software AG

Tamino API for .NET

Version 8.2.2

October 2013

tamino XML Server

This document applies to Tamino API for .NET Version 8.2.2.

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

Document ID: TAN-DOC-822-20130226

Table of Contents

Tamino API for .NET

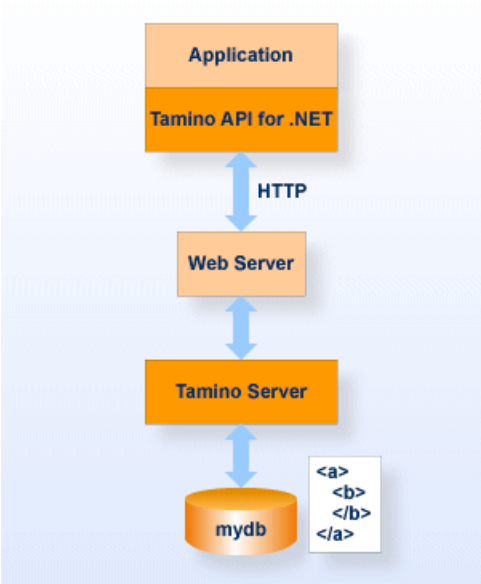
This document provides information for using the Tamino API for .NET, which is available on Microsoft Windows platforms.

The information is in the form of narrative descriptions, examples (code snippets and runnable programs), and reference documentation, which is derived from the source code of the API classes. The documentation is intended for software developers who are writing applications that use Microsoft .NET to access information stored in a Tamino database. You should be familiar with Tamino, with Microsoft's .NET Framework, and with the C# programming language (although the application can be written in any programming language that is supported by .NET).

1 Introduction

The Tamino API for .NET provides an object-oriented programming interface to the Tamino XML Server for .NET applications.

If you are writing applications for Microsoft's .NET Framework, you will find the Tamino API for .NET is the ideal interface for all accesses to the Tamino database.



The API, which is completely written in C#, supports the .NET XML classes for processing XML documents in a Tamino database.

The `TaminoConnection`, `TaminoCommand`, `TaminoTransaction` and `TaminoDataAdapter` classes adopt the basic architecture of a .NET data provider, which serves as a bridge between the application and the database. The `TaminoDataAdapter` class allows you to work with disconnected .NET datasets.

The `TaminoResponse`, `TaminoQueryResponse` and `TaminoDocument` classes provide convenient access to the Tamino serialization format. The `TaminoPageIterator` and `TaminoItemIterator` classes allow easy use of the Tamino-specific cursoring facilities.

The installation includes the API assembly, the comment documentation for the classes in the API, and a number of **samples**. The samples, which are provided as C# source code, demonstrate how to work with the API in various ways.

I Tamino API for .NET Component Profile and Set-up

2 Tamino API for .NET Component Profile and Set-up

This chapter lists the prerequisites and procedures for successfully installing the Tamino API for .NET.

Component Profile

Here you will find general information about the Tamino API for .NET and how to deploy it.

Tamino Component Profile for the Tamino API for .NET	
Supported Platforms	Any hardware and operating-system platform that supports both Software AG's Tamino XML Server and Microsoft's .NET.
Location of Installed Component	*TaminoInstallDir*/*SDK*/*TaminoAPI4DotNET* (henceforth called *TaminoAPIDir*)
Component Files	**Library:** *TaminoAPIDir*/*lib*/*TaminoAPI.dll* *TaminoAPIDir*/*lib*/*TaminoAPI.xml*
	Documentation: *TaminoInstallDir*/*Documentation*
	Samples: *TaminoAPIDir*/*samples*
Bundled Software	none

Installation

The Tamino API for .NET is installed with the Tamino XML Server installation. For more information, refer to the installation section of the Tamino XML Server documentation.

The installation procedure installs *TaminoAPI.dll* automatically. In order to use it, you must either reference the DLL explicitly in your .NET project, or register it in the normal manner.

Deployment

To work with the Tamino API for .NET

1 You need a running Tamino database server in order to be able to use an API. If you want to access an existing Tamino database, you need to know the database URI, for example: *http://localhost/tamino/mydb*).

2 The *TaminoAPI.dll* is a managed .NET assembly. The easiest way to develop an application with Microsoft Visual Studio .NET is explicitly to add a reference to the *TaminoAPI.dll* from your Visual Studio .NET project. When adding this reference, ensure that *TaminoAPI.xml* is in the same directory as *TaminoAPI.dll*. If this is not the case, the Visual Studio .NET tooltips will not work correctly.

3 *TaminoAPI.dll* is a strong-named assembly. You can use either of these techniques to use it from your application:

- Copy *TaminoAPI.dll* into your application directory;
- Use the appropriate .NET tools to install *TaminoAPI.dll* in the global assembly cache (GAC).

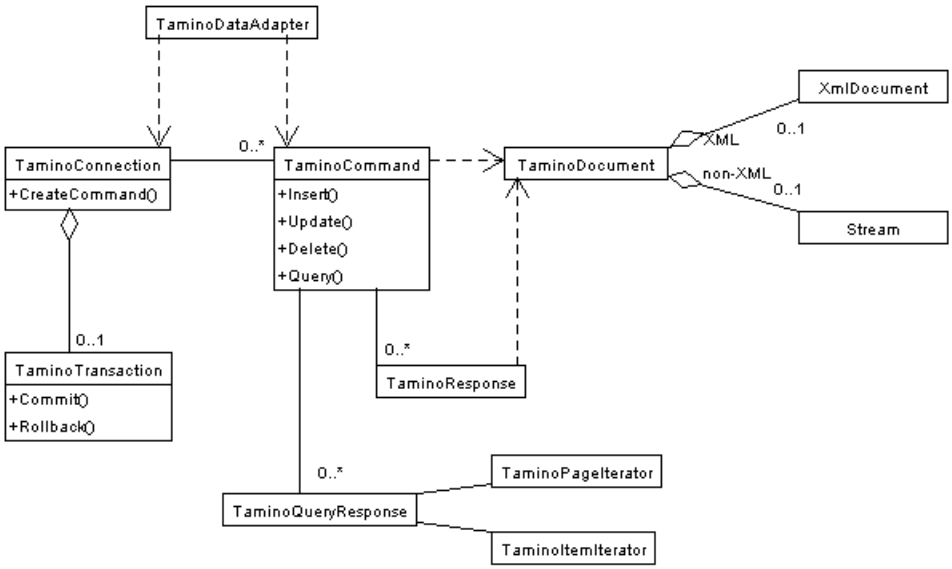
To use the Tamino API for .NET examples

- Please refer to the README file, which you can find at the location *TaminoAPIDir*/*samples/readme.txt*. This includes information for building the samples, loading test data, running the samples, and cleaning up.

II Architectural Overview

3 Architectural Overview

The following graphic shows a high-level UML class diagram of the Tamino API for .NET:



The most frequently used API classes include the following:

`TaminoConnection`
: Represents a connection to a Tamino database. A connection is necessary in order to be able to create `TaminoCommand`s so that the application can perform operations on the Tamino database.

`TaminoCommand`
: Provides basic Tamino database commands. The class can handle XML data as well as non-XML (binary) data.

`TaminoResponse`
: Represents information that Tamino returns in response to a command invocation.

`TaminoQueryResponse`
: Represents information that Tamino returns in response to a query command invocation. Methods are provided to permit iteration over the result set.

`TaminoPageIterator`
: Provides page-by-page iteration over the result set of a Tamino query.

`TaminoItemIterator`
: Provides item-by-item iteration, either over the whole result set of a Tamino query or over a single page of the result set.

`TaminoTransaction`
: Represents a Tamino database transaction.

`TaminoDataAdapter`
: Provides support for filling a dataset from Tamino and writing changes made to the dataset back to Tamino.

The basic API classes allow you to use the full power of XML support in the .NET Framework. Programming using `TaminoDataAdapter` and datasets is useful for Rapid Application Development (RAD) and simple XML data.

Example

The following example illustrates a very simple program:

```
// create connection
TaminoConnection = new TaminoConnection("http://localhost/tamino/mydb");

// open connection
connection.Open(TaminoConnectionMode.AutoCommit);

// create command
TaminoCommand command = connection.CreateCommand("mycollection");

// do simple insert
TaminoResponse response = command.Insert(new TaminoDocument(xmlDoc));
Trace.Assert(response.ReturnValue == "0", response.ErrorText);

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

Note: A large set of **samples** is provided as part of the installation. They demonstrate various facets of using the Tamino API for .NET.

III Programming with the Tamino API for .NET

An application can access XML documents that are stored in a Tamino database in one of two ways:

- Using the .NET classes in the System.Xml namespace;
- Using .NET datasets.

The latter choice, i.e. using .NET datasets, is generally recommended for applications that process less complex ("flat") XML document structures. In this case, the application uses the `TaminoDataAdapter` class, as described in the chapter **Working with Datasets and the `TaminoDataAdapter`**.

In order to process Tamino documents using the System.Xml classes, the application works directly with the core classes such as `TaminoConnection` and `TaminoCommand`. Before it can process Tamino documents, the application must:

1. Create a `TaminoConnection` object (see **Working with Connections**);
2. Get a `TaminoCommand` or `TaminoSchemaCommand` object from the `TaminoConnection` object (see **Working with Commands**).

The following chapters describe how to perform particular tasks.

4 Working with Connections

Before an application can run any commands against the Tamino database, it must create and open a `TaminoConnection` object.

This section describes the following functions for managing connections:

Creating a Connection

When creating a `TaminoConnection` object, the URL of the Tamino database must be specified as a parameter.

Example

```
TaminoConnection connection = new TaminoConnection("http://myserver/tamino/mydb");
```

A `TaminoConnection` object can be opened and closed repeatedly.

Opening a Connection

A connection can be opened in one of several different connection modes. The connection mode defines how transactions are processed within this connection. For more information about the different connection modes, see dotnetapiref/SoftwareAG.Tamino.Api.TaminoConnection-Mode.html.

Example

```
connection.Open(TaminoConnectionMode.AutoCommit);
```

Closing a Connection

You should call the `Close` method in the `TaminoConnection` object when the connection is no longer required. A connection is *not* implicitly released when the connection object falls out of scope or is reclaimed by the garbage collector.

Optimizing Multithreaded Performance

Under certain circumstances, you may experience performance degradation when running a multithreaded application against a single Tamino server. This is caused by the Microsoft .NET Framework, which uses a single ServicePoint per unique domain URI. By default, a ServicePoint supports just two permanent HTTP connections, in accordance with the HTTP specification.

If your system is affected by this problem, you can improve performance by setting `ServicePointManager.DefaultConnectionLimit` to a value greater than 2. Please note that you should only do this if you are experiencing performance problems.

5 Working with Commands

All data processing is done via two classes: `TaminoCommand` and `TaminoSchemaCommand`. The `TaminoCommand` class is used to process Tamino documents, whereas the `TaminoSchemaCommand` class provides methods for manipulating Tamino document type information.

Each command object is always associated with a specific connection object. The `TaminoConnection` class provides methods for creating `TaminoCommand` and `TaminoSchemaCommand` objects. It is not essential for the connection to be open when creating commands, although as good programming practice we recommend opening the connection.

Creating TaminoCommand Objects

Use the `CreateCommand` method in the `TaminoConnection` class to create a `TaminoCommand` object. The method takes the Tamino collection name as a parameter. All commands of a specific `TaminoCommand` object are executed relative to this Tamino collection. The only exception to this rule is if the application works with URIs (see chapter **Working with URIs**).

Examples

```
...
TaminoCommand command1 = connection.CreateCommand("mycollection");
connection.Open(TaminoConnectionMode.AutoCommit);
...
TaminoCommand command2 = connection.CreateCommand("mycollection2");
...
```

Creating TaminoSchemaCommand Objects

Use the `CreateSchemaCommand` method in the `TaminoConnection` class to create a `TaminoSchemaCommand` object. In contrast to `TaminoCommand` objects, a `TaminoSchemaCommand` object is not associated with a specific Tamino collection.

6 Inserting, Retrieving, Updating and Deleting XML Documents

The `TaminoCommand` class provides methods for inserting, retrieving, updating and deleting documents. To get a `TaminoCommand` object, use the `CreateCommand` method in the `TaminoConnection` object.

The methods work with two types of object:

- An object that implements the `ITaminoDocument` interface;
- `TaminoUri` objects.

This chapter describes the most common ways of using those objects together with the `Insert`, `Retrieve`, `Update` and `Delete` methods of the `TaminoCommand` class. The chapter **Working with URIs** describes more advanced usage of the `TaminoUri` class.

ITaminoDocument, TaminoDocument and TaminoUri

When working with Tamino documents, several different kinds of information are needed:

- The actual data as stored in Tamino;
- Information that characterizes the document, for instance whether it is XML or not;
- Information that identifies the document in Tamino.

The `ITaminoDocument` interface encapsulates all three kinds of information. It allows a Tamino XML document to be passed as a self-describing object within the application.

An object of the `TaminoUri` class represents information that describes how to find a document in Tamino.

ITaminoDocument and TaminoDocument

The `Data` property of the `ITaminoDocument` interface holds the actual XML document. In addition, the interface contains properties including the Tamino document type and the document name that identify a Tamino XML document uniquely. Depending on the operation, this identifying information must be specified before the Insert, Update, Delete is called; alternatively, this information is set from the `TaminoCommand` object before the method call returns.

The value of the `DocName` property is either the internal ID used by Tamino, or a unique name that the application has given to the document. An internal ID always starts with a leading "@". If an application-specific unique name exists, the unique name takes preference.

The Tamino API for .NET contains two classes that implement the `ITaminoDocument` interface:

- SoftwareAG.Tamino.Api.TaminoDocument
- SoftwareAG.Tamino.Api.Serialization.TaminoXmlDocument

This chapter focuses on the `TaminoDocument` object. `TaminoXmlDocument` is more appropriate when processing query results; for more information, see the section **Processing Document Lists**.

TaminoUri

A Tamino URI can be ether absolute or relative. In this chapter, all `TaminoURI` objects are relative to the Tamino collection of the `TaminoCommand` object. They have the following structure:

```
Tamino_document_type/document_name
```

The document name can be either the Tamino internal D with a leading "@" or the unique name that the application has given to the document. For more advanced usage scenarios of the `TaminoUri` class, see the chapter **Working with URIs**.

The following describes how to insert, update, retrieve and delete XML documents. For information about handling error situations, see the chapter **Handling Responses and TaminoExceptions**.

Inserting XML Documents

An application can insert XML documents with or without specifying a unique name. In each case the `TaminoCommand` class sets the `DocType` property to the correct value before the method returns. If no unique name has been specified when calling `Insert`, then the `DocName` property of the `ITaminoDocument` interface will contain the Tamino internal ID after the call.

The following shows how to insert an XML document without specifying a unique name. See also the InsertDocuments example in the SimpleSamples.

Example

```
...
// create a document
XmlDocument doc = new XmlDocument();
doc.LoadXml(...);

TaminoDocument tamDoc = new TaminoDocument(doc);
command.Insert(tamDoc);
...
```

To insert an XML document with a unique name, set the `DocName` property accordingly before calling the `Insert` method. This name must not start with "@". See also the InsertWithName example in the Advanced Samples.

Example

```
...
// create a document
XmlDocument doc = new XmlDocument();
doc.LoadXml(...);

TaminoDocument tamDoc = new TaminoDocument(doc);
tamDoc.DocName = "MyUniqueName";
command.Insert(tamDoc);
...
```

If you want to insert a document with a unique name and the XML document uses namespaces, you must observe the following rule: If the name of the Tamino document type is not the same as the fully-qualified name of the document's root element, then you must set the `DocType` property explicitly before calling the `Insert` method.

Updating and Deleting XML Documents

Before updating or deleting an XML document, the DocName and DocType properties must contain valid values. This will automatically be the case if, for example, a preceding Insert or Retrieve has been processed.

If the application creates a new `TaminoDocument` object from scratch, it must set the `DocName` property. The `DocType` property is automatically set to the fully qualified name of the document's root element. If namespaces are being used and the fully-qualified name of the document's root element does not match the name of the corresponding Tamino document type, the application must set the correct DocType value explicitly.

When processing a delete command, it is not necessary to provide the XML document itself. Therefore another option is to use the overloaded `Delete` method, which takes a `TaminoUri` object as parameter.

Examples

```
...
TaminoDocument tamDoc = new TaminoDocument(doc);
command.Insert(tamDoc);
...
command.Delete(tamDoc);
...
```

```
...
command.Delete(new TaminoUri("./MyDocType/@1"));
...
```

```
...
TaminoDocument tamDoc = new TaminoDocument(doc);
tamDoc.DocName = "@1";
command.Update(tamDoc);
...
```

```
...
TaminoDocument tamDoc = new TaminoDocument(doc);
tamDoc.DocName = "@2";
tamDoc.DocType = "pre:MyDocType";
command.Update(tamDoc);
...
```

Retrieving XML Documents and Document Properties

The Tamino API for .NET provides methods for retrieving documents and document properties . The most common cases are described below:

- Retrieve a Document using its Identity
- Retrieve a Document as a Programming Language Type
- Retrieve a Document as a Stream or XmlReader
- Retrieve Document Properties using Identity
- Re-Read a Document from Tamino

Retrieve a Document using its Identity

The application knows the identity of the document and wants to retrieve the document.

Example

```
...
TaminoDocument tamDoc = command.Retrieve(new TaminoUri("Property/@1"));
...
```

Retrieve a Document as a Programming Language Type

.NET supports the generation of a programming language type from an XML schema. These classes can then be used to process an XML document using the programming language types instead of the DOM API. The Tamino API for .NET allows an XmlDocument to be retrieved as a programming language. All types which can be serialized and deserialized using .NET's `XmlSerializer` class are supported.

Example

```
...
Property property =
(Property) command.Retrieve(new TaminoUri("Property/@1"), typeof(Property));
...
```

Retrieve a Document as a Stream or XmlReader

For more convenient further processing, it is sometimes better to have a `Stream` or `XmlReader` object returned instead of a `TaminoDocument`.

Examples

```
...
Stream stream = command.RetrieveStream(new TaminoUri("Property/@1"));
...
```

```
...
XmlReader reader = command.RetrieveReader(new TaminoUri("Property/@1"));
...
```

Retrieve Document Properties using Identity

The class `TaminoDocumentProperties` represents properties, which can be retrieved from Tamino for a single document. One example of these properties is the time when a Tamino document was most recently modified. (For more information about the returned time format, see the Tamino XML Server documentation.)

Example

```
...
string lastmodifiedBefore = ...;
TaminoDocumentProperties properties = command.RetrieveProperties("Property/@1");
if ( properties.LastModified != lastmodifiedBefore )
 Console.WriteLine("Document has changed");
...
```

Re-Read a Document from Tamino

If the application has already read a `TaminoDocument` or `TaminoXmlDocument`, `TaminoCommand` provides facilities for re-reading the data from Tamino:

Example

```
...
TaminoTransaction tx = connection.BeginTransaction();
TaminoDocument tamDoc = new TaminoDocument(doc);
TaminoResponse response = command.Insert(tamDoc);
tx.Commit();
...
tx = connection.BeginTransaction();
// reload the data
command.Retrieve(tamDoc);
...
tx.Commit();
...
```

7 Handling Responses and TaminoExceptions

The Tamino API for .NET uses two concepts to inform the application about errors returned from the Tamino XML Server. These two concepts are described in the following sections:

Return Values and Tamino Messages

Data handling methods in the Tamino API for .NET, such as inserting, updating, deleting and querying Tamino documents, can return `TaminoResponse` and `TaminoQueryResponse` objects. If a Tamino error occurs, these methods do not throw a `TaminoException`. This is in order to meet the varying requirements of different kinds of applications. Some applications might prefer to check for errors using programming language concepts; others might prefer to use an XSL transformation on the returned Tamino XML document in order to present the Tamino error messages to their clients. Whenever a Tamino API for .NET method returns a `TaminoResponse` or `TaminoQueryResponse` object, the application should check the `ReturnValue` property of the class. Methods that return `Stream` or `XmlReader` objects also do not throw exceptions for returned Tamino error messages. When consuming the stream or reader, the application should check for error messages in the returned documents. For more information about the Tamino response format, see the Tamino XML Server documentation.

Example

```
...
TaminoResponse response = command.Update(tamDoc);
if (response.ReturnValue != "0") throw new ApplicationException(response.ErrorText);
...
```

TaminoExceptions

A method with a return type other than `TaminoResponse`, `TaminoQueryResponse`, `Stream` or `XmlReader` will throw a `TaminoException` if the Tamino Server returns an error. Most of these methods are designed to control flow handling rather than data handling. `TaminoException` is derived from the .NET class `System.ApplicationException`. This allows the application to access detailed information about the error in the customary .NET way. The `Message` property of the `TaminoException` class will hold the error number and error text returned from Tamino.

Example

```
...
try
{
   ...
   TaminoTransaction tx = connection.BeginTransaction();
   ...
}
catch ( TaminoException e )
{
   ...
   Console.WriteLine(e.Message);
   ...
}
...
```

A TaminoExceptions is also thrown when the Tamino API for .NET discovers an error situation. In this case the `TaminoException` will contain Tamino API for .NET-specific error numbers and error texts. Error numbers and texts are listed below:

API (module=PI) Related Messages

TANPIE0001: {0} operation failed: {1} – Reason: The specified operation failed for the specified reason.

TANPIE0005: Unknown database version {0} – Reason: Database version is unknown/not handled.

TANPIE0010: Connection is already in the Open state. – Reason: Another Open request has been attempted on the connection without having done a Close request.

TANPIE0100: Iterator is closed – Reason: Program attempted to use a closed iterator.

TANPIE0200: Invalid document name {0} – Reason: The document name is invalid for the operation.

TANPIE0201: Document name missing – Reason: The document name is missing.

TANPIE0202: Document type missing – Reason: The document type is missing.

TANPIE0203: Document names differ {0} != {1} – Reason: The document names differ between the document and URI.

TANPIE0204: Document types differ {0} != {1} – Reason: The document types differ between the document and URI.

TANPIE0300: Incorrect ID length. Expected {0}, got {1} – Reason: The length of the ID was incorrect.

DataSet (module=DS) Related Messages

TANDSE0001: No update information found for query result – Reason: No update information could be found for one or more items in the query result.

TANDSE0002: Cannot delete or update {0}. Document has been changed –

TANDSE0003: DataSet schema and Tamino query result do not match. – Reason: The DataSet tables are empty although the Tamino Query result is not.

TANDSE0004: AutoCommit connection not allowed. – Reason: This operation is not supported for connections with mode AutoCommit.

TANDSE0005: No support for updates for this kind of view. No table found for element {0}. – Reason: Updates are only supported for complete subtrees of the XML document. The DataSet schema must reflect this subtree.

TANDSE0006: Query is not of the right format to fill for updates. – Reason: Use TaminoXQueryBuilder.BuildXQuery() to build queries to fill for updates.

TANDSE0007: Same prefix for different namespaces is not supported. –

TANDSE0008: Invalid or missing TaminoQueryItemMapping for table {0}. – Reason: The specified columns and/or tables do not exist.

TANDSE0009: Schema could not be found in Tamino. –

TANDSE0010: At least one of the specified TaminoQueryItemMapping did not identify the element uniquely within the Tamino document. –

TANDSE0011: TaminoAdapterBehavior.ReadOnly not allowed for this operation. –

Transaction (module=TX) Related Messages

TANTXE0001: Transaction already in progress – Reason: Operation is not permitted when a transaction is already in progress.

TANTXE0002: Transaction not in progress – Reason: Operation is only permitted when a transaction is in progress.

TANTXE0010: Transaction not allowed – Reason: Transactions are not allowed in this connection mode.

Serialization (module=SN) Failures

TANSNE0001: Operation not allowed – Reason: You cannot change the content type for XML documents.

TANSNE0002: The query item is not a document – Reason: The item found in the result is only a subtree of the Tamino document.

Internal Assertion (module=AS) Failures

TANASE0001: Connection is not in Open/Connecting state: {0}. – Reason: Connection is not in the Open/Connecting state.

TANASE0002: Empty database URL – Reason: One of the connection/collection/URL strings is empty or null.

TANASE0003: Unhandled parameter type {0} – Reason: Source not yet implemented in HTTP request handler

TANASE0004: Command requires query langauge {0} – Reason: The command needs the specified query language in order to be able to work.

TANASE0005: Response document does not contain object properties. – Reason: The response document does not contain an ino:object element.

8 Performing Queries and Update Queries

The Tamino XQuery language can be used both to query Tamino XML documents and also to update them. For more information about the Tamino XQuery language, please refer to the Tamino Server documentation.

The Tamino API for .NET provides the `Query` method of the `TaminoCommand` to execute both kinds of transaction. The `Query` method always returns a `TaminoQueryResponse` object. Iterators can be requested from this `TaminoQueryResponse` object to process the returned query result set.

When querying data from the database, the returned query result is built according to the specified query. In the case of a query that updates XML data in Tamino, the returned query result set consists of a list of information that identifies the affected XML documents in Tamino. The application can use the serialization class `TaminoUpdateItem` to access this identifying information easily.

Examples

```
...
// querying data from Tamino
TaminoQuery query = new TaminoQuery ("input()/Property");
TaminoQueryResponse queryResponse = command.Query(query);
...
```

```
...
// updating data in Tamino
TaminoQuery updateQuery =
  new TaminoQuery("update replace for $p in input()/Property//Name where
                  $p='Hugo Maier' with <Name>Hugo Meier</Name>");
TaminoQueryResponse updateResponse = command.Query(updateQuery);
TaminoUpdateItem updateItem =
  (TaminoUpdateItem) updateResponse.GetSingleItem(typeof(TaminoUpdateItem));
Console.WriteLine("collection="+updateItem.collection+" doctype"+
                  updateItem.doctype+" id="+updateItem.id);
...
```

Using Tamino Cursors

If a query is expected to return a very large result set, it is generally better not to return the complete query result set all at once. Instead, the application can specify that the result set should be retrieved page by page. This is done by specifying a page size in the `Query` method of the `TaminoCommand`. If a page size greater than 0 is specified, the Tamino API for .NET will use the Tamino cursor.

A cursor can only be used for querying data from Tamino. It cannot be used in conjunction with an XQuery update expression.

A cursor can only be used within the scope of a local or global transaction (see dotnetapiref/SoftwareAG.Tamino.Api.TaminoConnectionMode.html).

Example

```
TaminoConnection connection = new TaminoConnection("http://myserver/tamino/mydb");
connection.Open(TaminoConnectionMode.LocalTransaction);
TaminoTransaction transaction = connection.BeginTransaction();
TaminoCommand command = connection.CreateCommand("mycollection");

// 10 items per page
TaminoQueryResponse qr = command.Query(new TaminoQuery("input()/Property"),10);
... ↩
```

Closing a Query Result Set

You should call the `Close` method in the `TaminoQueryResponse` object when the query result set is no longer required, in order to release resources such as Tamino cursors. Resources are not implicitly released when the `TaminoQueryResponse` object falls out of scope or is reclaimed by the garbage collector.

9 Processing Query Results

Each application processes data differently when it receives a response from Tamino as a result of a query. The Tamino API for .NET provides several methods and classes for accessing query results appropriately, satisfying various requirements for further processing.

Accessing Query Results as Stream and XmlReader

For further processing, it may be convenient to access the results of a query as a `Stream`; alternatively, it may be more convenient to access them as an `XmlReader`. These options are supported by the methods `QueryStream` and `QueryReader` of the `TaminoCommand` class, respectively.

The QueryXmlReader sample in the **QuerySamples** demonstrates using the `QueryReader` method to apply an XSL transformation to the query result.

Example

```
...
XmlReader reader = command.QueryReader(query);
XPathDocument xpathDoc = new XPathDocument(reader);

XslTransform xslTransform = new XslTransform();
xslTransform.Load("mystylesheet.xsl");

xslTransform.Transform(xpathDoc, null);
...
```

Accessing Single Value Query Results

A query may return just one simple value as its result. The `GetSingleItem` method in the `TaminoQueryResponse` enables you to access a simple result simply. See also the IntegerQuery example in the QuerySamples.

Example

```
TaminoQuery query = new TaminoQuery("count(input()/Property")
TaminoQueryResponse qr = command.Query(query);
int result = (int) qr.GetSingleItem(typeof(int));
```

You can also specify a complex type as the parameter to the `GetSingleItem` method. Any type that can be serialized and deserialized by .NET's `XmlSerializer` class is supported.

Iterating Through Query Results

The Tamino API for .NET provides two classes for iterating over query result sets: `TaminoPageIterator` and `TaminoItemIterator`. Your application can use the convenient `foreach` statement to iterate over the query results.

Both classes can be used to process query results, whether a Tamino cursor has been used for the query or not. Tamino cursor usage is completely hidden from the application. However, if a cursor was not used for the query, all the query result items are returned in one page and therefore there is only one page to iterate over.

The `TaminoItemIterator` can be used to iterate over the complete query result. Item iteration crosses page boundaries seamlessly.

Example

```
...
TaminoQueryResponse qr = command.Query(query, 10); // pagesize=10
foreach (XmlNode node in qr)
{
   Console.WriteLine(node.OuterXml);
}
qr.Close();
...
```

The `TaminoItemIterator` can also be used in conjunction with a `TaminoPageIterator`, so that the application can navigate pagewise through the query result, and then request a `TaminoItemIterator` to iterate over the results within a single page.

Example

```
...
TaminoQueryResponse qr = command.Query(query, 10); // pagesize=10
TaminoPageIterator pageIt = qr.GetPageIterator();
foreach (XmlDocument page in pageIt)
{
   TaminoItemIterator itemIt = pageIt.GetItemIterator();
   foreach (XmlNode node in itemIt)
   {
      Console.WriteLine(node.OuterXml);
   }
}
qr.Close();
...
```

An application can iterate forwards and backwards over the query results.

Example

```
...
TaminoQueryResponse qr = command.Query(query, 10);  // pagesize=10
TaminoPageIterator pageIt = qr.GetPageIterator();
while(pageIt.Next())
{
   XmlDocument page = pageIt.GetPage();
   ...
}

while (pageIt.Previous())
{
   XmlDocument page = pageIt.GetPage();
   ...
}
qr.Close();
...
```

Note: After modifying items in a page and writing the modifications back to the database, you will see the old values when you iterate over this page again.

See also the QueryPageIterator, QueryItemIterator and QueryNoTaminoCursor examples in the QuerySamples.

Processing Document Lists

If the query result set is a list of complete root elements of Tamino XML documents, the application can use the serialization class `TaminoXmlDocument` to access the individual result items within this list, make modifications, and write the modified items back to the Tamino database.

Example

```
TaminoQuery query =
    TaminoXmlDocument.BuildQuery(null, "input()/Property[@Category='Buy']");

TaminoQueryResponse qr = command.Query(query, 10); // pagesize=10
TaminoItemIterator itemIt = qr.GetItemIterator();

while(itemIt.Next())
{
   TaminoXmlDocument doc =
       (TaminoXmlDocument) itemIt.GetItem(typeof(TaminoXmlDocument))

   XmlElement elem = (XmlElement) doc.Data;
   // ...make some changes to elem
```

```
   TaminoResponse response = command.Update(doc);
   ...
}
```

You can find the complete sample code in the ProcessingDocumentList sample of the QuerySamples.

Similarly, you can use the class `TaminoXmlDocumentProperties` to get a list of document properties which you can subsequently use to access the documents themselves. For more information, see the Tamino API for .NET Reference Documentation.

That's all you need to know in order to use document lists successfully in your own applications. If you want to know more about this technique, the following paragraphs describes how it works internally.

Processing Document Lists: Internals

1. `TaminoXmlDocument.BuildQuery` modifies the specified query expression so that the query result will look like this:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:response ...>
   <xq:query>
   <![CDATA[ namespace
    tf='http://namespaces.softwareag.com/tamino/TaminoFunction' for $p in
    input()/Property[@Category='Buy'] return <TaminoDocument
    doctype={tf:getDoctype($p)}
    docid={tf:getInoId($p)}>{$p}</TaminoDocument>
    ]]>
   </xq:query>
   ...
   <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:tf="http://namespaces.softwareag.com/tamino/TaminoFunction">

      <tan:TaminoDocument tan:docid="1" tan:doctype="Property"
                          ↩
xmlns:tan="http://namespaces.softwareag.com/tamino/TaminoAPI4DotNet">
         <Property Category="Buy" PropertyType="2-Room-Apartment">
            ...
         </Property>
      </tan:TaminoDocument>
      <tan:TaminoDocument tan:docid="2" tan:doctype="Property"
                          ↩
xmlns:tan="http://namespaces.softwareag.com/tamino/TaminoAPI4DotNet">
         <Property Category="Buy" PropertyType="4-Room-Apartment" >
            ...
         </Property>
      </tan:TaminoDocument>
   </xq:result>
</ino:response>
```

2. `TaminoXmlDocument` can be serialized and deserialized using .NET's `XmlSerializer` class. After deserializing the first result item by calling `iterator.GetItem(typeof(TaminoXmlDocument))`, the public fields and property of the `TaminoXmlDocument` object contain the following values:

DocName	"@1"
DocType	"Property"
ContentType	"text/xml"
IsXml	True

Date	`<Property Category="Buy" PropertyType="2-Room-Apartment">`
	`  ...`
	`</Property>`

3. `TaminoXmlDocument` implements the `ITaminoDocument` interface. This allows it to be passed on to the `Update` method of the `TaminoCommand` class.

Processing Node Lists

An application can always update any node of a Tamino XML document by performing XQuery updates via the `Query` method of the `TaminoCommand` class in a descriptive way (see **Performing Queries and Update Queries**). However, if the query result set comprises a list containing uniform subtrees of a Tamino XML document, it may be better if the application uses a more object-oriented technique to access the individual result items, make modifications, and write the changes back to the Tamino database. The ProcessingNodeList example in the QuerySamples demonstrates using a simple class like `MyXmlNode` to process node lists more conveniently.

Example

```
...
string expr = "input()/Property/ContactPerson";

MyXmlNode.Selector = ".//ContactPerson";
MyXmlNode.Field = "Name";
TaminoQuery query = MyXmlNode.BuildQuery(expr);
TaminoQueryResponse queryResponse = command.Query(query, 10); // pagesize=10

TaminoItemIterator itemIt = queryResponse.GetItemIterator();

while (itemIt.Next())
{
   MyXmlNode item = (MyXmlNode) itemIt.GetItem(typeof(MyXmlNode));

   XmlNode contact = item.Data;
    // make some modifications
   TaminoQuery updateQuery = item.BuildUpdate();
```

```
   TaminoQueryResponse updateResponse = command.Query(updateQuery);
   ...
}
...
```

Processing Node Lists: Internals

To update an individual query result item, the item must be uniquely identified in the Tamino database. In this example, the `MyXmlNode` class uses a similar concept to the **<unique> constraints** as it is defined in XML Schema: the application specifies sub-tree uniqueness within the scope of the XML document that contains the sub-tree. This is done by setting MyXmlNode.Selector and MyXmlNode.Field accordingly. The `MyXmlNode` class uses the same mechanism as the `TaminoXmlDocument` class to identify the XML document.

The following steps are performed when modifying the phone number for a `ContactPerson` item:

1. `MyXmlNode.Selector = ".//ContactPerson"` and `MyXmlNode.Field="Name"` specify how a single `ContactPerson` element can be uniquely identified within a Property document: each `ContactPerson` element is identified uniquely by its "Name" child. Notice that the scope of uniqueness is just a single document. The "Name" value does not have to be unique over all Property documents.
2. `MyXmlNode.BuildQuery(expr)` modifies the specified query expression so that the query result will contain identifying information for the XML documents affected by this query.

   A single item in the query result set might look like this:

   ```
   <MyXmlNode doctype='Property' inoid='10'>
     <ContactPerson>
       <Name>Glenn Carter</Name>
       <Phone>1-361-842-3875</Phone>
       <Email>Glenn.Carter@automailer.com</Email>
     </ContactPerson>
   </MyXmlNode>
   ```

3. When `iterator.GetItem(typeof(MyXmlNode))` is executed for the item in step 2 above, the properties and fields of the `MyXmlNode` instance will contain the following values:

TaminoInoId	"10"
DocType	"Property"

Data	<ContactPerson> <Name>Glenn Carter</Name> <Phone>1-361-842-3875</Phone> <Email>Glenn.Carter@automailer.com</Email> </ContactPerson>
m_oldfield	"Glenn Carter"

4. Assume that the application changes the `Data` property so that it contains the new phone number:

```
<ContactPerson>
  <Name>Glenn Carter</Name>
  <Phone>9-999-999-9999</Phone>
  <Email> Glenn.Carter@automailer.com </Email>
</ContactPerson>
```

5. `item.BuildUpdate()` uses the MyXmlNode.Selector and MyXmlNode.Field settings and the m_oldfield and Data values to create the following update query:

```
namespace tf='http://namespaces.softwareag.com/tamino/TaminoFunction' update
replace for $p in input()/Property//ContactPerson where tf:getInoId($p)=10 and
$p/Name="Glenn Carter" return $p with <ContactPerson><Name>Glenn
Carter</Name><Phone>9-999-999-9999</Phone><Email>Glenn.Carter@automailer.com</Email></ContactPerson>
```

Binding Query Results to Web Controls

`TaminoQueryResponse`, `TaminoItemIterator` and `TaminoPageIterator` implement .NET's `IEnumerable` and `IEnumerator` interfaces. This allows an application to bind query results directly to .NET Web controls.

The following example queries all names of contact persons from Tamino and shows them in a list box. You can find the Property schema and corresponding example data in the samples.

Example

```
...
protected System.Web.UI.WebControls.ListBox ListBox1;
...
TaminoQuery query = new TaminoQuery("input()/Property/ContactPerson/Name"));
TaminoQueryResponse qr = command.Query(query);
ListBox1.DataSource = qr;
ListBox1.DataTextField = "InnerText";
ListBox1.DataBind();
...
```

10 Processing Transactions

For a general introduction to transaction-oriented programming with Tamino, see also the chapter *Transactions Guide*.

When opening a `TaminoConnection`, you need to specify how commands are to be handled by the Tamino XML Server by using the TaminoConnectionMode enumeration. The enumeration has the following values:

- AutoCommit;
- LocalTransaction;
- GlobalTransaction.

These values specify the scope of a command's transaction.

Additionally, the behavior within a transaction can be specified in more detail. For more information, refer to these sections:

- **Modifying Transaction Behavior**
- **Transaction Timeouts**

This chapter comprises the following sections:

Note: On multithreaded transaction usage: It is the user's responsibility to ensure that multiple threads do not attempt to execute transactional commands simultaneously. In other words, the operations `BeginTransaction`, `Commit` and `Rollback` must not be performed on the same `TaminoConnection` or `TaminoTransaction` by two or more threads simultaneously.

AutoCommit

Each command runs within the scope of a single transaction. The result of each command is automatically committed by the Tamino XML Server. This means that each command affects the database immediately after it has been successfully processed.

Example

```
...
// open connection in AutoCommit mode
connection.Open(TaminoConnectionMode.AutoCommit);

// delete some document
TaminoResponse response = command.Delete(someDocumentUri);
Trace.Assert(response.ReturnValue == "0", response.ErrorText);
// if we get here the delete worked - oops can't undo the delete now!

// close the connection
```

```
connection.Close();
...
```

LocalTransaction

The application marks the beginning and end of a transaction. All commands executed between the beginning and end of the transaction run within the scope of the transaction. To complete a transaction, the application can decide whether to commit or rollback all the changes made by all the commands. This allows the application to group a set of commands and either commit either all the changes or none of them. Using local transactions you can only group commands associated with the same Tamino database.

The following sample shows how to delete a document and then commit the delete. After being committed, the changes to the database cannot be rolled back!

Example

```
...
// open connection in LocalTransaction mode
connection.Open(TaminoConnectionMode.LocalTransaction);

// begin transaction
TaminoTransaction tx = connection.BeginTransaction();

// delete some document
TaminoResponse response = command.Delete(someDocumentUri);
Trace.Assert(response.ReturnValue == "0", response.ErrorText);
// if we get here the delete worked - could still undo the delete if need be

// commit the preceding commands [in this case the Delete]
tx.Commit();
// oops can't change our mind now!

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

The following sample shows how to delete a document and then rollback the delete. This undoes the delete of the document, leaving the database in the same state as it was in before the method `BeginTransaction` was called (assuming that this program is the only program accessing the database).

Example

```
...
// open connection in LocalTransaction mode
connection.Open(TaminoConnectionMode.LocalTransaction);

// begin transaction
TaminoTransaction tx = connection.BeginTransaction();

// delete some document
TaminoResponse response = command.Delete(someDocumentUri);
Trace.Assert(response.ReturnValue == "0", response.ErrorText);
// if we get here the delete worked - could still undo the delete if need be
// we do undo the Delete by doing a Rollback

// rollback the preceding commands [in this case the Delete]
tx.Rollback();
// phew - undid the Delete!

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

GlobalTransaction

You can use global/distributed transactions to group commands associated with different databases. All the commands run within the scope of a distributed transaction. For a general introduction to distributed transactions in .NET, see ms-help://MS.NETFrameworkSDK/cpguidenf/html/cpconprocessingtransactions.htm.

To participate in a distributed transaction, the Tamino XML Server needs to know the transaction ID. The transaction ID of the currently-active distributed transaction can either be determined automatically by the Tamino API for .NET, or it can be explicitly specified by the application. In either case, the transaction ID is communicated to the Tamino XML Server.

If the transaction ID is to be determined automatically, then the .NET Framework property `System.EnterpriseServices.ContextUtil.Transaction` must specify a distributed transaction that implements the Microsoft COM+ interface `ITipTransaction`. This would be satisfied by a .NET application running within the MS/DTC (Microsoft's Distributed Transaction Monitor, which can coordinate commits/rollbacks in different databases) transactional environment with the application specifying that it "Requires Transaction" using the System.EnterpriseServices.TransactionAttribute. Note that the `TaminoConnection.Open` method *must* execute within the context of the distributed transaction. The `TaminoConnection.Close` method may *not* be used within an automatically-determined distributed transaction.

A `TaminoConnection` used in GlobalTransactionMode must not be closed if the transaction was an automatically-created transaction (as opposed to a user-created global transaction). An attempt to close such a connection would result in the Tamino server throwing an exception.

The following sample shows the code needed to use an automatically-determined transaction:

Example

```
...
// method/application with the .NET attribute
// [TransactionAttribute(TransactionOption.Required)]

// open connection in GlobalTransaction mode
connection.Open(TaminoConnectionMode.GlobalTransaction);

// insert a document
TaminoResponse response = command.Insert(tamDoc);
Trace.Assert(response.ReturnValue == "0", response.ErrorText);

// do NOT close the connection
...
```

Instead of using transactional attributes, the distributed transaction can also be explicitly under the control of the application. If it wishes to perform Tamino operations in the context of this transaction, then it must supply the transaction ID to Tamino via the `TaminoConnection.Open` method invocation. The `TaminoConnection.Close` method may be used after the distributed transaction has been terminated.

Example

```
...
// user starts distributed transaction in some manner

// obtain transaction id as TIP URL
TaminoTxId txId = new TaminoTxId(tipUrl);

// open the connection in GlobalTransaction mode
connection.Open(txId);

// insert a document
TaminoResponse response = command.Insert(tamDoc);
Trace.Assert(response.ReturnValue == "0", response.ErrorText);

// user terminates distributed transaction in some manner

// safe to close connection
connection.Close();
...
```

Modifying Transaction Behavior

To ensure consistency, Tamino provides a sophisticated locking concept. For detailed information regarding the Tamino locking concept, please refer to the Tamino XML Server documentation. The application can specify the locking strategy to be applied for commands within transactions. The application can specify the locks at three different granularities within the API. The granularities, in order from most specific to least specific, are:

- a single command within a single transaction;
- all commands within a particular transaction;
- all commands within all transactions of a connection.

The most specific settings take precedence over the least specific settings.

Single Command

The locking strategy for the command can be specified by setting the individual `TaminoCommand` properties: IsolationLevel, LockMode and Lockwait. If these properties are set to anything other than their default values, then the requested lock value is applied to the next command. It also applies to subsequent commands until the lock value is reset to its default value by the application. Note that setting the `IsolationLevel` property per command is only allowed for connections in AutoCommit mode.

Example

```
...
connection.Open(TaminoConnectionMode.AutoCommit);
command.Lockwait = TaminoLockwait.No;
TaminoQueryResponse qr = command.Query(query);
command.Lockwait = TaminoLockwait.Default;
...
```

If no lock values are set in AutoCommit mode, then the default lock values are automatically applied by the Tamino XML Server itself.

All Transaction Commands

You can specify the locking strategy for all the commands within a specific transaction This is done by setting the lock values on the `TaminoConnection.BeginTransaction` invocation. This applies the same lock values to all the commands executed within the transaction.

Example

```
...
connection.Open(TaminoConnectionMode.LocalTransaction);
TaminoTransaction tx =
    connection.BeginTransaction(TaminoIsolationLevel.Default,
        TaminoLockMode.Protected, TaminoLockwait.Yes);
command.Query(query);
command.Delete(uri);
tx.Commit();
...
```

If no lock values are set in LocalTransaction mode, then the default lock values are automatically applied by the Tamino XML Server itself.

All Connection Commands

You can specify the locking strategy for all the commands for a connection. This is achieved by setting the lock values on the `TaminoConnection.Open` invocation. This causes all the commands executed on the connection to have the same lock values applied.

Example

```
...
connection.Open(TaminoConnectionMode.LocalTransaction,
    TaminoIsolationLevel.Default, TaminoLockMode.Unprotected,
        TaminoLockwait.Default);
TaminoTransaction tx = connection.BeginTransaction();
command.Query(query);
tx.Commit();
...
```

As noted above under the precedence rules, it is possible for a subsequent BeginTransaction invocation to provide a different set of lock values that would apply to that particular transaction.

Transaction Timeouts

You can set the timeouts that the Tamino Server applies to transactions on a connection-by-connection basis. You specify timeout intervals by passing in an instance of `TaminoPreference` when creating the instance of `TaminoConnection`.

`TaminoPreference` has two fields that allow you to specify the transaction timeouts:

TransactionTimeout
Specifies the maximum time interval in seconds between Commit/Rollback calls.

NonActivityTimeout
Specifies the maximum time interval in seconds between database operations.

By default, the Tamino Server uses database-specific values. You can use the Tamino Manager to view and modify these values.

Example

```
...
// create suitable preferences
TaminoPreference preference = new TaminoPreference();
preference.TransactionTimeout = 60;
preference.NonActivityTimeout = 30;

// create connection
TaminoConnection connection = new TaminoConnection(uri, preference);

// open connection in LocalTransaction mode
connection.Open(TaminoConnectionMode.LocalTransaction);

// begin transaction
TaminoTransaction tx = connection.BeginTransaction();

// do commands + Rollback/Commit operations with specified timeouts applied

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

11 Working with Datasets and the TaminoDataAdapter

The TaminoDataAdapter class enables you to use .NET's DataSet object together with Tamino XML data. The DataSet object is a table/column-oriented object which is suited for disconnected working and can easily be bound to Windows Forms and Web Forms controls. For more information about .NET datasets, see Microsoft's .NET Framework documentation.

However, a table/column-oriented object is not appropriate for complex XML data. For processing a list of complex XML data, and in order to get the full benefit of Tamino's native XML support, your application should use the TaminoCommand and TaminoItemIterator/TaminoPageIterator objects instead (see '**Using Tamino Cursors**' and '**Iterating through Query Results**').

For processing a list of simple XML data or a list of simple subtrees of complex XML documents, the TaminoDataAdapter provides a convenient option to the TaminoItemIterator/TaminoPageIterator. It allows the application to fill a dataset with data from Tamino, modify the dataset and write the changes back to Tamino just by calling the Update method of the TaminoDataAdapter object.

Since Tamino is a native XML database, the behavior of the TaminoDataAdapter differs slightly from the corresponding behavior of the SQL-oriented data adapters of the .NET Framework. The behavior of the individual functions is described below in detail in the following sections:

Creating and Using TaminoDataAdapter Objects

A TaminoDataAdapter object can be used to:

- define tables and columns in a dataset,
- populate a dataset with Tamino XML data,
- write a modified dataset back to Tamino.

It is not mandatory to define tables and columns in a dataset before populating it with Tamino data. If no tables and columns are defined, the schema will be inferred from the data when the dataset is populated. One disadvantage of this is that the process of inferring a schema from XML is not deterministic; therefore the result (i.e., the schema) relies heavily on the specific data from which the schema is inferred. For more information about inference limitations, see Microsoft's .NET documentation (ms-help://MS.NETFrameworkSDK/cpguidenf/html/cpconinferringdatasetrelationalstructurefromxml.htm).

The same TaminoDataAdapter object can be used to fill and update multiple datasets. A TaminoDataAdapter object is always tied to a specific TaminoConnection object and can only handle data belonging to one Tamino collection.

Updates must be done by the same TaminoDataAdapter object that populated the dataset. This is because the table/column-oriented DataSet object does not hold all information about native XML data, and therefore the TaminoDataAdapter object has to keep additional information for subsequent updates.

Creating `TaminoDataAdapter` Objects

Collection and connection properties must be specified when creating a `TaminoDataAdapter` object.

An application that wants to leave the handling of connections and transactions to the `TaminoDataAdapter` object should just specify the database URL and the collection name as strings. The `TaminoDataAdapter` object will then create and use its own `TaminoConnection`/`TaminoCommand` object internally and handle the opening and closing of connections as well as transactions without any intervention from the application.

An application that prefers to customize transactional settings during command execution, or control connections and transactions on its own, can pass `TaminoConnection` or `TaminoCommand` objects in the constructor. The `TaminoDataAdapter` will then use the passed objects for command execution.

Examples

```
...
TaminoDataAdapter adapter =
      new TaminoDataAdapter("http://myserver/tamino/mydb", "mycollection");
...
```

```
...
TaminoConnection connection = new TaminoConnection("http://myserver/tamino/mydb");
TaminoDataAdapter adapter =
      new TaminoDataAdapter(connection, "mycollection");
...
```

```
...
TaminoConnection connection = new TaminoConnection("http://myserver/tamino/mydb");
TaminoCommand command = connection.CreateCommand("mycollection");
TaminoDataAdapter adapter = new TaminoDataAdapter(command);
...
```

Disconnected Working

Datasets are designed for disconnected (offline) working. To support this, the `FillSchema`, `Fill` and `Update` methods of the `TaminoDataAdapter` class handle opening and closing of the associated connection objects according to the following rules:

- If the connection is closed, it is opened by the `TaminoDataAdapter` object and closed again before the method call returns.
- If the connection is open, it remains open when the method call returns.

Similar rules apply to the transaction handling. However, the rules for transaction handling also depend on whether `Fill` or `Update` is executed and on whether filling is done for read-only or update mode.

`TaminoDataAdapter` Behavior

The same methods of the `TaminoDataAdapter` object can be used to fill a dataset for update or readonly mode. However, filling for readonly is faster, uses fewer resources and poses fewer limitations on the data with which the dataset can be filled. You should therefore decide whether your application needs to modify the data or not before the dataset is filled and set the `Behavior` property of the `TaminoDataAdapter` object accordingly.

A `DataSet` object that has been filled for readonly can subsequently be filled for update. However, once a `DataSet` object has been filled for update, it cannot be filled for readonly. You should therefore fill a `DataSet` object either always for readonly or always for update.

Examples

```
...
DataSet ds = new DataSet();
adapter.Behavior = TaminoAdapterBehavior.ReadOnly;
adapter.Fill(ds);
...
```

`TaminoDataAdapter` and Synchronized XmlDataDocuments

.NET supports the synchronization of an XmlDataDocument object with a corresponding `DataSet` object. The same `DataSet` object may only be synchronized with exactly one XmlDataDocument object. The `TaminoDataAdapter` class uses synchronized XmlDataDocuments when filling datasets for later updates. An application can use the `TaminoDataAdapter` object's `GetTaminoResponse` method to access the specific XmlDataDocument that is synchronized with the dataset. An application must not synchronize another XmlDataDocument object with a `DataSet` object that has been filled or should be filled for update by a `TaminoDataAdapter` object.

Examples

```
...
DataSet ds = new DataSet();
adapter.Behavior = TaminoAdapterBehavior.UpdateByItemMapping;
adapter.Fill(ds);
XmlDataDocument xdoc = adapter.GetTaminoResponse(ds);
...
```

Releasing Resources

Especially when working with data in update mode, an application should call the `TaminoDataAdapter` object's `Dispose` method when it has finished using the dataset together with the `TaminoDataAdapter` object.

Examples

```
...
DataSet ds = new DataSet();
...
adapter.Dispose(ds); // release resources for this DataSet
...
```

```
...
adapter.Dispose();  // release resources for all DataSets handled
                    // by the adapter
...
```

Defining Dataset Tables and Columns

An application can use the `FillSchema` methods to define tables and columns in the dataset. Depending on the data with which the dataset should be populated, the application can use different parameters to specify the dataset schema. To populate the dataset with complete root elements of Tamino documents, the application can specify the name of the corresponding schema in Tamino. To populate the dataset with subtrees of XML documents, the application can specify an `XmlReader` object to read the corresponding schema.

Examples

```
...
XmlTextReader reader = new XmlTextReader("c:\\tests\\myschema.xsd");
adapter.FillSchema(ds, reader);
...
```

```
...
adapter.FillSchema(ds, "Property"); // schema name within Tamino
...
```

For a complete code sample, see the FillSchema example in the DataSetSamples.

XmlSchema Validation for Inserted XML Elements

There is an overloaded method of the `FillSchema` method which takes an additional parameter. This method is important if you plan to do updates and these updates imply the addition of new XML elements. If you modify the XmlDataDocument directly using the DOM API, insertion should work fine and you should not use this overloaded method.

If you work with the DataSet object directly, the DataSet object will not maintain the correct sequence of inserted child elements in relation to the parent. Please note that not only inserting new rows, but also modifying a row can imply the insertion of a new XML element. This is the case if the corresponding column has not previously contained a value. By default, the dataset will insert a new child at any position within the parent XML element. This position may not be the same as the position defined in the corresponding Tamino document type, and a subsequent call to the Update method would fail.

The application can use the overloaded FillSchema method to tell the TaminoDataAdapter object to use the specified schema for validation when new elements are inserted. The TaminoDataAdapter will then try to correct the element sequence according to the schema.

Example

```
...
DataSet ds = new DataSet();
adapter.Behavior = TaminoAdapterBehavior.UpdateByItemMapping;
adapter.FillSchema(ds, "Property", true);
...
```

Correction capabilities are limited in the current version of the Tamino API for .NET. The parent element in which the sequence of the child elements should be corrected must be a global XML element in the sense of the XML Schema recommendation (see **http://www.w3.org/TR/xmlschema-0/**). The TaminoDataAdapter cannot correct the element ordering for inserts resulting from calls to the DataSet.RejectChanges() method. This means: If you delete complete rows and call RejectChanges(), then the correct ordering is not guaranteed. You can avoid this problem by using TaminoDataAdapter.Fill instead of DataSet.RejectChanges() to undo changes in the dataset.

The status of the correction implementation is "experimental", meaning that the TaminoDataAdapter may not be able to correct the sequence if the schema is very complex.

Populating a Dataset for Read-Only

The Fill methods can be used to populate a dataset with Tamino data. To populate a dataset for read-only, the Behavior property of the TaminoDataAdapter must be set to "TaminoAdapterBehavior.ReadOnly" before calling the Fill methods.

The XQuery that is used to populate the dataset can either be set via the Query property of the TaminoDataAdapter class, or it can be passed as a parameter to the Fill method. The Query property contains the TaminoQuery object that was used for the most recently called Fill method.

Example

```
...
DataSet ds = new DataSet();
adapter.Behavior = TaminoAdapterBehavior.ReadOnly;
adapter.Query    = new TaminoQuery("input()/Property/Address");
adapter.Fill(ds);
...
adapter.Fill(ds); // (re)filling several times using the same query
...
```

```
...
DataSet ds = new DataSet();
adapter.Behavior = TaminoAdapterBehavior.ReadOnly;
adapter.Fill (ds, new TaminoQuery("input()/Property/Address"));
...
```

See also the FillReadOnly example in the DataSetSamples.

Loaded Data and Tables

Note: When the `Fill` method is called, all rows of all tables in the dataset are cleared.

If the dataset contains tables before the `Fill` method is called, then `XmlReadMode.IgnoreSchema` is used for filling. If the dataset does not contain any tables, then `XmlReadMode.InferSchema` is used for filling.

Connection and Transaction Handling

If the `TaminoConnection` object associated with the `TaminoDataAdapter` is not open, the connection is opened in "TaminoConnectionMode.AutoCommit" and it is closed before the `Fill` method returns.

Populating a Dataset for Update

The `Fill` methods can be used to populate a dataset with Tamino data. To populate a dataset for update, the `Behavior` property of the `TaminoDataAdapter` must be set to "TaminoAdapterBehavior.UpdateByItemMapping" before calling the `Fill` methods. If "TaminoAdapterBehavior.UpdateByItemMapping" has not been set before filling, the `TaminoDataAdapter` object will not create nor keep the necessary information for subsequently updating the data in Tamino. Calling the `Update` method later with the dataset will throw a `TaminoException`.

The `TaminoQuery` object that is used to populate a dataset is created using the `BuildXQuery` method in the `TaminoXQueryBuilder` class. This extends the passed XQuery expression to enrich the returned Tamino data with identifying information for subsequent updates.

Example

```
...
DataSet ds = new DataSet();
adapter.Behavior = TaminoAdapterBehavior.UpdateByItemMapping;
TaminoQuery query =
  TaminoXQueryBuilder.BuildXQuery(null, "input()/Property/Address",
                                  TaminoAdapterBehavior.UpdateByItemMapping);
adapter.Fill(ds, query);
...
```

See also the XmlUpdate and RowUpdate examples in the DataSetSamples.

Loaded Data and Tables

Note: When the `Fill` method is called, all rows of all tables in the dataset are cleared.

Similarly as when filling for readonly, `XmlReadMode.IgnoreSchema` is used for filling if the dataset already contains tables and `XmlReadMode.InferSchema` is used if it does not. An additional restriction applies to filling for update: the schema of the dataset must reflect the hierarchical structure of the complete XML subtree that is returned from Tamino. Let us assume that the query returns Sample subtrees like the following:

```
<Sample>
  <Description>filling for update sample</Description>
  <Person>
     <Phone>999999999</Phone>
     <Name>
        <First>FirstName</First>
        <Last>LastName</Last>
     </Name>
     <Email>myemail@mycompany.com</Email>
  </Person>
</Sample>
```

The `Fill` method will, for instance, throw a `TaminoException` if the dataset does not contain a corresponding table for the `Person` element and therefore does not maintain the following nested relations: Sample_Person, Person_Name. You usually can avoid such problems by applying an appropriate schema to the dataset.

There is one special case that can occur even with schemas: If the root element of your single query items does not contain any attributes and only contains children which themselves have children, like <AttributeLessRoot> below, .NET will not create a table for <AttributeLessRoot>; instead it will apply the name "AttributeLessRoot" to the whole dataset.

```
<AttributeLessRoot>
   <ChildWithChildren>
      <Child1>Child1</Child1>
      <Child2>Child2</Child2>
   </ChildWithChildren>
</AttributeLessRoot>
```

The Tamino API for .NET will throw a `TaminoException` when the dataset is filled for update. You can easily avoid this problem: note that the <AttributeLessRoot> itself does not contain any useful information that you might want to modify from within your clients. Instead of querying for <AttributeLessRoot> elements, query for <ChildWithChildren>.

Connection and Transaction Handling

Filling a dataset for update requires the connection to be opened in "TaminoConnectionMode.LocalTransaction". If the connection is already open in "TaminoConnectionMode.AutoCommit", an exception will be thrown. The query is always executed with lock mode "TaminoLockMode.Shared".

Modifying Data in a Dataset

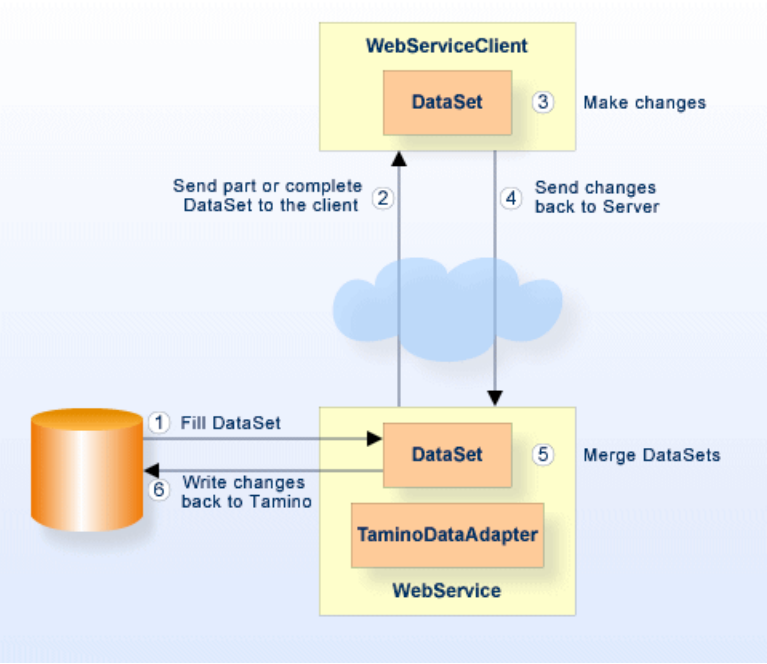
After a dataset has been filled for updates, the application will in general want to modify data. The following describes the most common ways of updating the data in the dataset.

Binding the Dataset to a Control

Datasets can easily be bound to Windows Forms or Web Forms controls, for instance the DataGrid control. The control displays the data which is bound to it, and the data can then be modified interactively through the control. See the RowUpdate example of the DataSetSamples for a running example.

Merging Datasets

Another useful feature of datasets is that they can be used together with Web services. A Web service scenario with the `TaminoDataAdapter` can be implemented as follows: A client application might request a filled dataset from a Web service, make some modifications to the dataset, and send a dataset that just includes the changes back to the Web service. The Web service might then merge the received changes into the original dataset and write the changes back to Tamino.



Modifying the Associated XML document

After a dataset has been filled for update by the `TaminoDataAdapter`, an XmlDataDocument is synchronized with the `DataSet` object. Synchronization means that changes made to either of the two objects are reflected in the other object. This allows an application to modify the data in the dataset by modifying the XmlDataDocument using the XML DOM interface. An application can access the synchronized XmlDataDocument through the `TaminoDataAdapter` class's `GetTaminoResponse` method. See also the XmlUpdate example of the DataSetSamples.

Handling Sequence Problems of Inserted XmlElements

If you do not make the modifications by modifying the associated XML document, you might run into problems with the sequence of inserted child elements. To overcome those problems, you can switch on the validation capability of the `TaminoDataAdapter` object. See "**XmlSchema Validation for inserted XML Elements**".

Example

```
...
XmlDataDocument xdoc = adapter.GetTaminoResponse(ds);
ds.EnforceConstraints = false;
...
// do modifications via DOM interface
...
ds.EnforceConstraints = true:
...
```

Writing Dataset Modifications Back to Tamino

The `TaminoDataAdapter` class's `Update` method allows modifications made to the dataset to be written back to the Tamino database. When the `Update` method is called, the `TaminoDataAdapter` looks for inserted, updated and deleted items in the dataset. If no modifications are found, the method returns immediately. If any modifications are found, it creates corresponding XQuery update expressions for Tamino and executes the updates. An application should not rely on individual items being updated in a defined sequence.

After successfully updating the data in Tamino, the `TaminoDataAdapter` calls the `AcceptChanges()` method in the `DataSet` object.

Note: The `Update()` method does not implicitly refill the dataset. If you have used the `Update()` method to update your modified items in the database and you want to modify the same items again, you must call the `Fill` method before making the modifications. Otherwise you will get an error message during the next update, stating that the items have been modified in the meanwhile.

Disconnected Working and Optimistic Concurrence

Datasets are designed to support disconnected (offline) working, in the sense that reading the data (for filling the dataset) and subsequent updating are not usually done within the same connection and/or transaction. One impact of this is that the data may be changed by another application between the Fill and the Update. To avoid overwriting changes that have already been made by other applications, the `TaminoDataAdapter` refuses to update a Tamino document if it has been changed in Tamino since the Fill. Notice that even if the dataset has been filled with subtrees of Tamino documents, the `TaminoDataAdapter` always checks for any modifications of the whole documents.

Updating Root Elements of Tamino Documents

If the dataset rows correspond to complete root elements of Tamino documents and their contents, an application can simply call the `Update` method taking just the `DataSet` object as parameter. The unit of update is always the complete XML tree starting with the root element.

Example

```
int count = adapter.Update(ds);
```

Updating XML Subtrees of Tamino Documents

The unit of update is always the complete subtree as returned from Tamino when the dataset was filled. Within those subtrees all modifications are allowed: inserting, deleting and updating XML subtrees. However, adding a complete new subtree to the query result is not supported because

the TaminoDataAdapter would not know into which Tamino document and at which point within the document the new subtree should be inserted.

In contrast to root elements, the application must specify a TaminoQueryItemMapping object for each different kind of subtree in the dataset. These TaminoQueryItemMapping objects must provide information specifying how to identify uniquely a single subtree within its parent Tamino XML document. TaminoQueryItemMappings are very similar to unique constraints in XML schemas (see also: **http://www.w3.org/TR/xmlschema-0/** - specifyingUniqueness). An XPath selector is specified to select a set of elements within its including XML document. “Fields” are then specified that must be unique within the scope of the set of selected elements. Notice that the scope of this uniqueness is only a single document and not a document set.

The following example shows how to specify TaminoQueryItemMappings.

Assume that the dataset has been filled with <Person> subtrees like the following:

```
<Person>
   <Phone>999999999</Phone>
   <Name>
      <First>FirstName</First>
      <Last>LastName</Last>
   </Name>
   <Email>myemail@mycompany.com</Email>
</Person>
```

According to .NET's rules for mapping XML to dataset tables and columns, the dataset will contain the following tables:

Person:

Phone	Email

Name:

First	Last

Further assume that the combination of first name and last name uniquely identifies a single Person subtree within a TaminoDocument. Remember that the scope of this uniqueness is only the document and not the document set.

The application would then specify the following TaminoQueryItemMapping for <Person> subtrees:

```
TaminoQueryItemMapping[] mapping = new TaminoQueryItemMapping[1];
TaminoFieldMapping[]     fields  = new TaminoFieldMapping[2];

mapping[0] =
    new TaminoQueryItemMapping
    (
        "Person",    // table name
        ".//Person", // XPath to select Person element sets
                     // within their including Tamino document
        fields       // fields defining uniqueness of Person
                     // elements within the element set.
    );

fields[0] = new TaminoFieldMapping("Name", "First");// table name, column name
fields[1] = new TaminoFieldMapping("Name", "Last"); // table name, column name

int count = adapter.Update(ds, mapping);
```

Connection and Transaction Handling

The `Update` method requires the connection to be opened in "TaminoConnectionMode.LocalTransaction". If the connection is already open in "TaminoConnectionMode.AutoCommit", an exception is thrown.

If a transaction is not already active when the `Update` method is called, the `TaminoDataAdapter` object starts a new transaction and either does a commit or a rollback before the `Update` method returns. For details of the error handling rules, see "**Customizing the Update Behavior**".

Customizing the Update Behavior

An application can customize the Update behavior of the `TaminoDataAdapter` object by setting the following properties:

ContinueUpdateOnError

In most cases, the `Update` method performs multiple update actions in Tamino, which are all executed within the scope of a single transaction. In some cases, the application might want to commit the transaction even though some of the actions have failed. For example, it might be impossible to update one of the documents because another application has changed it in the meanwhile; nevertheless, all other changes should be written back to the Tamino. The behavior is as follows:

Value of ContinueUpdateOnError	Description
False	This is the default setting. At the first thrown exception, the processing of update actions is stopped and the `Update` method throws an exception. If a transaction had already been started before the `Update` method was called, no Rollback is done. If the transaction was started by the `TaminoDataAdapter`, a Rollback is done before the exception is thrown.
True	The `TaminoDataAdapter` continues to process update actions. Instead of throwing an exception, it puts the text of a caught exception into the RowError information of the row, the item which caused the error is mapped to. This allows the application to handle these errors after the `Update` method returns. There is one exception to this rule: If SingleItemUpdate (see below) is set to false, the very last step of the update processing is the execution of an update command which contains all collected update actions. If an error occurs in this very last step, an exception is always thrown and the transaction is rolled back.

SingleItemUpdates

The behavior is as follows:

Value of SingleItemUpdates	Description
True	This is the default setting. For each item to be updated, the `TaminoDataAdapter` issues a single update command to Tamino. This setting is useful if the dataset contains only few and/or very large updated items, and/or the application would like to have detailed control over individual command execution.
False	The `TaminoDataAdapter` minimizes the number of round-trips to the Tamino Server. As far as possible, it combines individual update actions into one big update command. This setting is useful if the dataset contains many small updated items.

Working with TaminoDataAdapter Events

The `TaminoDataAdapter` exposes two events to which an application can respond in order to gain detailed control over the individual update actions that occur when processing the `Update` method. The TaminoUpdating and TaminoUpdated events are similar to .NET's RowUpdating and RowUpdated events, see ms-help://MS.NETFrameworkSDK/cpguidenf/html/cpconaddingremovingadonetproviderevents.htm.

Event	Description
TaminoUpdating	This event signifies that an update operation for a changed item in the dataset is about to begin. The application can respond to this event to do validation checks or modify the item before it is updated, to customize the command that will be used for the update, to skip the updating of single items, to cancel the update, and so on.
TaminoUpdated	This event signifies that an update operation for a changed item in the dataset has been completed. The application can respond to this event by handling errors that occurred during the update, by deciding whether update processing should continue with the next item or whether it should be cancelled, and so on.

For a working example of using `TaminoDataAdapter` events, see the TaminoUpdatingEvent example in the DataSetSamples.

`TaminoUpdatingEventArgs` and `TaminoUpdatedEventArgs`

When a TaminoUpdating event is raised, a `TaminoUpdatingEventArgs` object is passed to the event handlers. When a TaminoUpdated event is raised, a `TaminoUpdatedEventArgs` object is passed accordingly.

The `TaminoUpdatingEventArgs` object provides the following properties:

Property	Description
`StatementType`	The type of the update operation: "INSERT", "UPDATE", "DELETE".
`Command`	The `TaminoCommand` object to perform the update operation. The application can, for instance, change transactional properties.
`QueryExpression`	The XQuery update expression for the update operation. For an "INSERT" operation this is always null. The application can modify this expression.
`Namespaces`	A collection of namespace declarations for the XQuery update expression. For an "INSERT" operation this is always null. The application can modify the namespace declaration.
`Node`	The item as an XML element. The application can use this property to modify the item.
`Row`	The row to which the XML element is mapped within the dataset. The application can use this property to modify the item.
`Status`	Indicates whether an error has occurred. When the event is raised, the value is either "Continue" or "ErrorsOccurred". The application can set the Status to a different value.
`Errors`	The caught exception if status is "ErrorsOccurred". The application can set its own exception.

The `TaminoUpdatedEventArgs` object provides the following properties:

Property	Description
StatementType	The type of the update operation: "INSERT", "UPDATE", "DELETE". If SingleUpdateItem is set to false, this property is always set to "UPDATE".
Command	The TaminoCommand object that was used to perform the update operation. An application can, for instance, reset transactional properties that it had changed during the TaminoUpdating event.
TaminoResponse	The Tamino response object returned from the update operation.
Node	The item as an XML element. If SingleUpdateItem is set to false, this property is always null.
Row	The row to which the XML element is mapped within the dataset. If SingleUpdateItem is set to false, this property is always null.
Status	Indicates whether an error has occurred. When the event is raised, the value is either "Continue" or "ErrorsOccurred". The application can set the Status to a different value.
Errors	The caught exception if status is "ErrorsOccurred". The application can set its own exception.

Event Raising and Execution Order

The time when the events are raised depends on whether SingleItemUpdates is set to true or false.

If SingleItemUpdate is "true", for each modified item the behavior is as follows:

1. Create update expression.
2. Check whether the Tamino document that contains this item has been modified since the dataset was filled.
3. Raise TaminoUpdating event.
4. Check Status property returned from the event handler and continue or abort update processing.
5. Issue update command to Tamino.
6. Raise TaminoUpdated event.
7. Check Status property returned from the event handler and continue with the next item or abort update processing.

If SingleItemUpdate is "false", the behavior is as follows:

1. For each modified item:
   a. Create update expression.
   b. Raise TaminoUpdating event.
2. For each item that requires an INSERT operation:
   a. Issue command to Tamino.
   b. Raise TaminoUpdated event.

c. Check `Status` property returned from the event handler and continue or abort update processing.

3. For all affected Tamino documents: Check whether they have been changed since the dataset was filled.
4. Create an update command that combines all update actions for the remaining items.
5. Issue the update command to Tamino.
6. Raise TaminoUpdated event.

Limitations of the Dataset Support

Since Tamino is a native XML database and the dataset is a table/column-oriented object, the support provided for datasets by the `TaminoDataAdapter` is limited:

- The `TaminoDataAdapter` uses the .NET Framework to map XML to the dataset. This mapping has some restrictions: see the .NET documentation for limitations.
- When using the `TaminoDataAdapter`, a single `DataSet` object can only contain the result of one query at one point in time. The `Fill` method always clears all tables before filling is done. This means that a dataset cannot be filled from several different Data Sources.
- As described in the **section "Working with Datasets and the `TaminoDataAdapter`", section "Populating a Dataset for Update", subsection "Loaded Data and Tables"**, some restrictions about the dataset schema apply to filling for updates.
- If updating should be a process, the `TaminoDataAdapter` is a stateful object in the sense that filling and updating must be done by the same `TaminoDataAdapter` object.
- As described in the **section "Working with Datasets and the `TaminoDataAdapter`", section "Creating and Using `TaminoDataAdapter` Objects", subsection "`TaminoDataAdapter` and Synchronized XmlDataDocuments"**, the `TaminoDataAdapter` uses a synchronized XmlDataDocument when filling a dataset for update. This synchronization has several consequences; for example, the schema of the dataset cannot be modified after the dataset has been filled. Another consequence is that once you have filled a single `DataSet` object for update, you cannot fill it again for readonly.
- As described in "**XmlSchema Validation for inserted XML Elements**", the sequence of child elements is not always maintained when updating and inserting new rows by modifying the `DataSet` object directly or using the `DataSet.RejectChanges` method.
- When updating complete documents with default namespaces, the `TaminoDataAdapter` adds a namespace prefix to each element.
- Updating subtrees that use default namespaces is not currently supported.

12 Security

In order to pass the user's name, domain and password to the Tamino XML Server for security purposes, use the class `TaminoUserInfo`. When passing passwords, secure transmission (for example, HTTPS) *must* be used. The username and password are sent as Base64 encoded text, which is very easy to decode (Base64 is not an encryption technique) and is therefore insecure unless an encrypted transmission protocol is used.

Example

```
...
// create user security info
TaminoUserInfo userInfo = new TaminoUserInfo("name", "domain", "password");

// create connection
TaminoConnection =
    new TaminoConnection("https://localhost/tamino/mydb", userInfo);

// open connection
connection.Open(TaminoConnectionMode.AutoCommit);

// create command
TaminoCommand command = connection.CreateCommand();

// insert document
command.Insert(tamDoc);
...
```

13 Encoding

The property `TaminoPreference.RequiredEncoding` specifies to the Tamino XML Server the encoding to be used when returning a retrieved document or a query response. This is useful for programmers who wish to deal with documents in some encoding (for example, EUC-JP) other than the default encoding (UTF-8). Note that the encoding is only apparent at the byte stream level, i.e. if a document is stored to a file or is read as a stream. It is not apparent in a DOM tree, as the document appears as Unicode. The encoding would also appear in the XML document prolog (for example, encoding="EUC-JP").

Example

```
...
// create preferences for EUC-JP
TaminoPreference pref = new TaminoPreference();
pref.RequiredEncoding = "EUC-JP";

// create connection with preferences
TaminoConnection connection =
    new TaminoConnection("http://localhost/tamino/mydb", pref);
connection.Open(TaminoConnectionMode.AutoCommit);

// create command
TaminoCommand command = connection.CreateCommand();

// retrieve document in EUC-JP encoding
Stream stream = command.RetrieveStream(someUri);

// write stream to console
Console.Write("EUC-JP: ");
for (int b=stream.ReadByte() ; b != -1 ; b=stream.ReadByte())
{
    Console.Write(b.ToString("X2")+",");
}
Console.WriteLine();
...
```

14 Processing Tamino Schema Commands

The class `TaminoSchemaCommand` provides methods for defining and undefining Tamino doctypes via XML schema documents. It also includes methods for requesting information about existing doctypes in a Tamino database. For an example that shows how to use the `Define` and `Undefine` methods, see the Define and Undefine examples in the SimpleSamples.

The following example shows how to read an XML schema with a specified name in a specified collection from Tamino and access it via the .NET class `System.Xml.Schema.XmlSchema`.

Example

```
...
TaminoSchemaCommand schemaCommand = connection.CreateSchemaCommand();
XmlNodeList nodeList = schemaCommand.GetSchema("mycollection", "myschemaname")
// in this example the maximal count of nodes in nodeList is 1.

foreach (XmlNode node in nodeList )
{
 XmlNodeReader reader = new XmlNodeReader(node);
 XmlSchema     schema = XmlSchema.Read(reader, null);
    ...
}
...
```

15 Working with Non-XML Data

The following methods of the `TaminoCommand` class can also be used to process non-XML data:

- `Insert`
- `Delete`
- `Retrieve`
- `RetrieveProperties`
- `RetrieveStream`
- `Update`

Example

```
...
...
TaminoConnection connection = new TaminoConnection ("http://myserver/tamino/mydb");
connection.Open(TaminoConnectionMode.AutoCommit);

// Tamino provides collection ino:etc to insert non-XML data
TaminoCommand command = connection.CreateCommand("ino:etc");

FileStream stream = new FileStream("MyBinary.gif", FileMode.Open);
TaminoDocument doc = new TaminoDocument(stream, "image/gif");
doc.DocName = "MyBinary.gif";
doc.DocType = "ino:nonXML"; // ino:nonXML is the doctype for non-XML data
                            // in collection ino:etc
TaminoResponse response = command.Insert(doc);
Trace.Assert(response.ReturnValue == "0", response.ErrorText);
...
```

For more information about the ino:etc collection and the ino:nonXML document type, please see the Tamino Server documentation.

16 Working with URIs

The methods of a single `TaminoCommand` object are by default executed relative to the Tamino collection for which this `TaminoCommand` object was created. However, some applications have relative or absolute URIs at hand for the documents they want to manipulate. The `TaminoCommand` class has methods for `Insert`, `Retrieve`, `Update` and `Delete`, each of which takes a `TaminoUri` parameter to specify the URI explicitly. This also allows a Tamino document to be manipulated in a different collection than the collection for which this command object was originally created.

Example

```
...
TaminoConnection connection = new TaminoConnection ("http://myserver/tamino/mydb");
connection.Open(TaminoConnectionMode.AutoCommit);
TaminoCommand command = connection.CreateCommand("mycollection");

// relativeUri is an URI relative to the Tamino collection "mycollection"
// in the database "http://myserver/tamino/mydb"
TaminoUri relativeUri =  new TaminoUri("./MyDocType/MyDocumentName");
TaminoDocument tamDoc = command.Retrieve(uri);

// Note that after the Retrieve call:
// tamDoc.DocType has the value "MyDocType" and
// tamDoc.DocName has the value "MyDocumentName"

// URI for different collection
TaminoUri absoluteUri =
        new TaminoUri("http://myserver/tamino/mydb/mycollection2");

// this will insert the document with the name "MyDocumentName" into
// the collection "mycollection2"
TaminoResponse response = command.Insert(absoluteUri, tamDoc);
….
```

Similarly, an application could even execute commands in different Tamino databases. However, this has impacts on the transactional behavior: the connection object from which a `TaminoCommand`

object was created usually defines the transactional context for the command execution. This is also true when working with `TaminoUri` objects as long as the `TaminoUri` string matches the connection string of the `TaminoConnection`. If the strings do not match, the command is executed in TaminoConnectionMode.AutoCommit using Tamino default transactional settings.

17 Processing Diagnostics

Diagnostic tests can be executed for a connection that is in either of the states `TaminoConnectionState.Open` or `TaminoConnectionState.Closed`. The diagnostic subject is passed to the `Diagnose` method of the `TaminoConnection` object as a string. Please refer to the Tamino Server documentation for a list of the diagnostic subjects that are supported by Tamino.

You can use the class `TaminoDiagnoseResponse` to access the response from a particular diagnose request. See also the TaminoSerialization sample in the AdvancedSamples.

```
...
TaminoResponse response = connection.Diagnose("ping");
TaminoDiagnoseResponse diagnoseResponse =
  TaminoDiagnoseResponse.CreateDiagnoseResponse(response);
Console.WriteLine("Subject="+diagnoseResponse.Message.Messageline[0].Subject);
Console.WriteLine("Value=" + diagnoseResponse.Message.Messageline[0].Value);
...
```

Hint: Use the `DbVersion` property of the `TaminoConnection` object to access the version number of the Tamino database.

IV Samples

18 Samples

The samples subdirectory, *TaminoAPIDir*/*samples*, contains a number of samples demonstrating the use of the Tamino API. The samples are provided as C# source code and they are grouped by subject.

The subjects and samples are organized analogously to the Visual Studio .NET solutions and projects also included in this directory.

Example: In the directory *TaminoAPIDir*/*samples*/*DataSetSamples*/*MergeDataSets*, you can find a sample for merging datasets. The file *TaminoAPIDir*/*samples*/*DataSetSamples*/*MergeDataSets*/*MergeDataSets.cs* contains the example implementation using the Tamino API for .NET. If you have Microsoft Visual Studio .NET installed, you can simply open the file *TaminoAPIDir*/*samples*/*DataSetSamples*/*DataSetSamples.sln* with Visual Studio .NET and start working with the dataset samples.

For more information about the Tamino API for .NET code samples, please refer to the README file, which you can find at the location *TaminoAPIDir*/*samples*/*readme.txt*. In this file you can find information about building the samples, loading test data, running the samples, and cleaning up.

Samples: Documentation

To view the *documentation* of the Tamino API for .NET code samples, please select this link: ../navig/samples_navig.htm

Alternatively, if you prefer to use Microsoft's HTML Help system, you can use the file *TaminoInstallDir*/*Documentation*/*en*/*dotnetapi*/*samples*/*Documentation.chm*.

Note: Depending on your PC's security settings, you may have to copy this file to a local hard disk in order to be able to use it.

V Measuring Operation Duration

19 Measuring Operation Duration

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

Tamino API for .NET can help you to accomplish this task: it provides some basic mechanisms for gathering detailed information about execution times. This enables the application developer and the database designer to get more detailed information than would be available without this functionality. Tamino API for .NET does not include statistical tools to summarize or process the measurement results. Since standard Windows concepts are used to provide the measured values, any tools available on the Windows platform can be used instead.

The next section gives an overview of the operations that can be measured and the different values that can be measured. The following section describes the architecture and technical concepts used for measuring and monitoring. The final section in this chapter guides you through samples that demonstrate how to use the "Measuring Operation Duration" feature.

Operations and Measured Times

Measuring support is provided for most operations that submit requests to Tamino. The measurements are categorized into:

- Operations of the API core classes;
- Operations of the `TaminoDataAdapter` class.

Relationship between the Core API and `TaminoDataAdapter` Operations

Support for the `TaminoDataAdapter` class is layered on top of support for the core API classes such as `TaminoCommand`. A `TaminoDataAdapter` operation usually performs one or more `TaminoCommand` operations, and then some additional DataSet-related processing. Tamino API for .NET allows you to enable the measurement of core API operations and `DataSet` related operations separately. For the execution of `TaminoDataAdapter` operations, this means that *if* measuring the core API operations is enabled *and* measuring the DataSet-related operations is disabled, *then* the core API operations performed by the `TaminoDataAdapter` are measured nevertheless.

Measured Times for Core API Operations

The following times can be measured for all core API data operations:

TotalOperationDuration
The time taken from the start of the operation until the end of the operation. This is the total processing time within the API, including XmlParseDuration and TotalCommunicationDuration.

XmlParseDuration
The time taken for the parsing of 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 TaminoServerDuration.

TaminoServerDuration
The time taken for the request to be processed by the Tamino Server.

Measured Times for `TaminoDataAdapter` Operations

The following times can be measured for `TaminoDataAdapter` operations:

TotalOperationDuration
The time taken from the start of the operation until the end of the operation. This includes TotalCoreOperationDuration.

TotalCoreOperationDuration
The total time taken for processing *all* core operations of the Tamino API for .NET within the adapter operation.

The following time is additionally provided for Fill operations:

HierarchyValidationDuration
To update data from a `DataSet`, the data and the `DataSet` have to meet some requirements. See also **Populating a Dataset for Update > Loaded Data and Tables**. The `TaminoDataAdapter` performs some checks when a `DataSet` is filled for updates. HierarchyValidationDuration is the time spent performing these checks when a Fill operation is executed.

The following times are additionally provided for Update operations:

CollectModificationDuration
The first step when performing an Update operation is to collect the modifications done in the `DataSet`. The inserted, deleted and modified rows in the `DataSet` and their corresponding XML subtrees must be located. CollectModificationDuration is the time spent collecting the modified XML subtrees.

InsertCorrectionDuration
DataSets have limitations regarding the ordering when inserting elements. The `TaminoDataAdapter` includes some experimental code to correct the order in some common cases (see also **XmlSchema Validation for Inserted XML Elements**). InsertCorrectionDuration is the time spent trying to correct the ordering during an Update operation.

Accuracy of Measured Times

All times are measured in milliseconds.

Depending on the performed operation, not all of the measured values will necessarily be non-zero. For example, when deleting a document, there is no need to parse a response document. XmlParseDuration is zero for operations, such as TaminoCommand.QueryStream(), in which the returned response document is not parsed by the Tamino API for .NET. 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 on different computers.

When a .NET application is running, the .NET garbage collector will probably be started from time to time. This implies that even if you run exactly the same operation with exactly the same data twice, the measured times can differ. Garbage may be collected whilst the operation is in progress, and this can increase the measured times. Furthermore, the division between the different measured durations is not absolute. It is important to understand that the measurement feature of the Tamino API for .NET cannot answer the question: “Exactly how long does operation x with data y take?”; rather, it helps you to answers questions like: “Where does the application spend most of its time?” or: "Does the program run faster after making the change?"

Architecture and Technical Concepts

The Tamino API for .NET takes advantage of the performance monitoring concepts that are supported in the Microsoft platform. In order to satisfy different use cases, two alternatives are supported:

- Windows Performance Counters;
- Windows Management Events.

Both concepts have in common that the application being measured and the component that is monitoring or logging the measured values are loosely coupled. This means that the application and the monitor or logger are independent executables accessing shared Performance Counter/Management Event objects.

- Performance counters are preferable if you want to monitor durations over time using the Microsoft Performance Monitor. This is typically the case if you have a long-running application and are interested in its behavior over time (rather than in every executed operation or details about the executed operation).
- If you want information about every single operation, or if you are interested in operation details such as the URL accessed by the operation, it would be better to use Tamino API for .NET’s ability to fire management events.

Whichever concept you plan to use, the performance counters and/or management events cannot be seen on your system until you have executed a simple installation step, which is described below. The following gives a brief overview of how to measure and monitor an application using the different concepts.

Installation and Uninstallation

To install the necessary resources for the performance counters and management events provided by the Tamino API for .NET, run the Installer tool of the .NET Framework:

1. Open a command prompt in the directory *TaminoAPI4DotNET**lib*, where *TaminoAPI4DotNet* is the installation directory of the Tamino API for .NET.
2. Invoke the command `installutil TaminoAPI.dll`

To uninstall the Tamino API for .NET related resources, do the following:

1. Open a command prompt in the directory *TaminoAPI4DotNET**lib*, where *TaminoAPI4DotNet* is the installation directory of the Tamino API for .NET.
2. Invoke the command `installutil /u TaminoAPI.dll`

For more information about the Installer tool, see the Microsoft .NET Framework documentation.

Unfortunately, versions 1.0 and 1.1 of the Microsoft .NET Framework do not uninstall the Windows Management Instrumentation (WMI) schema that was added for the management events `TaminoCommandEvent` and `TaminoDataAdapterEvent` during installation (see **Using Management Events**). You can remove the schema by following these steps:

1. Open a command prompt
2. Invoke the Windows Management Instrumentation Tester by entering the command `wbemtest.exe`
3. Select **Connect...**
4. In the field `Server\Namespace` enter "root\TaminoAPI", then select **Login**
5. Select **Delete Class...**. In the pop-up window, enter the target class name `TaminoCommandEvent`. Select **OK**.
6. Select **Delete Class...**. In the pop-up window, enter the target class name `TaminoSchemaCommandEvent`. Select **OK**.
7. Select **Delete Class...**. In the pop-up window, enter the target class name `TaminoConnectionEvent`. Select **OK**.
8. Select **Delete Class...**. In the pop-up window, enter the target class name `TaminoCoreApiEvent`. Select **OK**.
9. Select **Delete Class...**. In the pop-up window, enter the target class name `TaminoDataAdapterEvent`. Select **OK**.

Activating Measurement

To activate measurement, add an appropriate key/value pair to the <appSettings> section in the .NET application configuration file of the application that is to be measured. The value must be either ""true"" or ""false"". If a key is not defined, its default value is ""false"".

Application Configuration Files

The application configuration file must be located in the application directory. Its name must be the same as the name of the application, with the extension ".config".

Supported Keys

The following keys control measurement of the Core API operations:

TaminoCommandPerformanceCounters
Switches on measuring for the `TaminoCommand` operations. The measured values are provided as Windows performance counters.

TaminoCommandManagementEvents
Switches on measuring for the `TaminoCommand` operations. The measured values are provided as Windows management events.

TaminoSchemaCommandPerformanceCounters
Switches on measuring for the `TaminoSchemaCommand` operations. The measured values are provided as Windows performance counters.

TaminoSchemaCommandManagementEvents
Switches on measuring for the `TaminoSchemaCommand` operations. The measured values are provided as Windows management events.

TaminoConnectionPerformanceCounters
Switches on measuring for the `TaminoConnection`/`TaminoTransaction` operations. The measured values are provided as Windows performance counters

TaminoConnectionManagementEvents
Switches on measuring for the `TaminoConnection`/`TaminoTransaction` operations. The measured values are provided as Windows management events.

The following keys control measurement of the `TaminoDataAdapter` operations:

TaminoDataAdapterPerformanceCounters
Switches on measuring for the `TaminoDataAdapter` operations. The measured values are provided as Windows performance counters.

TaminoDataAdapterManagementEvents
Switches on measuring for the `TaminoDataAdapter` operations. The measured values are provided as Windows management events.

Examples

The following example switches on measuring for the `TaminoCommand` operations and for the `TaminoDataAdapter` operations. All measured values are provided as Windows performance counters and also as Windows management events.

```
<configuration>
  <appSettings>
    <add key=" TaminoCommandPerformanceCounters " value="true" />
    <add key=" TaminoCommandManagementEvents " value="true" />
    <add key=" TaminoDataAdapterPerformanceCounters " value="true" />
    <add key=" TaminoDataAdapterManagementEvents " value="true" />
  </appSettings>
</configuration>
```

The following example switches on measuring for the `TaminoCommand` operations. The measured values are provided as Windows performance counters.

```
<configuration>
  <appSettings>
    <add key=" TaminoCommandPerformanceCounters " value="true" />
  </appSettings>
</configuration>
```

Optimization and Measurement

TaminoAPI4DotNET optimizes the reading of Tamino response documents. After executing a command, the response document is not read if the returned HTTP header contains all the required information; this saves a significant amount of overhead. For example, TaminoCommand.Query() always reads the response document, because information is required that is not contained in the HTTP header; by contrast, TaminoTransaction.Rollback() normally does not read the response document.

The variables XmlParseDuration and TaminoServerDuration are always set to zero if the response document is not read. You can set the following switch, which forces TaminoAPI4DotNET to read the response document always:

```
<configuration>
  <appSettings>
    <add key="TaminoReadResponseDocument" value="true" />
  </apSettings>
</configuration>
```

Setting this switch yields more measurement values. However, it disables the default optimization in TaminoAPI4DotNET, so operations may take slightly longer.

Using Performance Counters

Windows performance counters allow applications, services and devices to publish performance data that can be monitored using the Microsoft Performance Monitor tool. Although performance counters are not new with .NET, their use has been simplified by the .NET Framework.

Performance Counters and Categories

Windows itself comes with numerous predefined performance counters; other products such as .NET add their own performance counters that allow measurement of the .NET Framework and managed code that runs in the .NET Framework. For example, there are .NET specific performance counters that provide information about the .NET garbage collector.

Just as classes are grouped together in namespaces, so are performance counters grouped together in categories. The installation step described above sets up the following categories:

- Tamino API for .NET `TaminoCommand`;
- Tamino API for .NET `TaminoSchemaCommand`;
- Tamino API for .NET `TaminoConnection`;
- Tamino API for .NET `TaminoDataAdapter`.

The Tamino API for .NET `TaminoCommand` Category

The Tamino API for .NET `TaminoCommand` category comprises the following performance counters:

Delete:TaminoServerDuration	Delete:TotalCommunicationDuration	Delete:TotalOperationDuration	De
Insert:TaminoServerDuration	Insert:TotalCommunicationDuration	Insert:TotalOperationDuration	In
NextPage:TaminoServerDuration	NextPage:TotalCommunicationDuration	NextPage:TotalOperationDuration	Ne
PreviousPage:TaminoServerDuration	PreviousPage:TotalCommunicationDuration	PreviousPage:TotalOperationDuration	Pr
Query:TaminoServerDuration	Query:TotalCommunicationDuration	Query:TotalOperationDuration	Qu
Retrieve:TaminoServerDuration	Retrieve:TotalCommunicationDuration	Retrieve:TotalOperationDuration	Re
Update:TaminoServerDuration	Update:TotalCommunicationDuration	Update:TotalOperationDuration	Up

The NextPage:* and PreviousPage:* performance counters measure the durations when iterating through multi-page result sets (`TaminoPageIterator`, `TaminoItemIterator`). All other performance counters apply to the corresponding methods in the `TaminoCommand` class.

The Tamino API for .NET `TaminoSchemaCommand` Category

The Tamino API for .NET `TaminoSchemaCommand` category comprises the following performance counters:

Define:TaminoServerDuration	Define:TotalCommunicationDuration	Define:TotalOperationDuration	Define:
Undefine:TaminoServerDuration	Undefine:TotalCommunicationDuration	Undefine:TotalOperationDuration	Undefi

The Tamino API for .NET TaminoConnectionCommand Category

The Tamino API for .NET TaminoConnectionCommand category comprises the following performance counters:

Close:TaminoServerDuration	Close:TotalCommunicationDuration	Close:TotalOperationDuration	Close:X
Commit:TaminoServerDuration	Commit:TotalCommunicationDuration	Commit:TotalOperationDuration	Comm
Diagnose:TaminoServerDuration	Diagnose:TotalCommunicationDuration	Diagnose:TotalOperationDuration	Diagno
Open:TaminoServerDuration	Open:TotalCommunicationDuration	Open:TotalOperationDuration	Open:X
Rollback:TaminoServerDuration	Rollback:TotalCommunicationDuration	Rollback:TotalOperationDuration	Rollbac

The Tamino API for .NET `TaminoDataAdapter` Category

The Tamino API for .NET `TaminoDataAdapter` category comprises the following performance counters:

Fill:HierarchyValidationDuration
Fill:TotalCoreOperationDuration
Fill:TotalOperationDuration
Update:CollectModificationDuration
Update:InsertCorrectionDuration
Update:TotalCoreOperationDuration
Update:TotalOperationDuration

Tamino API for .NET Performance Counter Instances

A performance counter can have several instances. An instance contains the counter values. Instances and performance counter values are transitory. Note that instances apply to a category as a whole, rather than to individual counters. This means that all counters within a category have each instance defined for the category.

A performance counter instance only exists as long as at least one component is referencing the corresponding category. A referencing component can be the application that is writing the performance counter values, the Microsoft Performance Monitor, or some other application that is monitoring the counters in this category. For example, as long as the Performance Monitor is open and actively monitoring the Tamino API for .NET `TaminoCommand` category, all performance counter instances in this category are retained.

Each performance counter in each Tamino API for .NET category can have several instances. For an application that is being measured, the friendly name of the .NET application domain (Sys-

tem.AppDomain.CurrentDomain.FriendlyName) is used as the instance name. This allows assigning the measured values to different applications being measured simultaneously on the same system.

Example: Assume that measuring is switched on for the assemblies MyAssembly.exe and AnotherAssembly.exe, and assume that both executables are running simultaneously. For the Query:TaminoServerDuration performance counter the following two instances would exist: MyAssembly.exe and AnotherAssembly.exe.

Monitoring and Logging Performance Counters

You can use the Microsoft Performance Monitor tool to monitor and log performance counters. By means of the logging facilities you can, for instance, make the Performance Monitor write the measured values into a *.csv file; subsequently you can view this file via Excel or display it graphically using the graphical System Monitor part of the Performance Monitor tool.

Microsoft also provides a WMI extension for Visual Studio .NET for download.

Using Management Events

Windows Management Instrumentation (WMI) is an object-oriented framework for accessing management-related information. WMI is extensible, allowing applications to publish and access application-specific objects. The System.Management namespace in .NET provides a set of classes for working conveniently with WMI-based management information. For more information about WMI in .NET, see the .NET Framework documentation.

The Tamino API for .NET has been instrumented to provide performance information in the form of WMI management events. Instances of Tamino API for .NET specific management event classes are published at runtime. In contrast to performance counters, management events can expose all different kinds of properties and methods in a true object-oriented way.

Tamino API for .NET Management Event Classes

Within WMI, all management events that can be fired by the Tamino API for .NET are defined in the namespace "root\TaminoAPI".

Event Classes and Provided Properties for the Core API

`TaminoCommandEvent`
: Provides information about executed operations of `TaminoCommand` objects.

`TaminoSchemaCommandEvent`
: Provides information about executed operations of `TaminoSchemaCommand` objects.

`TaminoConnectionEvent`
: Provides information about executed operations of `TaminoConnection`/`TaminoTransaction` objects.

`TaminoCommandEvent`, `TaminoSchemaCommandEvent` and `TaminoConnectionEvent` are subclasses of `TaminoCoreApiEvent`.

The following information is provided by all subclasses of `TaminoCoreApiEvent`:

AppDomain
The Application Domain of the application.

ThreadId
The ID of the thread on behalf of which the operation was performed.

OperationType
The type of the executed operation.

Url
The URL of the Tamino database on which the operation was executed.

TaminoServerDuration
The TaminoServerDuration value.

TotalCommunicationDuration
The TotalCommunicationDuration value.

XmlParseDuration
The XmlParseDuration value.

TotalOperationDuration
The TotalOperationDuration value.

Event Classes and Provided Properties for the `TaminoDataAdapter` Class

`TaminoDataAdapterEvent`
Provides information about executed operations of `TaminoDataAdapter` objects.

Information provided by `TaminoDataAdapterEvent`:

AppDomain
The Application Domain of the application.

ThreadId
The ID of the thread on behalf of which the operation was performed.

OperationType
The type of the executed operation: Fill, Update, FillSchema.

TotalOperationDuration
The TotalOperationDuration value.

TotalCoreOperationDuration
The TotalCoreOperationDuration value.

HierarchyValidationDuration
The HierarchyValidationDuration value.

InsertCorrectionDuration
The InsertCorrectionDuration value.

CollectModificationDuration
The CollectModificationDuration value.

Firing `TaminoCoreApiEvent` and `TaminoDataAdapterEvent` Instances

Whenever an operation of the `TaminoCommand` class is executed, a `TaminoCommandEvent` instance is created and fired. This means that when running an application, there will be a `TaminoCommandEvent` object for each executed operation. The same is true for `TaminoSchemaCommandEvent`, `TaminoConnectionEvent` and `TaminoDataAdapterEvent` objects.

An event instance only exists as long as some component is referencing the instance. A component can be either the application that is firing events or another application that is watching fired events.

Watching Management Events

If you are using Visual Studio .NET, a convenient way to watch management events at development time is to download Management (WMI) Extensions for Visual Studio .NET RTM Server Explorer from the Microsoft download center. Another option is to use the .NET Managed Classes (included in the .NET framework) to write a simple event viewer that meets your requirements. The MeasuringSamples include a simple example event watcher.

Running the Samples

The following shows how to measure operation durations for the CoreAPIPerformanceCounters sample and the CoreAPIManagementEvents that are included in the measuring samples.

Prerequisites for Running the Samples

Before you can run the samples, you must:

- Install the resources as described in the **Installation and Uninstallation** section above;
- Build the samples, for example by running `buildAll.cmd` in the *`TaminoAPI4DotNET`*/*samples* directory;
- Run the script `LoadSampleData.cmd` in the *`TaminoAPI4DotNET`*/*samples* directory.

Executed Operations

Both examples perform the following core API operations on the property `Doctype` in the APISimpleSamples2 collection:

1. Create a `TaminoConnection` object and open the connection;

2. Insert XML documents;
3. Execute an XQuery;
4. Delete the inserted documents using an XQuery-Update statement;
5. Close the connection.

Running the MeasuringSamples

To run the samples, you can use the script `runCoreMeasuringSamples.cmd`, which you can find in the *TaminoAPI4DotNET* directory. The script runs the following two samples one after the other:

1. CoreAPIPerformanceCounter;
2. CoreAPIManagementEvents.

Before running them, it automatically copies the application configuration file to *TaminoAPI4DotNET/samples/MeasuringSamples/CoreAPIPerformanceCounter/bin/Release* and *TaminoAPI4DotNET/samples/MeasuringSamples/CoreAPIManagementEvents/bin/Release* respectively. If you do not use the `runCoreMeasuringSamples.cmd` script to run the samples, make sure that the configuration files are in the application directory before starting the application.

The CoreAPIPerformanceCounter Sample

In order to measure and monitor operation durations, you must perform the following steps:

1. Activate measuring for your application by setting the appropriate entries in the application configuration file;
2. Monitor the running application.

The following sections first explain these two steps, and then provide some step-by-step guidelines for monitoring durations for TaminoCommand.Insert operations using the Microsoft Performance Monitor.

The Application Configuration File

The file *TaminoAPI4DotNET/samples/MeasuringSamples/CoreAPIPerformanceCounter/CoreAPIPerformanceCounter.exe.config* includes the following entry:

```
<configuration>
  <appSettings>
    <add key="TaminoCommandPerformanceCounters" value="true" />
    <add key="TaminoSchemaCommandPerformanceCounters" value="true" />
    <add key="TaminoConnectionPerformanceCounters" value="true" />
  </appSettings>
</configuration>
```

Monitoring Performance Counter Instances

Monitoring can be done using the Microsoft Performance Monitor. One way to start the Performance Monitor is to open a command prompt window and enter the command `perfmon.exe`. Another way is via the menu sequence **Start** > **Settings** > **Control Panel** > **Administrative Tools** > **Performance**.

Right-click in the details pane (that is the right-hand pane) of the Performance Monitor, then select **Add Counters...** You will see the available performance counters in the Tamino API for .NET `TaminoCommand` category. No instances are visible, because they are not created until the application is started. Note also that the instances are only retained as long as some application is referencing the corresponding category.

In order to understand the next steps, we must look at how performance counters are monitored and which applications we would usually monitor.

Let's look at the TaminoCommand.Insert operations executed by CoreAPIPerformanceCounter.exe. When the first TaminoCommand.Insert is executed, a performance counter instance with the name "coreapiperformancecounter.exe" is created. Remember that performance counter instances apply to the category as a whole. This means that after the first TaminoCommand.Insert, the following counters exist:

Counter Name	Instance Name
Delete:TaminoServerDuration	coreapiperformancecounter.exe
Delete:TotalCommunicationDuration	coreapiperformancecounter.exe
Delete:TotalOperationDuration	coreapiperformancecounter.exe
Delete:XmlParseDuration	coreapiperformancecounter.exe
Insert:TaminoServerDuration	coreapiperformancecounter.exe
Insert:TotalCommunicationDuration	coreapiperformancecounter.exe
Insert:TotalOperationDuration	coreapiperformancecounter.exe
Insert:XmlParseDuration	coreapiperformancecounter.exe
Query:TaminoServerDuration	coreapiperformancecounter.exe
Query:TotalCommunicationDuration	coreapiperformancecounter.exe
Query:TotalOperationDuration	coreapiperformancecounter.exe
Query:XmlParseDuration	coreapiperformancecounter.exe
Retrieve:TaminoServerDuration	coreapiperformancecounter.exe
Retrieve:TotalCommunicationDuration	coreapiperformancecounter.exe
Retrieve:TotalOperationDuration	coreapiperformancecounter.exe
Retrieve:XmlParseDuration	coreapiperformancecounter.exe
Update:TaminoServerDuration	coreapiperformancecounter.exe
Update:TotalCommunicationDuration	coreapiperformancecounter.exe

Counter Name	Instance Name
Update:TotalOperationDuration	coreapiperformancecounter.exe
Update:XmlParseDuration	coreapiperformancecounter.exe

Whenever the application executes a TaminoCommand.Insert, the values of the following counters are modified:

Counter Name	Instance Name
Insert:TaminoServerDuration	coreapiperformancecounter.exe
Insert:TotalCommunicationDuration	coreapiperformancecounter.exe
Insert:TotalOperationDuration	coreapiperformancecounter.exe
Insert:XmlParseDuration	coreapiperformancecounter.exe

When monitoring a performance counter, the Microsoft Performance Monitor interrogates the current values of the monitored counters every X seconds. This implies that performance counters are well suited for monitoring application behavior over time; they are less well suited for measuring the duration of a single call.

When should we select the instances for monitoring in the Performance Monitor? Considering long-running applications and monitoring behavior over time, the answer is: First start the application, then select the instances for monitoring/logging.

Note: Since CoreAPIPerformanceCounter.exe is not really a long running application, it includes some Thread.Sleep calls in order to make it run longer.

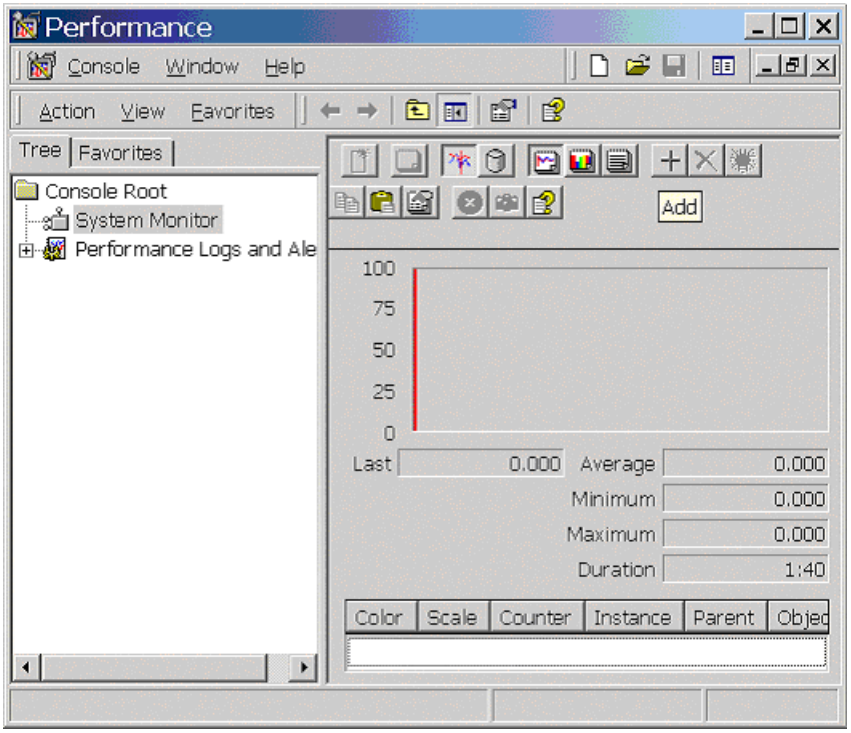
Hint: Instead of selecting the performance counter instances by hand, you can also use System Monitor Templates to easily start monitoring pre-defined performance counters. The CoreApiPerformanceCounter sample includes a suitable template: *TaminoAPI4DotNET/samples/MeasuringSamples/CoreApiPerformanceCounter/CoreApiPerformanceCounter.htm*. To use this template, do the following in the Microsoft Performance Monitor:

1. If necessary, expand **Performance Logs and Alerts**;
2. Right-click **Counter Logs**;
3. Select **New Log Settings From…**;
4. Select *CoreApiPerformanceCounter.htm*.

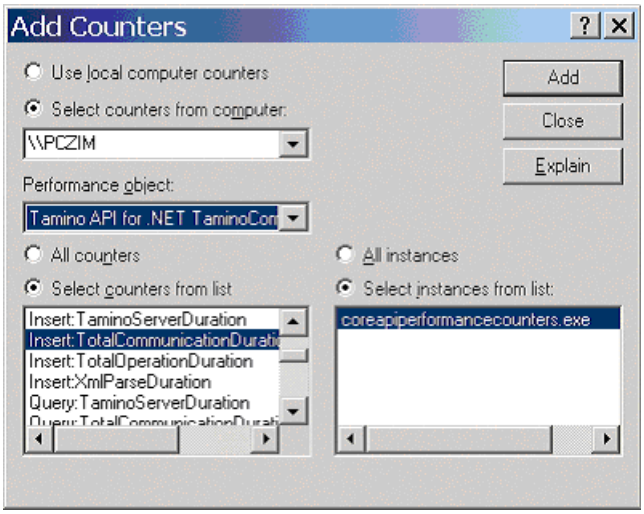
Monitoring TaminoCommand.Insert Step-by-Step

The following summarizes the preceding descriptions.

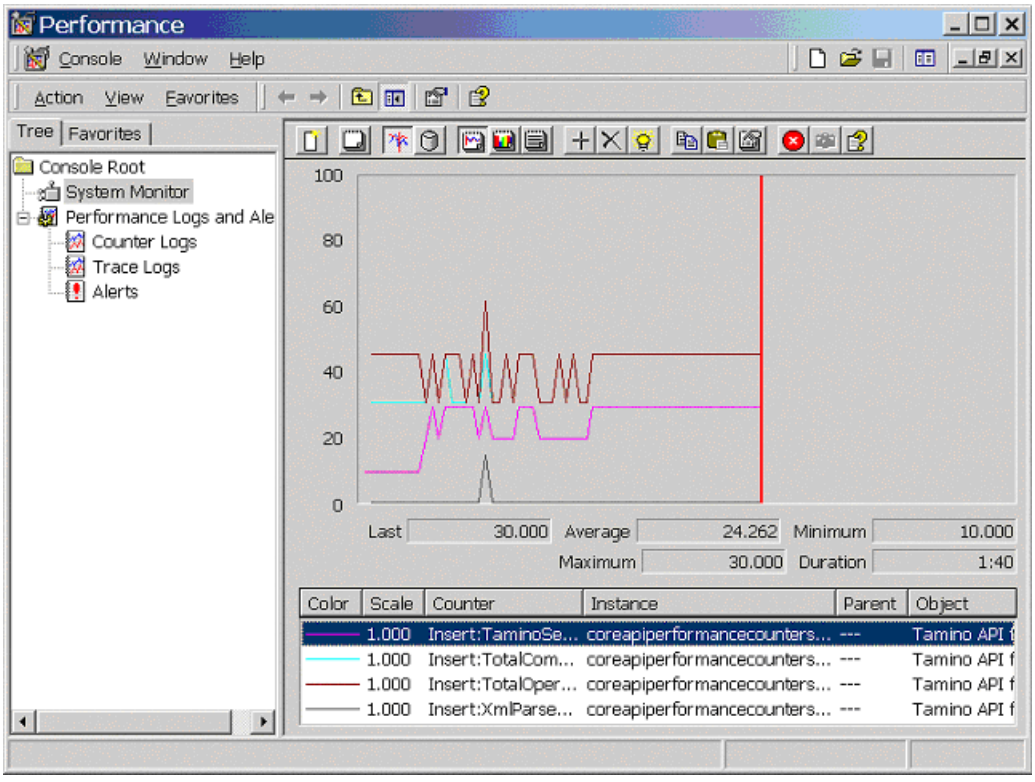
1. Open a command prompt. Change to the *TaminoAPI4DotNET/samples* directory.
2. Enter the command `runCoreMeasuringSamples.cmd`
3. Start the program `perfmon.exe`
4. Right-click in the right-hand pane, then select **Add Counters...**



5. Add the four counters related to the TaminoCommand.Insert operation:



6. If the application has already finished, start `runCoreMeasuringSamples.cmd` again. After a couple of minutes, you will see a graph similar to the following:



The CoreAPIManagementEvents Sample

To measure and monitor operation durations, perform the following steps:

1. Enable measuring for your application via the application configuration file.
2. Start an event watcher.
3. Run the application to be measured.

The Application Configuration File

The file *TaminoAPI4DotNET/samples/MeasuringSamples/CoreAPIPerformanceCounter/CoreAPIManagementEvents.exe.config* includes the corresponding entry:

```
<configuration>
  <appSettings>
    <add key="TaminoCommandManagementEvents" value="true" />
    <add key="TaminoSchemaCommandManagementEvents" value="true" />
    <add key="TaminoConnectionManagementEvents" value="true" />
  </appSettings>
</configuration>
```

Watching Management Events

The measuring samples include the source code for a simple event watcher. The events that an application would like to watch for are specified using the WMI Query Language (WQL). The following shows the basic code to register an `EventArrived` method, which will be called for each fired `TaminoCoreApiEvent` object:

```
WqlEventQuery query = new WqlEventQuery("TaminoCoreApiEvent");
query.QueryString = "SELECT * FROM TaminoCoreApiEvent";
ManagementScope scope = new ManagementScope(@"root\TaminoApi");
apiWatcher = new ManagementEventWatcher(scope,query);
apiWatcher.EventArrived += new
EventArrivedEventHandler(this.EventArrived);
```

The following code in the `EventArrived` methods simply inserts all the application-specific properties of the fired event into a `DataSet`:

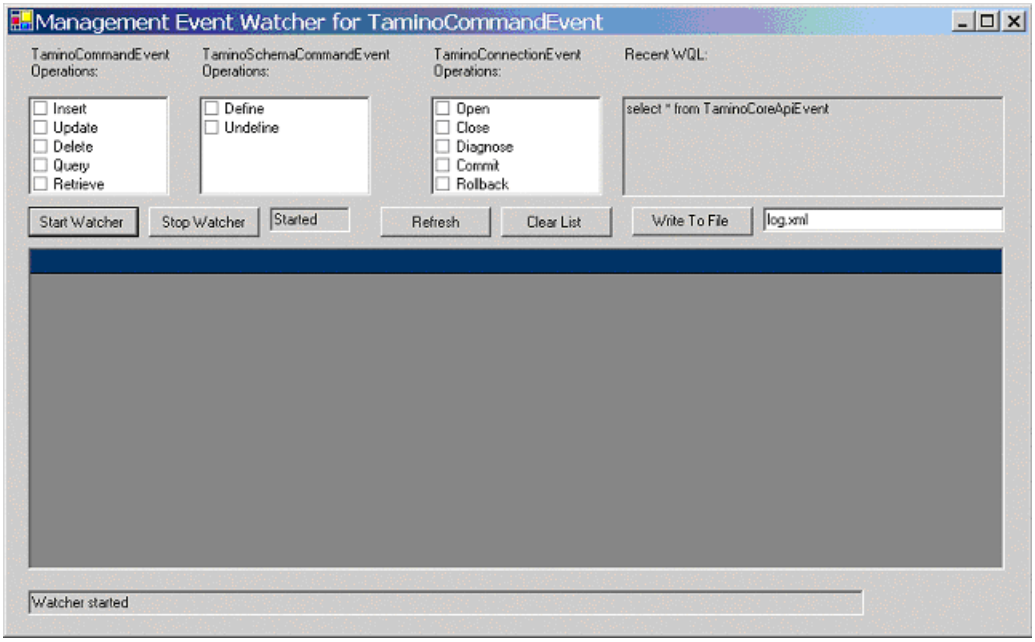
```
DataSet ds = new DataSet();
DataRow row = coreDs.Tables[0].NewRow();

foreach ( PropertyData data in e.NewEvent.Properties)
{
   row[data.Name] = data.Value.ToString();
}
ds.Tables[0].Rows.Add(row);
```

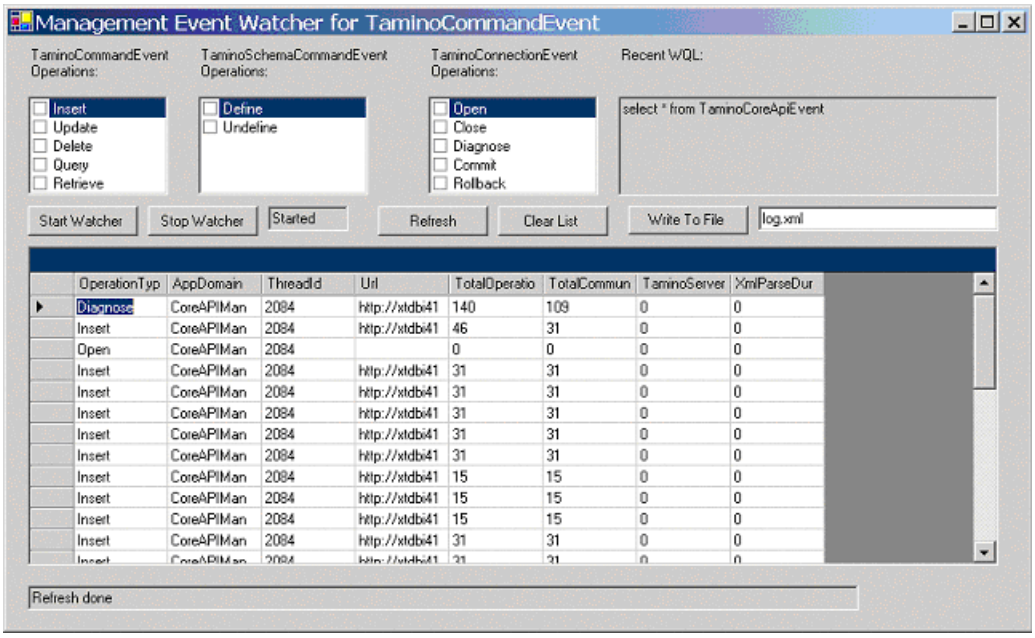
Remember that the Tamino API for .NET includes three subclasses of the `TaminoCoreApiEvent` class: `TaminoCommandEvent`, `TaminoSchemaCommandEvent` and `TaminoConnectionEvent`. In WQL you can also formulate more specific queries; for instance, you might only want to listen for `TaminoCommandEvent` objects with the OperationType "Insert".

Watching TaminoCoreApiEvent Objects Step-by-Step

1. Open a command prompt. Change to the *`TaminoAPI4DotNET`/samples* directory.
2. Enter the command `runManagementEventWatcher.cmd`
3. Select **Start Watcher**.



4. Start the application by entering the command `runCoreMeasuringSamples.cmd` at the command prompt.
5. In the EventWatcher, select **Refresh**. Select the `TaminoCoreApiEvent` table.



6. Select **Stop Watcher**.
7. Select **Write To File**. This writes to the application directory a file similar to the following:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
…
 <TaminoCoreApiEvent>
    <OperationType>Insert</OperationType>
    <AppDomain>CoreAPIManagementEvents.exe</AppDomain>
    <ThreadId>2084</ThreadId>
    <Url>http://xtdbi4141-rc07/tamino/zimdb1/APISimpleSamples2</Url>
    <TotalOperationDuration>46</TotalOperationDuration>
    <TotalCommunicationDuration>31</TotalCommunicationDuration>
    <TaminoServerDuration>0</TaminoServerDuration>
    <XmlParseDuration>0</XmlParseDuration>
  </TaminoCoreApiEvent>
 …
 <TaminoCoreApiEvent>
    <OperationType>Query</OperationType>
    <AppDomain>CoreAPIManagementEvents.exe</AppDomain>
    <ThreadId>2084</ThreadId>
    <Url>http://xtdbi4141-rc07/tamino/zimdb1/APISimpleSamples2</Url>
    <TotalOperationDuration>62</TotalOperationDuration>
    <TotalCommunicationDuration>31</TotalCommunicationDuration>
    <TaminoServerDuration>50</TaminoServerDuration>
    <XmlParseDuration>31</XmlParseDuration>
  </TaminoCoreApiEvent>
…
</NewDataSet>
```

VI API Reference Information

20 API Reference Information

To view the reference documentation of the Tamino API for .NET classes, please select this link: ../navig/ndoc_navig.htm

Alternatively, if you prefer to use Microsoft's HTML Help system, you can use the file *TaminoInstallDir*/*Documentation/en/dotnetapi/dotnetapiref/Documentation.chm*.

Note: Depending on your PC's security settings, you may have to copy this file to a local hard disk in order to be able to use it.

Index

Symbols

A

C

D

H

I

N

O

P

Q

R

S

T

U

X