



digital  
vision  
group

# JSON and XML Databases

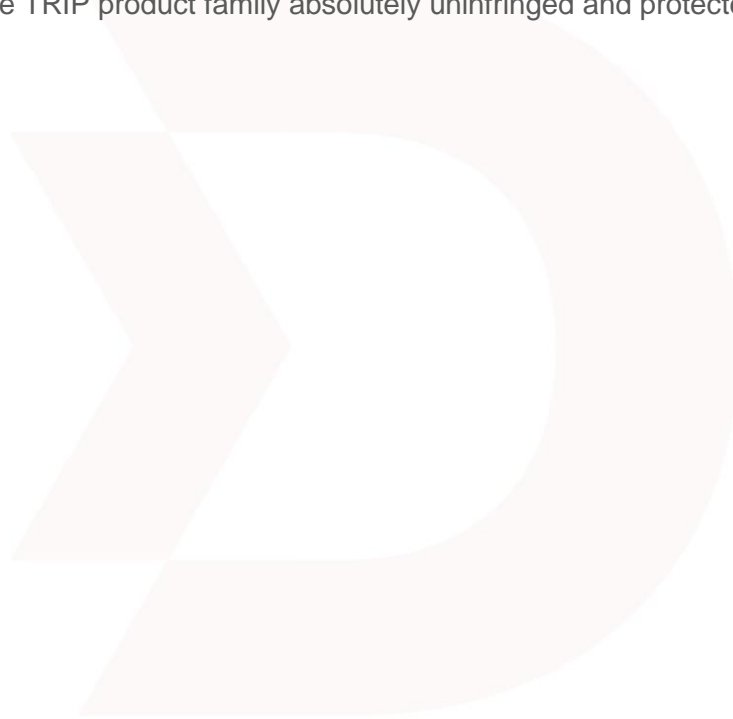
TRIPsystem  
Product Documentation

## End User License Agreement

All rights to this software, its documentation and logotypes of the TRIP product family and software (altogether “Software”) supplied by DVG Operations GmbH (DVG) are exclusively owned by DVG.

The transfer of this Software, solutions or parts thereof requires the prior written agreement of DVG. Furthermore, the customer has the right to use licensed Software and / or process solutions supplied by DVG to the extent specified in his contract with DVG.

The free-to-use non-commercial version doesn't require a prior written agreement with DVG but such customers, organizations and/or third parties agree by using the software and / or solution of DVG to be strongly obliged to keep all rights to this software, documentation and logotypes of the TRIP product family absolutely un infringed and protected.



## Table of Contents

<b>Introduction.....</b>	<b>5</b>
About this Document .....	5
XML .....	5
JSON .....	5
<b>Installation and Upgrade .....</b>	<b>6</b>
<b>Features.....</b>	<b>7</b>
Overview.....	7
Design .....	7
Overview.....	7
Storage Structure .....	8
XLink and XInclude support (removed) .....	9
Querying and Search Results .....	9
Application Programming .....	9
<b>How-To .....</b>	<b>10</b>
Create a JSON/XML database .....	10
Via TRIPclassic .....	10
Via TRIPmanager .....	10
Using jxtool .....	10
JSON/XML Database Character Set.....	10
Query the Database .....	10
Using CCL .....	10
Using XPath.....	11
Use the Toolkit API Calls.....	11
JSON/XML C API Functions .....	11
Backward-Compatibility Functions .....	11
<b>TdblImport</b> .....	12
<b>TdbExport</b> .....	13
Client-Side API calls .....	14
TRIPjxp and TRIPnxp.....	14
TRIPclient (TRIPcom).....	14
TRIP Java Toolkit .....	15
<b>Appendix A - API Reference.....</b>	<b>16</b>
The filter_data structure .....	16
<b>Appendix B - Encodings.....</b>	<b>18</b>
Supported Encodings for XML Documents .....	18
Supported Encodings for JSON Documents.....	19
<b>Appendix C – Settings for tdb.conf .....</b>	<b>20</b>
<b>Appendix D - Supported XPath Syntax .....</b>	<b>21</b>
Definitions .....	21
Node .....	21
Node Set.....	21
Location Step.....	21
Axis Types .....	22
Functions .....	22
Predicate Usage.....	23
The existence of an attribute .....	23
Attribute value check .....	23
Exact text node contents .....	23
Truncated text node contents.....	24

Truncated attribute text node contents .....	24
Comparison Operators .....	24
Node position.....	24
Multiple predicates in a single location step .....	25
Additional Examples .....	25
Expression over Axis "child" .....	25
Expression with axis "attribute" in predicate.....	25
Expression with axis "parent" .....	26
Expression with multiple axes in a predicate .....	26
Expression with axis "descendant" and multi-axis predicate .....	26
Expression with axes "ancestor-or-self", "parent" and "descendant" .....	26
<b>Appendix E – The jxtool.....</b>	<b>27</b>
Introduction.....	27
Purpose and Use Cases .....	27
Command Line Interface .....	27
Administration.....	28
Importing and Exporting XML and JSON documents .....	28
Bulk Import .....	28
XPath Based Querying.....	29

# Introduction

## About this Document

This document describes the JSON and XML database functionality as it is available in TRIPsystem 8.2. Descriptions of APIs and examples for their use is found with respective SDK product (TRIPjxp, TRIPnxp, etc).

XML database functionality in TRIP versions older than 8.0 were available in the separately installed TRIPxml product. This functionality has since TRIP version 8.0 been integrated into TRIPsystem, although still requiring it to be enabled in the license.

## XML

XML has traditionally been used as a data interchange format between different systems and applications. XML, being a flexible way to represent information, has also seen use as format for long term data storage, which is represented by TRIP and other XML database managers.

## JSON

JSON, even more so than XML, is a format for data interchange and message passing. It has since its inception also largely replaced XML in structured storage as is evident from its status as defacto standard format for NoSQL document stores. From version 8.0 of TRIP, storage of JSON documents is also supported.

## Installation and Upgrade

The JSON and XML functionality is part of the TRIPsystem installation, and does not require a separate installation step.

When upgrading from a pre-8.0 version of TRIP where TRIPxml is in use, that version of TRIPxml will no longer be used, even if re-installed. Old TRIPxml installations can be removed after a successful upgrade to TRIP 8.0 or later.

**NOTE:** Upgrading a TRIP installation where a TRIPxml version 1.x is in use is not supported. You should in such cases first upgrade to TRIPxml version 2 or 3 (on a TRIPsystem 6.x or 7.x installation) and migrate your XML databases as per the TRIPxml documentation. When you have verified that your XML databases are working with the new version of TRIPxml, you can proceed to upgrade TRIPsystem.



## Features

### Overview

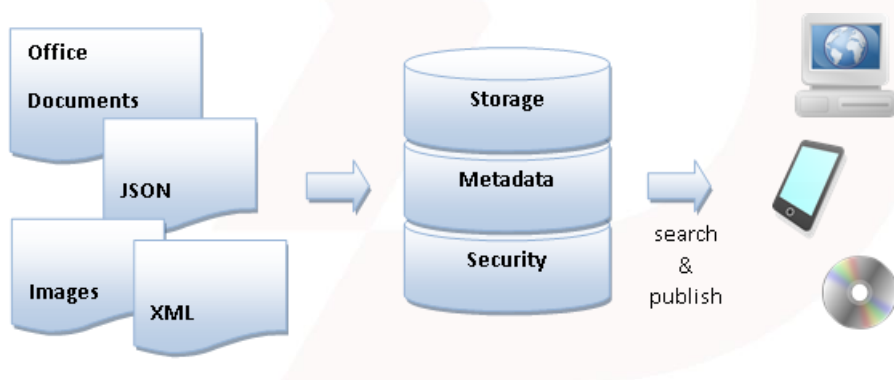
The focus of XML and JSON in TRIP is the storage of documents with a fair amount of textual data. The following points illustrate the related essential core features:

- Ability to represent and store XML and JSON documents with their complete structure.
- One single TRIP database design supports all kinds of XML and JSON documents, plus any kind of unstructured document or file.
- Support TRIP queries to find XML and JSON documents and sections within the documents.
- TRIP queries against a JSON/XML database can utilize the structure of the stored data.
- Optional storage of related DTD or schema for validation and stylesheet files for rendering of XML documents

### Design

#### Overview

Although this feature set has been through some fairly major internal changes since its first version, the basic design remains. It allows using TRIP as a hybrid XML and JSON database with storage, search and retrieval while also supporting unstructured data such as office documents and images.



Other JSON and XML features include:

- Programmatic access from both server (TRIP toolkit) and client (TRIPjxp, TRIPnpx, TRIPclient and TRIPjtk).
- Storage of any XML, JSON or other kind of file.
- Seamless integration with TRIPcof full text search enables stored unstructured documents (e.g. office files) to be searchable.
- Use XPath as query language across the entire document set of JSON and XML documents.

## Storage Structure

JSON and XML documents are basically structured like trees, which is also how their data is represented in TRIP.

The storage structure for JSON/XML databases is slightly different in TRIP 8 than in previous releases of the add-on product TRIPxml. JSON/XML databases created in this version of TRIP are therefore not backward compatible with older versions of TRIP that use TRIPxml, although old database versions from TRIPxml 2.x and 3.x can also be used with TRIP 8 (albeit without support for JSON documents).

The table below shows the minimal TRIP database design for any database that will store XML and JSON data. Each record in the database stores one document. The main differences in the current design compared to previous versions are **highlighted**.

XML documents can optionally also be stored as binary copies in the D\_XMLDOC field. The reason for this redundancy was to improve retrieval performance for very large documents, although it is less of a concern in the current version.

Field Name	Field Number	Part Field	Field Type	Description
D_XMLDOC	1	No	STRING	Binary copy (XML only)
D_META	2	No	TEXT	Document metatags
D_DOCTYPE	3	No	TEXT	Document doctype tag
D_DOCSIZE	4	No	INTEGER	Size (in bytes) of XML document stored in D_XMLDOC
D_DOCNAME	5	No	PHRASE	File name of the document
D_URLBASE	6	No	PHRASE	URL to document on the web, excluding the document name itself.
D_URLALIAS	7	No	PHRASE	Alias for the URL
N_ID	8	No	INTEGER	Tree node identity
N_PARENT	9	No	INTEGER	Parent node identity
N_SEQPATH	10	No	PHRASE	Path to node using node sequence numbers.
N_PATH	11	No	PHRASE	Complete path to node
N_SEQNO	12	No	INTEGER	Sequence number of the node
N_NAME	13	No	PHRASE	Node name
N_TYPE	14	No	PHRASE	Node type (element, attribute, text, ...)
N_NAMESPACE	15	No	PHRASE	Node namespace
N_MIME	16	No	PHRASE	What data is stored in the node, e.g. image/jpeg
N_ENCODING	17	No	PHRASE	Type or representation of the node data, e.g. base64
N_CVALUE	18	Yes	STRING	Node value (binary)
N_TVALUE	19	No	1*TEXT	Node value (text only)
N_NVALUE	20	No	NUMBER	Node value (numeric)
N_DVALUE	21	No	DATE	Node value (date)
D_PROPNAME	22	No	PHRASE	Property name field
D_PROPVAL	23	No	PHRASE	Property value field
DAV_NONXMLBLOB	24	No	STRING	Field for storage of non-XML files.
D_DOCTEXT	25	No	TEXT	Field for storage of text extract of non-XML files.
D_ID	26	No	PHRASE	Document/record identity field
N_JSONTYPE	27	No	PHRASE	JSON node type
N_RESERVED	50	No	PHRASE	Max reserved field number



## XLink and XInclude support (removed)

The XLink protocol is an XML-based technique to create links between information on the web. XInclude is a small XML protocol that allows one XML file to include another. For more information on these protocols, see the World Wide Web Consortium's web site [www.w3.org](http://www.w3.org).

The XML functionality as available in the older add-on product TRIPxml supported these two protocols by maintaining a link database. This behavior is not available in the JSON/XML functionality as integrated with TRIPsystem. Please contact your TRIP distributor if you are interested in using this functionality or have an older TRIPxml installation using XLink and/or XInclude that you wish to upgrade.

## Querying and Search Results

Querying an XML or JSON document is like querying a tree. The following points describe the possibilities of queries against a JSON/XML database.

- Search for specific XML elements / JSON object members
- Search for information in specific XML elements
- Perform full text search on entire document content
- Boolean logic in search conditions (and, or, not)
- Use CCL or XPath

A search results from a JSON/XML database is a regular TRIP search set. When retrieving a document from such a set, the default is to return the entire document to the application. A single XML document containing selected fragments from each document in the search result is another possibility.

XPath is supported as query language for both XML and JSON documents. Using XPath instead of CCL makes querying XML and JSON data using their own structures more efficient than is possible by just using CCL. For more information on XPath in TRIP, refer to Appendix D - Supported XPath Syntax.

The following points summarize retrieval:

- Retrieval of entire documents
- Retrieval of fragments of documents found by a search. These fragments do not necessarily have to be the same parts that matched the query.
- Possibility to, no matter how the document was originally stored, retrieve the entire document as either JSON or XML.
- Possibility to get the words in the text that matched the query condition marked for highlighting (for XML output only).

For more information about search and retrieval of XML documents, please refer to the section "Query the database" in the "Howto" chapter.

## Application Programming

The API routines for programmatic access to JSON/XML functionality in TRIP have the following features:

- Search using XPath expressions or a simplified, abbreviated form of CCL that enables the use of element and member names as field names.
- Import routines to store an XML document into TRIP, with optional validation according to DTD or XML schema.
- Import routines to store a JSON document into TRIP.
- Retrieval routines for both entire XML documents as well as parts of the documents and related DTDs and stylesheets stored in the database.
- Access from TRIPsystem, TRIPnxp and TRIPjxp.
- Limited access from TRIPhighway and from the older SDKs TRIPclient and TRIPjtk.

## How-To

### Create a JSON/XML database

#### Via TRIPclassic

XML databases can be created using TRIPclassic.

Select Databases from the Administration menu. In the database menu, select create/modify database from the DB Design menu. Enter the name of the database to be created. Press '6' on the numeric keypad (or 'Ctrl-K' followed by '6') to bring up the special database options dialog. Set the XML database to 'Y'. Press ENTER (or 'Ctrl-E') to confirm, then finally press ENTER (or 'Ctrl-E') again to save the new database.

#### Via TRIPmanager

JSON/XML databases can also be created using TRIPmanager. Right-click the "Databases" node to bring up the context menu. Select the "New Database..." option and click next. On the "general properties" page of the wizard, check the option "Database should be XML enabled".

When you get the question "Do you want to specify the field collection for this new database / thesaurus?" you should only answer "yes" if you wish to customize the default JSON/XML database design. If you do customize it, please do not change the definition for any of the pre-defined fields since this may render the database unusable.

#### Using jxtool

The jxtool command line program (documented in Appendix E) provides a server-side command line interface for basic administration of JSON/XML databases, including their creation. Use the --create option to create a database named via the -d option.

### JSON/XML Database Character Set

JSON/XML databases are Unicode-enabled by default. It is not recommended to change this setting.

Older versions of the TRIPxml add-on used whatever character set TRIPsystem was set up to use, even if the imported XML data happened to be in Latin-1 or gb2312. Unfortunately, this means that the only reliable way to get old XML data converted to a Unicode database without risk for corruption, is to re-import the XML documents into a new Unicode JSON/XML database.

### Query the Database

#### Using CCL

A JSON/XML database can be queried using CCL like any other TRIP database. There is one big difference, however. The difference is that with a JSON/XML database, you can pose a query using the element names of the stored documents. Note that if there is an element and a field with the same name in a JSON/XML database, the element will always be used.

For example, a database containing documents that use the CHAPTER element can be queried like this:

```
FIND CHAPTER=height
```

which will search for documents that have an element "CHAPTER", that contains – directly or in a sub element – the text "height".

In addition to using tag names in search conditions, as described above, you can use part of the path, like this:

```
FIND BOOK/CHAPTER=height
```

When you pose queries like this using CCL, you'll notice that TRIP expands the XML-oriented query to an ordinary CCL query.

## Using XPath

The recommended way to query JSON/XML databases is to use XPath. This is possible via the classes TdbSearch, TdbCclCommand and TdbRecordSet in TRIPjxp and TRIPnpx version 2.1 and later. Please refer to the "TRIPjxp & TRIPnpx Programmer's Guide" document for detailed information.

The XPath syntax supported is a subset of XPath 1.0 as specified by the W3C. See the appendix for a description of supported XPath syntax.

## Use the Toolkit API Calls

### JSON/XML C API Functions

The TRIPtoolkit contains eight special APIs for the interaction of JSON/XML databases:

- TdbPutXmlBuffer
- TdbPutJsonBuffer
- TdbPutXmlFile
- TdbPutJsonFile
- TdbGetXmlBuffer
- TdbGetJsonBuffer
- TdbExecuteXPath
- TdbGetXmlFragments

Please refer to the TRIPtoolkit Reference manual for API reference and usage examples.

## Backward-Compatibility Functions

The TRIPtoolkit API functions TdbImport and TdbExport were used to programmatically interact with TRIPxml from C/C++ prior to the release of TRIPsystem 8.0. Both of these functions use a structure called filter\_data. This structure is used for all parameters, including input options and return data. For a detailed description of this structure, see "Appendix A - API Reference".

Although there is nothing wrong with continued use of TdbImport and TdbExport, the regular C API calls (see above) should be used whenever possible when creating server-side TRIP Toolkit applications. The reason for this is that TdbImport and TdbExport are low-level functions and require a lot more care in usage than their ordinary counterparts.

## TdblImport

The TdblImport function is used to import XML and JSON documents.

### Prototype

```
int TdblImport ( filter_data* data );
```

Applicable values for the tdb\_options member:

NAME	VALUE	DESCRIPTION
IEOPT_FILENAME	1	The buffer member contains file name
IEOPT_FILEPTR	2	The buffer member contains file pointer (FILE*)
IEOPT_MEMORY	4	The buffer member points to memory area

Applicable mask values for the filter\_options member:

NAME	VALUE	DESCRIPTION
FOXML_NEWREC	1	Create a new record.
FOXML_REPLACE	2	The opposite of FOXML_NEWREC. Is implicitly set if FOXML_NEWREC is not set. The record_control and cursor fields are mandatory in combination with this option.
FOXML_VALIDATE	4	Specifies that the caller wishes to validate the XML document before it is imported.
FOXML_NOBLOB	64	Specifies that the caller does not wish to store a copy of the document in the D_XMLDOC field.
FOXML_STREAM	256	Specifies that the caller wishes to perform stream-oriented I/O.

URL information may be stored by using the filter\_arguments parameter in the filter\_data structure. Since this structure is generic, i.e. not specific to JSON/XML, each particular TRIP module using these functions may define different syntactical and semantical rules for the filter\_arguments parameter.

There are defines several filter arguments defined for use with JSON/XML. The terminating character for an argument line is a newline character. The line thus follows this syntax:

```
name = value '\n'
```

So, defining the URL <http://www.myweb.com/mysite/index.html>, the line will look like this:

```
URL=http://www.myweb.com/mysite/index.html
```

Please note that in the database, the URL is actually stored in two parts. The filename itself (e.g. index.html) is stored in the field D\_DOCNAME, and the other part of the URL (e.g. <http://www.myweb.com/mysite>) is stored in the field D\_URLBASE.

## Memory buffer import

Memory buffer import means that the caller provides all the data required in a memory area pointed to by 'buffer' in one go. The 'buffer\_length' specify the size of the entire document, and 'blockno' must be set to 0. The option for this is IEOPT\_MEMORY.

## File-oriented import

File-oriented import comes in two flavors. One in which a file name is specified in a character string pointed to by buffer, and one in which a file pointer is provided in buffer. In case a file pointer is provided, neither the TdblImport function, nor the filter function must close the file. The options are IEOPT\_FILENAME and IEOPT\_FILEPTR, respectively.

**NOTE:** The IEOPT\_FILEPTR is only supported on Linux/Unix. Attempting to use this import option on Windows may cause the application to turn unstable or crash.

### Stream-oriented import

Stream-oriented import can be used when importing large files into a JSON/XML database from a web site or a client application (based on TRIPjtk or TRIPclient). This is also the only way XML validation of system-local DTDs or schemas can be done when the DTD or schema is not available on the TRIP server machine.

To use streamed import, add FOXML\_STREAM to the filter\_options member of filter\_data. It is important also to assign either IEOPT\_FILENAME or IEOPT\_MEMORY to the tdb\_options member of filter\_data. You must also add a FILEURL parameter to the filter\_arguments string, specifying the URL from which you want TRIP to load the XML file, e.g. "FILEURL=http://mybox:1234/data.xml".

### Validation

If you want to validate an XML document when you are importing it, add FOXML\_VALIDATE to the filter\_options member of filter\_data. If XML\_SCHEMA is present and set to 1 in the server-side TRIPrcs file, then XML schemas can be validated. Otherwise, only DTDs are validated.

Note that if you are performing a blob-oriented import of an XML document referring to a DTD using a relative path and the DTD file has not been imported into the XML database, you will get a validation error.

### TdbExport

The TdbExport function is used to export JSON and XML documents, with optional hit markup for XML documents.

#### Prototype

```
int TdbExport ( filter_data* data );
```

Applicable values for the tdb\_options member:

NAME	VALUE	DESCRIPTION
IEOPT_FILENAME	1	The buffer member contains file name
IEOPT_FILEPTR	2	The buffer member contains file pointer (FILE*)
IEOPT_MEMORY	4	The buffer member points to memory area
EXPORT_ALLOC	32	Allocate memory for exported data (the buffer member is changed to point to newly allocated memory). Used with IEOPT_MEMORY.
EXPORT_FILEAPP	256	Declares that export filter should append XML document to a file. Used in combination with either IEOPT_FILENAME or IEOPT_FILEPTR.

Applicable mask values for the filter\_options member:

NAME	VALUE	DESCRIPTION
FOXML_GETBYID	8	Retrieve document by URL or record name (ID). Specify URL by adding the URL parameter to the filter_arguments member. Specify ID by adding the ID to the filter_arguments member.
FOXML_REMAKE	16	Recreate the xml document from its parts. Do not use data stored in the D_XMLDOC field.
FOXML_HILIGHT	32	Insert hit-markup in the extracted xml document, so the displaying application can highlight the hits. The option FOXML_REMAKE must also be set.
FOXML_STREAM	256	Specifies that the caller wishes to perform stream-oriented I/O.

### Memory buffer export

Memory buffer oriented export returns all the required data to the caller in memory via the 'buffer' field in one go. The 'buffer\_length' field specifies the size of the entire blob, and the 'blockno' must be set to 0. The option for this is IEOPT\_MEMORY.

This type of export also comes in another flavor, and that is that the filter routine allocates a large-enough buffer to contain the requested data. If this is the case, the filter routine will set buffer to allocated memory containing the data and buffer\_length to the size of the buffer. The option for this is IEOPT\_MEMORY | EXPORT\_ALLOC.

**NOTE:** If EXPORT\_ALLOC is used on Windows, the buffer returned in the 'buffer' field must be deallocated using the HeapFree() Win32 API function on the heap returned by the GetProcessHeap() Win32 API function. Using the regular free() function to release this memory will on Windows cause a crash or put the application into an unstable state.

### File-oriented export

File-oriented export comes in two basic flavors. One in which a file name is specified in a character string pointed to by buffer, and one in which a file pointer is provided in buffer. The options are IEOPT\_FILENAME and IEOPT\_FILEPTR, respectively. If combined with the EXPORT\_MKFILE option, the file name (or pointer) is created by the filter routine and returned in buffer. It will in this case not be necessary (or required) to provide anything in buffer by the caller.

For both types of file-oriented export applies that either EXPORT\_FILEAPP or EXPORT\_FILETRUNC must be used. No default action exists.

**NOTE:** The IEOPT\_FILEPTR is only supported on Linux/Unix. Attempting to use this export option on Windows may cause the application to turn unstable or crash.

### Stream-oriented export

Stream-oriented export is preferably used when exporting large files from a JSON/XML database to a client application (based on TRIPjtk or TRIPclient).

To use streamed import, add FOXML\_STREAM to the filter\_options member of filter\_data. It is important also to assign either IEOPT\_FILENAME or IEOPT\_MEMORY to the tdb\_options member of filter\_data. You must also add a FILEURL parameter to the filter\_arguments string, specifying the URL to which you want TRIPxml to upload the XML file with a HTTP POST message, e.g. "FILEURL=http://mybox:1234/data.xml".

## Client-Side API calls

### TRIPjxp and TRIPnxp

TRIPjxp and TRIPnxp version 8.0 provide the most complete APIs for development of TRIPxml applications. Please refer to the TRIPjxp and TRIPnxp documentation for details.

### TRIPclient (TRIPcom)

Users of the TRIPclient SDK can from version 2.5-0 use COM objects specific to import and export of XML documents. In addition, the Record object has since version 2.0-1 the XML-specific methods *CopyFromXMLFile* and *CopyToXMLFile*.

Please refer to TRIPclient documentation for details.

## TRIP Java Toolkit

TRIPjtk (now end-of-life) supported programmatic access of TRIPxml through its classes XMLRecord and XMLLink. Version 1.0 of TRIPjtk supported TRIPxml up to version 2.0, and version 1.1 and later of TRIPjtk supported TRIPxml up to version 3.1.





## Appendix A - API Reference

This appendix describes the TRIP toolkit APIs to use for low-level access of the JSON/XML functionality. The recommended way is to use the functionality integrated into TRIPnpx and TRIPjpx version 2.1 and later. TRIPnpx and TRIPjpx are documented separately.

### The filter\_data structure

This is the structure used by the API functions TdbImport and TdbExport:

Data Type	Member Name	In/Out	Description/Usage
TdbHandle	record_control	In/Out	The record control of the record into which an XML document is to be imported, or the record that contains an XML document that is to be exported. If the filter created a record control, it is returned in this member.
TdbHandle	cursor	In/Out	For imports; a record cursor. For exports; a cursor to the D_XMLDOC field. Is optional for exports. If the filter created a cursor, it is returned in this member.
TdbHandle	filter_address	In/Out	Contains a handle to the called API routine on return. This handle may be used on subsequent calls to boost performance. Not fully implemented in this version!
char	filter_name[32]	In	The name of the filter to call. For XML, the name of the import filter is "tripxmlput", and the export filter name is "tripxmlget".
char	filter_lib_env[32]	In	Not used with JSON/XML.
void*	buffer	In/Out	Field with many uses, dependent on what the tdb_options member say is contained herein. May be allocated memory, placeholder for allocated memory, name of file, or file pointer.
int	buffer_length	In/Out	Length of buffer.
char*	filter_arguments	In	Filter-specific string of arguments. Set to NULL or empty string if no arguments are passed.
int	arg_length	In	Length in bytes of the content of the filter_arguments member, including the terminating NULL-character.
int	filter_options	In	Filter-specific options.
int	tdb_options	In	Import/export specific options.
int	blockno	In	Block number (only for block-oriented i/o – see below). Set to 0 for final block, 1 for first in a sequence of several, etc.
int	errorcode	Out	Filter-specific error code (may also be informational or warning).
char	errortext[256]	Out	Textual, filter-specific, error message.



Valid values for the filter\_options member

Filter Option Name	Value	Description
FOXML_NEWREC	1	TdbImport: Create a new record.
FOXML_REPLACE	2	TdbImport: The opposite of FOXML_NEWREC. Is implicitly set if FOXML_NEWREC is not set. The record_control and cursor fields are mandatory in combination with this option.
FOXML_VALIDATE	4	TdbImport: Tells the XML parser to validate the XML record if any internal/external DTD subset have been seen.
FOXML_GETBYID	8	TdbExport: Retrieve document by URL or record name (ID). Specify URL by adding the URL parameter to the filter_arguments member. Specify ID by adding the ID to the filter_arguments member.
FOXML_REMAKE	16	TdbExport: Recreate the xml document from its parts. Do not use data stored in the D_XMLDOC field.
FOXML_HILIGHT	32	TdbExport: Insert hit-markup in the extracted xml document, so that the displaying application can highlight the hits. The option FOXML_REMAKE must also be set.
FOXML_NOBLOB	64	TdbImport: Do not store the original document.
FOXML_STREAM	256	Specifies that the caller wishes to perform stream-oriented I/O.

Valid values for the tdb\_options member

Tdb Option Name	Value	Description
IEOPT_FILENAME	1	The buffer member contains file name.
IEOPT_FILEPTR	2	The buffer member contains file pointer (FILE*).
IEOPT_MEMORY	4	The buffer member points to memory area.
EXPORT_ALLOC	32	TdbExport: Declares that export filter should allocate memory for data (the buffer member is changed to point to newly allocated memory). Used in combination with IEOPT_MEMORY.
EXPORT_FILEAPP	256	TdbExport: Declares that export filter should append XML document to a file. Used in combination with either IEOPT_FILENAME or IEOPT_FILEPTR.

## Appendix B - Encodings

### Supported Encodings for XML Documents

Via the integration to the ICU library, most major encodings are supported for XML documents including:

- ASCII
- UTF-8
- UTF-16 (Big/Small Endian)
- UCS4 (Big/Small Endian)
- EBCDIC code pages
- GB2312 and BIG5
- IBM037 and IBM1140 encodings
- ISO-8859-1 (aka Latin1)
- Windows-1252

For a more complete list of encodings, see the ICU homepage <http://icu-project.org> or IANA's list of character set names at <http://www.iana.org/assignments/character-sets>.

Even though the integration with the ICU library provides many alternatives to choose from, the best choice in most cases is either UTF-8 or UTF-16. Advantages of these encodings include:

- *The best portability.* These encodings are more widely supported by XML processors than any others, meaning that your documents will have the best possible chance of being read correctly, no matter where they end up.
- *Full international character support.* Both utf-8 and utf-16 cover the full Unicode character set, which includes all of the characters from all major national, international and industry character sets.
- *Efficient.* Utf-8 has the smaller storage requirements for documents that are primarily composed of characters from the Latin alphabet. Utf-16 is more efficient for encoding Asian languages. But both encodings cover all languages without loss.

A second choice of encoding would be any of the others listed above. This works best when the xml encoding is the same as the default system encoding on the machine where the XML document is being prepared, because the document will then display correctly as a plain text file. For systems in countries speaking Western European languages, the encoding will usually be iso-8859-1.

A word of caution for Windows users: The default character set on Windows systems is windows-1252, not iso-8859-1. While this Windows encoding is recognized, it is a poor choice for portable XML data because it is not as widely recognized by XML processing tools. If you are using a Windows-

based editing tool to generate XML, check which character set it generates, and make sure that the resulting XML specifies the correct name in the encoding="..." declaration.

### Supported Encodings for JSON Documents

JSON documents must be encoded in UTF-8. The usage of any other character set is not supported.



## Appendix C – Settings for tdbb.conf

The following tdbb.conf settings under the nonprivileged section are specific to the JSON/XML functionality. All settings that have a valid value set of 0 and 1 are boolean – i.e. 0=off, 1=on.

Setting	Valid Values	Default Value	Description
XML_LINKS	0, 1	0	Link support
XML_LINK_DB	special	special	Link database name if other than XML_LINK_DB
XML_NAMESPACES	0, 1	1	Namespace support.
XML_SCHEMA	0, 1	0	Schema support
XML_STRICT_SCHEMA	0, 1	0	Strict schema validation
XML_STRICT_VALIDATION	0, 1	1	If documents are imported with validation, the strict validation means that the parser will load any external references required to validate the document such as DTD files, etc.
XML_NOENTITYRESOLVER	0, 1	0	<p>The entity resolver is used to redirect requests for DTDs and schemas to the proper location. E.g. if a DTD has been imported into an XML database, the entity resolver will export the DTD from there.</p> <p>Turning on this setting means that an entity resolver will not be created. This will break any import where validation is enabled and a DTD or XML schema file needs to be loaded!</p>

## Appendix D - Supported XPath Syntax

This appendix is an overview of the part of the XPath 1.0 syntax supported by TRIPsystem for queries against JSON/XML databases. Refer to the W3C (<http://www.w3.org/TR/xpath/>) for a complete description of XPath 1.0.

### Definitions

#### Node

A node is a node in the tree structure that makes up an XML document. There are several types of nodes, e.g. element, attribute and text;

```
<element attribute="">text</element>
```

In a JSON/XML database, every node is stored in a tuple (associated subfields across more than one field), with an optional associated text value in a part record with the same number as the tuple (subfield) number.

#### Node Set

A node set is what an XPath expression evaluates to. A node is usually an element, an attribute or text. Every node in a node set "knows" from which position in the document it comes, and can therefore be used as context for further XPath expressions.

In TRIP terminology, a node in a node set is a reference to a specific tuple, or subfield, in a record in a JSON/XML database.

#### Location Step

Every location step is an XPath expression in itself, and as such it has a context. A context is the XML node(s) relative to which the expression is to be evaluated. A bit like ".././somedir" in a file system is relative to the current directory.

A location step consists of the three parts *axis*, *node test* and *predicate* in the form `axis::nodetest[predicate]`.

1. An *axis* determines the direction of the selection that the expression is to go, relative to context. The most common axis type is "child". The child axis says that the expression matches nodes that has the context node (or nodes) as direct parents.
2. A *node test* makes a selection of nodes from the specified axis. A node test can be the name of an element, a node type, a wildcard, etc.
3. A *predicate* is a filter of sorts. It further limits the set of nodes that matches the location path expression. This filtering can be made in many different ways. A common one is to specify specific attributes and values of specific attributes. Only the nodes that matches the condition of the predicate will be included in the final node set for the current location step.

For example, in the expression:

```
child::para[position()=1]
```

*child* is the axis, *para* is the node test and *[position()=1]* is the predicate. In plain English, this means "select the first *para* element that is *child* to one of the nodes in the context node set".

## Axis Types

Axis	Description
child	Selects of nodes that are direct children to nodes in the context node set.
attribute	Selection of nodes that are attributes to (element) nodes in the context node set.
descendant	Selects nodes that have any of the nodes in the context node set as parent or ancestor. In other words, this selects the sub trees where the nodes in the context node set are roots.
ancestor	Selects nodes that are parents or ancestors to the nodes in the context node set.
ancestor-or-self	Selects nodes that are part of the context node set, or are parents or ancestors to nodes in the context node set.
descendant-or-self	Selects nodes that are part of the context node set, or have a node in the context node set as parent or ancestor.
self	Selects all nodes from the context node set.
following-sibling	Selects nodes that have the same parent as any of the nodes in the context node set, and are located after the context node in question.
preceding-sibling	Selects nodes that have the same parent as any of the nodes in the context node set, and are located before the context node in question.
parent	Selects nodes that are parents to nodes in the context node set.
following	Selects all nodes that follows the nodes in the context node set.
preceding	Selects all nodes that precedes the nodes in the context node set.

## Functions

The following functions are supported for use in predicates:

Function	Description
position()	<p>Evaluates to the position of a context node. Must be used as an lvalue in a comparison. For example:</p> <pre>//child::para[position()=1]</pre> <p>which also can be written as:</p> <pre>//child[1]</pre>

Function	Description
last()	<p>Evaluates to the last node in the context node set. Must be used as an rvalue in a comparison with position(). For example:</p> <pre>//child::para[position() = last() - 1]</pre> <p>Which selects the second-to-last node from the context node set.</p>
contains(nodeset,value)	<p>Selects nodes that have TEXT contents (as descendant nodes) that contains the specified value. For example:</p> <pre>//sect1[contains(para, 'welcome')]</pre> <p>This expression selects sect1 elements that have para children that contains TEXT nodes in which the word "welcome" can be found.</p>

## Predicate Usage

### The existence of an attribute

TRIPsystem supports the following predicate syntax for checking if an attribute exists. These expressions are equivalent:

```
[@ID]
```

```
[attribute::ID]
```

Applied to a location step, this limits the selected nodes to those that have the specified attribute ("ID" in this example).

### Attribute value check

The most common predicate is probably the one that checks the value of an attribute. These expressions are equivalent:

```
[@lang="EN"]
```

```
[attribute::lang="EN"]
```

Applied to a location step, this limits the selected nodes to those that have the specified attribute with the specified value.

### Exact text node contents

Whether or not to use an attribute or a text node under an element to represent a particular value is in many cases not a clear-cut choice. The equivalent of doing an attribute value check for the text content of an element is this:

```
[. = "Enterprise"]  
[text() = "Enterprise"]
```

Applied to a location step, this limits the selected nodes to those that have exactly the specified value as their text contents. No truncation is performed.

### Truncated text node contents

The only XPath function supported by TRIPsystem that can perform a truncated search is the `contains()` function. If there is a text node that contains "they welcomed us to their party", then we can use this predicate expression:

```
[contains(., "welcome")]  
[contains(text(), "welcome")]
```

Applied to a location step, this limits the selected nodes to those that have exactly the specified value as their text contents. No truncation is performed.

### Truncated attribute text node contents

A variant on the truncated text node contents, we replace the first argument of the `contains` function with an expression that evaluates to an attribute.

```
[contains(@name, "TRIP")]  
[contains(attribute::name, "TRIP")]
```

Applied to a location step, this limits the selected nodes to those that have an attribute "name" whose value contains "TRIP". So, we would for instance be able to find "TRIPjxp" this way.

### Comparison Operators

All the six normal comparison operators are supported; `=`, `!=`, `>`, `<`, `>=`, and `<=`.

Note that comparison is always text based. This means that a non-equi comparison with numerical data may not yield the expected results.

### Node position

Perhaps more useful in fragment retrieval than in querying, limiting the node set by node position is also possible with TRIPsystem. The functions `position()` and `last()` can be used here, as well as the abbreviated form of just using a number



Selecting a node at a specific position from the context node set, the following expressions are equivalent:

```
[position()=1]
```

```
[1]
```

Selecting a range of nodes:

```
[position() < 10]
```

Selecting The last node in the set:

```
[position() = last()]
```

```
[last()]
```

Verifying that the context node set has a certain size (here exactly 2 nodes):

```
[last()=2]
```

Selecting nodes at the end of the set, e.g. the last two ones:

```
[position() >= last() - 1]
```

## Multiple predicates in a single location step

The keywords "and" and "or" can be used within a predicate. Multiple predicates can be listed one after the other within the same location step, in which case there is an implicit "and" operation between each predicate.

This combination exemplifies both variants:

```
[@type="S" or contains(., "TRIP")][position()<10]
```

This would select the first nine elements that either have an attribute "type" having the value "S" or contains the text "TRIP".

## Additional Examples

### Expression over Axis "child"

The following expressions are equivalent:

```
/child::doc/child::sect1
```

```
/doc/sect1
```

### Expression with axis "attribute" in predicate

The following expressions are equivalent:

```
/child::doc/child::sect1/child::title[attribute::ID="chhist"]  
/doc/sect1/title[@ID="chhist"]
```

## Expression with axis "parent"

The following expressions are equivalent:

```
/descendant-or-self::node()/attribute::ID/parent::*  
//@ID/..
```

## Expression with multiple axes in a predicate

This selects the "para" elements that are located somewhere under a "sect1" element whose direct child element "title" has an attribute "ID" with the value "chhist":

```
//para[ancestor::sect1/title/@ID="chhist"]
```

## Expression with axis "descendant" and multi-axis predicate

Here we want the "para" elements located somewhere under a "sect1" element whose direct child element "title" has an attribute "ID" with the value "chhist":

```
//sect1[title/@ID="chhist"]/descendant::para
```

## Expression with axes "ancestor-or-self", "parent" and "descendant"

This expression selects the "sect1" elements under which there is an element "title" whose attribute "ID" contains the text "hist". Furthermore, we want to make sure that there are "para" elements under the selected "sect1" elements, and that these "para" elements or one of their ancestors have an attribute "lang" with the value "EN".

```
//para/ancestor-or-self::node()[@lang="EN"]/  
descendant::title[contains(@ID,"hist")]/parent::sect1
```

Note: this is all a single, long line. The line break is for clarity only.

## Appendix E – The jxtool

This appendix describes the jxtool command line program provided with TRIPsystem as a simple CLI interface to manage and use JSON/XML databases and experiment with XPath based queries.

### Introduction

#### Purpose and Use Cases

While TRIPclassic and TRIPmanager can be used for administering JSON/XML databases, the jxtool allows such administrative tasks to also be done from the server-side command line. These tasks include creation and deletion of JSON/XML databases, and the granting and revoking of access rights to such databases to/from users and groups.

TRIP applications using JSON or XML data will normally use the APIs in TRIPjxp or TRIPnxp to store and retrieve such data. The jxtool is an alternative for scenarios that involve server-side importing and/or exporting of JSON and/or XML files.

The third functionality of the jxtool is its use as a simple, exploratory XPath query interface that can be used to query JSON/XML databases. The main intended use case for this is as an aide to develop and try out XPath query expressions for subsequent use in TRIPjxp or TRIPnxp based applications.

#### Command Line Interface

Argument	Description/Usage
-u username	The name of a TRIP user with which to authenticate.
-p password	The password of the specified TRIP user
-d database	The name of a JSON/XML database to administer or query
-g userOrGroup	Grant READ access to the specified database to this named TRIP user or group.
-G userOrGroup	Grant WRITE access to the specified database to this named TRIP user or group.
-x userOrGroup	Revoke access to the specified database from this named TRIP user or group.
-s location	Storage location for the named database. Used with the argument -create if the default TDBS_BASES location is not desired.
-f filename	Name of JSON or XML file to import or write, used with the arguments --get and --put.
-r rid	Record ID of JSON or XML record to retrieve.
-m format	Document format for import and export, used with the arguments --get and --put. Valid values are "json" or "xml".
-i [scriptfile]	XPath query script to execute. Omit the scriptfile argument value to read from stdin.
--create	Create a JSON/XML database
--drop	Delete a JSON/XML database
--get	Retrieve JSON or XML document, as specified using the -r rid argument.
--put	Store a JSON or XML document, as specified using the -f filename argument.
--split-array	Split an input JSON array into separate records (with --put)
--quiet	Run in quiet mode; no prompts or other messages.
--echo	Echo input statements to stdout. Used with the -i scriptfile argument.

## Administration

Create a JSON/XML database by authenticating as a user with file manager (FM) rights and specify the name of the database to create, optionally with a storage location.

```
jxtool -u username -p password -d myjsondb -s TDBS_BASES --create
```

Similarly, to remove a database:

```
jxtool -u username -p password -d myjsondb --drop
```

Granting read or write access to the database can be done at the same time as the database is being created:

```
jxtool ... -d myjsondb --create -G jsonupdaters -g public
```

Or separately, at a later stage:

```
jxtool ... -d myjsondb -G jsonupdaters -g public
```

Similarly, to revoke access from a database:

```
jxtool ... -d myjsondb -x public
```

Note that when removing a database, explicitly revoking access to it is not required since the removal operation will take care of also removing any and all access right records associated with the database.

## Importing and Exporting XML and JSON documents

The jxtool utility can be used to perform server-side imports and exports of JSON and XML documents, one document at a time. While not designed for bulk operation it can be used in similar scenarios.

To import an XML document:

```
jxtool ... -d myjsondb --put -m xml -f mydoc1.xml
```

To import a JSON document:

```
jxtool ... -d myjsondb --put -m json -f mydoc1.json
```

Omitting the `-m` argument causes jxtool to try to determine the file format itself.

```
jxtool ... -d myjsondb --put -f examinethis.json
```

Regardless of source format, the file is stored in such a manner that the same record can be exported as either XML or JSON. For example, assuming that the JSON or XML file to export resides in record 42:

```
jxtool ... -d myjsondb --get -r 42 -m xml -f retrieved.xml
```

```
jxtool ... -d myjsondb --get -r 42 -m json -f retrieved.json
```

## Bulk Import

Importing a JSON document that at its root is an array, jxtool can via the `--split-array` argument be told to import each element in this top-level array separately. A JSON document with an array holding ten objects, will therefore result in ten records being added.

This kind of bulk import requires that the array elements are either objects or arrays themselves. If the top-level array contains elements of other types, the import will fail with an error message saying “unable to split JSON array containing atomic values.”

For example, if we have a JSON document with the following contents:

```
[
  {
    "id":1,
    "first_name":"Jeanette",
    "last_name":"Penddreth",
    "gender":"Female"
  },
  {
    "id":2,
    "first_name":"Giavani",
    "last_name":"Frediani",
    "gender":"Male"
  },
  {
    "id":3,
    "first_name":"Noell",
    "last_name":"Bea",
    "gender":"Female"
  },
  {
    "id":4,
    "first_name":"Willard",
    "last_name":"Valek",
    "gender":"Male"
  }
]
```

We can import it into four separate records using jxtool:

```
> jxtool ... -d myjsondb --put --split-array -f bulk_example.json
```

```
Found 4/4 valid elements
Sub-document #1 imported as record #1
Sub-document #2 imported as record #2
Sub-document #3 imported as record #3
Sub-document #4 imported as record #4
```

## XPath Based Querying

Starting jxtool with no other arguments than username, password and database name, causes it to enter into a simple XPath query mode. The following commands are available here:

Command	Description/Usage
HELP;	Show list of available commands
BYE;	Logout and exit the jxtool
BASE database;	Open different JSON/XML database
XGET query fspc;	Execute an XPath query and fetch the specified fragments
XFIND query;	Execute an XPath query
[NO]EXECUTE;	Toggle execution of parsed XPath statements.
[NO]HIGHLIGHT;	Toggle highlighting of results

The XFIND command can be used to query JSON/XML databases. The XGET command extends the XFIND by also allowing the retrieval of a fragment set, where the data to fetch from each found record is defined by a second XPath statement.

