ERROR_NO (ODBC callback only)	SQL_CALLB_ERROR (ODBC callback only)	SQL error when calling a callback	An ODBC error occurred. Use SxsGetSqlMsg() or the return code of the just called ODBC callback function to get the SQLRETURN value, then refer to the ODBC manual for more information.
SXS_CALLB_ILLEGAL_HDL_TYPE_NO (ODBC callback only)	ILLEGAL_HANDLE_TYPE (ODBC callback only)	Callback used an invalid handle type	Wrong use of SxsSQLAllocHandle(). Only statement handles may be allocated in callbacks.
SXS_CALLB_ILLEGAL_SQL_CMD_NO (ODBC callback only)	ILLEGAL_SQL_COMMAND (ODBC callback only)	Commit or rollback not allowed	There was an attempt to perform a commit or rollback. This is not allowed in callbacks.
SXS_CALLB_INVALID_BUFFER_PTR_NO (ODBC callback only)	INVALID BUFFER (ODBC callback only)	Invalid data buffer pointer	An ODBC callback function with a parameter of type StringBuffer (Java) was called, but the passed variable had not been allocated using SxsAlloc() / new StringBuffer.

Examples

Error Handling for an XML Callback Call

Direct

```
sxdint ret = SxsXMLXql(collection, query, &response);

if (ret)
{
    m_callback->ThrowException(SXS_CALLB_NO_MEM_NO,
                                m_callback->AllocString(L"xql callback failed"));
}
```

Java

```
int ret = SxsXMLXql(collection, query, response);

if (ret != 0) {

    int msgNo = SxsGetMsgNo();                  // identify the callback error

    switch (msgNo) {

        case INO_ERROR:

            StringBuffer msgBuf = new StringBuffer();
            int inoMsgNo = SxsGetInoMsgNo();    // Get the Tamino Server message ↩
number
           ret = SxsXMLGetMessage(msgBuf);     // Get the Tamino Server messageline
          SxsException(inoMsgNo, msgBuf);     // Signal exception to the SXS caller
            break;

        default:

           String msg = SxsGetMsgText();       // Get the SXS message corresponding
                                                 // to callback error
           SxsException (msgNo, msg);          // Signal exception to the SXS caller

    }
}
```

Error Handling for an ODBC Callback Call

Java

```
int ret = SxsSQLExecute(StmtHdl);

if (ret != 0) {

    int msgNo = SxsGetMsgNo();                  // identify the callback error

    switch (msgNo) {

        case INO_ERROR:

            StringBuffer msgBuf = new StringBuffer();
            int inoMsgNo = SxsGetInoMsgNo();    // Get the Tamino Server message ↩
number
           ret = SxsXMLGetMessage(msgBuf);     // Get the Tamino Server messageline
          SxsException(inoMsgNo, msgBuf);     // Signal exception to the SXS caller
            break;

        case SQL_CALLB_ERROR:
           int sqlMsgNo = SxsGetSQLMsgNo();    // Get the SQL message number from ↩
ODBC
           String msg = SxsGetMsgText();       // Get the SXS message for ODBC error
           SxsException(sqlMsgNo, msg);        // Signal exception to the SXS caller
            break;

        default:
           String msg = SxsGetMsgText();       // Get the SXS message corresponding
                                                 // to callback error
           SxsException(msgNo, msg);           // Signal exception to the SXS caller

    }
}
```

Exceptions

If a server extension function throws an exception, the function call fails and returns with an error to the Tamino Server. The Tamino Server rolls back the current XML transaction and inserts the exception text (error code and error text as far as available) into the response document of the request that called the server extension function.

Direct

Developers can throw their own C++ exceptions. They must be derived from std::exception.

```
throw MyException ("This is an error");
```

A pointer to such an exception object can also be thrown:

```
throw new MyException ("This is an error");
```

Additionally, the callback method

```
m_callback->ThrowException (errnr, msg)
```

can be called.

The following method can be used to generate a warning rather than an exception:

```
sxdvoid SxsWarning (sxdlong warnNum, sxdstring msg);
```

Java

Developers can implement their own Java exceptions. These and any standard Java exceptions are handled as described above.

Additionally, the method `void SxsException (int errNo, String msg)` can be used to signal the caller that an exception occurred in the server extension, but it does not throw a Java exception itself. This ensures that the session is rolled back and that the error number and message appear in the response document, even if the developer does not throw a Java exception or if the corresponding exception is caught later.

The following method can be used to generate a warning rather than an exception:

```
void SxsWarning (int warnNum, String msg);
```

All Infrastructures

Note: Exceptions thrown from event functions do not make sense, because they are only processed when the current request has terminated. They are caught, but they are not shown to the user.

Version Numbers

A two-part version number of the form *major*.*minor* is associated with each Tamino server extension. By default, the version number "1.0" is allocated when a server extension is created. You can change the version number at any time during the server extension's lifecycle.

For more information about the relevance of the server extension version number, see **Upgrading a Server Extension**.

Direct

To change the version number of a Direct server extension, edit a comment in the server extension's header file. Assuming the server extension is called *MySXS* and its corresponding header file is *MySXS.h*, you will find a comment in the file *MySXS.h* of the form:

```
/**
 * Tamino Server Extension MySXS
 * @version 2.3
 * @author Bob
 */
```

Edit the comment `@version 2.3` as required. The version number is incorporated automatically when this file is processed during compilation to produce the wrapping structures and methods.

Java

To change the version number of a Java server extension, change the constructor of the static field `sxsVersion`. For example:

```
static final SXSVersion sxsVersion = new SXSVersion(2, 3); // version now 2.3
```

9 Debugging Java Server Extensions

The simplest way to debug a Java server extension is to attach a Java debugger to the running JVM. By specifying appropriate X-Tension Java options, you can make the JVM initialization wait during Tamino server startup until a debugger is attached.

Any Java IDE that supports JDWP (Java Debug Wire Protocol) should be suitable. NetBeans (from **http://www.netbeans.org/**), for example, uses the following JVM options:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=xxxx,server =y
```

where *xxxx* should be replaced by the relevant address.

The following generic steps should be followed to debug a Java server extension.

Caution: This should only be done on a test database, as it may result in significant performance degradation.

To debug a Java server extension

1. Add the required string to the X-Tension Java options (as documented in *Administrating Tamino Server Extensions – Specifying Java Options*).
2. Start the Tamino Server. The server start process will wait until you have attached the Java debugger.
3. Set breakpoint(s) as required in the Java sources of your server extension.
4. Start the Java IDE and attach it to the Tamino server's JVM (usually by entering the host and port number specified in the JVM options). The start process of the Tamino server will now complete.
5. Execute the server extension you want to debug. Execution stops if it reaches a breakpoint, and you can use all of your Java IDE's debugging functionality.

6 To stop debugging, let the server extension execute until it terminates. Stop the server with Tamino Manager and end the Java IDE debugging session.

7 Remove the debugging string from the X-Tension Java options.

8 Restart the Tamino server in normal operation mode.

10 Tracing Tamino Server Extensions

SXS Trace is a mechanism that enables you to obtain information from the interface between Tamino Server and server extension functions. In cases where a Tamino server extension behaves unexpectedly, this information may help to determine whether the problem lies in the server extension or the Tamino Server. Server extension developers can define their **own traces** to add specific information to the SXS Trace.

SXS Trace is activated and deactivated from the Tamino Manager.

The information written by SXS Trace can be accessed using standard XQuery or X-Query queries. All these queries are to be entered in the **Query** field of the Tamino Interactive Interface.

SXS Trace supports the following actions:

Activating SXS Trace

SXS Trace is activated for a whole database. All server extension functions called from XML requests are traced.

SXS Trace should be deactivated whenever it is no longer needed. It is deactivated by default each time the Tamino Server is started, in order to prevent unintended tracing.

To activate SXS Trace

1 Start the Tamino Manager.

Expand the **Databases** object.

Start and expand the database containing the server extensions that you want to trace.

2 Select the **Server Extensions** object.

3 Choose **Extension Settings**.

The **X-Tension Settings** page appears.

4 Mark the X-Tension Trace Switch. Confirm with OK.

Deactivating SXS Trace

No server extension function called from XML requests will be traced after deactivating SXS Trace. However, SXS Trace information that has been recorded remains.

To deactivate SXS Trace

1 Start the Tamino Manager.

Expand the **Databases** object.

Start and expand the database containing the server extensions that you no longer want to trace.

2 Select the **Server Extensions** object.

3 Choose **Extension Settings**.

The **X-Tension Settings** page appears.

4 Remove the check mark for X-Tension Trace Switch. Confirm with OK.

Programming User-Defined Trace Information

Server extension developers can define their own information to be included in the SXS Trace by using the following function:

Java call

```
SXSTrace(String text);

SxsTrace(String text);
```

To activate and retrieve the SXS Trace information, see the sections *Activating SXS Trace* and *Viewing SXS Trace Information*.

Direct

```
SXSTrace (sxdstring text);
```

Viewing SXS Trace Information

As long as SXS Trace is activated, trace information is automatically written and stored in XML documents of the collection ino:SXS-Trace in the Tamino Server. Insertions into ino:SXS-Trace are committed immediately after each entry; they are not part of the current transaction.

The information comprises the function name, date and time, session ID, the parameter values when entering a server extension function and after the return of this function, the parameter values passed to and returned from a callback, and exceptions. Server extension developers can define their own information to be included in the SXS Trace. For details, see the section *Programming User-Defined Trace Information*.

Database administrators can use standard XQuery or X-Query queries on the collection ino:SXS-Trace in the **XQuery** or **X-Query** tab of the Tamino Interactive Interface to access the trace information.

The following examples show how special trace information can be retrieved in XPath format.

Start the Tamino Interactive Interface from the Tamino program group under the Windows **Start** menu. In the **Database URL** field, enter "http://localhost/tamino/*your_database*" and in the **Collection** field, enter "ino:SXS-Trace".

Then enter one of the following commands in the **Query** field and choose the **Query** button:

ino:SXS-Trace
Shows all trace information stored in Tamino.

ino:SXS-Trace[FunctionEnter/@Name="MySXF"]
Shows all trace information for the "MySXF" server extension function.

ino:SXS-Trace/FunctionEnter[@Name="MySXF"]
Shows all trace information about the call of the "MySXF" server extension function.

ino:SXS-Trace/FunctionReturn[@Name="MySXF"]/Parameter
Shows all trace information about the return parameters of the "MySXF" server extension function call.

ino:SXS-Trace[FunctionEnter/@Name="MySXF"]/CallbackEnter
Shows all trace information about all callbacks that were used in the "MySXF" server extension function.

ino:SXS-Trace[FunctionEnter/@Name="MySXF"]/Exception
Shows all trace information about an exception that resulted from a call of the "MySXF" server extension function.

Deleting SXS Trace Information

SXS Trace information in the collection ino:SXS-Trace must be deleted manually, using the Tamino Interactive Interface.

To delete the SXS Trace

1 Start the Tamino Interactive Interface from the Windows **Start** menu. In the **Database URL** field enter "http://localhost/tamino/*your database*" and in the **Collection** field enter "ino:SXS-Trace";

2 Choose the **Delete** tab and enter

```
ino:SXS-Trace
```

in the **Delete Query** field.

This deletes all the trace information from the collection ino:SXS-Trace. You can also delete information selectively from the collection ino:SXS-Trace by specifying appropriate X-Query queries in the **Delete** field of the Tamino Interactive Interface.

Note: Deleting information from ino:SXS-Trace is irreversible, i.e. it is not possible to rollback the delete operation.

11 X-Tension Tools

The information in this section is broken down as follows:

Modifying the Public Java Classpath

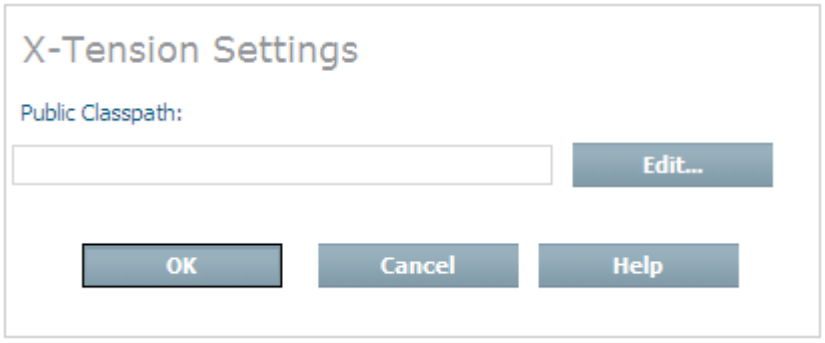
You can modify the public classpath for Java server extensions. A modified classpath applies to all Java server extensions in all Tamino databases on this host. During execution of a Java server extension, the public classpath precedes the standard classpath setting.

To add all the JAR files in a directory (e.g. *D:\X\Y*) to the private classpath, you can specify the directory name followed by **.jar* in the classpath (e.g. *D:\X\Y*.jar*). Other expressions with wildcard characters are not supported.

To modify the public classpath

1 Start the Tamino Manager.

Select the X-Tension Tools node and choose **X-Tension Settings** from the context menu. The **X-Tension Settings** dialog appears:



You can either enter the new classpath in the **Public Classpath** field directly or select the **Edit** button, which opens the **Modify Classpath** dialog:

2 In the **Modify Classpath** dialog, you can use the **Browse...** button to navigate to the required location, then choose the **Precede Path** or **Append Path** button to add the location to the start or the end of the classpath.

At this stage, the individual paths that make up the complete classpath are displayed on separate lines. The classpath is constructed by appending these paths in the order shown, starting from the top. You can change the position of any path in the list by selecting it and using the **Move Up** and **Move Down** buttons to move it to the required position. The part of the display in which the path names are listed cannot be scrolled horizontally, so path names that are wider than the available display area are not completely visible. In this case, you can select the path name, which then appears in the **Selected Path** field. This field can be scrolled horizontally.

To delete a path name from the list, select it and use the **Delete** button. To delete all of the path names from the list, use the **Delete all** button.

When you have defined the required list of paths, choose **OK** to return to the **X-Tension Settings** dialog.

3 In the **X-Tension Settings** dialog, choose **OK** to save the public classpath definition.

Analyzing Arbitrary Objects

If you have any DLL, EXE, TLB, CLASS, or JAR files that were not created by the **Tamino X-Tension Builder** and you want to determine whether they contain classes or methods that could be used as server extension functions, you can use the X-Tension Object Analyzer. If there are any suitable classes or methods, you can use the X-Tension Object Analyzer again to create a server extension package, which is what you need when you want to install a server extension into a database.

The X-Tension Object Analyzer is started from the Tamino Manager.

The X-Tension Object Analyzer must not be run more than once in parallel.

The typical steps to create a server extension package are:

- Analyzing an Object
- Entering Additional Information for a Server Extension Object
- Entering Additional Information for a Direct Server Extension Object
- Entering Additional Information for a Java Server Extension Object
- Selecting Server Extension Functions
- Deselecting Server Extension Functions
- Modifying Function Parameters
- Creating a Server Extension Package
- Deleting a Server Extension Object

A successfully analyzed object that does not satisfy your requirements can be **deleted** from the Tamino Manager.

Analyzing an Object

To find out whether a DLL, EXE, TLB, CLASS or JAR file contains code that could be used as a server extension function, the file must be analyzed.

To analyze an object

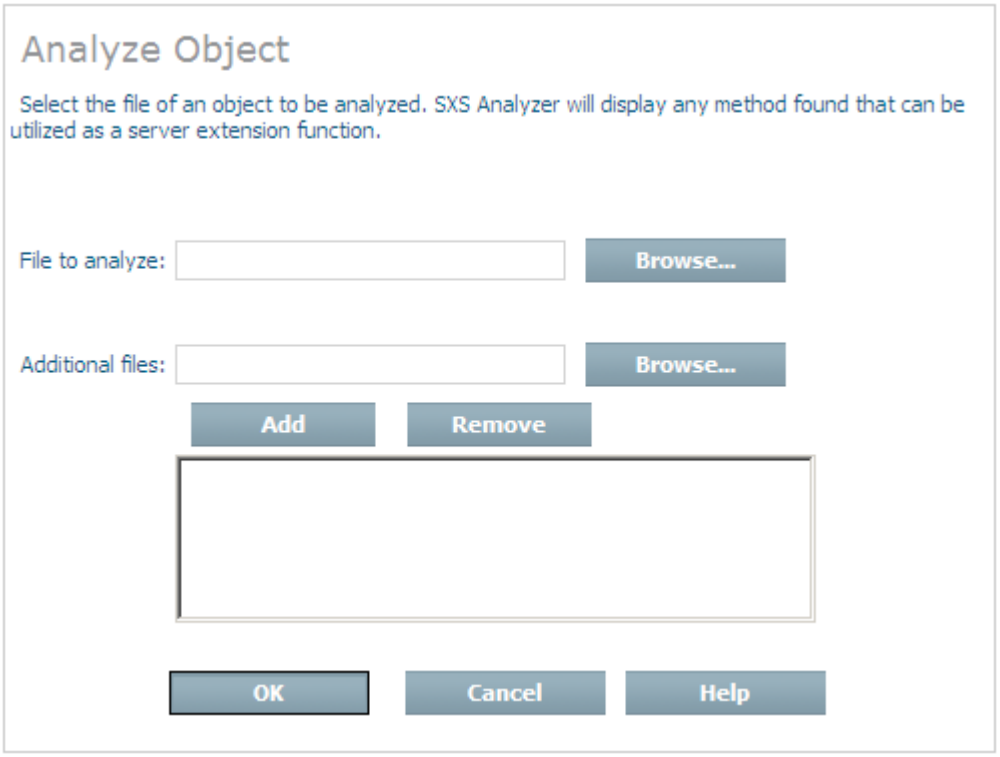
1 Start the Tamino Manager.

2 Select and expand the X-Tension Tools.

3 Select the **Object Analyzer**.

4 In the context menu, choose **Analyze Object**.

5 The **Analyze Object** page appears:



Enter the full path of the file to be analyzed (including the file name) in the **File to analyze** text box, or choose **Browse...** and select the drive, directory and file.

In the **Additional files** text box you can specify more CLASS or DLL files by entering the full path of the file to be analyzed (including the file name), or by choosing **Browse...** and selecting the drive, directory and file.

Choose **Add** to add the files you want. Note that a file that you have selected but not added to the list will not be included in the analyzed object. Repeat these steps as often as required. To delete a file from the list, select it and choose **Remove**.

Choose **OK** to continue.

6 If the specified file is a JAR file, the **Analyze Java Class** page appears:

Select a file from the **Class to analyze** text box and choose **OK** to continue. Ensure that this file is the main or entry class.

The **Job Monitor** page appears, informing you about success or failure of the analysis.

If the object was successfully analyzed, its name is now included under the expanded X-Tension Object Analyzer object in the Tamino Manager.

Entering Additional Information for a Server Extension Object

Additional information can be entered for a server extension object and its functions. You can modify the name of a server extension object, its description, the author and the external name of server extension functions.

To enter additional information for a server extension object

1 Start the Tamino Manager.

Select and expand the **Object Analyzer** to display the analyzed server extension objects in the tree view.

2 Select the server extension object for which you want to enter additional information.

3 Choose **Modify Extension**.

4 The **Modify Server Extension** page appears.

The **Extension Name** text field contains the name of the object. This name can be changed if required. If you enter a new name for the object, this name will be valid after packaging the object.

In the **Description** text field you can enter a brief description of the server extension. The descriptions of CLASS and JAR files that were not developed with the current X-Tension Builder can be changed; the descriptions of DLL files cannot be changed.

In the **Author** text field you can enter the name of the developer of the server extension.

In the **Function Name** text field you can enter the external name of the server extension function.

XML metacharacters such as "&", "<" and ">" are not allowed in the fields **Extension Name**, **Author**, **Description** and **Function name**.

Choose **OK** to save the entries.

The **Job Monitor** page appears, informing you about success or failure of the modifications.

If the message reports success, you must **select a function** from the server extension object before you can create a server extension package.

Entering Additional Information for a Direct Server Extension Object

Additional information can be entered for a server extension object and its functions, and the name of a server extension object can be modified. Help files that you specify here are included in the server extension package.

To enter additional information for a Direct server extension object

1 Start the Tamino Manager.

Select and expand the **Object Analyzer** to display the analyzed server extension objects in the tree view.

2 Select the Direct server extension object for which you want to enter additional information.

3 Choose **Edit Details**.

4 The **Edit Details of Object** page appears.

The path of the object is displayed in the **Object pathname** text field. The path cannot be changed from this dialog.

The **Extension name** text field contains the name of the object. This name can be changed if required. If you enter a new name for the object, this name will be valid after packaging the object.

In the **Short description** text field you can enter a brief description of the server extension. The description can be changed in the case of EXE files, but not in the case of DLL files.

In the **Author name** text field you can enter the name of the developer of the server extension.

XML metacharacters such as "&", "<" and ">" are not allowed in the fields **Author name** and **Short description**.

If known, enter the full path of the object's help file (including the file name) in the **Help file** text box; otherwise, choose **Browse...** and select the drive, directory and file. Help files must be of type HTM, HTML, XML or TXT.

If there are other help files associated with the object, enter the full path name of the file (including the file name) in the **Additional help file** text box or choose **Browse...** and select the drive, directory and file. Choose **Add** to add the help file to the list of additional help files. Note that a file you have selected but not added to the list will not be included in the analyzed object. Repeat these steps as often as required. To delete a help file from the list, select the file and choose **Remove**.

Choose **OK** to save the entries.

The **Job Monitor** page appears, informing you about success or failure of the modifications and the help file specifications.

If the message reports success, you must **select a function** from the server extension object before you can create a server extension package.

Entering Additional Information for a Java Server Extension Object

Additional information can be entered for a server extension object and its functions, and the name of a server extension object can be modified. Help files that you specify here are included in the server extension package.

To enter additional information for a Java server extension object

1 Start the Tamino Manager.

Select and expand the **Object Analyzer** to display the analyzed server extension objects in the tree view.

2 Select the Java server extension object for which you want to enter additional information.

3 Choose **Edit Details**.

4 The **Edit Details of Object** page appears.

The path of the object is displayed in the **Object pathname** text field. The path cannot be changed from this dialog.

The **Extension name** text field contains the name of the object. This name can be changed if required. If you enter a new name for the object, it will be valid after packaging the object.

In the **Short description** text field you can enter a brief description of the server extension. The descriptions of CLASS and JAR files that were not developed with the current X-Tension Builder can be changed; the descriptions of DLL files cannot be changed.

In the **Author name** text field you can enter the name of the developer of the server extension.

XML metacharacters such as "&", "<" and ">" are not allowed in the fields **Extension name**, **Author name** and **Short description**.

The value of the private classpath can be changed in the **Private Classpath** field. You can use the **Edit** button to open a subdialog in which you can specify the private classpath. See the section *Dialog for Setting the Private Classpath* for usage details.

If known, enter the full path of the object's help file (including the file name) in the **Help file** text box; otherwise, choose **Browse...** and select the drive, directory and file. Help files must be of type HTM, HTML, XML or TXT.

If there are other help files associated with the object, enter the full path name of the file (including the file name) in the **Additional help file** text box, or choose **Browse...** and select the drive, directory and file. Choose **Add** to add the help file to the list of additional help files.

Note that a file you have selected but not added to the list will not be included in the analyzed object. Repeat these steps as often as required. To delete a help file from the list, select it and choose **Remove**.

Choose **OK** to save the entries.

The **Job Monitor** page appears, informing you about success or failure of the modifications and the help file specifications.

If the message reports success, you must select a function from the server extension object before you can create a server extension package.

Selecting Server Extension Functions

To be able to create a server extension package, at least one function must have been selected from the analyzed object. You can choose from one of the following options:

Selecting a Server Extension Function with Defaults;

Selecting and Modifying a Server Extension Function;

Selecting all Server Extension Functions;

Modifying a Server Extension Function and Parameters.

Selecting a Server Extension Function with Defaults

To select a server extension function with default settings for a server extension object

1 Start the Tamino Manager.

Select and expand the X-Tension Tools.

Select and expand the **Object Analyzer**.

2 Expand a server extension object to display the associated server extension functions in the tree view.

3 Select the server extension function for which you want the default settings.

4 Choose **Select with Defaults**.

The **Select function ...** page appears, showing whether selecting the function was successful.

Selecting and Modifying a Server Extension Function

To select and modify a server extension function

1 Start the Tamino Manager.

Select and expand the X-Tension Tools.

Select and expand the **Object Analyzer**.

2 Expand a server extension object to display the associated server extension functions in the tree view.

3 Select the server extension function that you want to modify.

4 Choose **Select and Modify**.

5 The **Select and Modify Function** page appears.

The **Function name** text field contains the name of the function. It cannot be changed.

Select the appropriate type for the function from the **Function type** drop-down list box.

In the **Short description** text field you can enter a brief description of the server extension function.

XML metacharacters such as "&", "<" and ">" are not allowed in the field **Short Description**.

If known, enter the full path of the object's help file (including the file name) in the **Help file** text box; otherwise, choose **Browse...** and select the drive, directory and file. Help files can be of any type, but HTM, HTML, XML or TXT is recommended.

If there are other help files associated with the function, enter the full path name of the file (including the file name) in the **Additional help file** text box or choose **Browse...** and select the drive, directory and file, then choose **Add** to add the help file to the list of additional help files. Note that a file you have selected but not added to the list will not be included in the analyzed object. Repeat these steps as often as required. To delete a help file from that list, select the file and choose **Remove**.

Choose **OK** to save the entries and to select the function automatically.

The **Job Monitor** page appears, informing you about success or failure.

Selecting all Server Extension Functions

To select all Server Extension functions

1 Start the Tamino Manager.

Select and expand X-Tools.

Select and expand the **Object Analyzer** to display the analyzed server extension objects in the tree view.

2 Select the server extension object whose functions you want to select.

3 Choose **Select all Functions**.

The **Job Monitor** page appears, informing you about success or failure of the selection.

Modifying a Server Extension Function and Parameters

In the Modifying a Server Extension Function and Parameters dialog, the external name, the description of a server extension function and the XML data types of function parameters can be changed.

To modify a server extension function and parameters

1 Start the Tamino Manager.

Select and expand the X-Tension Tools.

Select and expand the **Object Analyzer**.

2 Expand a server extension object to display its associated server extension functions in the tree view.

3 Select the server extension function that you want to modify.

4 Choose **Modify Function**.

5 The **Modify Function** page appears.

The **Name** text field contains the name of the function. It cannot be changed.

In the **External Name** text field you can enter the external name of the server extension function.

In the **Description** text field you can enter a brief description of the server extension function.

XML metacharacters such as "&", "<" and ">" are not allowed in the field **Description**.

The parameter name can be changed for Java class files.

The XML data type of a parameter can be changed if a list box is displayed.

Choose **OK** to save the entries and to select the function automatically.

The **Job Monitor** page appears, informing you about success or failure.

Deselecting Server Extension Functions

If selected functions are not needed, you can **deselect all functions**, or you can **deselect individual functions**.

Deselecting all Server Extension Functions

To deselect all server extension functions

1 Start the Tamino Manager.

Select and expand X-Tools.

Select and expand the **Object Analyzer** to display the analyzed server extension objects in the tree view.

2 Select the server extension object whose functions you want to deselect.

3 Choose **Deselect all Functions**.

The **Job Monitor** page appears, informing you about success or failure of the deselection.

Deselecting Individual Server Extension Functions

To deselect one server extension function

1 Start the Tamino Manager.

Select and expand X-Tools.

Select and expand the **Object Analyzer**.

2 Expand a server extension object to display the associated server extension functions in the tree view.

3 Select the server extension function that you want to deselect.

4 Choose **Deselect Function**.

The **Job Monitor** page appears, informing you about success or failure of the deselection.

Modifying Function Parameters

When a server extension function has been selected, its parameters can be modified.

To modify a function parameter

1 Start the Tamino Manager.

Select and expand **X-Tension Tools**.

Select and expand the **Object Analyzer**.

2 Expand one of the displayed server extension objects.

3 Expand a selected server extension function to display the associated parameters in the tree view.

4 Select the parameter that you want to modify.

5 Choose **Modify Parameter**.

6 The **Modify Parameter** page appears:

The internal name of the parameter is displayed in the **Parameter Name** text box. This can be changed in the case of Java parameters.

The other parameters such as the host language type (the example shows the host language type to be "Java") and **direction** are displayed but they cannot be changed. The XML data type can be changed if a list box is displayed.

For Init functions, the default values must be specified in the **Default Value** text field.

Choose **OK** to modify the parameter.

The **Job Monitor** page appears, indicating whether the modification was successful or not.

Creating a Server Extension Package

When you have **analyzed an object**, **selected the functions** you want, and **made any modifications** you want, you can create a server extension package.

To create a server extension package

1 Start the Tamino Manager.

Select and expand **X-Tension tools**.

Select and expand the **Object Analyzer**.

2 Select the server extension object that you want to pack.

3 Choose **Pack Extension**.

4 The **Pack Server Extension** page appears.

The **Destination directory** text box contains the full path of the directory in which the server extension package will be created. The default directory is *...\Tamino\Server Extensions\Pack*. You can change the directory by entering a path or choosing **Browse...** to select a drive and directory.

Tick the **Overwrite existing package** check box if you want to overwrite an existing package with the same name.

Choose **OK** to create the server extension package.

The **Job Monitor** page appears, informing you about success or failure of the creation of the package file.

The package file (*your-file.sxp*) can be installed in a Tamino database as described in the section **Installing a Tamino Server Extension**.

Deleting a Server Extension Object

If you discover that a server extension object that was successfully analyzed is not useful for your purpose, you can delete it.

To delete a server extension object

1 Start the Tamino Manager.

Select and expand **X-Tools**.

Select and expand the **Object Analyzer** object.

2 Select the server extension object that you want to delete.

3 Choose **Forget Object** to delete it.

The **Job Monitor** page appears, informing you about success or failure of the deletion.

Viewing a Package File

To find out whether a package contains valid code that is useful for your purpose

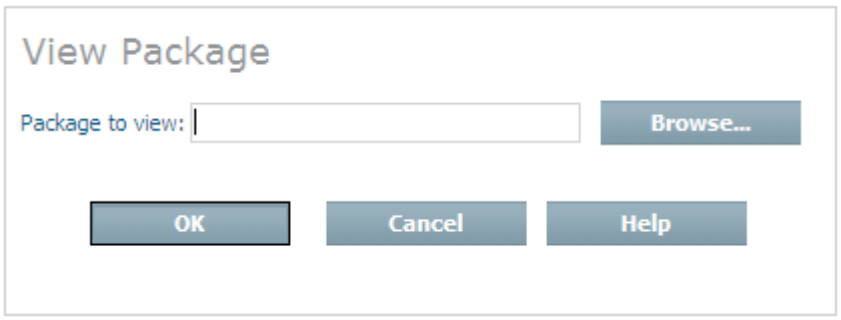
1 Start the Tamino Manager.

Select and expand **X-Tension Tools**.

Select and expand **Package Viewer**.

In the context menu, choose **Analyze Object**.

2 The **View Package** page appears:



Browse to an **.sxp* file and select it. Choose **OK**.

The name of the server extension package appears in the tree view. Here the package can be expanded and viewed, just like any other server extension object. After viewing, the package file is automatically deleted when the Package Viewer tree is collapsed.

If the package is useful, you can install the **.sxp* file in the database.

12 Tamino Server Extension Examples

Complete examples of all types of server extensions written in C++ and Java are provided in the Tamino installation directory. In addition, an XSLT server extension example is described in detail in the appendix ***Example: XSLT Server Extension*** of this document.

The following examples, which are available in the Tamino installation directory, are named after the infrastructure and function type: Direct Mapping, Direct Query, Direct Trigger, Java Mapping, Java Query, Java Trigger. Each example includes all sources and information to build, administrate and execute the corresponding example case. On Windows platforms, these are hosted in the sub-directories:

Examples\Server Extensions\Direct_Mapping

Examples\Server Extensions\Direct_Query

Examples\Server Extensions\Direct_Trigger

Examples\Server Extensions\Java_Mapping

Examples\Server Extensions\Java_Query

Examples\Server Extensions\Java_Trigger

Examples\Server Extensions\Java_Query_XSLT

The corresponding UNIX directory names are: *Examples/Server_Extensions/...*

These directories contain all the files that are generated when creating package files using the Tamino X-Tension Builder. Detailed explanations and use-cases are included in the *Readme.txt* files.

The **.sxp* files contain, among other things, the executable files that can be installed and executed in Tamino databases. The *Install.xml* files describe the interface to the server extension, which is

necessary for administration and execution. A copy of the *Install.xml* file is included in the **.sxp* file for easy administration.

Some of the supplied pre-built packages (**.sxp* files) are compiled and packed for Microsoft Windows platforms only. You may need to rebuild the projects and create new **.sxp* files to suit a different target platform. Please check the *Readme.txt* file for further information.

The **.cpp* and **.java* files are source files that can be opened in a text editor to view the function code.

An example of a shadow function is currently not available.

A Example: XSLT Server Extension

The XSLT server extension, written in Java, is designed to perform XSLT transformations on XML documents in the Tamino database. Database queries can extract the input XML document(s) and the XSLT stylesheet from Tamino, and the resulting document can be redirected back to the client or written to another collection within the same Tamino database. There are several configuration options available to control the behavior of the XSLT server extension.

The Tamino kit includes sample files that you can use for testing the server extension. They are available under the Tamino installation directory *Examples/Server Extension/Java_Query_XSLT* in the following subdirectories:

examples
Contains the sample schemas, input data and XSLT stylesheets;

config
Contains the configuration schema and sample configurations for Saxon and Xalan;

src
Contains the source code (Java), *Install.xml*, internal interface HTML documentation for the server extension and the prebuilt X-Tension package file (*sxp*).

The following sections are available:

Requirements

Tamino	The current release of Software AG Tamino XML Server.
XSLT Processor	Any Java XSLT processor implementing TrAX (Transformation API for XML), such as Saxon or Xalan. ■ Saxon is available at **http://saxon.sourceforge.net/**. ■ Xalan can be downloaded from **http://xml.apache.org/xalan-j/**.
Java API for XML Parsing	The Java API for XML Parsing, *jaxp.jar*, is available at **http://jaxp.java.net/**.

Known Limitations

The known limitations in the current implementation of the XSLT server extension are listed here:

- There is currently no mechanism for maintaining cache consistency if the XSLT stylesheet is changed in the Tamino database. This could become an issue if you are developing a stylesheet in the database. In this case the cache should be turned off.

Installation

The JAR files previously mentioned in the **Requirements** section, namely *jaxp.jar* and the XSLT processor, must be added to the class path before installing the XSLT server extension software package. The section **Configuration** describes how to select the XSLT processor of your choice. The XSLT server extension is contained in the package *XSLT.sxp*. Refer to the section ***Installing a Tamino Server Extension*** for information about installing server extensions and adding JAR files to the Java class path.

Query Functions for Transformation

The XSLT server extension has two separate query functions to perform XSLT transformations on XML data in Tamino. Both of them add wrapper elements to the input data and the result document. These wrapper elements are fully configurable through the SXS-Configuration document, and also through the run-time configuration parameters `input_wrapper` and `result_wrapper`. The default values are "XSLT_input" and "XSLT_result". If these parameter values are empty, no input or result wrapper is applied before or after the transformation, respectively. If an input wrapper is applied, thus adding a new root element, the result of the input query expression passed as the first argument (see below) does not have to be a single document; it may consist of zero or more documents. Furthermore, note that a stylesheet may require some adaptation if an input wrapper is being applied.

The `media-type` attribute of the `xsl:output` element in the stylesheet can be used to control the value of the content-type HTTP header that is returned to the client.

transform

This query function performs an XSLT transformation on an XML document extracted from a Tamino collection using an XSLT stylesheet queried from another collection. The result is returned to the client.

The following parameters are provided:

Parameter	Meaning	XML Type	Explanation
1	Query expression	XML-OBJ	The query expression that extracts from Tamino the XML input document to be transformed. This query refers to the collection passed in the URL.
2	Stylesheet Collection	xs:string	The name of the collection in Tamino that contains the XSLT stylesheet.
3	Stylesheet Query	xs:string	The query expression to extract the XSLT stylesheet from Tamino.

Usage of this function is illustrated in the section **A Simple Transformation Example**.

Administrative Issues

This section provides information on administrative issues involved with using the XSLT server extension.

Storing Stylesheets

There are two strategies for loading stylesheets:

- Load them into the collection *ino:etc* without providing a schema;
- Define a schema, for instance by using the enclosed schema file *xsl_stylesheet.tsd*. This file defines a doctype named *xsl:stylesheet* in the collection *stylesheet*.

Configuration

A Tamino SXS server extension query function is provided in order to manage the configuration of the XSLT server extension in conjunction with an SXS-Configuration XML document inside Tamino.

Syntax:

```
configuration (configkey, configvalue)
```

For more information on the syntax, please refer to the section *Query Function Call Syntax*.

configkey	*configvalue*	**Description**
`cache_state`	`ON/OFF`	Sets the state of the cache either *on* or *off*.
`cache_size`	`1 - 255`	Specifies the maximum number of stylesheets that can be held in the cache.

configkey	*configvalue*	**Description**

`transformer_factory`	*Class Package*	Specifies the XSLT transformation factory Java class package to use, e.g. for:
		■ Saxon (default):
		`com.icl.saxon.TransformerFactoryImpl`
		■ Xalan:
		`org.apache.xalan.processor.TransformerFactoryImpl`

`show`	`(ignored)`	Shows the current configuration values. The resulting XML document is described below.
`refresh`	`(ignored)`	Refreshes the current configuration with the values contained in the SXS-Configuration for this server extension.
`update`	`ON/OFF`	Specifies the update state of the server extension. This determines whether the values contained in the SXS-Configuration for this server extension should be updated. If the update state is set to *off*, any changes made to this server extension via this configuration function apply to the current session only.
`input_wrapper`	`string`	Specifies the name of the input wrapper. Default is `XSLT_input`.
`result_wrapper`	`string`	Specifies the name of the result wrapper. Default is `XSLT_result`.

The XML document returned by `configuration("show","")` is as follows:

```
...
<xql:result>
    <XSLT>
        <configuration>
            <show>
                <value name="cache_state">ON</value>
                <value name="cache_size">128</value>
                <value ↩
name="transformer_factory">com.icl.saxon.TransformerFactoryImpl</value>
                <value name="input_wrapper">XSLT_input</value>
                <value name="result_wrapper">XSLT_result</value>
                <value name="update">ON</value>
            </show>
        </configuration>
    </XSLT>
</xql:result>
...
```

Based on the included schema file *SXS-Configuration.tsd*, a collection named *SXS-Configuration* (containing a single doctype "SXS-Configuration") can be defined.

The following is an example of an "SXS-Configuration" document:

```
<SXS-Configuration name="XSLT">
    <value name="cache_state">ON</value>
    <value name="cache_size">128</value>
    <value name="transformer_factory">com.icl.saxon.TransformerFactoryImpl</value>
    <value name="input_wrapper">myOwnInput</value>
    <value name="result_wrapper">myOwnResult</value>
    <value name="update">ON</value>
</SXS-Configuration>
```

Sample configuration documents for use with Saxon (default) and Xalan are enclosed in this package:

- *SXS-Configuration_Saxon.xml*
- *SXS-Configuration_Xalan.xml*

Cache Management

The XSLT server extension provides a Tamino SXS server extension query function in order to manage the stylesheet cache.

Syntax:

```
cache (operation)
```

For more information on the syntax, please refer to the section ***Query Function Call Syntax***.

<*operation*>	Description
`show`	The `cache("show")` operation queries the stylesheet cache and returns an XML document describing its current contents.
`clear`	The `cache("clear")` operation deletes all entries in the stylesheet cache.
`delete=`*`cache_key`*	The `cache("delete=`*`cache_key`*`")` operation requires the addition of a cache key value to delete an individual stylesheet from the stylesheet cache. To list valid cache key values, use the `show` operation.

The following shows a typical XML document returned by the `cache("show")` operation:

```
...
<xql:result>
   <XSLT>
      <cache>
         <show_cache>
            <cache_entries>
               <entry key="ino:etc/xsl:stylesheet[@ino:id=12]">
                  <value ↩
name="transformer_factory">com.icl.saxon.TransformerFactoryImpl</value>
                  <value name="loaded">Mon Sep 17 04:12:16 EDT 2003</value>
                  <value name="last_used">Tues Sep 20 09:32:47 EDT 2003</value>
                  <value name="usage_count">19</value>
               </entry>
               <entry key="ino:etc/xsl:stylesheet[@ino:docname='mytransform']">
                  <value ↩
name="transformer_factory">com.icl.saxon.TransformerFactoryImpl</value>
                  <value name="loaded">Mon Sep 17 03:15:26 EDT 2003</value>
                  <value name="last_used">Weds Sep 21 10:22:57 EDT 2003</value>
                  <value name="usage_count">3</value>
               </entry>
            </cache_entries>
         </show_cache>
      </cache>
   </XSLT>
</xql:result>
...
```

A Simple Transformation Example

In order to provide some hands-on experience, the following section gives step-by-step instructions for performing a simple transformation. The following example documents are contained in this kit:

Document	Description
NASCAR_Drivers.tsd	The example data collection schema
NASCAR_Drivers.xml	The example data
NASCAR_Drivers_to_XML.xsl	The example stylesheet

The name of the Tamino database used in this example is "MyDB".

Tip: As an alternative to HTTP, you may use the Tamino Interactive Manager to perform queries.

To perform a simple transformation

1 Define the schema (*NASCAR_Drivers.tsd*) in the database.

2 Process the input data (*NASCAR_Drivers.xml*) into collection *NASCAR_Drivers*.

Note: This file contains multiple documents.

3 Process the XSLT stylesheet (*NASCAR_Drivers_to_XML.xsl*) into the *ino:etc* collection for "MyDB". Note the "ino:id" for the processed document (`ino:id=1` for this example).

4 Verify the existence of the following documents in the database:

http://localhost/tamino/MyDB/NASCAR_Drivers?_xql=NASCAR_Drivers
http://localhost/tamino/MyDB/ino:etc?_xql=xsl:stylesheet[@ino:id=1]

5 Perform the transformation using X-Query:

```
http://localhost/tamino/MyDB/NASCAR_Drivers?_xql=XSLT.transform
(NASCAR_Drivers,'ino:etc','xsl:stylesheet[@ino:id=1]')
```

6 Perform the transformation using XQuery:

```
http://localhost/tamino/MyDB/NASCAR_Drivers?_xquery= {?serialization
method="XSLT.transform" parameter='ino:etc'
parameter='xsl:stylesheet[@ino:id=1]'?} for $d in input()/NASCAR_Drivers return
$d
```

where:

`http://localhost/tamino/MyDB/NASCAR_Drivers`
specifies the host, the database and the input data collection;

`?_xql=`
is the command verb to perform a query (the query expression follows after the = sign);

`XSLT.transform`
uses the `transform` query function contained in the XSLT server extension;

`NASCAR_Drivers`
is the query expression that extracts the input data from the collection;

`'ino:etc'`
is the collection that contains the XSLT stylesheet;

`'xsl:stylesheet[@ino:id=1]'`
is the query expression that extracts the stylesheet.

Index

T

X