software AG

Tamino

Tamino API for Java

Version 10.1

April 2018

WEBMETHODS

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

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

Document ID: TAJ-DOC-101-20180413

Table of Contents

Tamino API for Java

This document provides information about the Tamino API for Java. This API is completely object-oriented, has a very flexible design, and allows convenient access to data stored in the Tamino XML Server. It is implemented in Java.

This document is for software developers who want to create applications on the basis of data stored in the Tamino XML Server.

Release Notes	The latest features and modifications since the last release.
Introduction	What is the Tamino API for Java? A brief summary.
Tamino API for Java Component Profile and Set-up	The components offered by the Tamino API for Java, for manipulating data in a Tamino XML Server.
Say Hello!	A very simple example that inserts a friendly greeting into an existing database.
Architectural Overview	An overview that explains the general architecture.
Doing More with the API	Further examples that show you how to use the Tamino API for Java.
Tamino API for Java: Javadoc Reference Information	The reference documentation for the Tamino API for Java in Javadoc format.
Object Model: Javadoc Reference Information	The reference documentation for the JDOM object model in Javadoc format.
Webserverless Access via the Tamino API for Java	How to access Tamino through the Tamino API for Java without a web server.
Performance Tips and Tricks	Useful hints to streamline a Tamino API for Java application's design.
Measuring Operation Duration	Find out where your application spends most of its execution time.
Examples in Code	An overview of references to complete code listings of the examples discussed in this documentation.

I Release Notes

1 Release Notes

This document contains an overview of the features of the Tamino API for Java that have been introduced or modified since the original release with Tamino V.3.1.

Binary Compatibility

All programs that ran under version 4.1 should run under version 8.x, with the following exception:

- Exceptions have introduced a new parameter (`Locale`) in their constructors.

Global Transactions

Version 8.x supports global transactions using the two-phase commit (2PC) concept. In order to use this feature, call the overloaded `useGlobalTransactionMode()` method of the `TConnection` interface.

Secured Database Connections

Version 8.x supports secure database connections, which can be created with the help of security tokens. The factory class `TConnectionFactory` has an overloaded method `newConnection()` that accepts SecurityTokens as parameter. In order to use this feature, the file *cstUtils.jar* must be added to the execution classpath. This file is normally located in *$INODIR\$INOVERS\SDK\TaminoAPI4J\lib*.

Prepared XQuery

The prepared query has now been simplified.

Old style:

```
declare variable $y:string as xs:string external for $q in
  input()/DataTypes[string=$y:string] return $q
```

New style:

```
declare variable $y as xs:string external for $q in
  input()/DataTypes[string=$y] return $q
```

The following code sample shows typical usage:

```
// Get tamino connection, use local transaction mode
connection = getConnection();
connection.useLocalTransactionMode();

// Get XML object accessor
accessor = connection.newXMLObjectAccessor();

// Prepare XQuery using connection object
TPreparedXQuery query = connection.prepareQuery("declare variable $y as xs:string ↩
external for $q in
 input()/DataTypes[string=$y] return $q");
// Bind external variables
query.bindString(new QName("y"),"String");

// Execute prepared XQuery
TResponse response = accessor.xquery(query);
```

Support for Internationalization and Localization

A connection can now be created by specifying a locale. See the Javadoc of `TConnectionFactory` for further details.

Exceptions

All exception classes that are inherited from `TException` can use Local to specify internationalized messages. The class hierarchy of the `TException` interface can be found at javadoc/JavaTaminoAPI/com/softwareag/tamino/db/api/common/TException.html

Note: Existing programs that were written to run with earlier versions of Tamino and throw exceptions in their code or custom exceptions derived directly or indirectly from `TException` will cause compilation errors if used with Tamino version 8.x.

Use of New Serialization Pragma

With version 4.4, Tamino enables you to customize the presentation of the result of an XQuery. With this new facility Tamino XQuery can manage both first-rate database querying and front-end publishing in a single step, thus making the handling of XML data even easier and faster than ever. The output of such an XQuery was always serialized using a standardized response wrapper.

The `TXMLObjectAccessor` will handle wrapped or unwrapped XML results and `TNonXMLObjectAccessor` will handle nonXML results. In order to find out whether a query will return wrapped or unwrapped XML data, the accessor must know the serialization specification. The serialization specification can be specified as a part of an XQuery expression, or it can be set using setter methods of `TXQuery` class.

Code Sample

Retrieve response as plain text

```
TXMLObjectAccessor accessor = connection.newXMLObjectAccessor(
TAccessLocation.newInstance(COLLECTION), TDOMObjectModel.getInstance() );
TXQuery xquery = new TXQuery("for $q in input()/Account return $q");
xquery.setOutputMethod(TOutputMethod.TEXT);
TResponse response = accessor.xquery(xquery);
System.out.println("Response as a STRING -->" + response.getQueryContentAsString());
```

OR

```
TXMLObjectAccessor accessor = connection.newXMLObjectAccessor(
TAccessLocation.newInstance(COLLECTION), TDOMObjectModel.getInstance() );
TXQuery xquery = new TXQuery("{?serialization method='text' ↩
media-type='text/plain'?}for $q in
input()/Account return $q");
TResponse response = accessor.xquery(xquery);
System.out.println("Response as a STRING -->" + response.getQueryContentAsString());
```

Retrieve response as nonXML document

```
TNonXMLObjectAccessor accessor = connection.newNonXMLObjectAccessor(
TAccessLocation.newInstance(COLLECTION));
TXQuery xquery = new TXQuery("declare namespace tf =
\"http://namespaces.softwareag.com/tamino/TaminoFunction\" " +
"declare namespace ino= \"http://namespaces.softwareag.com/tamino/response2\" " +
"input()/ino:nonXML[tf:getInoId(.)=5]/..");
xquery.setOutputMethod(TOutputMethod.INODOCUMENT);
TResponse response = accessor.xquery(xquery);
//Get inputStream from response.
TInputStream inputStream = response.getQueryContentAsStream(); ↩
```

Use of Cursor Parameter _scroll

Until the previous release, the `_scroll` parameter had always been set to "yes", even if the user did not intend to move backward in a cursor. The current implementation optionally enables a user to set the `_scroll` parameter to "no", in order to gain better performance.

Code Sample

Using XML Accessor

```
xmlAccessor.setScrollType(TScroll.NO);
TResponse result = streamAccessor.query(query, 5);
```

Using Stream Accessor

```
streamAccessor.setScrollType(TScroll.NO);
TInputStream inputStream = streamAccessor.openCursor(TQuery.newInstance(docType), ↩
1, 5);
```

Use of _maximumrequestduration Parameter

In Tamino, the response time of a single request depends on a number of factors, e.g. the presence of indexes, locks of concurrent transactions, amount of data to be searched or sorted, etc. Thus, an application can be blocked for a considerable time because a single request takes quite some time. In particular, similar requests might have performed well in the past. Often, such a situation is resolved only by timeouts e.g. in the HTTP layer or the transaction timeout in Tamino. On the other hand, a user might issue a request that – intentionally or unintentionally – consumes a lot of Tamino resources and blocks other requests.

Until now the XML maximum query duration property was available with Tamino. But this property is applicable for XQuery and X-Query only. A new parameter, `_maximumrequestduration`, has been introduced to limit the maximum elapsed time for all requests to a database.

Code Sample

```
// Get Accessor object to execute requests.
TXMLObjectAccessor accessor = connection.newXMLObjectAccessor(
  TAccessLocation.newInstance(COLLECTION), TDOMObjectModel.getInstance());

// Set maximum request duration in seconds.
accessor.setMaximumRequestDuration(2);
```

Cancel Requests

Currently it is not possible to stop requests sent to Tamino through the Tamino API for Java. Once started, a query executes (i.e., consumes CPU cycles, main memory, and I/O capacity) until all results are delivered or until an error condition occurs. The user has no chance to cancel a long running request (and thus stop resource consumption).

In the current version of Tamino, the server provides this feature. Application-owned requests can be cancelled using the Tamino API for Java.

Code Sample

```
/*
 * In order to cancel any request, it is required to set as cancellable.
 * It is recommended to set the client application name for ease of tracing / ↩
debugging.
 */
accessor.setCanBeCancelled(true);
accessor.setApplicationName("myApplication");
//Execute requests (XQuery, X-Query, process, define etc) using accessor
//... ...

/* In order to cancel above request, client application need to call cancelRequests()
 * on same accessor object concurrently.
 */
accessor.cancelRequests();

// Note that cancelRequests() will cancel all active requests issued using same ↩
accessor object.
```

Xquery Prepare and Execute

In order to separate the compilation of an XQuery from its execution, Tamino 4.4 provides an interface for preparing and executing queries. Two new commands have been introduced: `_prepare` and `_execute`.

To support this feature in Tamino API for Java , the class `com.softwareag.tamino.db.api.accessor.TPreparedXQuery` has been implemented.

The lifetime of a prepared query is determined by a connection (session). Therefore the `TConnection` interface provides a factory method for creating an instance of `TPreparedXQuery`. Note that a prepared query can only be used in the context of a session. This means that `LocalTransactionMode` must be used when working with prepared queries. A prepared query can be executed using an XML object accessor.

In order to pass parameters to a query, XQuery provides external variables. An external variable gets its value from the evaluation context (client application). The Tamino API for Java provides several external variable binding APIs.

Code Sample

```
// Get tamino connection, use local transaction mode
connection = getConnection();
connection.useLocalTransactionMode();

// Get XML object accessor
accessor = connection.newXMLObjectAccessor();

// Prepare XQuery using connection object
TPreparedXQuery query = connection.prepareQuery("declare variable $y:string as ↩
xs:string external for $q in
 input()/DataTypes[string=$y:string] return $q");
// Bind external variables
query.bindString(new QName("http://www.softwareag.com","y","string"),"String");

// Execute prepared XQuery
TResponse response = accessor.xquery(query);
```

User Schema with Cyclic Imports

The Tamino API for Java now supports multiple or cyclic schema DEFINE and UNDEFINE operations.

Code Sample

Define Schema Cluster

```
//Get Tamino connection and schema definition accessor
TConnection connection = connectionFactory.newConnection(URI);
tsd3Accessor = connection.newSchemaDefinition3Accessor(TDOMObjectModel.getInstance());

//create schema documents as TXMLObjects
TXMLObject[] schemaObjects = new TXMLObject[2];
schemaObjects[0] = schemaObject1;
schemaObjects[1] = schemaObject2;
//define schema cluster
tsd3Accessor.define(schemaObjects);
```

Undefine Schema Cluster

```
//Get Tamino connection and schema definition accessor
TConnection connection = connectionFactory.newConnection(URI);
tsd3Accessor = connection.newSchemaDefinition3Accessor(TDOMObjectModel.getInstance());

//create undefined items
TUndefineItem[] undefineItems = new TUndefineItem[2];
undefineItems[0] = new TUndefineItem("collectionName","schema1name"); ↩
```

Weaker Locking on Long-Lasting Index Access

A new parameter has been introduced in Tamino 4.4 to support weaker locking on long-lasting index access operations:

```
_QUERYSEARCHMODE={"approximative"|"accurate"|"nonserialized"}
```

Where:

"**accurate**"
represents the current behavior (this is the default);

"**approximative**"
specifies the new behavior;

"**nonserialized**"
is similar to mode approximative, but omits post-processing.

In order to support this feature in the Tamino API for Java, the new class `TQuerySearchMode` is introduced to represent the `_QUERYSEARCHMODE` parameter value.

The `TQuery` class represents the XQuery expression that is to be executed. To the send parameter `_QUERYSEARCHMODE` with XQuery, the following getter/setter APIs have been added in the `TQuery` class:

```
setQuerySearchMode(TQuerySearchMode mode)
TQuerySearchMode getQuerySearchMode()
```

You can find detailed information about the query search mode in the *X-Machine Programming* documentation.

Code Sample

```
TQuery query = new TQuery("Account");
query.setQuerySearchMode(TQuerySearchMode.APPROXIMATE);
TResponse response = accessor.query(query);
```

Domain Authentication

It is possible to specify a domain when creating a `TConnection` instance.

Isolation Levels

The Tamino XML Server uses an enhanced locking algorithm, providing several isolation levels. Each isolation level has different effects on parallel transactions. The interface `TConnection` in the package `com.softwareag.tamino.db.api.connection` provides access to the transactional parameters which define the behavior of Tamino when dealing with concurrent access of data. The parameters are:

```
LockwaitMode
IsolationLevel
LockMode
IsolationDegree
```

Additionally, a subset of the transactional parameters may also be defined with the accessors. For detailed information, see the Javadoc reference section in the online documentation.

Tip: The previous API version returned default values if the transactional parameters were not changed explicitly. Now null values indicate the Tamino default values.

Schema definition

Methods in the interfaces `TSchemaDefinition3Accessor` and `TStreamAccessor` in the package `com.softwareag.tamino.db.api.accessor` allow you to specify a `TDefineMode`. This permits definition of a schema with validation. For detailed information, see the Javadoc reference section in the online documentation.

Explain Query

The class `TQueryBuilder` offers methods to build a `TQuery` instance for an "explain" query. It can be used to retrieve information about query execution for analysis and optimization.

Query Count

The interface `TXMLObjectIterator` in the package `com.softwareag.tamino.db.api.objectModel` has additional methods to get the total number of resulting objects for a query issued to the Tamino XML Server. This query count is only available for queries that have been issued using the `query` method, specifying a quantity (i.e. when the Tamino cursoring mechanism is being used). The Tamino XML Server only returns a query count if the calculation does not require the creation of the result set as a whole. In other words, Tamino XML Server only returns a query count if it is not expensive to calculate.

Tamino XQuery

The Tamino API for Java supports two query languages: Tamino X-Query and Tamino XQuery (note the hyphen).

XQuery is supported by the new classes `TXQuery` and `TXQueryBuilder` in the package `com.softwareag.tamino.db.api.accessor`. XQueries can be executed in the interfaces `TXMLObjectAccessor` and `TStreamAccessor`.

X-Query, the former query language of the Tamino XML Server, can still be used, e.g. for migration purposes.

Retry Handler

If a request to the Tamino XML Server fails, a retry handler will retry the request. In the case of a transaction timeout (session lost), the retry handler will set up a new session to the Tamino XML Server and associate it with the `TConnection` object. This is only possible if the last successful command was `commit` or `rollback`. For all other failures, the retry handler will retry the request up to 10 times, with an increasing time interval between successive retries.

You can disable retries by calling `TPreference.getInstance().setUseRetryHandler(false)` in the package `com.softwareag.tamino.db.api.common`.

Tip: If you create a new connection from the `TConnectionFactory` to determine whether a database is available, you should disable the retry handler before doing so. Failure to do so results in a long waiting time if the database is inaccessible.

Pre- and Postconditions

The exceptions `TPreconditionViolation` and `TPostconditionViolation` in the package `com.softwareag.tamino.db.api.common` are no longer part of the API. It is not recommended to catch pre- and postcondition violations thrown by the API in client components. Nevertheless, the API components may still throw runtime exceptions in the case of a contract violation.

Xerces V2.5.0

The Tamino API for Java comes with the Xerces V2.5.0 Java XML parser.

JDOM 1.0

The Tamino API for Java comes with JDOM 1.0.

General Entities

Previous versions of this API set the Apache-Xerces-specific feature *http://apache.org/xml/features/nonvalidating/load-external.dtd*. This feature simplified the parsing of XML documents, because the DTD was not searched and therefore no problems arose if the DTD did not exist or was non accessible. However, this caused general entities to be lost; they were neither translated nor kept.

In the current version of the API, the parser by default attempts to translate general entities.

If you want to maintain compatibility with older versions of the API, set the preference property: `TPreference.getInstance().setUseApacheLoadExternalDTD( false );`.

XML Documents: Encoding Information

The encoding information stored in the database currently gets lost. For `TXMLObjectAccessor`, this is because the underlying SAX parsers ignore the encoding information. For the `TStreamAccessor` implementation, this happens because no encoding information is recognized.

Connection Pooling

The Tamino API for Java offers the possibility to pool physical Tamino connections. For more information, see the reference documentation of the class `TConnectionPoolManager` in the package `com.softwareag.tamino.db.api.connection`.

List Databases

The Tamino API for Java offers the possibility to retrieve the list of available databases for a given Tamino XML Server. For more information, see the reference documentation of the class `TConnectionFactory` in the package `com.softwareag.tamino.db.api.connection` and in particular the methods `getDatabases()`.

II Introduction

2 Introduction

The Tamino API for Java offers an object-oriented programming interface to the Tamino XML Server. You can use it to write client applications that access and manipulate data stored in Tamino databases. As such it performs the same task as the HTTP Client API for Java, but does so in a more flexible way and is not restricted to a specific object model. You can use the Tamino API for Java with Tamino XML Server, Version 4.1.1.4 and later.

The Tamino API for Java:

- supports different models for accessing data in Tamino (DOM, JDOM, SAX, and stream);
- supports standards JAXP and DOM2;
- supports local and global (XA) transactions;
- introduces well-defined exception handling with localization support;
- hides the native interface and communication logic of the Tamino XML Server

The following section introduces you to the individual components of the Tamino API for Java. We then present a simple “Hello World” example with which you can become acquainted with programming the Tamino API for Java.

III Tamino API for Java Component Profile and Set-up

3 Tamino API for Java Component Profile and Set-up

The Tamino API for Java can be used to access and manipulate data stored in the Tamino XML Server as well as for applications which deploy a Tamino XML database.

Component Profile

Here you will find general information about the APIs and how to deploy them.

The Tamino API for Java is automatically installed if you choose the option "Complete" of the Tamino XML Server installation. For more information, refer to the installation section of the Tamino XML Server documentation.

Component Profile for the Tamino API for Java	
Supported Platforms	All platforms supported by Tamino XML Server.
Required Software	The Tamino API for Java can be used with all versions of the Tamino XML Server not older than version 2.3.1. Please check the documentation of the Tamino XML Server for the supported features. As prerequisite for developing and running applications using the Tamino API for Java you need a fully installed Java. The Tamino API for Java has been tested successfully using Java 7.
Location of Installed Component	*<TaminoInstallDir>/SDK/TaminoAPI4J* (henceforth called *<TaminoAPIDir>*)
Component Files	**Library - Tamino API for Java:** *<TaminoAPIDir>/lib/TaminoAPI4J.jar* *<TaminoAPIDir>/lib/TaminoAPI4J-l10n.jar* *<TaminoAPIDir>/lib/TaminoAPI4J-l10n_en.jar* **Note:** Installing additional language packs requires that the localized messages *.JAR* files of the corresponding locale are included: *<TaminoAPIDir>/lib/TaminoAPI4J-l10n_<locale>.jar* **Library - Additional *.jar* Files:** *<TaminoAPIDir>/lib/cstUtils.jar* *<TaminoAPIDir>/lib/utx.jar* *<TaminoAPIDir>/lib/xts.jar* *<TaminoAPIDir>/lib/javaJDOM.jar* **Tracing:** *<TaminoAPIDir>/lib/log4j.jar* Part of the bundled third-party software as mentioned below. **Examples:**

Component Profile for the Tamino API for Java	
	<TaminoAPIDir>/examples/JavaTaminoAPIExamples.jar
Bundled Software	In *<TaminoAPIDir>/lib* you will find the required versions of: Apache Xerces 2 (*xercesImpl.jar*, *xmlParserAPIs.jar*) JDOM (*jdom.jar*) *log4j.jar*

Important: The Tamino XML Server supports the new schema definition language based on XML Schema (XSD), whereas earlier versions support TSD4 and TSD3 (versions up to Tamino XML Server v4.4) or TSD2 (versions up to Tamino XML Server v2.3.1).

Deployment

To work with the Tamino API for Java

You need a running Tamino XML Server in order to deploy an API. If you want to access an existing Tamino database, you must know its URI (such as `http://localhost/tamino/mydb`).

1 Set your `CLASSPATH` environment variable so that it includes all the *.jar* files contained in the directory *<TaminoAPIDir>/lib*. For information on how to set an environment variable, please consult the documentation for your operating system.

2 Make sure that programs of the Java Runtime Environment such as the compiler `javac` or the interpreter `java` are either of version 1.3 or of version 1.4 . You can check the version by entering java -version.

To use the examples of the Tamino API for Java

- Examples that use the Tamino API for Java are packaged in the file *JavaTaminoAPIExamples.jar*. You can extract them into the current directory by using the `jar` utility:

  ```
  jar xf <TaminoAPIDir>/examples/JavaTaminoAPIExamples.jar
  ```

  The *.jar* archive expands to a directory hierarchy. In the subdirectory *com/software-ag/tamino/db/api/examples*, you will find the examples. The source code as well as class files are provided.

Supported Character Encodings

The Tamino API for Java supports the standard character encodings and their well known aliases, as shown in the following list. Please observe that Tamino XML Server may support other encodings than those listed here.

Encoding Name	Well known aliases
Big5	950, cp950, csBig5, ibm-1370_VSUB_VPUA, x-big5
cp850	850, csPC850Multilingual, IBM850
cp857	857, csIBM857
cp860	860, csIBM860, IBM860
cp861	861, cp-is, csIBM861, IBM861
cp862	862, cp867, cspc862latinhebrew
cp863	cp863, csIBM863, IBM863
cp864	csIBM864
cp865	865, csIBM865, IBM865
cp866	866, csIBM866
cp868	868, cp-ar, csIBM868, IBM868
cp869	869, cp-gr, csIBM869
EUC-JP	csEUCPkdFmtJapanese, eucjis, Extended_UNIX_Code_Packed_Format_for_Japanese, ibm-33722_VPUA, ibm-eucJP, X-EUC-JP
EUC-KR	csEUCKR, ibm-970_VPUA, ibm-eucKR, X-EUC-KR
gb18030	ibm-1392
GB2312	1383, chinese, cp1383, csGB2312, csISO58GB231280, EUC-CN, gb, gb2312-1980, GB_2312-80, ibm-1383, ibm-1383_VPUA, ibm-eucCN, iso-ir-58, X-EUC-CN
GBK	CP936, ibm-1386_VSUB_VPUA, MS936, zh_cn, windows-936
IBM01140	CCSID01140, CP01140, cpibm1140, ebcdic-us-37+euro
IBM01141	CCSID01141, CP01141, cpibm1141, ebcdic-de-273+euro
IBM01142	CCSID01142, CP01142, cpibm1142, ebcdic-dk-277+euro, ebcdic-no-277+euro
IBM01143	CCSID01143, CP01143, cpibm1143, ebcdic-fi-278+euro, ebcdic-se-278+euro
IBM01144	CCSID01144, CP01144, cpibm1144, ebcdic-it-280+euro
IBM01145	CCSID01145, CP01145, cpibm1145, ebcdic-es-284+euro
IBM01146	CCSID01146, CP01146, cpibm1146, ebcdic-gb-285+euro
IBM01147	CCSID01147, CP01147, cpibm1147, ebcdic-fr-297+euro
IBM01148	CCSID01148, CP01148, cpibm1148, ebcdic-international-500+euro
IBM01149	CCSID01149, CP01149, cpibm1149, ebcdic-is-871+euro
IBM037	cpibm37, ebcdic-cp-us, ebcdic-cp-ca, ebcdic-cp-wt, ebcdic-cp-nl, cp37, cp037, 037

Encoding Name	Well known aliases
IBM1026	CP1026, csIBM1026, Ibm-1026_STD
IBM273	273, CP273, cpibm273, csIBM273, ebcdic-de
IBM277	277, csIBM277, cpibm277, EBCDIC-CP-DK, EBCDIC-CP-NO, ebcdic-dk
IBM278	278, cp278, cpibm278, csIBM278, ebcdic-cp-fi, ebcdic-cp-se, ebcdic-sv
IBM280	280, CP280, cpibm280, csIBM280, ebcdic-cp-it
IBM284	284, CP284, cpibm284, csIBM284, ebcdic-cp-es
IBM285	285, CP285, cpibm285, csIBM285, ebcdic-cp-gb, ebcdic-gb
IBM290	cp290, csIBM290, EBCDIC-JP-kana
IBM297	297, cp297, cpibm297, csIBM297, ebcdic-cp-fr
IBM367	
IBM420	420, cp420, csIBM420, ebcdic-cp-ar1
IBM424	424, cp424, csIBM424, ebcdic-cp-he
IBM500	500, CP500, cpibm500, csIBM500, ebcdic-cp-be, ebcdic-cp-ch
IBM852	
IBM855	
IBM857	
IBM862	
IBM864	
IBM869	
IBM870	CP870, csIBM870, ibm-870, ibm-870_STD, ebcdic-cp-roece, ebcdic-cp-yu
IBM871	871, CP871, cpibm871, csIBM871, ebcdic-cp-is, ebcdic-is
IBM918	CP918, csIBM918, , ebcdic-cp-ar2, ibm-918_STD, ibm-918_VPUA
ISO-2022-CN-EXT	
ISO-2022-CN	
ISO-2022-JP	csISO2022JP
ISO-2022-KR	csISO2022KR
iso-8859-15	
ISO-8859-1	8859-1, cp819, csISOLatin1, IBM819, ISO_8859-1:1987, iso-ir-100, l1, latin1
iso-8859-2	8859-2, 912, cp912, csISOLatin2, ISO_8859-2:1987, iso-ir-101, l2, latin2
iso-8859-3	8859-3, 913, cp913, csISOLatin3, iso-ir-109, l3, latin3
iso-8859-4	8859-4, 914, cp914, csISOLatin4, ISO_8859-4:1988, iso-ir-110, l4, latin4
iso-8859-5	8859-5, 915, cp915, csISOLatinCyrillic, cyrillic, ISO_8859-5:1988, iso-ir-144
iso-8859-6	1089, 8859-6, arabic, asmo-708, cp1089, csISOLatinArabic, ecma-114, ISO_8859-6:1987, iso-ir-127
iso-8859-7	813, 8859-7, cp813, csISOLatinGreek, ecma-118, elot_928, greek, greek8, ISO_8859-7:1987, iso-ir-126
iso-8859-8	916, cp916, csISOLatinHebrew, Hebrew, 8859-8, ISO_8859-8:1988, iso-ir-138

Encoding Name	Well known aliases
iso-8859-9	8859-9, 920, cp920,latin5, csISOLatin5, ISO_8859-8:1989, iso-ir-148, l5
KOI8-R	cp878, cskoi8r, koi8
Shift_JIS	943, cp943, cp932, csShiftJIS, csWindows31J, MS_Kanji, pck, sjis, windows-31j, x-sjis
TIS-620	874, cp874, cp9066, ms874, windows-874
US-ASCII	ANSI_X3.4-1968, ASCII, ANSI_X3.4-1986, cp367, csASCII, ISO_646.irv:1983, ISO_646.irv:1991, ISO646-US, iso-ir-6, us
UTF-16BE	cp1201, UTF16_BigEndian, x-utf-16be
UTF-16LE	cp1200, UTF16_LittleEndian, x-utf-16le
UTF-8	cp1208, cp65001
UTF-16	csUnicode, ISO-10646-UCS-2, ucs-2
windows-1250	cp1250
windows-1251	cp1251
windows-1252	cp1252
windows-1253	cp1253
windows-1254	cp1254
windows-1255	cp1255
windows-1256	cp1256
windows-1257	cp1257
windows-1258	cp1258

Reference Documentation

The reference documentation for the Tamino API for Java. You will find the documentation about packages and classes/interfaces after the respective sections: the reference documentation for the Tamino API for Java follows next.

IV

4 Say Hello!

Before discussing the details of the API let us take a look at a very simple example that does nothing else than insert a friendly greeting into an existing database. The section **describing the Tamino API package** explains where you will find the example files used below; please refer also to the section "How To Create a Tamino Database" in the Tamino documentation. The following examples show you how to perform the most common operations on a database:

- establish a connection to a Tamino database
- obtain an access method to handle either XML or non-XML data
- perform common operations such as insert, update, delete, as well as querying a database
- use a local transaction mode
- close a database connection

Four examples are provided that demonstrate these tasks:

XMLGreeting: Inserting an XML Document

The class `XMLGreeting` shown below accomplishes the following tasks that are typical for an application accessing data stored in a Tamino database:

1. establish a connection to the existing Tamino database `mydb`
2. obtain an access method to handle XML data using the Document Object Model (DOM)
3. insert an XML document into the database
4. query the database to retrieve the document inserted previously
5. close a database connection

It is assumed that a database called `mydb` has already been created and is running. We use two string constants that hold the URI of the database and the XML instance to be inserted:

```
  // URI of the Tamino database, please edit accordingly
  public final static String DATABASE_URI = "http://localhost/tamino/mydb";

  // XML document to be written to the connected database
  public final static String XML = "<Greeting by='XMLGreetingApplication'>Hello ↩
World</Greeting>";
```

Please note that Tamino considers this as a valid XML document. Strictly speaking, this document satisfies all conditions of an XML document except that the XML declaration (`<?xml version="1.0"?>`) is missing.

We define `xmlObject` as an instance of `TXMLObject`. For representing the document structure, we choose DOM as one of several available object models. So `xmlObject` holds a DOM representation of that greeting document:

```
  // Put the XML content into a StringReader
  StringReader stringReader = new StringReader( XML );

  // Instantiate an empty TXMLObject instance using the DOM object model
  TXMLObject xmlObject = TXMLObject.newInstance( TDOMObjectModel.getInstance() );

  // Establish the DOM representation by reading the contents from the character ↩
input stream
  xmlObject.readFrom( stringReader );
```

Now we can go through the typical processing steps of a Tamino application:

1. **Establish a Connection**
   Use a factory method of `TConnectionFactory` to instantiate a `TConnection`. A `TConnection` object effectively represents a Tamino database session. You can use the `newConnection` method to establish a connection with a database that is specified by `DATABASE_URI`:

   ```
     // Establish the Tamino connection
     TConnection connection = TConnectionFactory.getInstance().newConnection( ↩
   DATABASE_URI );
   ```

2. **Obtain an Accessor**
   With the help of an Accessor object you can access the data that is to be read from or written to a database. Here we instantiate a new `TXMLObjectAccessor` object that is bound to the database connection established in the previous step. There are two parameters specifying the type of access: the first is the collection into which data is to be stored (`TAccessLocation`); here it is the default collection `ino:etc` that is always present in a Tamino database. The second is the object model, which is DOM in this example:

   ```
     // Obtain a TXMLObjectAccessor with a DOM object model
     TXMLObjectAccessor xmlObjectAccessor = connection.newXMLObjectAccessor(
                                         TAccessLocation.newInstance( "ino:etc" ),
                                          TDOMObjectModel.getInstance() );
   ```

3. **Insert the XML Document**
   The `xmlObjectAccessor` provides methods for performing queries and manipulating XML data such as inserting our `TXMLObject` instance into the default collection `ino:etc`. On insertion the XML object is assigned a unique ID (`ino:id`) that you can get by applying the object's method `getId`.

```
  // Invoke the insert operation
  xmlObjectAccessor.insert( xmlObject );

  // Show the ino:id of the document just inserted
  System.out.println( "Insert succeeded, ino:id=" + xmlObject.getId() );
```

4. **Query the Database**
   The `query` method of an `xmlObjectAccessor` expects as arguments a non-empty X-Query expression. We construct a query expression that retrieves the XML object stored with the previously assigned `ino:id` (which is 1 for the first item stored in a database): `/Greeting[@ino:id="1"]`. Here `Greeting` is the name of the document's doctype that is returned by the method `getDoctype()`. You can access the result set returned by Tamino with an instance of `TResponse`. The `getFirstXMLObject` method retrieves the first element of the result set

```
  // Prepare to read the instance
  TQuery query = TQuery.newInstance( xmlObject.getDoctype() + "[@ino:id=" + ↩
xmlObject.getId() + "]" );

      // Invoke the query operation
      TResponse response = xmlObjectAccessor.query( query );
      if ( response.hasFirstXMLObject() ) {
        StringWriter stringWriter = new StringWriter();
        response.getFirstXMLObject().writeTo( stringWriter );
        System.out.println( "Retrieved following instance:" + stringWriter );
      }
      else
        System.out.println( "No instance found!" );
```

5. **Close the Connection**
   The `TConnection` instance is also responsible for closing an open database connection. You can close the connection to the database `mydb` by using:

```
  connection.close();
```

The example file contains additional code for error handling which is not relevant for the discussion of the API basics.

Running the Example

You need to include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *xercesImpl.jar*, *xmlParserAPIs.jar*, *jdom.jar* and *log4j.jar* in your system's class path, all of which are included in the distribution of the Tamino API (see the **component profile**). Then you can run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.greeting.XMLGreeting
Insert succeeded, ino:id=1
Retrieved following instance:<Greeting by="XMLGreetingApplication" ino:id="1">Hello ↩
World</Greeting>
```

Note: Using *JavaTaminoAPIExamples.jar* only works if you have created a database called "mydb" on your local host. Otherwise you have to change the constant `DATABASE_URI` and recompile the code.

ProcessXMLGreeting: Operating in Local Transaction Mode

We extend and modify the example above and demonstrate how to operate in a local transaction mode. In addition to inserting data and querying the database, updating and deleting data are also demonstrated. We also use a different object model:

1. establish a connection to the existing Tamino database `mydb`
2. obtain an `Accessor` instance
3. insert a new document: `<Greeting></Greeting>`
4. The following tasks are performed in a local transaction mode:
   - retrieve first stored object
   - update objects in the database
   - commit the transaction if successful otherwise rollback
5. leave the local transaction mode
6. query the updated document
7. delete document
8. close the connection

We use the same constants as before.

1. **Establish a Connection**
   This is done the same way as in the previous example.
2. **Obtain a JDOM Accessor**
   The Accessor used here is instantiated in the same way as before, but in contrast to the first example it uses the JDOM object model.

```
  // Obtain the concrete TXMLObjectAccessor with an underyling JDOM object model
  accessor = connection.newXMLObjectAccessor( TAccessLocation.newInstance( ↩
collection ) ,
                                              TJDOMObjectModel.getInstance() );
```

3. **Insert the XML Document**
   Here we make use of `TResponse`. An object of that class is instantiated and stores the response from Tamino. The data will be inserted by calling the accessor's `insert` method.

```
  // TResponse represents an access response from Tamino.
  TResponse response = null;

  // Invoke the insert operation and obtain the response.
  response = accessor.insert( xmlObject );

  // Show the collection, doctype and id
  System.out.println( "Insert succeeded, ino:collection:" + ↩
xmlObject.getCollection() +
                      ", ino:doctype:" + xmlObject.getDoctype() + ", ino:id:" + ↩
xmlObject.getId() );
```

4. **Operate in a Transaction Block**
   We use a transaction block to query and update the database. Normally, every operation is running inside its own transaction block, and with the end of the operation the implicit transaction ends (*auto-commit*). Here we enter a local transaction mode, perform a query and an update operation in one transaction, which is finished by either a commit or rollback.

   a. **Open the Transaction Block**
      For entering the local transaction mode, we instantiate `TLocalTransaction` and call the `useLocalTransactionMode` method to bind it to the currently open connection.

```
  TLocalTransaction localTransaction = null;
  // Switch to the local transaction mode
  localTransaction = connection.useLocalTransactionMode();
```

   b. **Query the Database and Retrieve First Object**
      Again we use a `TResponse` object. You can retrieve the first stored object by calling `getFirstXMLObject`.

```
  // Invoke the query to obtain the document and to lock it
  TResponse response = accessor.query( xpath );

  // Obtain the TXMLObject from the response
  TXMLObject xmlObject = response.getFirstXMLObject();
  if ( xmlObject == null )
     return;
```

c. **Update the Database**
Update the current document after changing some text.

```
// Invoke the update
  response = accessor.update( xmlObject );
  System.out.println( "Update succeeded!" );
```

d. **Finish Transaction**
If every part of the transaction is successful, then commit the transaction:

```
localTransaction.commit();
```

If a `TAcessorException` is thrown, then a rollback is performed in the `catch` branch:

```
catch (TAccessorException accessorException)  {
    showAccessFailure( accessorException );
    localTransaction.rollback();
    throw accessorException;
  }
```

5. **Leave the Local Transaction Mode**
You must leave the local transaction mode by explicitly switching to the auto commit mode:

```
connection.useAutoCommitMode();
```

6. **Delete the Document**
As final operation we delete the second document:

```
TResponse response = accessor.delete( query );
System.out.println( "Deleted the document!" );
```

7. **Close the Connection**
This is done the same way as before.

Running the Example

You need to include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *xercesImpl.jar*, *xmlParserAPIs.jar*, *jdom.jar* and *log4j.jar* in your system's class path, all of which are included in the distribution of the Tamino API (see the **component profile**). Then you can run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.greeting.ProcessXMLGreeting
Insert succeeded, ino:collection:ino:etc, ino:doctype:Greeting, ino:id:2
Update succeeded!
Queried document:<Greeting ino:id="2" by="ProcessGreetingApplication">Hello World, ↩
updated :-)</Greeting>
Deleted the document!
```

Note: As in the previous example, using *JavaTaminoAPIExamples.jar* only works if you have created a database called "mydb" on your local host. Otherwise you have to change the source code and recompile it.

NonXMLGreeting: Inserting Non-XML Data

This example is nearly the same as the first example, but this time we want to insert a non-XML document. In the Tamino API a clear distinction is made between handling of XML documents and non-XML documents. Since the application logic is the same as before, we concentrate on the differences with respect to non-XML documents.

Again, a string constant will hold the document to be inserted into the database:

```
  // Non-XML document to be written to the connected database
  public final static String NON_XML = "Greeting by='XMLGreetingApplication':Hello ↩
World";
```

Please note that this document is a non-XML document, although it pretty much looks like the document used in the previous examples. It lacks any XML elements and thus it neither has a necessary root element nor can it be well-formed. It is a non-XML document used for an image or video file, or a text file that does not comply with the XML standard.

We define `nonXMLObject` as an instance of `TNonXMLObject`. In non-XML data there is no structural representation for which an object model could be used. The second argument is the name of the collection. Since it is `null`, the default collection `ino:etc` will be used. The next argument is the doctype name `ino:nonXML`, followed by the document name `NonXMLGreeting`. The last argument is the content type `text/plain`.

```
  // Put the XML content into a StringReader
  StringReader stringReader = new StringReader( NON_XML );

  // Instantiate an empty TNonXMLObject instance
  TNonXMLObject nonXMLObject = TNonXMLObject.newInstance( stringReader ,
                               null ,
                               "ino:nonXML" ,
                               "NonXMLGreeting" ,
                               "text/plain" );
```

1. **Establish a Connection**
   This is already described in the previous examples.
2. **Obtain an Accessor**
   We have to instantiate an object from `TNonXMLObjectAccessor`. As the data is non-XML an object model is not needed:

```
  // Obtain a TNonXMLObjectAccessor
  TNonXMLObjectAccessor nonXMLObjectAccessor = connection.newNonXMLObjectAccessor(
                                                  TAccessLocation.newInstance( ↩
"ino:etc" ) );
```

3. **Insert the Non-XML Document**
   There is a corresponding set of methods available for `TNonXMLObjectAccessor` instances so that inserting the non-XML document is analogous to inserting an XML document: it is also assigned a unique ID (`ino:id`) that you can retrieve after insertion:

```
  // Insert document and display its ino:id afterwards
  nonXMLObjectAccessor.insert( nonXMLObject );
  System.out.println( "Insert succeeded, ino:id=" + nonXMLObject.getId() );
```

4. **Query the Database**
   Analogous to the `XMLGreeting` example. However, we now use a `ByteArrayOutputStream` instead of a `StringWriter`:

```
  // Instantiate TQuery
  TQuery query = TQuery.newInstance( nonXMLObject.getDoctype() +
                                     "[@ino:id=" + nonXMLObject.getId() + "]" );
  // Invoke the query operation
  TResponse response = nonXMLObjectAccessor.query( query );
  if ( response.hasFirstNonXMLObject() ) {
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    response.getFirstNonXMLObject().writeTo( outputStream );
    System.out.println( "Retrieved following instance:" + outputStream );
  }
  else
    System.out.println( "No instance found!" );
```

5. **Close the Connection**
   This is done the same way as in the previous examples.

Running the Example

You need to include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *xercesImpl.jar*, *xml-ParserAPIs.jar*, *jdom.jar* and *log4j.jar* in your system's class path, all of which are included in the distribution of the Tamino API (see the **component profile**). Then you can run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.greeting.NonXMLGreeting
Insert succeeded, ino:id=1
Retrieved following instance:Greeting by='XMLGreetingApplication':Hello World
```

ProcessNonXMLGreeting: Operating in Local Transaction Mode with Non-XML Data

In this example the same tasks are performed as in `ProcessXMLGreeting`, except that non-XML data is used.

Running the Example

You need to include the same *.jar* files as before. Then you can run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.greeting.ProcessNonXMLGreeting
Insert succeeded, ino:collection:ino:etc, ino:doctype:ino:nonXML, ino:id:2
Update succeeded!
Queried document, Content:Greeting by='XMLGreetingApplication':Hello World
Deleted the document!
```

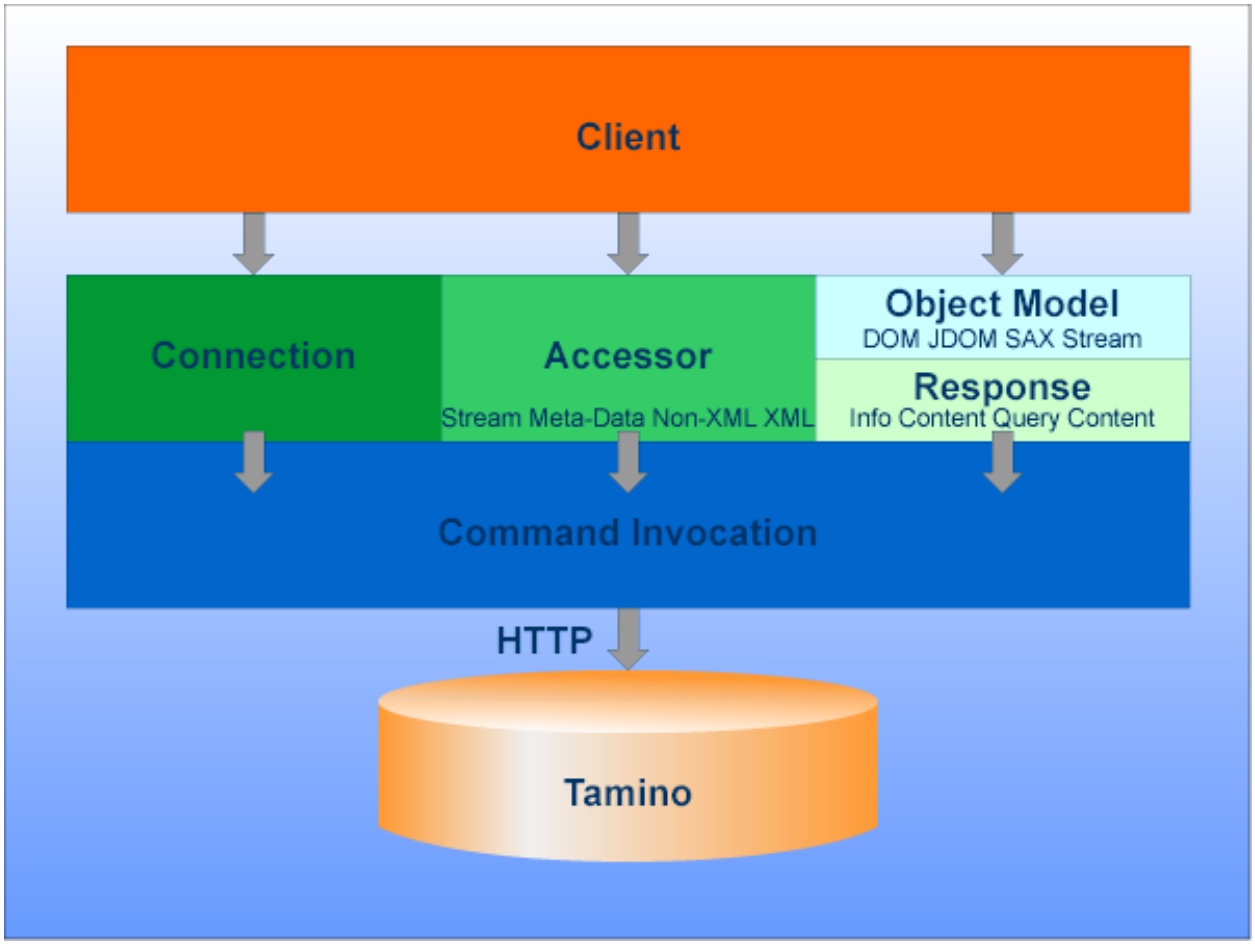
5 Architectural Overview

This chapter explains the general architecture of the Tamino API. The following sections are provided:

API Overview

This section introduces key concepts that are required for an understanding of the Tamino API architecture. The general concept follows a component and object-oriented development approach and is described here without referring to the features of a specific object-oriented programming language.

The following graphic shows the Tamino API architecture:



Tamino API Components

For the client application only the Connection, Accessor, Object Model and Response components are of interest. The Command Invocation component is only for internal use and is invisible to the client application.

Connection

The entry point to the API is the Connection component. It is needed to establish a connection to a Tamino database. Within the context of a Connection, Tamino specific services are executed and results are returned. An integral part of this component is to provide transaction support, i.e., the starting and finishing, by commit or rollback, of Tamino transactions.

Accessor

The Accessor provides high-level services to invoke specific operations on the Tamino database. This comprises insert, update, delete and query operations on Tamino XML documents as well as operations to access non-XML documents or meta data (e.g. what doctypes are stored under a specific collection).

In general, a distinction is made between access to XML or non-XML documents. The client uses an `Accessor` instance that is capable of accessing either XML or non-XML documents, but not both.

XML Object Model

This component describes how concrete documents that are storable in Tamino are represented. In general, XML and non-XML are treated differently. This is reflected by different classes within the object model. `TXMLObject` is the abstraction of XML documents whereas `TNonXMLObject` is used for the abstraction of non-XML documents.

Furthermore `TXMLObject` represents a facade to the underlying object model (DOM, JDOM, and others). This ensures a loose coupling between the API and the concrete XML object representation. Since each object model is represented by its own class, the client simply chooses the desired object model by instantiating the corresponding class such as `TDOMObjectModel`.

Response Builder

The role of this component is to build from low level results (XML or Non-XML documents within input streams) as they are returned by Tamino, high level response objects. Once the client has invoked a certain operation on an accessor, a high level Tamino response object is returned. The information that can be found in such a response object can be categorized as follows:

- **Response Information**
  Each Tamino response document contains general context information that is related to the previous request. This includes information such as return values, message codes, session ids, session keys etc. Generally, response information is always contained within a Tamino response document.
- **Query-related XML documents**
  XML documents that are returned in response to XML related queries are nested within Tamino's response documents. In this case the corresponding response object provides a `TXMLObjectIterator` that can be used to navigate in a type-safe manner over the resulting `TXMLObject` instances.
- **Query-related Non-XML documents**
  Non-XML documents are returned in response to non-XML related queries. In this case the corresponding response object provides a `TNonXMLObjectIterator` that can be used to navigate in a type-safe manner over the resulting `TNonXMLObject` instances.

Exception Handling

Each component that is part of the architecture is responsible for its own exception handling. Exceptions that can be thrown to the outside are always component specific. Therefore, the client only receives component specific exceptions during the interaction with the component.

Each component defines a base class exception that contains more detailed component specific exceptions. The base class exception's name consists of the component name with the suffix `Exception`. For example, the Connections component base class exception is named `TConnectionException`. Furthermore each of these base class exceptions is derived from the most common API exception `TException`.

However, the situation is different when pre- or postconditions are violated for operations that are invoked by the client. In the case of contract violation, an `ViolatedPrecondition` or `ViolatedPostcondition` exception is thrown by the affected component. These exceptions are applied for any type of pre- and postcondition violations.

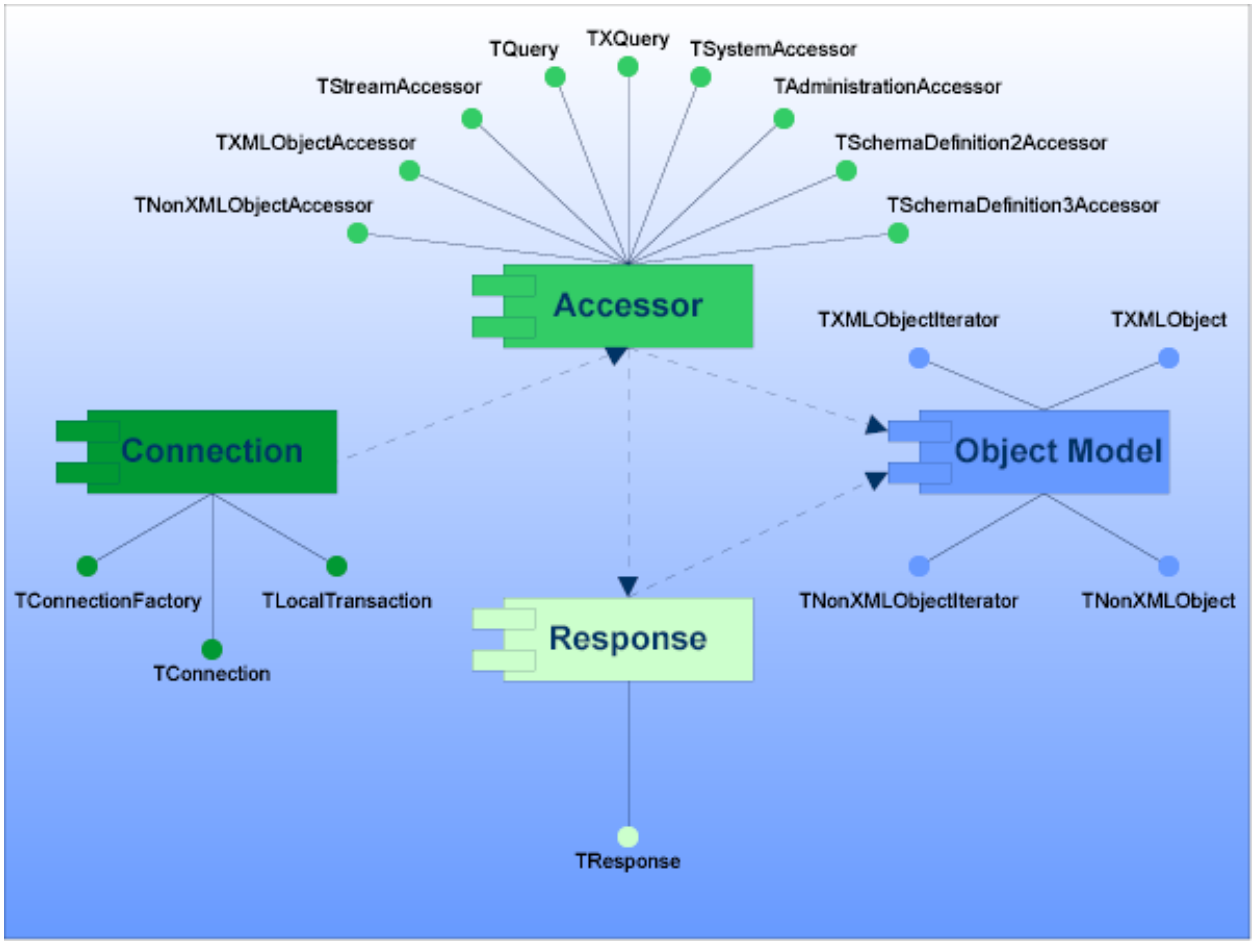
Each component can only throw either component specific exceptions or the API wide defined pre- and postcondition violation exceptions. It is not possible that a component directly throws an exception that originates from another internal client component. In these situations it is more

likely that an exception that might be thrown is carried piggyback so that the client can retrieve it.

Interfaces

This section introduces component-specific interfaces that the application developer has to be aware of for an in depth understanding of the API. The term interface here means any type of class that is part of a component's facade. According to UML this can be a concrete class, abstract class or an interface class which only defines a set of abstract operations. The interfaces a component offers can be distinguished in service oriented interfaces and exception specific interfaces. The first generally defines the services a component offers while the second defines the component's manner in exceptional conditions.

The UML component diagram shown below depicts the most relevant service interfaces that serve as facades to their owning components.



Please note that you use different classes when you process non-XML data instead of XML data:

XML Data	Non-XML Data
TXMLObjectAccessor	TNonXMLObjectAccessor
TXMLObject	TNonXMLObject
TXMLObjectIterator	TNonXMLObjectIterator

Connection

A client always gets access to the API by acquiring a concrete TConnection instance from a TConnectionFactory. TConnection introduces all operations that are needed to access the concrete accessor objects, so that this is the entry point to the Accessor component. Furthermore it provides access to the TLocalTransaction instance that is bound to the client's connection. TLocalTransaction contains all the operations to work with Tamino in a transaction context, which means that transactions can be started, committed or rolled back.

Accessor

The high level access to Tamino is accomplished by using a concrete TAccessor instance. A concrete accessor class implements the common TAccessor interface and provides a set of logically grouped access operations. Currently, there are the following concrete accessor classes: TNonXMLObjectAccessor for access of non-XML objects, TXMLObjectAccessor for access of XML objects, TStreamAccessor for the stream-based access of XML objects, TSystemAccessor for access to system information, TSchemaDefinition2Accessor for accessing database schema information in TSD2, TSchemaDefinition3Accessor for accessing database schema information in the current schema language, based on XML Schema, and TAdministrationAccessor for issuing Tamino XML Server _admin commands.

A TNonXMLObjectAccessor uses a given TNonXMLObject instance as the non-XML document representation for an insert or delete operation. TQuery is used for the retrieval of non-XML documents stored in Tamino. In the same way a TXMLObjectAccessor uses a given TXMLObject instance for an insert, update or delete operation. TQuery is the representation of a Tamino X-Query expression and is used for the XML document retrieval.

To obtain a concrete accessor, the client always has to use a TConnection instance that defines the database session with Tamino. Each accessor instance is logically bound to a specific database session. Once the client starts a transaction on the connection's TTransaction instance, the transaction context also affects all accessors that are currently running within the connection context. As a consequence a commit or rollback takes place on all accessor specific operations that have been invoked during the transaction context.

Object Model

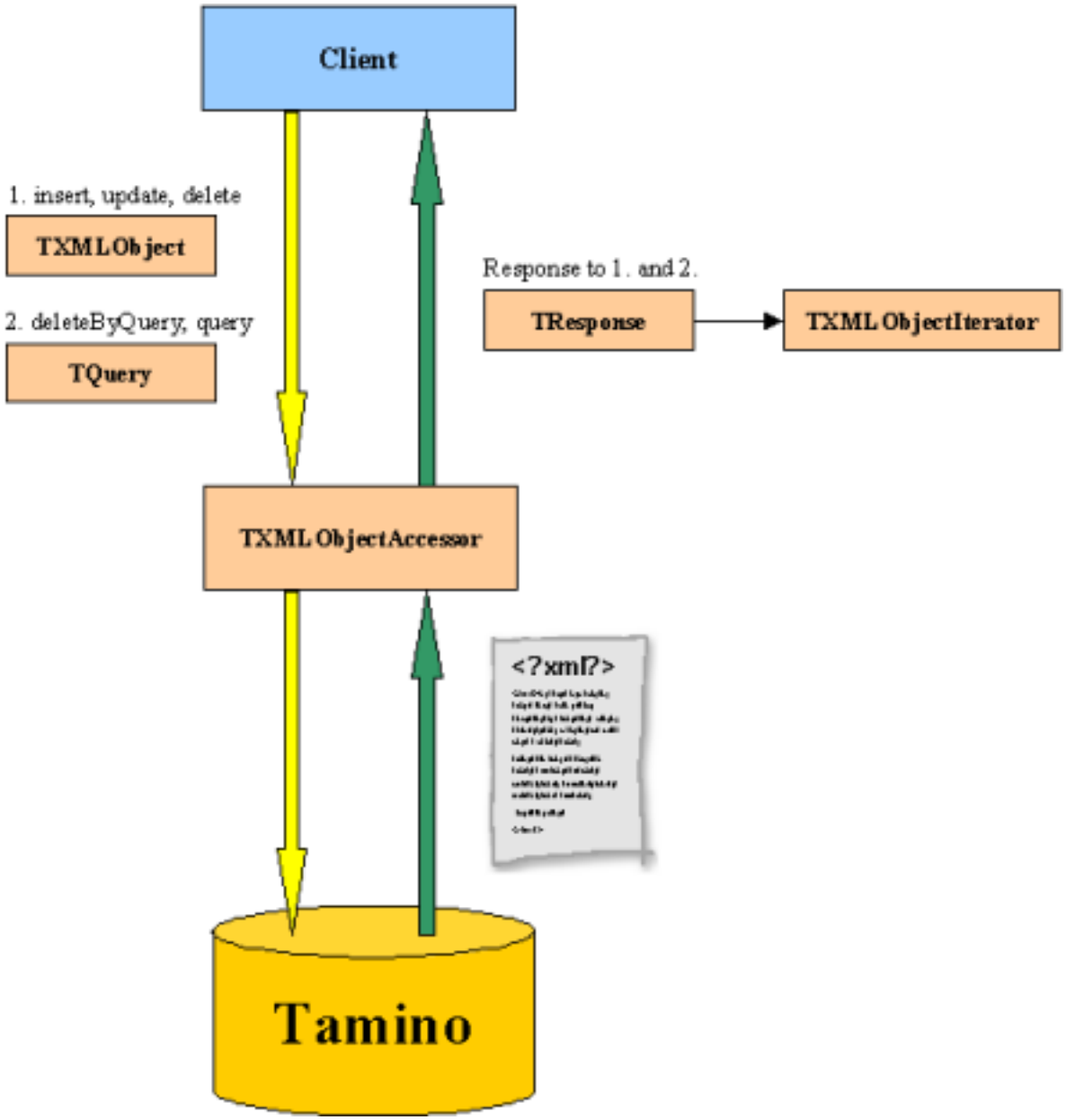
Documents that are storable for Tamino are either represented by the class `TXMLObject` for XML or by the class `TNonXMLObject` for non-XML. The client uses a non-XML accessor to pass `TNonXMLObject` instances back and forth to Tamino and proceeds in the same way concerning XML accessors and `TXMLObject` instances.

The common abstraction to all Tamino documents is `TDataObject` which also serves as a base class to `TNonXMLObject` and `TXMLObject`. `TNonXMLObject` basically contains the non-XML data as an input stream. This can be provided by the client as input data for insert operations to Tamino or as an input stream which Tamino returned as the result of a query operation.

The abstract base class `TXMLObject` represents a facade to the concrete underlying XML object model. A factory method can be used to instantiate a concrete `TXMLObject` class that serves as an adapter to the object model. In Java, `TXMLObject` supports the instantiation of instances that use DOM or JDOM as object model. Other object models can be dynamically plugged in when the client provides implementations for `TXMLObjectModel`. The purpose behind `TXMLObjectModel` is to provide meta-information that defines the concrete object model so that the API is able to work with it on demand.

Response

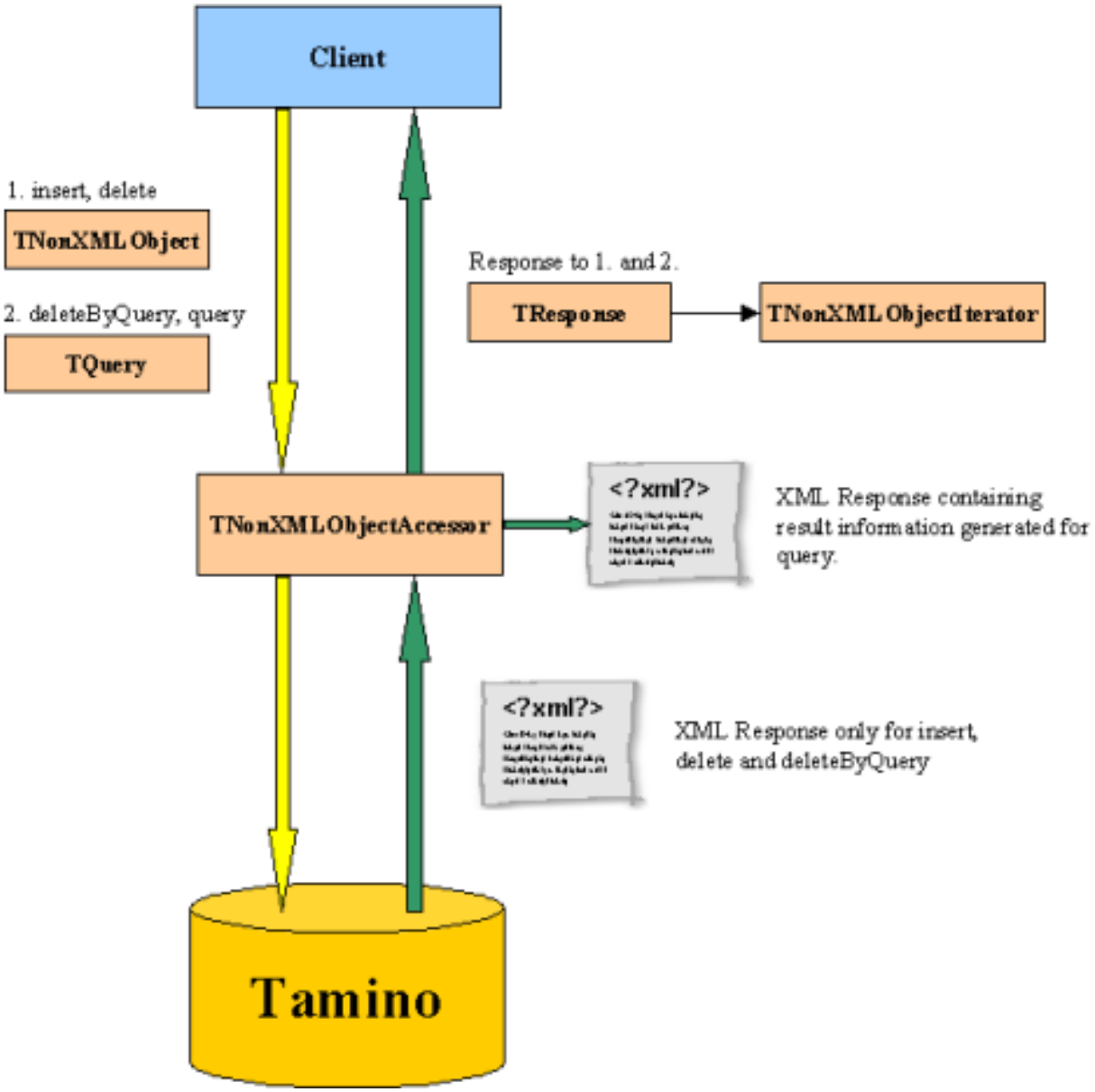
Tamino responds to a command invocation such as `_xquery`, `_xql`, `_process`, `_delete`, etc. with an XML response document. It contains general response information as well as the result set, which consists of the documents returned from Tamino. You can access this response information by an instance of `TResponse`. It always contains information on the execution result such as the response code.



When the client invokes a query operation for XML documents, the `TResponse` instance provides a `TXMLObjectIterator` that can be used to navigate in a type-safe manner over the result set of `TXMLObject` instances. However, if the client only wants to obtain the first document, `getFirstXMLObject` on the `TResponse` instance is invoked to avoid the additional indirection with the iterator.

The situation changes when the client wants to access non-XML documents. In this case Tamino does not always respond with XML result documents. When non-XML documents are retrieved, Tamino directly returns the non-XML documents rather than responding with XML response documents. The API however hides this mechanism and behaves for non-XML in much the same way as with XML access. As a consequence, the client works with non-XML documents in the

same way as when accessing XML documents. The non-XML classes are used to access Taminos non-XML documents. When the client invokes a query operation for non-XML documents with the `TNonXMLObjectAccessor`, the `TResponse` instance provides a `TNonXMLObjectIterator` that can be used to navigate in a type-safe manner over the result set of `TNonXMLObject` instances. If the client only wants to obtain a single non-XML document, the `getFirstNonXMLObject` operation on the `TResponse` instance can be used.



The API hides the fact that Tamino handles non-XML objects in a different way than XML objects.

Query

As already mentioned, a clear distinction is made between the handling and access of XML and non-XML documents.

An abstraction of Tamino query expressions formulated in X-Query is the class `TQuery` which is a single class needed to retrieve both XML documents and non-XML documents when using the SoftwareAG proprietary query language X-Query based on XPath.

For X-Query's successor XQuery there is the class `TXQuery` available to retrieve XML documents when using the query language XQuery propagated by the W3C. Retrieval of non-XML documents using XQuery is currently not implemented.

Access Methods and Response Processing

Access Methods

The Tamino API for Java provides several different ways to access and store documents in Tamino and different ways to process the response returned by Tamino. Different accessors offer access to Tamino on different abstraction levels.

The lowest level of access to Tamino is available through the `TStreamAccessor`. It treats all documents as streams of bytes or characters. It delivers the entire response document returned by Tamino as a single stream. The `TStreamAccessor` does not process or interpret the Tamino response document. The interpretation of the response document is up to the client program.

On a higher abstraction level, the Tamino API offers the interfaces `TXMLObjectAccessor` and `TNonXMLObjectAccessor`. These interfaces treat each document as a separate object represented by the `TXMLObject` and `TNonXMLObject` interfaces.These accessors process and interpret the response documents returned by Tamino and separate the result information (return value, message, cursor information etc.) from the result content. These accessors are capable of supporting different object models. An object model defines, how XML documents contained in Tamino responses are delivered to the client program and how the client program delivers documents to Tamino (DOM, JDOM, SAX, stream).

Note the differences between a `TStreamAccessor` and the stream object model. The stream object model offers stream access to XML documents contained in the result content, while the `TStreamAccessor` offers stream access to the whole Tamino response, without further interpretation of the response document. The stream object model works with `TXMLObject` instances, whereas `TStreamAccessor` works with `TInputStream` instances.

Note: The `TStreamAccessor` delivers the Tamino XML Server response as a stream without any XML parsing, thus being considerably faster than all other object model based accessors. Notably, the largest part of processing time when using the Tamino API for Java is spent

for parsing of XML documents. This should be taken into consideration when optimizing performance.

Apart from the previously mentioned accessors there are specialized accessors available which provide access to specific types of information. The `TSchemaDefinition2Accessor` and the `TSchemaDefinition3Accessor` provide methods to access and store schema information. The `TSystemAccessor` and the `TAdministrationAccessor` provide access to system information.

Response Processing

The result of a Tamino response is contained within the `xql:result` node of the Tamino response document. It can be a list of XML documents or a list of literals (strings, numbers etc.). The API offers access to a list of XML documents via the document iterators (`TXMLObjectIterator` and `TNonXMLObjectIterator`), which can be obtained from the `TResponse` class. However, there are convenience methods in the `TResponse` object to directly access the first document of such a list.

You can directly access literals in the result content with the `TResponse` method `getQueryContentAsString()`. Currently, the Tamino response does not contain any type information for literals, so they are always returned as strings and the client program has to convert them.

V Doing More with the API

We now provide further examples that show you how to use the Tamino API. In these examples we demonstrate:

- how to work with XML objects that are validated against a schema definition
- how to work with XML objects that contain non-ASCII characters
- how to add an XML object model
- how to perform “joins” (how to retrieve information from different doctypes into a single result set)
- how to test integrity constraints between document types
- how to test for unique keys

In the first example called “Get Personal”, we demonstrate XML document processing using the schema "Person".

In the second example called “SAX Package: Using SAX Object Model”, we demonstrate a simple application using the SAX object model.

In the third example called “Adding an Object Model”, we demonstrate how to add another XML object model to the API using the DOM4J object model.

In the fourth example called “All That Jazz”, more advanced concepts such as conceptual model representation, schema definition, joining documents are discussed.

6 Get Personal

In this example, we will demonstrate some basic XML document processing.

As data, we provide five XML documents intended to be used with the very simple "person" schema. The document instances contain basic personal information such as last name, first name, and sex. You will find these XML documents along with a Tamino schema definition file in the *examples/persons* directory of the Tamino documentation.

In order to try out these examples, you will need a running database with the "person" schema defined. Please refer to the Tamino documentation for how to create a Tamino database.

The following use cases are covered in these examples:

- establish a connection to a Tamino database
- use a system accessor for retrieving different information
- read some XML documents from files and insert them into the database
- perform different queries on these documents and list the results
- delete documents
- insert a schema definition into the database, creating a new collection
- insert some XML documents into the created collection
- show how a document fails the validation against the schema
- close a database connection

Two examples are provided that demonstrate these tasks

- `ProcessPersons`: Insert and query person documents
- `ProcessPersonsWithSchema`: Insert and query person documents using a schema

Inserting and Querying Documents

The class `ProcessPersons` performs the following tasks that are typical for an application accessing data stored in a Tamino database:

1. establish a connection to the existing Tamino database `mydb`
2. check the availability of the database connection
3. obtain a system access method and retrieve some system information
4. obtain an access method to handle XML data using the Document Object Model (DOM)
5. read some XML documents from files and insert them into the database
6. perform queries for all, some, the count of all and the availability of some documents
7. delete some documents by query and show the remaining documents

8. delete all documents
9. close a database connection

It is assumed that a database called `mydb` has already been created and is running. The necessary data files are stored within the classpath at the same location as the class `ProcessPersons` and will be read from there. So the examples are ready to run provided that Tamino is installed on the local host.

Running the Example

You need to include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *log4j.jar*, *xercesImpl.jar*and *xmlParserAPIs.jar* in your class path, all of which are distributed with the Tamino API. Then you can run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.person.ProcessPersons
ProcessPersons sample programm
==============================
Connecting to Tamino database http://localhost/tamino/mydb, ... server is alive

Here is some systeminformation
------------------------------

The Tamino server hosting http://localhost/tamino/mydb is version 4.2.1.1
(Server API version: 1.1, Tamino API for Java version: 4.2.1.1)

Insert and query and delete in default collection "ino:etc"
----------------------------------------------------------

Reading documents from file and insert into database

  Inserted: Atkins, Paul (ino:id="1" collection="ino:etc" doctype="person" )
  Inserted: Bloggs, Fred (ino:id="2" collection="ino:etc" doctype="person" )
  Inserted: Müller, Andreas (ino:id="3" collection="ino:etc" doctype="person" )
  Inserted: Müller, Karla (ino:id="4" collection="ino:etc" doctype="person" )
  Inserted: Atkins, Andreas (ino:id="5" collection="ino:etc" doctype="person" )

The query "person" returns 5 documents, which are:
  Atkins, Paul (ino:id="1" collection="ino:etc" doctype="person" )
  Bloggs, Fred (ino:id="2" collection="ino:etc" doctype="person" )
  Müller, Andreas (ino:id="3" collection="ino:etc" doctype="person" )
  Müller, Karla (ino:id="4" collection="ino:etc" doctype="person" )
  Atkins, Andreas (ino:id="5" collection="ino:etc" doctype="person" )

The query "//surname='Atkins'" returns "TRUE"
So list and then delete all "Atkins" documents

The query "person[//surname='Atkins']" returns 2 documents, which are:
  Atkins, Paul (ino:id="1" collection="ino:etc" doctype="person" )
  Atkins, Andreas (ino:id="5" collection="ino:etc" doctype="person" )
```

```
Deleted all documents for query "person[//surname='Atkins']"

The query "person" returns 3 documents, which are:
  Bloggs, Fred (ino:id="2" collection="ino:etc" doctype="person" )
  Müller, Andreas (ino:id="3" collection="ino:etc" doctype="person" )
  Müller, Karla (ino:id="4" collection="ino:etc" doctype="person" )

Deleted all documents for query "person"
```

Note: Using *JavaTaminoAPIExamples.jar* only works if you have created a database called "mydb" on your local host. Otherwise you have to change the constant `DATABASE_URI` and recompile the code.

Inserting and Querying Documents Using Schemas

The class `ProcessPersonsWithSchema` performs the following tasks that are typical for an application accessing data stored in a Tamino database:

1. establish a connection to the existing Tamino database `mydb`
2. check the availability of the database connection
3. obtain a system access method and retrieve some system information
4. read a schema definition from file and insert it into the database, creating a new collection
5. read some XML documents from files and insert them into the database
6. show how a document fails the validation against the schema
7. query and list all successfully inserted documents
8. delete all documents
9. close a database connection

It is assumed that a database called `mydb` has already been created and is running. The necessary schema and data files are stored within the classpath at the same location as the class `ProcessPersonsWithSchema` and will be read from there. So the examples are ready to run provided that Tamino is installed on the local host. Note that the class `ProcessPersonsWithSchema` uses some methods from the class `ProcessPersons` and thus cannot run without it.

Running the Example

You need to include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *log4j.jar*, *xercesImpl.jar*, and *xmlParserAPIs.jar* in your classpath, all of which are distributed with the Tamino API. Then you can run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.person.ProcessPersonsWithSchema
ProcessPersonsWithSchema sample programm
========================================
Connecting to Tamino database http://localhost/tamino/mydb, ... server is alive

Here is some systeminformation
------------------------------

The Tamino server hosting http://localhost/tamino/mydb is version 4.2.1.1
(Server API version: 1.1, Tamino API for Java version: 4.2.1.1)

Create collection "people" insert and delete some documents
-----------------------------------------------------------

Reading TSD3 schema from file and insert into database

  Inserted the schema for collection "people" and doctype "person"

Reading documents from files and insert into database

  Inserted: Atkins, Paul (ino:id="1" collection="people" doctype="person" )
  Inserted: Bloggs, Fred (ino:id="2" collection="people" doctype="person" )
  Inserted: Müller, Andreas (ino:id="3" collection="people" doctype="person" )

Can't insert: Müller, Karla (ino:id="" collection="people" doctype="person" )
  Reason: Line 8, Column 15:  [element 'occupation' in element 'person']

  Inserted: Atkins, Andreas (ino:id="4" collection="people" doctype="person" )

The query "person" returns 4 documents, which are:
  Atkins, Paul (ino:id="1" collection="people" doctype="person" )
  Bloggs, Fred (ino:id="2" collection="people" doctype="person" )
  Müller, Andreas (ino:id="3" collection="people" doctype="person" )
  Atkins, Andreas (ino:id="4" collection="people" doctype="person" )

Deleted all documents for query "person"

Deleted collection "people" with all content
```

Note: Using *JavaTaminoAPIExamples.jar* only works if you have created a database called "mydb" on your local host. Otherwise you have to change the constant `DATABASE_URI` and recompile the code.

7 SAX Package: Using SAX Object Model

The SAX package contains an example similar to the `ProcessXMLGreeting` example. It demonstrates the usage of the Tamino `TSAXObjectModel` , which represents the SAX object model. You should be familiar with the SAX object model to understand the `TSAXObjectModel` and this example.

Overview

Using the classes `TDOMObjectModel`, `TJDOMObjectModel` or `TStreamObjectModel` has advantages over using SAX for reasons of simplicity. This is because SAX requires the user to develop one or more event handlers. For example, when using the `TDOMObjectModel` the complete XML document is read into memory and represented as a tree. This allows you to manipulate any part of the document instantly, and does not require any programming on your part. This convenience comes, of course, at the expense of system resources and speed, as the complete XML document needs to be parsed and Java objects need to be instantiated to represent the complete DOM tree. This mechanism may result in considerable overhead, especially in those cases where your application only requires a small part of the XML document. In that case an approach based on SAX may provide a much more efficient alternative. Using SAX you must provide one or more event handlers for the parser. An event handler is a component that registers itself for callbacks from the parser when SAX events are fired.

The `TDOMObjectModel`, `TJDOMObjectModel` and `TStreamObjectModel` are factory classes providing a singleton instance by means of the factory method `getInstance()`. These object models are generic and can work with every type of XML document. In contrast, the `TSAXObjectModel` differs here, as it is not generic. For each type of XML document, a specific instance of the `TSAXObjectModel` must be created, specifying one or two sets of helper classes. Each set consists of an event handler (extention of the SAX `DefaultHandler`) and a class representing the XML node either as a document or an element. The event handler provides the processing logic specific to the XML node. The event handlers available with the SAX package are named `elementDefaultHandler` and `documentDefaultHandler`. The class (use `saxElementClass` or `saxDocumentClass`) must be an implementation of the `TSAXDocument` or `TSAXElement`. When deciding to use the `TSAXObjectModel` it is important to know whether it is to be used for processing query results or single XML documents (either programmatically instantiated or retrieved from Tamino) or both. For the processing of query results a `saxElementClass` and an `elementDefaultHandler` are required. If only single XML documents are retrieved from Tamino (via the `accessor retrieve` method) or if single XML documents are created (via a `TXMLObject newInstance` factory method), a `saxDocumentClass` and a `documentDefaultHandler` are required. The documented example demonstrates both.

The SAX package consists of following classes:

Class	Description
`ProcessGreeting`	The class with the main method. It actually does the same as the `ProcessXMLGreeting` example: It establishes a connection, gets an `Accessor` (`SAX accessor`), inserts, retrieves, updates and deletes an XML document `<Message>...</Message>` in local transaction mode.
`Greeting`	Represents an XML document. It implements the `TSAXElement` and `TSAXDocument` interfaces
`GreetingDefaultHandler`	Extends the SAX DefaultHandler. Its purpose is to handle all events for the Message XML documents
`DocumentDefaultHandler`	Extends the `TDocumentDefaultHandler` class. It does its work by delegating the events to a `GreetingDefaultHandler`.
`ElementDefaultHandler`	Extends the `TSAXElementDefaultHandler` class. It does its work by delegating the events to a `GreetingDefaultHandler`.

Note: Because the `ProcessGreeting` class has the same functionality as the `ProcessXMLGreeting` we only describe the SAX aspects in the following sections.

Note: When interpreting an InputStream using the SAX object model, user-specific implementations of the abstract `TSAXElementDefaultHandler` class are used. To clear the contents that might currently be available from `TSAXElementDefaultHandler`'s methods `getFirstElement()` and `getElementIterator()`, the interface now offers a `reset()` method which is called at the appropriate time before input stream interpretation starts. The default implementation of the `reset()` method is empty. If, for example, the user-specific implementation of `TSAXElementDefaultHandler` maintains a list of elements, thís list must be cleared in the `reset()` method.

Construct a TSAXObjectModel

```
// Instantiate the default handler that processes the sax events
greetingDefaultHandler = new GreetingDefaultHandler();

// Instantiate the document and element event handlers each of which
// delegates its events to the greetingDefaultHandler
docDefHandler = new DocumentDefaultHandler( greetingDefaultHandler );
elDefHandler = new ElementDefaultHandler( greetingDefaultHandler );

// Instantiate the specific TSAXObjectModel
saxObjectModel = new TSAXObjectModel("GreetingSAXObjectModel", Greeting.class, ↩
```

```
Greeting.class, docDefHandler, elDefHandler );

// Do the object model registration.
TXMLObjectModel.register( saxObjectModel );
```

Obtain a SAX Accessor

```
accessor = connection.newXMLObjectAccessor(TAccessLocation.newInstance( collection ↩
) , saxObjectModel );
    }
```

Running the Example

You need to include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *xercesImpl.jar*, *xmlParserAPIs.jar* and *log4j.jar* in your CLASSPATH. These files are included in the distribution of the Tamino API (see section **Component Profile**). You can then run the Java interpreter:

```
java com.softwareag.tamino.db.api.examples.greeting.SAX.ProcessGreeting
```

```
Going to insert <Greeting>Hello World</Greeting>
Insert succeeded, ino:collection:ino:etc, ino:doctype:Greeting, ino:id:1
Update succeeded!
Queried document:<Greeting ino:id="1">Hello World, updated :-)</Greeting>
Deleted the document!
```

Note: As in other examples delivered with the Tamino XML Server, using *JavaTaminoAPIExamples.jar* only works if you have created a database called “mydb” on your local host. Otherwise you will have to change the source code and recompile it.

8 DOM4J: Adding an Object Model

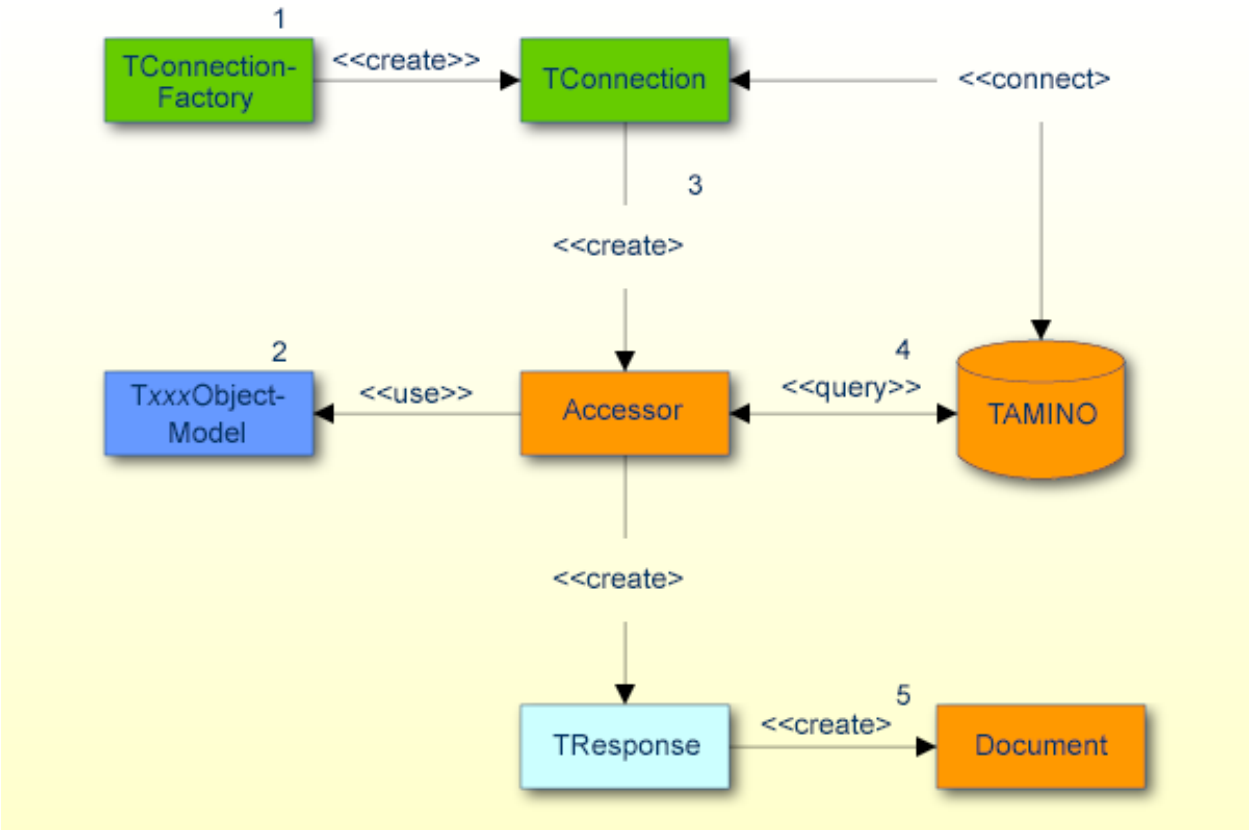
A client application handles XML documents by using an accessor to pass `TXMLObject` instances to Tamino. `TXMLObject` supports DOM and JDOM as object models, but you can add other object models as well. This chapter explains the steps involved using DOM4J as an example.

Object Models

The underlying XML object model is represented by the abstract base class `TXMLObject`. Using a factory method you can instantiate a `TXMLObject` that serves as an adapter to the object model. DOM and JDOM are directly supported as object models, but you can dynamically plug in other object models provided that the client offers an implementation for `TXMLObjectModel`. `TXMLObjectModel` provides meta-information that defines the object model so that it can be used within the API.

Using an Object Model

In the context of the Tamino API, you use an XML object model as follows (*xxx* here stands for an arbitrary object model):



1. Use a `TConnectionFactory` to create a `TConnection` instance for establishing a connection to a Tamino database

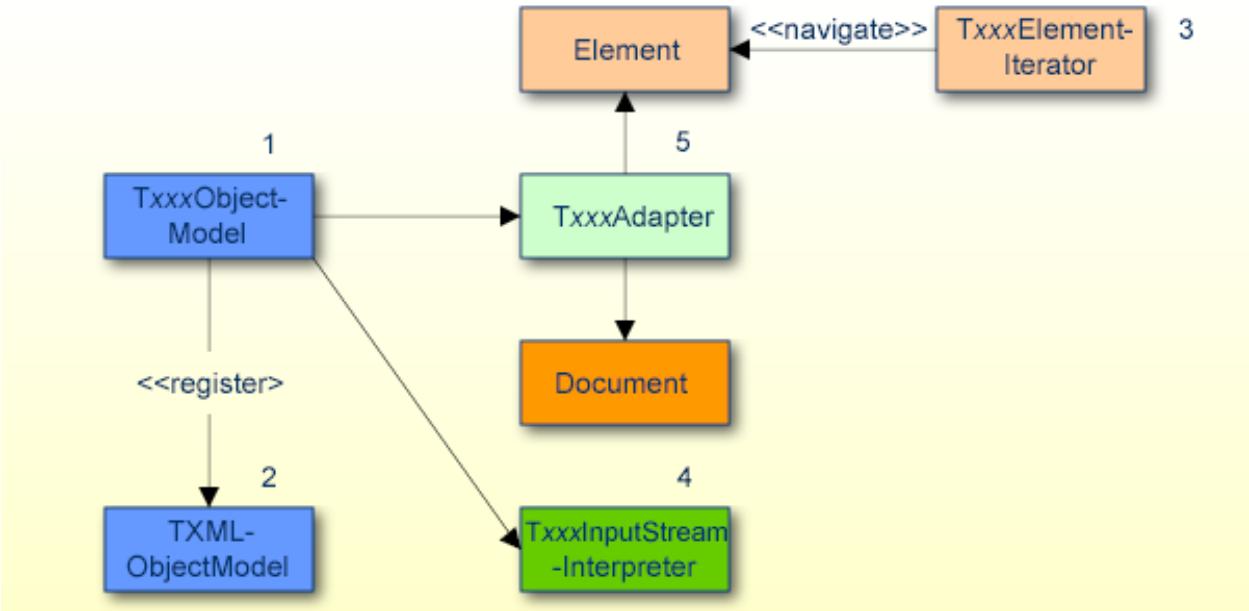
2. Create an instance of the `TxxxObjectModel`.
3. Let the `TConnection` create an Accessor for the `TxxxObjectModel`.
4. Now you can use the Accessor to send a query to Tamino.
5. The resulting `TResponse` instance is able to create the `Document` instance specific to that object model.

Writing an Object Model

An object model is defined by the following classes:

- the generic class or interface of the underlying object model that represents the root node type for an XML document; for DOM4J this is the `Element` interface.
- `TXMLObject` adapter implementation which adapts some interfaces of the underlying object model
- an implementation for interpreting the input stream

As a consequence, the abstract base class `TXMLObjectModel` defines operations to access the meta class for the node type, the `TXMLObject` adapter and the `TResponseBuilder`. To use a new object model you need to provide the appropriate `TXMLObjectModel` implementation along with the `TXMLObject` adapter and the `TxxxInputStreamInterpreter` implementation.



1. The `TxxxObjectModel` describes which xxx specific `Document`, `Element`, `TxxxAdapter` and `TxxxInputStreamInterpreter` is used by the xxx object model.
2. To use this object model it must be registered in the `TXMLObjectModel` with the method: `TXMLObjectModel.register(TXMLObjectModel xmlObjectModel)`.

3. A `TxxxElementIterator` is needed to navigate unidirectional and type safe over the xxx specific Elements.
4. `TxxxInputStreamInterpreter` interprets the TAMINO Stream and creates the xxx specific `Document` instance
5. The `TxxxAdapter` adapts to the xxx specific object model to the generic `TXMLObject` interface.

Assembling a New Object Model

In this section we use DOM4J as an example to show which classes are needed for the implementation of a new object model. You can use the example code available with this documentation (*<Tamino installaion directory>*/SDK/TaminoAPI4J/Documentation/inoapi/listings/dom4j/) as a template for supporting another object model in the Tamino API.

In the following sections we discuss the implementation of the classes needed based on the DOM4J object model implementation.

First, you must determine a short name for the object model, by which all classes are recognized as belonging to the adapter implementation of the same object model. Here, we use "xxx" as name to refer to the new object model adapter.

You need to create two packages, one containing the complete implementation of the object model adapter and another package with the implementation of the input stream interpreter. Using "xxx" the package names are as follows:

- `com.softwareag.tamino.db.api.objectModel.`*xxx*
- `com.softwareag.tamino.db.api.response.`*xxx*

TxxxAdapter

`TXMLObject` is the abstract super class to represent an XML object. To create the specific adapter class, this class must be extended. Example:

```
com.softwareag.tamino.db.api.objectModel.dom4j.TDOM4JAdapter
```

How to implement the `TxxxAdapter`

1 Copy the class

```
com.softwareag.tamino.db.api.objectModel.dom4j.TDOM4JAdapter
```

to the new package:

```
com.softwareag.tamino.db.api.objectModel.xxx
```

2 Rename the adapter class to:

```
com.softwareag.tamino.db.api.objectModel.TxxxAdapter
```

3 Replace the references to `org.dom4j.Element` and `org.dom4j.Document` with references to appropriate classes of the object model "xxx".

4 In the implementation of the object model, identify and use the methods that perform the following tasks:

- get the related Document
- remove an Attribute
- get an Attribute
- add an Attribute
- set a value
- get the qualified name
- get the root element

You also need a way to read an InputStream and create your specific Document instance. In DOM4J this is `SAXReader`, which is used in the `readFrom()` methods.

The best way to make these changes is to replace the import instructions in your new Adapter class (e.g. replace dom4j specific imports by imports for your new model), and to check any compiler messages about missing methods.

TxxxInputStreamInterpreter

How to implement the `Txxx`InputStreamInterpreter

1 Copy the class

```
com.softwareag.tamino.db.api.response.dom4j.TDOM4JInputStreamInterpreter
```

to the new package:

```
com.softwareag.tamino.db.api.response.xxx
```

2 Rename the class to:

```
com.softwareag.tamino.db.api.response.TxxxInputStreamInterpreter
```

3 In the import instructions replace the following classes

```
org.dom4j.Document;
org.dom4j.Element;
org.dom4j.Namespace;
org.dom4j.QName;
org.dom4j.DocumentException;
org.dom4j.io.SAXReader;
```

with the corresponding classes of the new object model.

4 In the implementation of the new object model, identify these classes and methods of DOM4J and change it appropriately for `TxxxInputStreamInterpreter`:

Class	Method
org.dom4j.Document	getRootElement()
org.dom4j.Element	element()
	elements()
	attributeValue()
	getText()
org.dom4j.Namespace	get()
org.dom4j.QName	
org.dom4j.DocumentException	
org.dom4j.io.SAXReader	read()

ObjectModel

`TXMLObjectModel` contains operations needed to define and control an XML object model. Each XML object model is defined by four class instances:

`Element`
: An `Element` class represents either an element or a fragment of the document tree.

`Document`
: A `Document` class represents the complete document.

`Adapter`
: An `Adapter` class adapts the physical object model to `TXMLObject`.

`InputStreamAdapter`
: An `InputStreamInterpreter` class determines for a physical object model the way to handle response documents returned by Tamino.

How to implement the T*xxx*ObjectModel

1 Copy the class

```
com.softwareag.tamino.db.api.objectModel.dom4j.TDOM4JObjectModel
```

to the new package:

```
com.softwareag.tamino.db.api.objectModel.xxx.TxxxObjectModel
```

2 Change the constructor

```
   protected TxxxObjectModel() {
       super( "xxx" ,
              xxx.Document.class ,
              xxx.Element.class ,
              com.softwareag.tamino.db.api.objectModel.xxx.TxxxAdapter.class ,
              ↩
com.softwareag.tamino.db.api.response.xxx.TxxxInputStreamInterpreter.class );
       }
```

You must use your specific `Document`, `Element`, `Adapter` and `InputStreamInterpreter` classes.

3 Change occurrences of `TDOM4JObjectModel` to T*xxx*`ObjectModel`.

Outputter

If your object model does not provide a class to serialize XML into a string, you have to write your own class.

How to implement an outputter class:

1 Copy the classes

```
com.softwareag.tamino.db.api.objectModel.dom4j.TDOM4JXMLOutputter
com.softwareag.tamino.db.api.objectModel.dom4j.TDOM4JNamespaceStack
```

into the package

```
com.softwareag.tamino.db.api.objectModel.xxx
```

2 Rename these classes as:

```
com.softwareag.tamino.db.api.objectModel.xxx.TxxxXMLOutputter
com.softwareag.tamino.db.api.objectModel.xxx.TxxxNamespaceStack
```

3 You must replace the following classes and methods:

Class	Method / Constant
org.dom4j.Attribute	getNamespace()
	getQualifiedName()
	getValue()
org.dom4j.CDATA	asXML()
org.dom4j.Comment	asXML()
org.dom4j.DocumentType	getPublicID()
	getSystemID()
	getElementName()
org.dom4j.Document	getRootElement()
org.dom4j.Element	getDocument()
	getNamespacePrefix()
	getParent()
	elements()
	getQualifiedName()
	getNamespace()
	additionalNamespace()
	attributes()

Class	Method / Constant
	getTextTrim()
	getText()
org.dom4j.Entity	asXML()
org.dom4j.Namespace	get()
	XML_NAMESPACE()
	NO_NAMESPACE()
	getUri()
	getPrefix()
org.dom4j.ProcessingInstruction	

4 In the implementation of *Txxx*XMLOutputter replace occurrences of TDOM4JNamespaceStack with *Txxx*NamespaceStack.

Putting Together

At this point, you have a complete implementation of all necessary classes. Compile these classes, include them and those of the original object model in the class path. You can then use the new object model in your client applications.

Using the DOM4J Object Model: A Sample

To use the implementation for the DOM4J object model you need the DOM4J implementation which you can download at **http://www.dom4j.org/**. You need to include the DOM4J classes and the adapter classes in your classpath. You will find the adapter classes in the file *JavaTaminoAPIExamples.jar* in the directory *<TaminoAPIDir>/examples*.

In the sample below you can see how to register the new object model so that the Tamino API knows about it. In the last line, an accessor is instantiated that uses the new object model.

```
// Constant for the database URI. Please edit to use your URI of interest.
private final static String DATABASE_URI = "http://localhost/tamino/mydb";

// Obtain the connection factory
TConnectionFactory connectionFactory = TConnectionFactory.getInstance();

// Obtain the connection to the database
TConnection connection = connectionFactory.newConnection( databaseURI );

// Instantiate the specific TxxxObjectModel
TxxxObjectModel xxxObjectModel = TxxxObjectModel.getInstance();
```

```
   // Register the object model
   TXMLObjectModel.register( xxxObjectModel );

   // Obtain the concrete TxxxObjectAccessor with an underlying xxx object model
   TXMLAccessor accessor = connection.newXMLObjectAccessor( ↩
TAccessLocation.newInstance( "ino:etc" , xxxObjectModel );
```

9 All that Jazz

We now present a few more use cases for the Tamino API. These examples cover a multi-document join, the validation of integrity constraints, and the validation of unique keys.

All three examples introduce not only into programing with the Tamino API but also into the programming with DOM and JDOM. The Tamino API has a pluggable object model interface and allows to use various object models.

JDOM is a popular (albeit non-standard) API that provides a simpler interface than the standard DOM implementations. Because it allows to merge different documents easily we use it for our first example where we join multiple documents.

The preferred object model, however, is DOM which we will use for the following two examples. DOM is a W3C recommendation and has bindings into many programming languages, which is not the case for JDOM.

The DOM Level 2 specification is available at **http://www.w3.org/TR/DOM-Level-2-HTML/**. You will find a tutorial **http://www.w3schools.com/xml/dom_intro.asp**.

We have chosen to implement a small knowledge base about jazz musicians and jazz music. In this scenario, jazz musicians play together in different types of collaborations (jam session, project, band) and produce results in form of albums.

For the following examples readers should have general knowledge about XML and DOM and should have practical experience with Tamino, especially with the Tamino Schema Editor, the Tamino Interactive Interface, and the Tamino Manager.

The examples are presented under the following topics:

Preparation

The section describing the Tamino API package explains where you will find the example files used below.

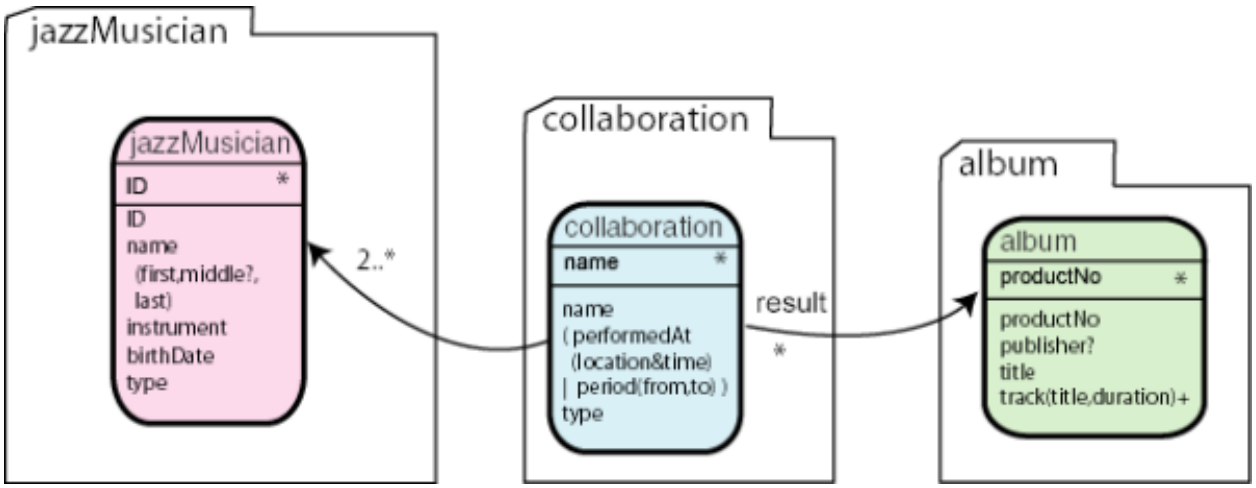
To set up the stage start the Tamino Manager and create a new database with the name `jazz`. A small database with the default settings will do.

Conceptual model

Our conceptual model of the jazz knowledge base consists of three document types:

- a document type `jazzMusician`. The property `type` determines the type of `jazzMusician`: an instrumentalist, a composer, or a singer. The property `ID` establishes a key for each musician which we construct from the last name and the first name: "GillespieDizzy", "ColtraneJohn", etc.

- a document type `collaboration`. The property `type` determines the type of `collaboration`: a band, a project, or a jam session. A collaboration is either a single event (jam session) performed at a specific time and place, or it exists over a period of time. Each collaboration relates to at least two jazz musicians that take part in that collaboration. A collaboration can also relate to one or several albums which result from that collaboration.
- a document type `album`. Here we have properties such as product number (which acts also as key), publisher, and information about each track.



Schema Definition

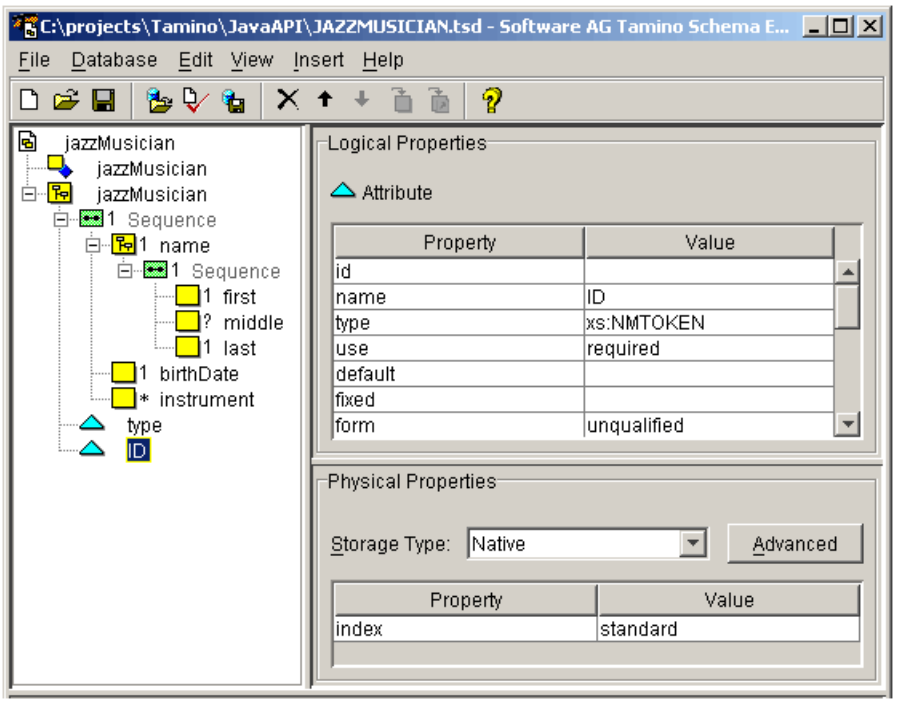
Based on this simple conceptual model we can now define our schemata with the help of the Tamino Schema Editor. All three schemata will be in a new Tamino collection which we will name `encyclopedia`.

Schema jazzMusician

We begin with `jazzMusician`. We set the schema name to "jazzMusician" and the collection name to "encyclopedia":

Property	Value
schemaName	jazzMusician
collectionName	encyclopedia
xmlns:xs	http://www.w3.org/2001/XMLSchema
xmlns:tsd	http://namespaces.softwareag.com/tamino/...
targetNamespace	
attributeFormDefault	unqualified
elementFormDefault	unqualified
id	
version	
created	
modified	

Then we define the document structure:



Here, we can see the complete schema as it appears in the Tamino Schema Editor. We have chosen to implement the properties `ID` and `type` as attributes, and we have declared `ID` as a standard index.

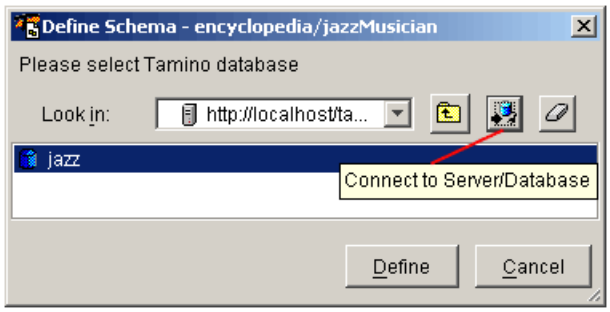
Both attributes have the type `xs:NMTOKEN`. The type of attribute `type` has been restricted by an enumeration to the values "instrumentalist", "jazzSinger", and "jazzComposer". The element `birthDate` has the type `xs:date`, and the other elements have the type `xs:string` or `xs:normalizedString`.

The resulting schema definition should look like this:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "jazzMusician">
        <tsd:collection name = "encyclopedia">
        </tsd:collection>
        <tsd:doctype name = "jazzMusician">
          <tsd:logical>
            <tsd:content>open</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "jazzMusician">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "name">
          <xs:complexType>
            <xs:sequence>
              <xs:element name = "first"
                          type = "xs:normalizedString"/>
              <xs:element name = "middle"
                          type = "xs:normalizedString"
                          minOccurs = "0"/>
              <xs:element name = "last"
                          type = "xs:normalizedString"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name = "birthDate" type = "xs:date">
        </xs:element>
        <xs:element name = "instrument"
                    type = "xs:string"
                    minOccurs = "0"
                    maxOccurs = "unbounded"/>
      </xs:sequence>
      <xs:attribute name = "type">
        <xs:simpleType>
          <xs:restriction base = "xs:NMTOKEN">
            <xs:enumeration value = "instrumentalist"/>
            <xs:enumeration value = "jazzSinger"/>
            <xs:enumeration value = "jazzComposer"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name = "ID"
                    type = "xs:NMTOKEN" use = "required">
```

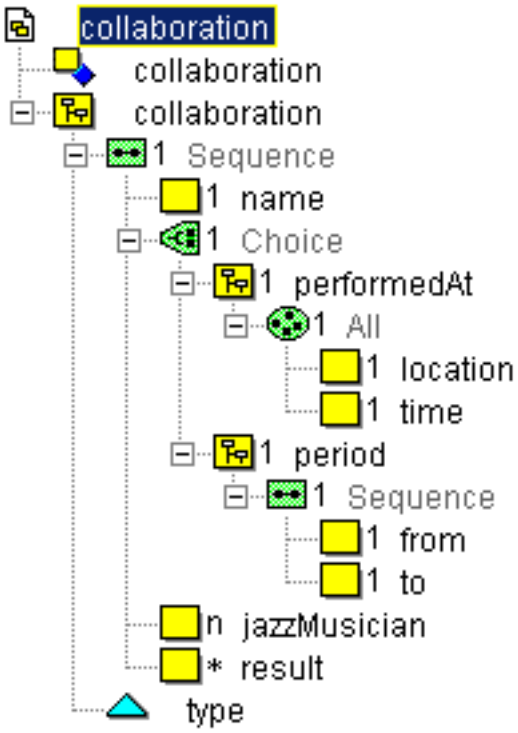
```
        <xs:annotation>
          <xs:appinfo>
            <tsd:attributeInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard/>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:attributeInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

We can now define this schema in the database. We can do this directly from the Tamino Schema Editor. In the **Database** menu, select **Define Schema**. If you are not already connected to the `jazz` database, choose the **Connect to Server/Database** button and enter your user name and password to connect. Then select the `jazz` database from the list and choose the **Define** button.



Schema Collaboration

Again, the property `type` is defined as an attribute of type `xs:NMTOKEN` restricted by the enumeration `[jamSession, project, band]`. The elements `jazzMusician` and `result` (which relates to `album`) are of type `xs:NMTOKEN`, too, and are declared as standard indexes. The element `time` is declared as type `xs:dateTime` while the elements `from` and `to` are declared as type `xs:date`. The other elements are declared as strings or normalized strings.



The resulting schema should look like this:

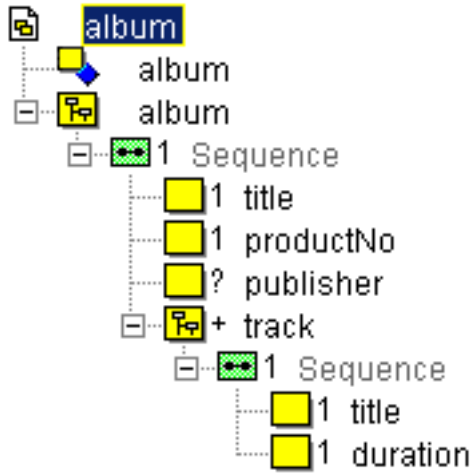
```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "collaboration">
        <tsd:collection name = "encyclopedia">
        </tsd:collection>
        <tsd:doctype name = "collaboration">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>  <xs:element name = "collaboration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "name" type = "xs:NMTOKEN">
        </xs:element>
        <xs:choice>
          <xs:element name = "performedAt">
            <xs:complexType>
              <xs:all>
```

```
            <xs:element name = "location"
                        type = "xs:normalizedString"/>
            <xs:element name = "time" type = "xs:dateTime"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
      <xs:element name = "period">
        <xs:complexType>
          <xs:sequence>
            <xs:element name = "from" type = "xs:date"/>
            <xs:element name = "to" type = "xs:date"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
    <xs:element name = "jazzMusician"
                type = "xs:NMTOKEN"
                minOccurs = "2"
                maxOccurs = "unbounded">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard/>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "result"
                type = "xs:NMTOKEN"
                minOccurs = "0"
                maxOccurs = "unbounded">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard/>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
```

```
      <xs:attribute name = "type" use = "required">
        <xs:simpleType>
          <xs:restriction base = "xs:NMTOKEN">
            <xs:enumeration value = "jamSession"/>
            <xs:enumeration value = "project"/>
            <xs:enumeration value = "band"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Schema Album

For document type `album` we have defined element `productNo` as a standard index of type `xs:NMTOKEN`. The element `duration` is declared as `xs:short` while all other elements are declared as strings or normalized strings.



The resulting schema should look like this:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "album">
        <tsd:collection name = "encyclopedia">
        </tsd:collection>
        <tsd:doctype name = "album">
          <tsd:logical>
            <tsd:content>open</tsd:content>
```

```
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "album">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "title" type = "xs:normalizedString">
        </xs:element>
        <xs:element name = "productNo" type = "xs:normalizedString">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
                      <tsd:standard/>
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name = "publisher"
                    type = "xs:string" minOccurs = "0"/>
        <xs:element name = "track" maxOccurs = "unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name = "title" type = "xs:string"/>
              <xs:element name = "duration" type = "xs:short"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

We define both schemata `collaboration` and `album` to the `jazz` database.

Populating the Database

Next, we add some test data to our database. We start with two famous jazz musicians. We create *parker.xml*:

```
<?xml version="1.0"?>
```

```
<jazzMusician type="instrumentalist" ID="ParkerCharlie">
  <name>
    <first>Charlie</first>
    <last>Parker</last>
  </name>
  <birthDate>1920-08-19</birthDate>
  <instrument>saxophone</instrument>
</jazzMusician>
```

and *dizzy.xml*:

```
<?xml version="1.0"?>
```

```
<jazzMusician type="instrumentalist" ID="GillespieDizzy">
  <name>
    <first>Dizzy</first>
    <last>Gillespie</last>
  </name>
  <birthDate>1917-10-21</birthDate>
  <instrument>trumpet</instrument>
</jazzMusician>
```

Then, we create a (fictional) jam session *post-election-jam.xml*:

```
<?xml version="1.0"?>
```

```
<collaboration type="jamSession">
  <name>post-election-jam</name>
  <performedAt>
    <location>Blues House</location>
    <time>1945-10-21T20:00:00</time>
  </performedAt>
  <jazzMusician>GillespieDizzy</jazzMusician>
  <jazzMusician>ParkerCharlie</jazzMusician>
  <result>BGJ-47</result>
</collaboration>
```

and an `album` document *blueshouse.xml* as the result of that session:

```
<?xml version="1.0" encoding = "UTF-8"?>
```

```
<album>
  <title>Blues House Jam</title>
  <productNo>BGJ-47</productNo>
  <track>
    <title>Post Election Jam I</title>
    <duration>1175</duration>
  </track>
  <track>
    <title>Post Election Jam II</title>
    <duration>1235</duration>
  </track>
</album>
```

We add these four documents to our encyclopedia `collection` in database `jazz`. We can use the Tamino Interactive Interface to do so.

Joining Documents

Now we are ready to write our first Java program to retrieve information from our database. What we want to do is to find a particular jazz musician and retrieve all the information about this musician. We want to include information about the collaborations in which the musician participated and about the albums that resulted from these collaborations. We don't want to list the albums with all details. The title, product number, and publisher will be sufficient.

This task requires us to join information from three document types. We have to retrieve the appropriate documents, and we must construct a new result document from the information combined.

Constructing Queries

To retrieve the right `jazzMusician` document is easy. We just query for the `ID` attribute with the query `jazzMusician[@ID='?']` where we replace `'?'` with the actual ID, for example with "ParkerCharlie".

To find the corresponding collaborations is just as easy. We simply query for the `jazzMusician` element in document type collaboration:

```
collaboration[jazzMusician='?']
```

Finding the resulting albums requires a bit more work. We first have to extract the content of element `result` from each `collaboration` instance. Then we have to use this value in the following query:

```
album[productNo='?']
```

Tip: It is a good idea to test these queries with the Tamino Interactive Interface before implementing them in Java.

The "main" method

We implement our example program as a Java class `MusicianCollaborationResult` that can be executed from the command line.

The key value (ID attribute) of the `jazzMusician` document that we want to retrieve is passed as a command line parameter to the main method. The `main` method is implemented as a class method. Therefore, it must first create a new instance of the class `MusicianCollaborationResult`. Then it calls the instance method `show` and passes the key value as a parameter to this method:

```
public static void main(String[] args) throws Exception  {
 MusicianCollaborationResult musicianCollaborationResult =
new MusicianCollaborationResult( DATABASE_URI , COLLECTION );
musicianCollaborationResult.show(args[0]);
}
```

Global constants

We have used here a few constants that we still have to declare:

```
// Constant for the database URI.
private final static String DATABASE_URI =
                        "http://localhost/tamino/jazz";
// Constant for the collection.
private final static String COLLECTION = "encyclopedia";
```

We also introduce a namespace constant for the namespace prefix `ino:`. This constant will later be used to identify Tamino-specific attributes such as `ino:id`. Namespaces are identified by URIs, and the `INO_NAMESPACE` constant establishes the connection between namespace prefix and namespace URI.

```
// Constant for ino namespace.
private final static Namespace INO_NAMESPACE =
        Namespace.getNamespace("ino",
            "http://namespaces.softwareag.com/tamino/response2");
```

Instance variables

Then we set up two instance variables. The variable `connection` will hold a Tamino connection object while the variable `accessor` will hold a `TXMLObjectAccessor` instance. Accessor objects provide the necessary methods to query, insert, update, or delete documents in a specific collection and via a specific connection.

```
// The database connection.
private TConnection connection = null;
// The accessor instance, here a high level TXMLObjectAccessor.
private TXMLObjectAccessor accessor = null;
```

Initialization

Both variables are initialized in the class constructor which is executed when the class method `show` creates a new `MusicianCollaborationResult` instance. The connection is not created directly. Instead, we first create an instance of a connection factory.

```
public MusicianCollaborationResult(String databaseURI,
                                   String collection)
                            throws TConnectionException {
// Obtain the connection factory
TConnectionFactory connectionFactory =
                      TConnectionFactory.getInstance();
```

Note: Instances of `TConnectionFactory`, `TXMLObjectAccessor`, `TDOMObjectModel`, and `TJDOMObjectModel` are not created with a `new` instruction but with the class method `getInstance()`.

Then we let the connection factory create the actual connection:

```
// Obtain the connection to the database
   connection = connectionFactory.newConnection( databaseURI );
```

Finally we create the accessor object:

```
// Obtain the concrete TXMLObjectAccessor
   //            with an underlying JDOM object model
   accessor = connection.newXMLObjectAccessor(
                TAccessLocation.newInstance( collection ),
                TJDOMObjectModel.getInstance() );
   }
```

Querying Tamino

Now, that we have established a connection and obtained an accessor object we are ready to query the database:

```
   // show result
   private void show(String keyValue)  throws Exception {
     try  {
       // Build a query and process it
       TResponse response =
          processQuery( "jazzMusician[@ID"+"='" + keyValue + "']" );
```

From the key value passed as a parameter we construct a query string as shown above and hand it over to the private method `processQuery()`. This method constructs a query object from a string parameter and passes it to the `query` method of the accessor object. It also handles the case of a `TQueryException`.

```
   // process query
   private TResponse processQuery(String s) throws Exception {
      TQuery query = TQuery.newInstance( s );
   try {
     // Invoke the query operation.
     return accessor.query( query );
           }
   catch (TQueryException queryException) {
     // Tell about the reason for the failure.
         showAccessFailure(queryException);
         return null;
   }
   }
```

Evaluating the Query Response

This method returns a Tamino response object which can contain one or several result documents. Since we assume that the ID of a jazz musician is unique, we expect at most a single result document. We retrieve this document from the response object and get its top level JDOM element (`jazzMusician`).

```
// Get first (and single) object
if (!response.hasFirstXMLObject())
     throw new Exception("Nothing found");
TXMLObject xmlObject = response.getFirstXMLObject();
// Get top level JDOM element
Element jazzMusician = (Element) xmlObject.getElement();
```

Next, we retrieve collaboration documents that match the key value (jazz musician ID):

```
response = processQuery(
           "collaboration[jazzMusician"+"='" + keyValue + "']" );
```

Because this query may result in several documents we use an iterator object to loop through all result documents:

```
// Iterate over result documents
TXMLObjectIterator collabIt = response.getXMLObjectIterator();
while (collabIt.hasNext()) {
    xmlObject = collabIt.next();
    // Get top level JDOM element
    Element collab = (Element) xmlObject.getElement();
```

Merging Documents

Because we later want to paste these `collaboration` elements into the `jazzMusician` document, we have to *clone* them. This operation removes the context (such as the parent information) from the current node and allows us to insert it into another context (i.e. to merge documents):

```
 // clone to remove context
 collab = (Element) collab.clone();
```

Note: This technique is specific to JDOM. While DOM Level 1 does not support the merging of documents at all, DOM Level 2 introduces an `import`method. This method must be used to import a foreign node into a target document before the foreign node can be added to a node of the target document.

Now we retrieve the content of all `result` elements in each `collaboration` document. Using JDOM methods, we first construct a list of all `result` child elements:

```
// Get a list of all direct children with name "result"
List resultChildren = collab.getChildren("result");
```

Then we iterate over this list and extract the text from each element:

```
// get iterator over children
ListIterator resultIt = resultChildren.listIterator();
// now loop over the "result" children
while (resultIt.hasNext()) {
  // get a single "result" child
  Element resultElement = (Element) resultIt.next();
  // get the content
  String resultID = resultElement.getText();
  // now read album records with resultID as key
```

By now we have obtained the values of all `result` elements in all collaborations of a jazz musician. We can use these values to retrieve the albums that are results of these collaborations. We do so by querying for `album` documents with a `productNo` equal to `resultID`:

```
  // now read album records with resultID as key
  response = processQuery("album[productNo"+"='"
                          + resultID + "']" );
```

Again we expect only a single result document per key:

```
  // Process the album if we have one
  if (response.hasFirstXMLObject()) {
     // get first (and only) result document
     xmlObject = response.getFirstXMLObject();
     // Get top level JDOM element
     Element album = (Element) xmlObject.getElement();
```

Once again, we clone this element and remove all `track` child elements because we are not interested in that information.

```
     // clone to remove context
     album = (Element) album.clone();
     // remove "track" elements
     album.removeChildren("track");
```

We also want to remove the Tamino specific `ino:id` attribute. To identify this attribute we use the constant INO_NAMESPACE that connects the prefix `ino:` with the Tamino namespace URI.

```
        // remove ino:id
        album.removeAttribute("id",INO_NAMESPACE);
```

Then, we simply add the remaining structure to our `collaboration` element stored in `collab` and close the loop:

```
        // add album to collaboration clone
        collab.addContent(album);
    }
```

By now we have joined document `album` to document `collaboration`. In the next step we join the resulting `collaboration` document to document `jazzMusician`.

Similarly, we remove the `result` elements from the `collaboration` element (because this information is already contained in the `productNo` attributes of the added `album` elements), and the `ino:id`.

```
    // remove "result" elements from collaboration
    collab.removeChildren("result");
    // remove ino:id
    collab.removeAttribute("id",INO_NAMESPACE);
```

Then we add the `collaboration` element to our jazz musician element and close the loop:

```
    // and add collab to jazzMusician
    jazzMusician.addContent(collab);
}
```

Done. What remains to do is to print the result:

```
// Output with JDOM output tool
XMLOutputter outputter = new XMLOutputter();
outputter.output(jazzMusician, System.out);
```

and to finally close the connection:

```
// Close the connection.
connection.close();
```

Running the Program

The complete listing is shown in MusicianCollaborationResult.java. This file contains also the necessary code for exception handling consisting of the private method `showAccessFailure` and `try`, `catch`, and `finally` clauses in the method `show`:

We can execute this program from the command line or from our favorite IDE. If we invoke the program with parameter "ParkerCharlie" we should get the following result:

```
<jazzMusician
  xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
  ino:id="2"
  type="instrumentalist"
  ID="ParkerCharlie">
  <name>
    <first>Charlie</first>
    <last>Parker</last>
  </name>
  <birthDate>1920-08-19</birthDate>
  <instrument>saxophone</instrument>
  <collaboration type="jamSession">
    <name>post-election-jam</name>
    <performedAt>
      <location>Blues House</location>
      <time>1945-10-21T20:00:00</time>
    </performedAt>
    <jazzMusician>GillespieDizzy</jazzMusician>
    <jazzMusician>ParkerCharlie</jazzMusician>
    <album>
      <title>Blues House Jam</title>
      <productNo>BGJ-47</productNo>
    </album>
  </collaboration>
</jazzMusician>
```

Testing Integrity Constraints

Our next example is a program that tests for integrity constraints between two document types before inserting a new document. When inserting a new `collaboration` document we want to make sure that the alternative elements `collaboration/performedAt/time` and `collaboration/period/from` do not contain events that take place before the birth date of each participating musician. To make such a test "waterproof" against competing transactions we must perform it in the same transaction as the insert operation. Therefore, prior to the test we establish a local transaction. If the test succeeds we perform the insert operation and explicitly commit the transaction. Otherwise, we simply rollback the transaction.

Again, we write a Java program that can be executed from the command line. The file name of the document which we wish to insert is specified as a parameter.

Preparing for Transactions

The basic setup (constants, instance variables, constructor) is similar to the previous example in section **Joining documents**. However, this time we use the DOM object model instead of JDOM. In addition to establishing a connection we make sure that we run in a transaction safe environment and therefore set the lock mode of the connection to "shared":

```
// Set lock mode to "shared"
connection.setLockMode(TLockMode.SHARED) ;
```

Also, the `main()` method looks similar to the one shown in the previous example:

```
public static void main(String[] args) throws Exception  {
  InsertConstraintCheck insertConstraintCheck =
      new InsertConstraintCheck( DATABASE_URI , COLLECTION );
  // perform the transaction
  insertConstraintCheck.processTransaction(args[0]);
}
```

Processing Transactions

The actual work is done in method `processTransaction()`. Here, we first setup a few variables, read the specified file into a DOM Tamino XML Object, and extract the DOM document object (`collaborationDoc`):

```
  TLocalTransaction myTransaction = null;
  boolean abortTransaction = false;
  try  {
    // Read file into a DOM Tamino XML object.
    // Instantiate an empty TXMLObject instance
    //    related to the DOM object model.
    TXMLObject collaborationObject =
          TXMLObject.newInstance( TDOMObjectModel.getInstance() );
    // Establish the DOM representation by reading the content
    //    from a file input stream.
    collaborationObject.readFrom( new FileInputStream(filename ));
    // get DOM document
    Document collaborationDoc =
          (Document) collaborationObject.getDocument();
```

Now we can start a new local transaction.

```
    // Set local transaction mode and get a transaction object
    myTransaction = connection.useLocalTransactionMode();
```

Validating Constraints

Then we begin with the tests for constraint violation. We first extract the start date of the collaboration – which is defined either by `collaboration/performedAt/time` or alternatively by `collaboration/period/from` – and convert the string value of these elements into a Java `Date` value. Because there are no other `time` and `from` elements in the document, we can access these elements on document level via the DOM method `getElementsByTagName`:

```
    // initialize start date
    Element startDateElement = null;
    // get a "from" elements if defined
    NodeList fromList =
             collaborationDoc.getElementsByTagName("from");
    if (fromList.getLength() > 0) {
      // get the only child
      startDateElement = (Element) fromList.item(0);
    } else {
      // alternatively, get the "time" element
      startDateElement =
   (Element) collaborationDoc.getElementsByTagName("time").item(0);
    }
    // get start date value
    String startDateValue = getText(startDateElement);
    // convert to Date
    Date startDate = toDate(startDateValue);
```

The conversion to the Java `Date` format is done with the private method `toDate` which is shown in the full listing. The text content of element `startDateElement` was extracted with the private method `getText`. Text content is treated in DOM as a separate child element, and consequently we first use the DOM method `getFirstChild()` followed by method `getData()`:

```
// get text content from element
private String getText(Element element) {
  return ((CharacterData) element.getFirstChild()).getData();
}
```

Now, we loop over all `jazzMusician` elements of the new `collaboration` document

```
    // Get jazzMusician elements
    NodeList collaborateurs =
        collaborationDoc.getElementsByTagName("jazzMusician");
    // now loop over the "jazzMusician" children
    for (int i=0; i < collaborateurs.getLength(); i++) {
```

We extract their content and use it to query the database for `jazzMusician` documents:

```
      // get a single "jazzMusician" child
      Element collaborateurElement =
             (Element) collaborateurs.item(i);
      // check if this item has content
      if (collaborateurElement.hasChildNodes()) {
        // get the string content
        String collaborateurID = getText(collaborateurElement);
        // Perform query for jazzMusicians
        // identified by collaborateurID
        TResponse response =
    processQuery("jazzMusician[@ID"+"='" + collaborateurID + "']");
```

For each query we check if we have found a document. If so, we extract the birth date, convert it to the Java `Date` format and compare it with the collaboration start date. If the birth date is larger than the collaboration start date, or if the referenced `jazzMusician` document did not exist, we report an appropriate error message and indicate that the transaction needs to be aborted.

```
   // Process the musician document if we have one
   if (!response.hasFirstXMLObject()) {
     abortTransaction = true;
     System.out.println("Error: Referenced jazzMusician "
                      +collaborateurID+" does not exist");
   } else {
     // get first (and only) result document
     TXMLObject jazzMusicianObject =
                response.getFirstXMLObject();
     // Get top level DOM element
     Document jazzMusicianDoc =
              (Document) jazzMusicianObject.getDocument();
     // get birthDate
     Element birthDateElement = (Element)
       jazzMusicianDoc.getElementsByTagName("birthDate").item(0);
     // get string value
     String birthDateValue = getText(birthDateElement);
     // convert to date
     Date birthDate = toDate(birthDateValue);
     // compare with startDate
     if (startDate.compareTo(birthDate) <= 0) {
        abortTransaction = true;
        // Report violation of integrity constraint
        System.out.println(
"Error: Collaboration start date before birth date of jazz musician "
        +collaborateurID);
     }
   }
```

Completing the Transaction

After we have looped through all collaborators we are ready to insert the new `collaboration` document into the database. If the indicator `abortTransaction` was set, we rollback the transaction. Otherwise we perform the insert operation and commit the transaction.

```
if (abortTransaction) {
  myTransaction.rollback();
  // Report abort of operation
  System.out.println("Error: Insert not performed");
} else {
  performInsert( collaborationObject );
  myTransaction.commit();
  // Show the collection, doctype and id
 System.out.println(
   "Message: Insert succeeded, ino:collection="
   + collaborationObject.getCollection() + ", ino:doctype="
   + collaborationObject.getDoctype() +", ino:id="
   + collaborationObject.getId() );
}
```

The actual insert operation is performed in the private method `performInsert()` which is quite similar to the previous `processQuery()` method, and which is shown in the full listing in InsertConstraintCheck.java.

Running the Example

When we execute this program with parameter *C:/projects/jazz/post-election-jam.xml* (or wherever else this file may be stored) we get a protocol similar to the following:

```
Message: Insert succeeded, ino:collection=encyclopedia, ino:doctype=collaboration, ↩
ino:id=3
```

However, if we change the `time` entry in this document from

```
    <time>1945-10-21T20:00:00</time>
```

into

```
    <time>1915-10-21T20:00:00</time>
```

and try to insert it again, we obtain:

```
Error: Collaboration start date before birth date of jazz musician GillespieDizzy

Error: Collaboration start date before birth date of jazz musician ParkerCharlie
```

```
Error: Insert not performed
```

Note: As a matter of fact, we would need similar checks when we update a `collaboration` document and, of course, when we update `jazzMusician` documents.

Testing for Unique Keys

The third example deals with the problem of inserting a document that has a primary (unique) key. Both our `jazzMusician` and `album` document are equipped with a primary key: `jazzMusician` with the attribute `ID`, and `album` with the element `productNo`. When we want to add one of these documents, we must make sure that a document with the same key value does not already exist in the database.

Strategy

This must be done in a transactionally safe way, and there are several ways to achieve that. For the purpose of this example, we have chosen an optimistic method which consists of the following steps:

1. Start a transaction.
2. IInsert the document with the lock mode set to "shared".
3. Retrieve all documents with the same key value and with lock mode set to "unprotected".
4. If there are more than one document returned, rollback the transaction. Otherwise commit the transaction.

However, this leaves us with a problem. Tamino does not allow us to change the lock mode of a connection within a transaction. The solution is to open two connections. Connection A is set to "shared" and performs the steps 1, 2, and 4. Connection B is set to "unprotected" and performs step 3.

Initialization

Consequently, we have to initiate two database connections and two accessor instances. Here are the definition for the instances variables:

```
// Database connection A
private TConnection connectionA = null;
// Accessor A
private TXMLObjectAccessor accessorA = null;
// Database connection B
private TConnection connectionB = null;
// Accessor B
private TXMLObjectAccessor accessorB = null;
```

The initialization, which is performed in the constructor of the class, looks like this:

```
public InsertUnique (String databaseURI,String collection)
                                throws TConnectionException {
  // Obtain the connection factory
  TConnectionFactory connectionFactory =
                                TConnectionFactory.getInstance();
  // Obtain the first connection to the database
  connectionA = connectionFactory.newConnection( databaseURI );
  // Obtain the concrete TXMLObjectAccessor with
  //    an underyling DOM object model
  accessorA = connectionA.newXMLObjectAccessor(
               TAccessLocation.newInstance( collection ) ,
               TDOMObjectModel.getInstance() );
  // Set local transaction mode to "shared"
  connectionA.setLockMode(TLockMode.SHARED) ;
  // Obtain the second connection to the database
  connectionB = connectionFactory.newConnection( databaseURI );
  // Obtain the second accessor
  accessorB = connectionB.newXMLObjectAccessor(
               TAccessLocation.newInstance( collection ) ,
               TDOMObjectModel.getInstance() );
  // Set lock mode of connection 2 to "unprotected"
  connectionB.setLockMode(TLockMode.UNPROTECTED) ;
}
```

Processing the transaction

Our example program `InsertUnique` is written in a generic way. It accepts as parameters the name of the XML file to be inserted and the name of the key to test (if the key is an attribute it is prefixed with an @). These parameters are passed to the private method `processTransaction()`.

Similar as in the previous example in section **Testing integrity constraints** we first read the document to be inserted from an XML file:

```
private void processTransaction(String filename, String key)
                                          throws Exception {
  TLocalTransaction myTransaction = null;
  try  {
    // Read file into a DOM Tamino XML object.
    // Instantiate an empty TXMLObject instance
    //     related to the DOM object model.
    TXMLObject xmlObject =
        TXMLObject.newInstance( TDOMObjectModel.getInstance() );
    // Establish the DOM representation
    //    by reading the content from a file input stream.
    xmlObject.readFrom( new FileInputStream(filename ));
    // get DOM document
    Document doc = (Document) xmlObject.getDocument();
    // get top level element
    Element root = (Element) xmlObject.getElement();
```

Then we try to get the value of the key element or key attribute. In case of an attribute we first must remove the @ from the key name which is done with method `substring(1)`.

```
    // get key value
    String keyValue = null;
    // check if key is an attribute or an element
    if (key.startsWith("@")) {
        // get attribute value
        keyValue = root.getAttribute(key.substring(1));
    } else {
      // get element node list
      NodeList nl = doc.getElementsByTagName(key);
      if (nl.getLength() == 0)
                    throw new Exception("Key not found");
      // get only element
      Element elem = (Element) nl.item(0);
      // get element content
      keyValue = getText(elem);
    }
    // Check for proper content
    if (keyValue == "") throw new Exception("Key not found");
```

The private method `getText` retrieves the text content of an element and is defined as in the previous example in section **Testing integrity constraints**.

Now we can start a transaction on connection A and insert the document:

```
    // Start the transaction
    myTransaction = connectionA.useLocalTransactionMode();
    // Insert the document
    performInsert( xmlObject );
```

Checking the unique constraint

Then we ask for the number of documents matching the key value. We do this through connection B which was set to lock mode "unprotected". If the number of matching documents is unequal 1 we rollback the transaction, otherwise we perform a commit.

```
    // Get number of matching documents
    int c = getCount( xmlObject.getDoctype()
                      + "["+key+"='" + keyValue + "']" );
    if (c == 1) {
      // Unique - commit the transaction
      myTransaction.commit();
      System.out.println("Transaction committed");
    } else {
      // Bad - rollback the transaction
      myTransaction.rollback();
      throw new Exception("Key not unique: "
```

```
                          +c+" occurrences. Transaction aborted.");
    }
```

Counting documents

The number of matching documents is determined by private method `getCount()`. This method simply wraps an XQuery `count()` function around the query string, performs the query, and converts the result into an integer format.

```
private int getCount(String path) throws Exception {
  try  {
    // Construct TQuery object
    TQuery query = TQuery.newInstance("count("+path+")");
    // perform the query
    TResponse response = accessorB.query(query);
    // get the number of documents found
    String s = response.getQueryContentAsString();
    // convert to integer
    return Integer.valueOf(s).intValue();
  }
  catch (TQueryException queryException)  {
    showAccessFailure( queryException );
    return 0;
  }
}
```

The complete code of class `InsertUnique` is contained in InsertUnique.java.

Running the Example

We can test this program by invoking it with the parameters

"C:\projects\jazz\dizzy.xml" and "@ID".

We should get the following result (because we previously have already added *dizzy.xml* to the database):

```
   java.lang.Exception: Key not unique: 2 occurrences. Transaction aborted.
     at ↩
com.softwareag.tamino.db.api.examples.jazz.InsertUnique.processTransaction(InsertUnique.java:123)
     at ↩
com.softwareag.tamino.db.api.examples.jazz.InsertUnique.main(InsertUnique.java:145)
   Exception in thread "main"
```

10 Reference Documentation

The reference documentation for the Tamino API for Java is provided in Javadoc format. In the HTML version, this link leads to the index page for the documentation in Javadoc.

In the PDF version of this document, you will find the documentation for the Tamino API for Java right after this section.

VI Webserverless Access Via the Tamino API for Java

11 Webserverless Access Via the Tamino API for Java

The Tamino API for Java, in common with the other Tamino APIs, communicates with the Tamino XML Server via an HTTP API through a web server. With this release of Tamino XML Server 4.1, a Tamino database can now also be accessed without a web server through the Tamino API for Java. This is achieved by the plug-in architecture of the API providing access the webserverless base C API.

Usage

The distinction, whether the Tamino API for Java should access the Tamino XML Server via the web server or without the web server, is done using the protocol tag in the URL. For example, if a database can be accessed via *http://localhost/tamino/mydb/*, a URL such as *wsl:///tamino/mydb/* would access the same database without going through the web server. When opening a connection to a Tamino database using the webserverless mode via the call `TConnectionFactory.getInstance().newConnection( DB_URI )`, the variable `DB_URI` must specify the protocol *`wsl`* instead of *`http`*, e.g. *wsl:///tamino/mydb*. Since all connections via WSL have to use the eXtended Transport Services (XTS) from Software AG, the hostname and the portnumber are left blank in the URL and the database is accessed through its unique database name, as in the above example *`mydb`*. This is the only change required when the webserverless mode is to be used, i.e. to replace the protocol in all `DB_URI`s by *`wsl`* and leave out the hostname and the portnumber.

In order to use the correct XTS server, which knows about the desired database, either set the environment variable `XTSDSURL` locally or adjust the entry in the file */etc/hosts*. The latter is normally done automatically when the XTS package is installed.

Installation

In order to use the webserverless mode of the Tamino API for Java, the following points must be considered:

1. **Windows:**

   The directory containing the library *modwsl.dll* must be in the path. This is usually done at installation time by adding the variable `%SAG_COMMON%` to the `path` variable.

   UNIX:

   The directory containing the library *libmodwsl.so* must be added to the environment variable `LD_LIBRARY_PATH`. This is usually done by calling the setup script */opt/sag/sagenv.new*.

2. The option `-Djava.protocol.handler.pkgs=com.softwareag.tamino.db.protocols` must be supplied at startup to the Java application, i.e. to the JVM.

Security Considerations

The webserverless mode requires that the Software AG product XTS is installed. It can e.g. be found on the Tamino XML Server distribution media.

In order to ensure that a particular database can only be accessed through a particular webserver (or webserverless client), one defines the webserver using the **Web Server Management** object in the tree-view frame of the System Management Hub. This webserver can then be assigned to a particular database via the **Web Servers** object in the tree-view frame.

Limitations

Since the webserverless mode does not require a web server to access Tamino, the application should not use direct HTTP calls to communicate with Tamino databases; it should only use methods provided by the Tamino API for Java.

Only databases of a Tamino XML Server with a version number of 4.1 or higher can be accessed via the webserverless mode.

VII Performance Tips and Tricks

12 Performance Tips and Tricks

Experience and tests using the Tamino API for Java in conjunction with Software AG's Tamino XML Server have brought to light some aspects that we would like to share with our customers. Awareness of the following tips and tricks can considerably improve the performance of an application based on the Tamino API for Java.

Using Cursoring

When using cursoring, find a good compromise for the cursor size (pagesize). A small cursor size requires a lot of requests to be sent to Tamino and therefore slows down the processing. A large cursor size locks a lot of memory on the client. Typically there is a certain size that you can find in practical tests, where an increase in cursor size does not speed up the whole process any more. Make practical tests, since changing the cursor size can make a big difference if you walk through a long result set.

Using XML Parsers

- While competing and new XML parsers like Piccolo (from Sourceforge) or XPP promise improved performance times, we were not able to measure a significant difference. XPP also requires you to switch to another object model for representing your documents (XmlNodes), which makes it not recommendable for most cases. Xerces, as a parser, is used for DOM and JDOM. DOM and JDOM perform on the same level. Sometimes one is better, sometimes the other, if we look at the performance of the creation process for new nodes in the object model.
- Xerces and DOM use the not very well known feature of deferred node creation. That means that sub nodes are only created when they are accessed. This feature is ON by default and it is not possible to see from the API which nodes actually have already been created. However, while this is a big advantage when reading large documents (with a lot of nodes) without actually accessing every node, it turns into a disadvantage once you touch the nodes because then they are created and consequently more time is needed.

Using Large XML Documents with Many Nodes

- When reading documents with a large number of individual nodes, even if you do not need to read all the nodes, it is helpful when the document can only be partially parsed and the complete document object model is only built when needed.
- Xerces uses a feature called deferred node creation, which means that while using the regular DOM interface, node levels below the current level are only created when the client tries to access them.

- Pull parsers require some more work from the client programmer, but they also support modes where the nodes are either transparently built as the client accesses the data (similar to deferred nodes creation in Xerces) or where events are read (similar to the SAX interface).

VIII Measuring Operation Duration

13 Measuring Operation Duration

Tamino is a fast, flexible, and highly scalable DBMS system. However, when writing high-performance, large-scale Tamino applications, the first step in finding potential bottlenecks is to discover where the application spends most of its time.

Tamino API for Java provides basic mechanisms for gathering detailed information about execution times. Tamino API for Java does not include statistical tools to summarize or process the measurement results. Instead, publicly available handler classes can be used to measure values.

• **Operation and Measured Values**	An overview of the operations for which measuring is supported and the different values that can be measured
• **Architecture and Technical Concepts**	The architecture and technical concepts used for measuring and monitoring
• **Controlling Duration Measurement**	The properties to control the measuring
• **Running the Get Personal Example with Duration Measurement**	An example which demonstrates how to use the "Measuring Operation Duration" feature

Operation and Measured Values

Measuring support is provided for the following operations in the Tamino API for Java:

Creating new connections in the `TConnectionFactory` class.
All operations of the various accessor interfaces with the exception of the operations inherited from superinterfaces `TAccessor` and `TInvalidatableAccessor`.
All operations in the interfaces `TXMLObjectIterator` and `TNonXMLObjectIterator`.
Creating new `TXMLObject` instances or initializes them by reading data from an `InputStream` or a `Reader`.

Measured Values

For the operations listed above, the following values can be measured:

`TotalOperationDuration`	The time taken from the start of the operation until the end of the operation. This is the total processing time within the Tamino API for Java, including the values measured by `XmlParseDuration` and `TotalCommunicationDuration`.
`XmlParseDuration`	The time taken to parse the XML response document.
`TotalCommunicationDuration`	The time taken from submitting the request to Tamino until the response document is completely received. Note that this includes the value for `TaminoServerDuration`.

`TaminoServerDuration`	The time taken to process the request in the Tamino XML Server.

Even if the expected result should produce values greater than zero, this will not necessarily be the case for all values measured. Moreover, this depends on the operation performed. For example, when deleting a document no relevant parsing of response documents needs to be done.

Accuracy of Measured Values

All values are measured in milliseconds. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the operating system and may in some cases be larger. For example, many operating systems measure time in units of tens of milliseconds. Hence fast operations may deliver a value of zero because their duration was below the minimum granularity. Also note that the sum of individual durations may not be exactly equal to the total operation duration. This is because measurements are made in different processes and possibly even on different computers.

The "Measuring Operation Time" feature of the Tamino API for Java cannot answer the question: *Exactly how long does operation x with data y take?*. Rather, it helps you to answers questions such as: *Where does the application spend most of its time?* or: *Does the program run faster after making a change?*

Architecture and Technical Concepts

The Tamino API for Java uses the popular log4j API to trace the measured values. log4j is an open source project to develop a logging package for Java. It allows the developer to control the output of log statements with arbitrary granularity. It is fully configurable at runtime using external configuration files. This documentation does not describe the log4j API. The latest log4j version and documentation can be found at **http://logging.apache.org/**.

Configuration of the log4j environment is typically done at the initialization of the application, preferably by reading a configuration file. Please see the log4j documentation for further information.

The log4j logger used by the Tamino API for Java for measuring operation duration is named `com.softwareag.tamino.db.api.logging`.

Controlling Duration Measurement

The following Java properties can be set to control the measuring:

com.softwareag.tamino.db.api.logging.DurationLogging	
Description	Specifies whether duration measurement in the Tamino API for Java is enabled.
True	Perform duration measurement.
False	Do not perform duration measurement.
Default	False

com.softwareag.tamino.db.api.logging.DurationLoggingLevel	
Description	Choose the level used by the duration logging for the messages sent to the underlying log4j logger.
Default	DEBUG

com.softwareag.tamino.db.api.logging.DurationLoggingPattern	
Description	A simple pattern to configure the output string used by the duration logging for the messages. The name of the operation and the measured duration values are symbolized by the following wildcards: %OPERATION% %TOTAL_OPERATION_DURATION% %TOTAL_COMMUNICATION_DURATION% %TAMINO_SERVER_DURATION% %XML_PARSE_DURATION%
Default	Timekeeper[operation=%OPERATION%, totalOperationDuration=%TOTAL_OPERATION_DURATION%, totalCommunicationDuration=%TOTAL_COMMUNICATION_DURATION%, taminoServerDuration=%TAMINO_SERVER_DURATION%, xmlParseDuration=%XML_PARSE_DURATION%]

com.softwareag.tamino.db.api.common.TOptimizationHints.isOn	
Description	The Tamino API for Java tries to optimize the interpretation of the Tamino response. A side effect is that sometimes the value for the TaminoServerDuration cannot be provided. This property does not really influence the duration measurement itself, so you may consider setting it to *False* to get as much measurement information as possible.

com.softwareag.tamino.db.api.common.TOptimizationHints.isOn	
Default	True

Running the Get Personal Example with Duration Measurement

The following section shows an example of how to measure operation duration.

- Prerequisites for Running the Samples
- Configure the log4j Environment
- Run Example with Measuring Activated

Prerequisites for Running the Samples

Include the *.jar* files *TaminoAPI4J.jar*, *JavaTaminoAPIExamples.jar*, *log4j.jar*, *xercesImpl.jar* and *xml-ParserAPIs.jar* in your class path. All files are distributed along with the Tamino API for Java.

Configure the log4j Environment

This sample log4j configuration file results in writing the output of the duration measurement in a file *Durations.txt*.

```
# Print only messages of the Tamino API for Java duration measurement.
# Set logger level to DEBUG and its only appender to
log4j.logger.com.softwareag.tamino.db.api.logging=DEBUG, A
# A is set to be a FileAppender.
log4j.appender.A=org.apache.log4j.FileAppender
log4j.appender.A.File=Durations.txt.
# A uses PatternLayout to configure the output string
log4j.appender.A.layout=org.apache.log4j.PatternLayout
log4j.appender.A.layout.ConversionPattern=%m%n
```

Run Example with Measuring Activated

The following command executes the Java interpreter:

```
java -Dcom.softwareag.tamino.db.api.logging.DurationLogging=true ↩
com.softwareag.tamino.db.api.examples.person.ProcessPersonsWithSchema
```

The output in the *Durations.txt* file will look like the following:

```
Timekeeper[operation=newConnection, totalOperationDuration=109, ↩
totalCommunicationDuration=31, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=isServerAlive, totalOperationDuration=312, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=281]
Timekeeper[operation=getServerVersion, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=getServerAPIVersion, totalOperationDuration=16, ↩
totalCommunicationDuration=16, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=getAPIVersion, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=newXMLObject, totalOperationDuration=15, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=readFrom, totalOperationDuration=16, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=16]
Timekeeper[operation=define, totalOperationDuration=390, ↩
totalCommunicationDuration=390, taminoServerDuration=370, xmlParseDuration=0]
Timekeeper[operation=newXMLObject, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=newXMLObject, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=readFrom, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=insert, totalOperationDuration=94, ↩
totalCommunicationDuration=78, taminoServerDuration=80, xmlParseDuration=16]
Timekeeper[operation=newXMLObject, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=readFrom, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=insert, totalOperationDuration=78, ↩
totalCommunicationDuration=62, taminoServerDuration=50, xmlParseDuration=16]
Timekeeper[operation=newXMLObject, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=readFrom, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=insert, totalOperationDuration=78, ↩
totalCommunicationDuration=62, taminoServerDuration=20, xmlParseDuration=16]
Timekeeper[operation=newXMLObject, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=readFrom, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=insert, totalOperationDuration=47, ↩
totalCommunicationDuration=31, taminoServerDuration=10, xmlParseDuration=16]
Timekeeper[operation=newXMLObject, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=readFrom, totalOperationDuration=0, ↩
totalCommunicationDuration=0, taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=insert, totalOperationDuration=47, ↩
totalCommunicationDuration=31, taminoServerDuration=10, xmlParseDuration=16]
Timekeeper[operation=query, totalOperationDuration=31, totalCommunicationDuration=15, ↩
taminoServerDuration=10, xmlParseDuration=16]
Timekeeper[operation=hasNext, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
```

```
Timekeeper[operation=query, totalOperationDuration=47, totalCommunicationDuration=47, ↩
taminoServerDuration=10, xmlParseDuration=0]
Timekeeper[operation=hasNext, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=next, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=hasNext, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=next, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=hasNext, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=next, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=hasNext, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=next, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=hasNext, totalOperationDuration=0, totalCommunicationDuration=0, ↩
taminoServerDuration=0, xmlParseDuration=0]
Timekeeper[operation=delete, totalOperationDuration=32, ↩
totalCommunicationDuration=32, taminoServerDuration=30, xmlParseDuration=0]
Timekeeper[operation=undefine, totalOperationDuration=578, ↩
totalCommunicationDuration=578, taminoServerDuration=551, xmlParseDuration=0]
```

IX Appendix: Examples in Code

14 Appendix: Examples in Code

This appendix contains instructions on where to find the Java files delivered with the documentation of the Tamino API for Java. The root directory of the specified file locations is the directory containing the Tamino API documentation, which is *<TaminoDocRootDir>SDK/TaminoAPI4J/Documentation/inoapi/*

Say Hello!

The *.java* files are located in the subdirectory */listings/examples/greeting*.

```
listings
 |
 > examples
    |
    > greeting
       |
       XMLGreeting.java
       XMLGreetingDOM4J.java
       ProcessXMLGreeting.java
       ProcessXMLGreetingDOM4J.java
       NonXMLGreeting.java
       ProcessNonXMLGreeting.java
       |
       > SAX
          |
          ProcessGreeting.java
          GreetingDefaultHandler.java
          Greeting.java
          ElementDefaultHandler.java
          DocumentDefaultHandler.java
  ↩
```

Persons

The *.java* files are located in the subdirectory */listings/examples/person*.

```
listings
 |
 > examples
    |
    > person
       |
       ProcessPersons.java
       ProcessPersonsDOM4j.java
       ProcessPersonsWithSchema.java
       ProcessPersonsWithSchemaDOM4J.java
```

All that Jazz

The *.java* files are located in the subdirectory */listings/examples/jazz*.

```
listings
 |
 > examples
    |
    > jazz
       |
       InsertConstraintCheck.java
       InsertConstraintCheckDOM4J.java
       InsertUnique.java
       InsertUniqueDOM4J.java
       MusicianCollaborationResult.java
       MusicianCollaborationResultDOM4J.java
```

DOM4J

The *.java* files are located in the subdirectories */listings/objectModel/dom4j* and */listings/response/dom4j*.

```
listings
   |
   > objectModel
       |
       > dom4j
          |
          TDOM4JAdapter.java
          TDOM4JElementIterator.java
          TDOM4JNamespaceStack.java
          TDOM4JObjectModel.java
          TDOM4JXMLOutputter.java
```

```
listings
   |
   > response
       |
       > dom4j
          |
          TDOM4JInputStreamInterpreter.java
```

Index