Matisse® XML Programming Guide

May 2013

MATISSE XML Programming Guide

PDF generated 2 May 2013

Contents

Intended Audience 7
Conventions 7

1 Using the XML Utilities 8
1.1 The mt_xml Utility 8
Return Status 8
Location 8

2 Importing XML Documents 9
2.1 A Simple Example 9
XML Example 9
ODL Example 9
2.2 Attribute Values 10
Integer 10
Real Number 11
MT_STRING 11
MT_CHAR 12
Boolean 12
Date 12
Timestamp 12
Interval 13
2.3 Importing Multiple Objects 13
XML Example 13
2.4 Using Relationships 14
ODL Example 14
XML Example 1 15
XML Example 2 16
2.5 Using MtPrimaryKey 17
XML Example 17
2.6 Updating Attributes and Relationships of Objects 18
Updating Attributes 18
Updating Relationships 19
2.7 Deleting Objects 21
2.8 Using the Oid Format 21
ODL Example 21
Creating Objects 22
Updating Attributes 22
Updating Relationships 23
Deleting Objects 24
2.9 Matisse List Data Types 24
Numerical Lists and Boolean Lists 24

MT_STRING_LIST 24
Other Lists 25
2.10 Matisse Data Types Restriction 25
2.11 Multiple transactions (--commit option) 25
2.12 Remapping Namespaces (--fn and --dn options) 26
2.13 Parallel Loading (--parallel option) 26
2.14 Current Limitations 26

3 Exporting XML Documents 27
3.1 Export Using SQL (-sql option) 27
3.2 Export Using OID (--oid option) 27
3.3 Exporting Primary Keys (--pkoid option) 27
3.4 Exporting Media Data (--emedia option) 28
3.5 Splitting XML Data (--size option) 28
3.6 Exporting from a Namespace (--ns option) 28
3.7 Exporting in parallel (--parallel option) 28
3.8 Exporting the whole Database 28
Using the -full option 28
XML example 29
Notes on importing 29

4 Matisse XML C Programming API 30
4.1 Environment 30
4.2 API References 30
`Error` 30
`ExportObjects` 30
`Failure` 32
`Free` 32
`ImportXML` 33
`MATISSEError` 35
`Success` 35
4.3 Types and Errors 36
`MTXML_INVALXML` 36
`MTXML_MATISSE_ERROR` 36
`MTXML_NOTENOUGHSPACE` 36
`MTXML_NULLPOINTER` 37
`MTXML_SYSTEMERROR` 37
`MTXML_TYPENOTSUPPORTED` 37

5 Programming API for Internal Objects 38
5.1 Environment 38
5.2 API References 38
`CloseInputStream` 38
`FreeObjectRep` 38
`NextObjectRep` 39
`OpenInputStream` 40

5.3 Classes and Types 40
MtXMLElement 41
MtXMLObjElement 41
MtXMLAttElement 42
MtXMLRelElement 43
MtXMLAttribute 43
5.4 Errors 43
MTXML_ENDOFSTREAM 43
MTXML_INVALSTREAM 44

Index 45

Tables

Table 1.1 `mt_xml` statuses 8

Introduction

The `mt_xml` utility simplifies the development of applications using XML with Matisse. The `mt_xml` utility allows you to manage XML documents in the database. With `mt_xml`, you can import XML data into a database or export objects from a database into an XML document.

Intended Audience

This document should be read by any developer handling XML documents with Matisse, regardless of the development environment.

Conventions

This document uses the following conventions:

Text

The running text is written in characters like these.

`Code`

All computer variables, code, commands and interactions are shown in this font. Also, any code and commands that the user must enter are shown on a gray background.

`variable`

In a program example, or in an interaction, a variable, which means anything that is dependent on the user environment, is written in italics.

`[class]`

In the schemas, classes are represented by their name between brackets (`[]`).

`attribute/`

In the schemas, attributes are represented by their name followed by the character `/`.

`relationship->`

In the schemas, relationships are represented by their name followed by the characters `->`.

References

References to another part of the Matisse documentation are made as shown here.

1 Using the XML Utilities

1.1 The mt_xml Utility

To import an XML document file *input.xml* into a database *example* on host *localhost*, use the following command:

```
% mt_xml -d example@localhost import -f input.xml
```

To export objects specified by SQL statement *sql* from database *example* on host *localhost* to the file *output.xml*, use the following command:

```
% mt_xml -d example@localhost export -f output.xml --sql
"<sql>"
```

You can use the standard input or the standard output instead of specifying *xml_file* by using the following options:

```
--in:reads XML document from the standard input
--out:writes XML document into the standard output
```

You can get a status report of the number of objects imported/exported by specifying the `-v` option. The status report is written to the standard error. The `--help` option provides a full description of the command line options.

Return Status

The `mt_xml` utility can return any status listed below.

Table 1.1 `mt_xml` **statuses**

Status	Code	Description
XML_SUCCESS	0	Successful. The whole XML document has been stored into the database as new objects.
XML_PSUCCESS	1	Successful. However, some elements in the XML document were not imported in the database, since they already existed in the database (see section 2.4).
XML_MATISSE_ERROR	2	Error regarding Matisse (for example, class not found).
XML_SYNTAX_ERROR	3	Error regarding XML syntax.
XML_NOSUCHFILE	4	xml_file specified in the command line was not found.

Location

`mt_xml` is located in $MATISSE_HOME/bin.

2 Importing XML Documents

The `mt_xml` utility adheres to the XML 1.0 standard specification. The document type definition (DTD) of XML documents need to follow the Matisse database schema, but they do not have to match exactly. Note that the XML document does not have to include any reference to the DTD.

2.1 A Simple Example

This example shows how to describe XML information and store it into the database. The following XML document contains the description of an instance of Employee.

XML Example

```
<!-- *** xml_example2.1 *** -->
<?xml version="1.0"?>
<!-- Note that the DTD does not have to be included -->
<!DOCTYPE Employee [
<!ELEMENT Employee (FirstName, MiddleInitial?,
       LastName, Birthday, SocialSecurityNumber,
       StartYear)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT MiddleInitial (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT Birthday (#PCDATA)>
<!ELEMENT SocialSecurityNumber (#PCDATA)>
<!ELEMENT StartYear (#PCDATA)>
]>

<Employee>
    <FirstName>Amy</FirstName>
    <MiddleInitial>F</MiddleInitial>
    <LastName>Martin</LastName>
    <Birthday>1967-02-09</Birthday>
    <SocialSecurityNumber>123-34-4567
       </SocialSecurityNumber>
     <StartYear>1995</StartYear>
</Employee>
```

The corresponding Matisse database schema, for example, can be defined in the ODL format as follows:

ODL Example

```
/**** odl_example2.1 ****/
interface Employee : persistent
```

```
{
  attribute String FirstName;
  attribute String MiddleInitial = MtString(NULL);
  attribute String LastName;
  attribute MtTimestamp Birthday;
  attribute String<12> SocialSecurityNumber;
  mt_entry_point_dictionary SSNDict
      entry_point_of SocialSecurityNumber
      make_entry_function "make_entry";
  mt_index SSNIdx
     criteria {Employee::SocialSecurityNumber MT_ASCEND};

  attribute Long StartYear;
};
```

After importing the above XML document, you'll get an instance of the class Employee in the database.

The values of the XML elements `<FirstName>`, `<MiddleInitial>`, and `<SocialSecurityNumber>` are stored as string values in the Matisse object since the corresponding Matisse attributes have the type String (see odl example 2.1). The values of the `<Birthday>` and `<StartYear>` elements are stored as MtTimestamp and Long type in the object respectively.

When an XML element has an invalid value as its corresponding Matisse attribute, a Matisse error (`MATISSE_INVALIDVALUE`) is raised.

If the XML Employee element includes an element, for example, Hobby, which has no corresponding Matisse attribute in a database, this element is transparently ignored, that is, no error or warning is returned.

Note that the element MiddleInitial of the element Employee in the DTD is optional since it is followed by "?" in the DTD. When an Employee element does not have a MiddleInitial element, the corresponding object in the database will have the default value (`MtString(NULL)`).

Note that all the Matisse attributes except for MiddleInitial require a value. If you don't provide any value for these attributes, the Invalid attribute type Matisse error is raised.

2.2 Attribute Values

This section explains the valid format for each Matisse attribute type. For the list types, please refer to section 2.9, Matisse List Data Types, on page 24.

Integer

This includes the Matisse types `MT_BYTE`, `MT_SHORT`, `MT_INTEGER`, and `MT_LONG`. The valid format for integer is as follows:

```
[+|-][0{x|X}]{0-9}*
```

If the number starts with 0 (zero) (except for the + or – sign), it is treated as an octal number. If the number starts with 0x, it is treated as a hexadecimal number. For example,

```
<integer>1122</integer>
```
(decimal number)

```
<integer>01122</integer>
```
(octal number)

```
<integer>-0x1122</integer>
```
(hexadecimal number)

If the number is out of range, an error is raised. For example, if the Matisse attribute type is `MT_SHORT` and the XML element is as follows:

```
<short>1234567890</short>
```

then you will get an error since valid value for the type `MT_SHORT` is between -32767 and 32767.

When the XML element has no value as shown below, no value is saved, that is, the corresponding Matisse attribute value remains undefined or unchanged:

```
<integer></integer>
```

Real Number

This includes the Matisse types `MT_FLOAT` and `MT_DOUBLE`. The valid format for real number is as follows:

```
[+|-][{0-9}*][.{0-9}*][{e|E}[+|-]{0-9}*]
```

The following examples are valid format for real numbers:

```
<double>123</double>
<double>123.</double>
<double>-.123</double>
<double>+1.23e05</double>
<double>123.E-5</double>
```

When the XML element has no value as shown below, no value is saved; the corresponding Matisse attribute value remains undefined or unchanged:

```
<float></float>
```

MT_STRING

When the XML element has no value as shown below, the corresponding Matisse attribute value will have an empty string:

```
<string></string>
```

MT_CHAR

This includes the Matisse types MT_CHAR and MT_ASCII_CHAR. When the XML element has more than one character, only the first character is stored and the rest is ignored. When the XML element has no value as shown below, no value is saved; the corresponding Matisse attribute value remains undefined or unchanged:

```
<char></char>
```

Boolean

The valid values for the Matisse attribute type MT_BOOLEAN are:

```
true
false
```

They are not case sensitive. When the XML element has no value as shown below, no value is saved; the corresponding Matisse attribute value remains undefined or unchanged:

```
<boolean></boolean>
```

Date

The valid format for the Matisse attribute type MT_DATE is:

```
YYYY-MM-DD
```

where YYYY is year number, MM is month number, and DD is the day number in the month.

For example, the following is a valid date:

```
<date>2001-01-10</date>
```

The next one is not valid, since the year 2001 is not a leap year:

```
<date>2001-02-29</date>
```

When the XML element has no value as shown below, no value is saved; the corresponding Matisse attribute value remains undefined or unchanged:

```
<date></date>
```

Timestamp

The valid format for the Matisse attribute type MT_TIMESTAMP is:

```
YYYY-MM-DD HH:mm:SS[.uuuuuu]
```

where YYYY is year number, MM is month number, DD is the day number in the month, HH is hour number (24 hour system), mm is minute number, SS is seconds number and uuuuuu is the micro-second number. The time is stored as GMT (Greenwich Mean Time).

For example, the following is a valid timestamp:

```
<timestamp>2001-01-10 23:24:00</timestamp>
```

The next one is not valid, since HH must be between 0 and 23:

```
<timestamp>2001-01-10 24:24:00</timestamp>
```

When the XML element has no value as shown below, no value is saved; the corresponding Matisse attribute value remains undefined or unchanged:

```
<timestamp></timestamp>
```

Interval

The valid format for the Matisse attribute type `MT_INTERVAL` is:

```
[+|-]DD HH:MM:SS[.uuuuuu]
```

where `DD` is number of days, `HH` is hour number, `MM` is minute number, `SS` is seconds number and `uuuuuu` is the micro-second number.

For example, the following is a valid timestamp:

```
<interval>+10 23:00:00.00</interval>
```

When the XML element has no value as shown below, no value is saved; the corresponding Matisse attribute value remains undefined or unchanged:

```
<boolean></boolean>
```

2.3 Importing Multiple Objects

You may want to import multiple objects from an XML file. However, the XML specification allows you to contain only one top element in an XML document. That is, an XML file contains only one object.

We have introduced a processing instruction to let the `mt_xml` utility handle an XML document containing multiple objects. Just insert the processing instruction line:

```
<?mt_xml version="2" container="yes"?>
```

Next, embrace all the elements you want to import with any start tag and end tag, such as `<MtContainer>` and `</MtContainer>`.

The following example contains two instances of the class Employee:

XML Example

```
<!-- *** xml_example2.2 *** -->
<?xml version="1.0"?>
<?mt_xml version="2" container="yes"?>
<MtContainer>
<Employee>
    <FirstName>Amy</FirstName>
    <MiddleInitial>F</MiddleInitial>
    <LastName>Martin</LastName>
    <Birthday>1967-02-09</Birthday>
```

```
    <SocialSecurityNumber>123-34-4567
       </SocialSecurityNumber>
    <StartYear>1995</StartYear>
</Employee>

<Employee>
    <FirstName>Rio</FirstName>
    <LastName>Kay</LastName>
    <Birthday>1958-03-09</Birthday>
    <SocialSecurityNumber>987-87-8765
       </SocialSecurityNumber>
    <StartYear>1989</StartYear>
</Employee>
</MtContainer>
```

2.4 Using Relationships

The above examples have no relationship between objects. Now we want to introduce the department to which the employees belong in our data model. We define a new class, Department, as well as a relationship between Employee and Department in the database as follows:

ODL Example

```
/**** odl_example2.2 ****/
interface Employee : persistent
{
  attribute String FirstName;
  attribute String MiddleInitial = MtString(NULL);
  attribute String LastName;
  attribute MtTimestamp Birthday;
  attribute String<12> SocialSecurityNumber;
  mt_entry_point_dictionary SSNDict
      entry_point_of SocialSecurityNumber
      make_entry_function "make_entry";
  mt_index SSNIdx
     criteria {Employee::SocialSecurityNumber MT_ASCEND};
  attribute Long StartYear;

  relationship Department MemberOf[0,1]
              inverse Department::Members;
};

interface Department : persistent
{
  attribute String<24> Name;
```

```
  mt_entry_point_dictionary DeptNameDict
      entry_point_of Name
      make_entry_function "make_entry";
  mt_index DeptNameIdx
     criteria {Department::Name MT_ASCEND};

  relationship List<Employee> Members
              inverse Employee::MemberOf;
};
```

The corresponding XML DTD and content should look like this:

XML Example 1

```
<!-- *** xml_example2.3 *** -->
<?xml version="1.0"?>
<?mt_xml version="2" container="yes"?>
<!-- DTD does not have to be included -->
<!DOCTYPE Employee [
<!ELEMENT Employee (FirstName, MiddleInitial?,
       LastName, Birthday, SocialSecurityNumber,
       StartYear, Department)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT MiddleInitial (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT Birthday (#PCDATA)>
<!ELEMENT SocialSecurityNumber (#PCDATA)>
<!ELEMENT StartYear (#PCDATA)>
<!ELEMENT Department (Name)>
<!ATTLIST Department MtRelationship (MemberOf) #REQUIRED>
<!ELEMENT Name (#PCDATA)>
]>

<MtContainer>
<Employee>
    <FirstName>Amy</FirstName>
    <MiddleInitial>F</MiddleInitial>
    <LastName>Martin</LastName>
    <Birthday>1967-02-09</Birthday>
    <SocialSecurityNumber>123-34-
4567</SocialSecurityNumber>
   <StartYear>1995</StartYear>
    <Department MtRelationship="MemberOf">
        <Name>Sales</Name>
    </Department>
</Employee>

<Employee>
```

```
    <FirstName>Rio</FirstName>
    <LastName>Kay</LastName>
    <Birthday>1958-03-09</Birthday>
    <SocialSecurityNumber>987-87-
8765</SocialSecurityNumber>
    <StartYear>1989</StartYear>
    <Department MtRelationship="MemberOf">
        <Name>Engineering</Name>
    </Department>
</Employee>
</MtContainer>
```

The element Employee has a new element, Department, that has the element attribute MtRelationship to specify the relationship between an Employee object and a Department object. Now you can know to which department each employee belongs.

Note that Matisse updates the inverse relationship automatically. In the above example, the Engineering Department object will be connected to the Employee object for Rio Kay through the relationship Members after the document is imported.

Also note that the element attribute MtRelationship must appear at the first place in its XML attribute list.

You may need to deal with multiple cardinality relationships. If for example, you want to let an employee belong to two departments at the same time, you will need to add two successor objects through a relationship.

The following example shows how to add two objects of the class Department through the relationship MemberOf.

XML Example 2

```
<!-- *** xml_example2.4 *** -->
<?xml version="1.0"?>

<Employee>
    <FirstName>Amy</FirstName>
    <LastName>Martin</LastName>
    ...
    <Department MtRelationship="MemberOf">
        <Name>Engineering</Name>
    </Department>
    <Department MtRelationship="MemberOf">
        <Name>Sales</Name>
    </Department>
</Employee>
```

2.5 Using MtPrimaryKey

When the `mt_xml` utility imports an XML document, it creates by default a new object for each element representing a class in a database schema. In the XML example 2.3 in section 2.4, two new objects of the class Employee and two new objects of the class Department are created in the database.

This is not acceptable when you import an XML document containing, for example, 20 employees of the department Sales and 10 employees of the department Engineering. You do not want to create 20 different objects of the department Sales.

In this case, you can use an instruction in the preprocessor, `MtPrimaryKey`, to specify an object in the database. The value of an element which has the `MtPrimaryKey` attribute is considered as a unique value to identify an object. For example, when using an associated Entry Point Dictionary on a PrimaryKey attribute:

```
<?mt_xml version="2" container=”yes”
     MtPrimaryKey=”Classname::Attribute”
     MtEntryPointDictionary=”DictionaryName”?>
```

The following example shows how to use `MtPrimaryKey` when using an associated Index on a PrimaryKey attribute:

```
<?mt_xml version="2" container=”yes”
     MtPrimaryKey=”Classname::Attribute”
     MtIndex=”IndexName”?>
```

In the following example, the Department object is identified by its value of the element Name. When the `mt_xml` utility is importing the employee Amy, no Department object with a value Sales as its Name exists in the database. Then a new object of the class Department is created. For the employee Robert, a new object of the class Department is not created. Instead, the existing object, Sales, is related to the employee Robert:

XML Example

```
<!-- *** xml_example2.5 *** -->
<?xml version="1.0"?>
<?mt_xml version="2" container=”yes”
     MtPrimaryKey=”Department::Name”
     MtIndex=”DeptNameIdx”?>

<MtContainer>
<Employee>
    <FirstName>Amy</FirstName>
    ...
    <Department MtRelationship="MemberOf">
        <Name>Sales</Name>
```

```
    </Department>
</Employee>

<Employee>
    <FirstName>Robert</FirstName>
    ...
    <Department MtRelationship="MemberOf">
        <Name>Sales</Name>
    </Department>
</Employee>
</MtContainer>
```

When you use this `MtPrimaryKey` feature, you need to define an entry point dictionary or index on the Matisse attribute corresponding to the XML element you put in `MtPrimaryKey`, as shown in the following ODL. Using an EntryPointDictionary (ODL):

```
interface Department : persistent
{
  attribute String Name;
  mt_entry_point_dictionary DeptNameDict
     entry_point_of Name
     make_entry_function "make_entry";
  ...
};
```

Using an Index (ODL):

```
interface Department : persistent
{
  attribute String<24> Name;
  mt_index DeptNameIndex
     criteria {Department::Name MT_ASCEND};
};
```

Note that when more than one object is found from a key value specified as `MtPrimaryKey`, an error is raised.

2.6 Updating Attributes and Relationships of Objects

Updating Attributes

When you want to update values of objects, you use the command line option `-update` or the XML attribute `MtAction="update"` and the `MtPrimaryKey` feature together. For example, suppose you need to change the last name of the employee Amy shown in XML example 2.5. You prepare the XML document like this:

```
<?mt_xml version="2" container=”yes”
     MtPrimaryKey=”Employee::SocialSecurityNumber”
     MtIndex=”SSNIdx”?>


<Employee>
    <LastName>Tesler</LastName>
  <SocialSecurityNumber>
       123-34-4567</SocialSecurityNumber>
</Employee>
```

And you type the command line:

```
% mt_xml -d database@host import -f xml_file --update
```

Then the `mt_xml` utility searches the object whose social security number is 123-34-4567 and updates the last name attribute of the object. Other attribute values and relationships remain the same.

Updating Relationships

Suppose you transfer Amy from the Sales department to the Engineering department. You will need to update the relationships. You prepare a new XML document using the `MtPrimaryKey` feature like this:

```
<?mt_xml version="2" container=”yes”
     MtPrimaryKey=”Employee::SocialSecurityNumber”
     MtIndex=”SSNIdx”
     MtPrimaryKey=”Department::Name”
     MtIndex=”DeptNameIdx”?>


<Employee>
    <SocialSecurityNumber>
       123-34-4567</SocialSecurityNumber>
    <Department MtRelationship=”MemberOf”>
       <Name>Engineering</Name>
    </Department>
</Employee>
```

This XML document is going to replace the current successor object through the relationship MemberOf of the Employee Amy object, with the Engineering department object.

If you want to make Amy belong to two departments at the same time (for instance, not only the Sales department as specified in XML example 2.3, but also the Engineering department), you are going to use a the XML attribute, `MtAction`, along with `MtRelationship` to specify the relationship successor operation. The operation is either “**`replace`**”, “**`append`**”, “**`remove`**”, “**`appendIfNotExist`**”, “**`forceAppend`**” or “**`removeIfExist`**”. The default operation is “**`replace`**”. A sample XML document to append the Engineering department object to Amy through the relationship MemberOf would look like this:

```
<?mt_xml version="2" container=”yes”
     MtPrimaryKey=”Employee::SocialSecurityNumber”
     MtIndex=”SSNIdx”
     MtPrimaryKey=”Department::Name”
     MtIndex=”DeptNameIdx”?>

<Employee>
   <SocialSecurityNumber>123-34-4567
      </SocialSecurityNumber>
   <Department MtRelationship=”MemberOf”
              MtAction=”append”>
      <Name>Engineering</Name>
   </Department>
</Employee>
```

The description of each operation is as follows:

clear Remove all the current successor objects.

replace Remove all the current successor objects, and then add the new object specified in the XML document. This is the default for `MtAction`.

append Add the new object to the end the object specified in the XML document, while keeping the current successor objects. If the object already exists in the current successors list, an error is returned.

remove Remove the object specified in the XML document from the current list of the successors. If the object does not exist in the current successors list, an error is returned.

appendIfNotExist Add the new object to the end the object specified in the XML document only if the object does not exist in the current list of successors. Keep the other successor objects in the list.

forceAppend Add the new object to the end the object specified in the XML document if the object does not currently exist in the list of successors. If the object already exists in the current list of successors, first remove the object from the list and then add the object to the end of the list. Keep the other successor objects in the list.

removeIfExist Remove the object specified in the XML document from the current list of the successors only if the object already exists in the list.

Note that the cardinality of the relationship MemberOf in the ODL definition needs to be updated so an employee can be a member of more than one department. For example:

```
relationship Department MemberOf [0,2]
   inverse Department::Members;
```

2.7 Deleting Objects

When you want to delete objects, you use the command line option `-update` and the `MtPrimaryKey` feature together with the XML attribute `MtAction="delete"`. For example, suppose you need to remove one `Employee` object from the database. You prepare the XML document like this:

```
<?mt_xml version="2" container="yes"
     MtPrimaryKey="Employee::SocialSecurityNumber"
     MtIndex="SSNIdx"?>


<Employee MtAction="delete">
<SocialSecurityNumber>
       123-34-4567</SocialSecurityNumber>
</Employee>
```

And you type the command line:

```
% mt_xml -d database@host import -f xml_file --update
```

Then the `mt_xml` utility searches the object whose social security number is `123-34-4567` and deletes the object.

2.8 Using the Oid Format

The `oid` processing instruction declares the oid as the primary key of an object. This `oid` value matches the oid of the object in the database for the update operations. For insert operations, the oid value matches the reference of the object inside the XML document. The `oid` value in the XML document is mapped to the oid of the object created in the database during the import processing. Just insert the processing instruction line where the value for `prealloc` represent the number of objects in the XML document:

```
<?mt_xml version="2" container="yes" oid="yes"
prealloc="1234"?>
```

ODL Example

```
/**** odl_example ****/
module xmlExample {

  interface Document : persistent {
    attribute Integer isbn;
    attribute String Title;
    mt_index DocIsbnIdx
      criteria {Document::isbn MT_ASCEND};

    relationship Category Genre[0,1]
      inverse Category::Documents;
  };
```

```
  interface Category : persistent
  {
    attribute String<6> Tag;
    attribute String Name;
    mt_index CatTagIdx
      criteria {Category::Tag MT_ASCEND};

    relationship Set<Document> Documents[0,-1]
      inverse Document::Genre;
  };
};
```

Creating Objects

Suppose you want to create 2 documents of the same category. You prepare a new XML document using the `oid` feature like this:

```
<?mt_xml version="2"
       container="yes"
       oid="yes"
       prealloc="3"?>
<MtContainer>

<xmlExample.Document oid="1">
<isbn MtBasicType="MT_INTEGER">1</isbn>
<title MtBasicType="MT_STRING">Document 001 title</title>
</xmlExample.Document>

<xmlExample.Document oid="2">
<isbn MtBasicType="MT_INTEGER">2</isbn>
<title MtBasicType="MT_STRING">Document 002 title</title>
</xmlExample.Document>

<xmlExample.Category oid="12">
<tag MtBasicType="MT_INTEGER">cat001</tag>
<name MtBasicType="MT_STRING">Category 01</name>
<xmlExample.Document oid="1" MtRelationship="Documents"/>
<xmlExample.Document oid="2" MtRelationship="Documents"/>
</xmlExample.Category>

</MtContainer>
```

Updating Attributes

Suppose you want to update the name of an existing category with the oid 4255. You prepare a new XML document using the `oid` feature like this:

```
<?mt_xml version="2"
```

```
        container="yes"
        oid="yes"
        prealloc="1"?>
<MtContainer>

<xmlExample.Category oid="4255" MtAction="update">
<name MtBasicType="MT_STRING">Category 02 updated</name>
</xmlExample.Category>

</MtContainer>
```

Updating Relationships

Suppose you want to link the document with oid 4248 to the category with oid 4255. You prepare a new XML document using the `oid` feature like this:

```
<?mt_xml version="2"
        container="yes"
        oid="yes"
        prealloc="2"?>
<MtContainer>

<xmlExample.Category oid="4255" MtAction="update">
<xmlExample.Document oid="4248" MtRelationship="Documents"
MtAction="append"/>
</xmlExample.Category>

</MtContainer>
```

Suppose you want to remove the link between the document with oid 4251 and the category with oid 4265. You prepare a new XML document using the `oid` feature like this:

```
<?mt_xml version="2"
        container="yes"
        oid="yes"
        prealloc="2"?>
<MtContainer>

<xmlExample.Category oid="4265" MtAction="update">
<xmlExample.Document oid="4251" MtRelationship="Documents"
MtAction="remove"/>
</xmlExample.Category>

</MtContainer>
```

Suppose you want to remove the links from the category with oid 4265. You prepare a new XML document using the `oid` feature like this:

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

```
<!-- Matisse XML Document -->

<?mt_xml version="2"
       container="yes"
       oid="yes"
       prealloc="1"?>
<MtContainer>

<xmlExample.Category oid="4265" MtAction="update">
<xmlExample.Document oid="0" MtRelationship="Documents"
MtAction="clear"/>
</xmlExample.Category>

</MtContainer>
```

Deleting Objects

Suppose you want to delete the document with oid 4248 and the category with oid 4252. You prepare a new XML document using the `oid` feature like this:

```
<?mt_xml version="2"
       container="yes"
       oid="yes"
       prealloc="2"?>
<MtContainer>

<xmlExample.Category oid="4252" MtAction="delete"/>

<xmlExample.Document oid="4249" MtAction="delete"/>

</MtContainer>
```

2.9 Matisse List Data Types

Numerical Lists and Boolean Lists

All the Matisse list data types are supported. When a Matisse attribute, for example, NumList, is a numerical list such as `MT_INTEGER_LIST` or `MT_DOUBLE_LIST`, the valid XML element has the following format:

```
...
<NumList>1 1 2 3 5 8 13</NumList>
...
```

If the NumList datatype is `MT_INTEGER_LIST`, the XML element will be stored as a list containing seven integers.

MT_STRING_LIST

When you want to store a string list into a Matisse attribute, for example, BookTitles, the corresponding XML elements have the following format:

```
 ...
<BookTitles>Designing XML applications</BookTitles>
<BookTitles>Programming Perl</BookTitles>
<BookTitles></BookTitles>
 ...
```

The Matisse attribute must be `MT_STRING_LIST`. The above elements are stored into a database as a string list that has three string values ("Designing XML applications", "Programming Perl", and an empty string "").

Other Lists

Other lists include `MT_TIMESTAMP_LIST`, `MT_DATE_LIST` and `MT_INTERVAL_LIST`. The corresponding XML elements have a comma-separated list of the formatted string of `MT_TIMESTAMP`, `MT_DATE`, or `MT_INTERVAL`. For example, a list of sunrise times should look like this:

```
<SunriseTime_List>2000-10-12 07:17:00, 2000-10-13 07:19:00,
2000-10-14 07:21:00 </SunriseTime_List>
```

2.10 Matisse Data Types Restriction

For the `mt_xml` utility to properly store XML element values into a database, you must define exactly one type, or any one type plus the type `MT_NULL`, for each Matisse attribute.

For example, if the Matisse attribute SocialSecurityNumber has two possible types, `MT_STRING` and `MT_INTEGER`, the `mt_xml` utility does not know which type to use.

In the current implementation, if a Matisse attribute has multiple types, the `mt_xml` utility simply selects the first non-`MT_NULL` type and tries to convert the XML value to the Matisse attribute type.

2.11 Multiple transactions (--commit option)

The `mt_xml` utility has an option `--commit <n>`. With this option, the utility stores <n> objects per transaction. For example, if an XML document contains 500000 objects and you specify the option `--commit 20000`, the utility iterates the following procedure until all the objects are loaded into the database:

1. Start a transaction.
2. Parse up 256 XML objects.

   If there is a parsing error, the utility exits with an error message, including the line number where the error occurs. If each object has a primary key, objects are sorted alphabetically on their primary key values.

3. Store the parsed data into the database and go to step 2 until <n> objects are stored.
   Note that an XML object can composed of multiple objects to be stored in the database.

   If there is a Matisse error, the utility exits with a Matisse error message.

4. Commit the transaction.

   If there is a Matisse error, the utility exits with a Matisse error message.

The smaller is the number of objects per transaction, the smaller is the risk of a transaction blocking other transactions due to index page updates; therefore, loading multiple XML files in parallel are running faster.

Note that the greater the number of transactions, the more overhead for transaction management is required.

If the number of objects per transaction is large, the program consumes more memory space to cache the objects being stored.

2.12 Remapping Namespaces (--fn and --dn options)

The `mt_xml` utility provides the `--fn` and `--dn` options to remap the data from a source namespace in the XML file into a destination namespace in the database.

```
% mt_xml -d database@host import -f <file> --fn proj.app1 -
--dn prod.client1
```

2.13 Parallel Loading (--parallel option)

The `mt_xml` utility provides the `--parallel <n>` option to import data with `n` tasks running in parallel. The XML data is imported from a multi-segment XML file. The number of tasks is limited by the number of XML file segments.

```
% mt_xml -d database@host import -f <file> --parallel 6
```

2.14 Current Limitations

No values of a XML attribute list are stored in a database. They are ignored except for MtRelationship, MtAction, and MtPrimaryKey.

Even if an XML document contains a DTD, the validity of the XML document's content against the DTD is not checked.

3 Exporting XML Documents

You can export objects in a database in XML format. You can specify objects by a SQL SELECT statement or by OID.

3.1 Export Using SQL (-sql option)

You can use an SQL statement to specify objects to be exported. For example, to export objects of the class Employee, whose last name starts with S, you type:

```
% mt_xml -d database@host export
     [-f <file> | --out] --sql "SELECT * FROM Employee
     WHERE LastName LIKE 'S%'"
```

The double quotation marks surrounding the SQL statement are for escaping characters such as * (asterisk) or ' (single quotation). The `mt_xml` utility reads all strings following "-sql" until the end of the command line. Instead of using double quotation marks as in the above example, you may type the following:

```
% mt_xml -d database@host export
     [-f <file> | --out] --sql SELECT
     \* FROM Employee WHERE LastName LIKE \'S%\'
```

Either way, the `mt_xml` utility gets the same string as the result of the echo command of the UNIX shell. For more information about SQL, refer to the *Matisse SQL Programmer's Guide*.

3.2 Export Using OID (--oid option)

To export objects in a database, you type:

```
% mt_xml -d database@host export
     [-f <file> | --out] --oid <oid> ...
```

The OID can be given either in decimal or in hexadecimal. For hexadecimal OIDs, the OID must be prefixed by 0x.

3.3 Exporting Primary Keys (--pkoid option)

When exporting using the `--sql` or `--oid` options discussed above, including the `--pkoid` option as well will preserve the objects' relationship using a primary key. See the *XML example* in the following section.

3.4 Exporting Media Data (--emedia option)

When exporting data from the database into an XML document, the media data are exported in external file located in the same directory as the XML document. To export media data into the XML document, you now need to add the -emedia option to the export command.

```
% mt_xml -d database@host export
       [-f <file> | --out] --emedia --sql SELECT
       \* FROM Employee WHERE LastName LIKE \'S%\'
```

3.5 Splitting XML Data (--size option)

The `-s` option specifies the XML data file max size therefore splitting XML data into multiple XML files named `<db name>_xds_<document id>.xml`. The file size is in Giga bytes.

```
% mt_xml -d database@host export
        -f <file> --size 2 --sql SELECT
       \* FROM Employee WHERE LastName LIKE \'S%\'
```

3.6 Exporting from a Namespace (--ns option)

The `-n` option specifies to only export the objects from classes defined inside a namespace hierarchy.

```
% mt_xml -d database@host export -f <file> --ns
examples.reports --full
```

3.7 Exporting in parallel (--parallel option)

The `--parallel <n>` option specifies to export data with `<n>` tasks running in parallel. The XML data is exported into multiple XML files named `<filename>_xds_a<documentid>.xml` and `<filename>_xds_r<documentid>.xml`.

```
% mt_xml -d database@host export -f <file> --parallel 6 --
full
```

3.8 Exporting the whole Database

Using the -full option

To export a whole database into a single XML file, use the following command:

```
mt_xml -d database@host export -f xml_file --full
```

It is not necessary to specify the `--pkoid` option with `--full`, primary keys are included automatically.

`mt_xml` does not export the schema, so to export the entire database content , you must use `mt_sdl` as well. The following commands will export the entire database and its schema:

```
mt_xml -d database@host export -f data.xml --full
mt_sdl -d database@host export --odl -f schema.odl
```

These commands may be used to transfer a database from one platform to another, for example from a Windows desktop to a Linux server.

XML example

For example, say that a database to be exported contains two objects, an Employee, and a Department, as well as a relationship between the two objects. The objects in the database would be exported in OID format as:

```
<?mt_xml version="2" container="yes"
          oid="yes"
          prealloc="2"?>
<MtContainer>

<Department oid="4377">
    <Name MtBasicType="MT_STRING">Sales</Name>
    <Employee oid="4375" MtRelationship="Members"/>
</Department>
<Employee oid="4375">
    <FirstName MtBasicType="MT_STRING">Amy</FirstName>
    <LastName MtBasicType="MT_STRING">Martin</LastName>
    <SSN MtBasicType="MT_STRING">123-34-4567</SSN>
    <Department oid="4377" MtRelationship="MemberOf"/>
</Employee>

</MtContainer>
```

Notes on importing

Before importing, edit the XML file so its preprocessor directive is as follows:

```
<?mt_xml version="2" container="yes"
  MtPrimaryKey="MtClass::MtName"
  MtEntryPointDictionary="MtNameDictionary"?>
```

When importing, use `mt_sdl` to load the schema first, then `mt_xml` to import the data as follows:

```
mt_sdl -d database@host import --odl -f schema.odl
mt_xml -d database@host import -f data.xml
```

4 Matisse XML C Programming API

When you want to write your own program to manage XML documents with Matisse, you can use the Matisse XML C Programming API.

4.1 Environment

Your program needs to include the C header file matisseXML.h in the directory $MATISSE_HOME/include. The shared library is $MATISSE_HOME/lib/libmatisseXML.so.

4.2 API References

All the C API functions begin with the prefix MtXML. Functions taking an `MtOID` (an object id) follow the MtXML prefix with an underscore (_). Functions with the prefix MtXML_M signify that memory is allocated by the Matisse XML library.

All of the APIs are listed below:

Error

Synopsis

```
#include "matisseXML.h"
MtString MtXMLError()
```

Purpose This function returns the string associated with the last generated Matisse XML error.

Result A string.

ExportObjects

Synopsis

```
#include "matisseXML.h"
MtXMLSTS MtXML_ExportObjects
(MtSize*  documentSize,
MtString xmlDocument,
MtSize   numObjects,
            ...)
        MtXMLSTS MtXML_ExportNumObjects
```

```
   (MtSize*  documentSize,
    MtString xmlDocument,
    MtSize   numObjects,
    MtOID*   objects)
MtXMLSTS MtXML_MExportObjects
   (MtSize*   documentSize,
    MtString* xmlDocument,
    MtSize    numObjects,
    ...)
MtXMLSTS MtXML_MExportNumObjects
   (MtSize*   documentSize,
    MtString* xmlDocument,
    MtSize    numObjects,
    MtOID*    objects)
```

Purpose These functions export objects that are stored in a database. The objects are exported in XML format.

Arguments

`documentSize` `INPUT/OUTPUT`

In input, specifies the size of the string space specified by the user. Can be used as an input argument only by those functions—MtXML_ExportObjects and MtXML_ExportNumObjects—that do not allocate memory for the string.

In output, gives the length of the string written.

`xmlDocument` `OUTPUT/INPUT`

For those functions—MtXML_ExportObjects and MtXML_ExportNumObjects—that do not allocate memory, this argument is a string space allocated in the calling program. After the function is called, this string will contain the XML document.

For those functions—MtXML_MExportObjects and MtXML_MExportNumObjects—that allocate memory, this argument is a pointer to a string allocated by the functions. In this case, the program must declare an MtString. After declaring it, the program must pass its address as the argument to the function.

In output, it contains the string of the exported XML document.

`numObjects` `INPUT`

The number of objects to be exported.

`objects` `INPUT`

The array of objects to be exported

Other INPUT Arguments The argument numObjects must be followed by the objects (type `MtOID`) to be exported.

Result

```
MTXML_SUCCESS
MTXML_NOTENOUGHSPACE
MTXML_NULLPOINTER
MTXML_TYPENOTSUPPORTED
MTXML_MATISSE_ERROR
    MATISSE_OBJECTDELETED
    MATISSE_OBJECTNOTFOUND
```

Description

The functions MtXML_ExportObjects and MtXML_ExportNumObjects do not allocate a string space to store the XML document of specified objects. The program that calls them must allocate adequate string space.

The functions MtXML_MExportObjects and MtXML_MExportNumObjects allocate a string space to store the XML document of specified objects. When calling these functions, a program must pass as its xmlDocument argument the address of a string. In output, this argument will point to a string that contains the XML document. To free the memory space allocated for the string, the program must call the function MtXMLMFree.

This function can be called inside a transaction or during a version access.

Failure

Synopsis

```
#include "matisseXML.h"
int  MtXMLFailure (MtXMLSTS status)
```

Purpose

This macro indicates whether a Matisse XML function has completed successfully (see also, `Success`).

Arguments

`status` INPUT

The status returned by a Matisse XML function.

Result

Zero (0) if the status corresponds to a success; a non-null integer otherwise.

Free

Synopsis

```
#include "matisseXML.h"
MtXMLSTS MtXMLMFree(void* value)
```

Purpose

This function frees the memory allocated by the functions that allocate memory (MtXMLM* and MtXML_M*).

Arguments

`value` INPUT

A value allocated by one of the functions that allocate memory (MtXMLM* and MtXML_M*).

Result `MTXML_SUCCESS`

Description When the program calls one of the Matisse XML functions that begin with the letters MtXMLM or MtXML_M, Matisse XML allocates memory to store the value. When the value is not needed anymore, the program must free the value with this function.

ImportXML

Synopsis

```
#include "matisseXML.h"
MtXMLSTS MtXMLImportXML
            (MtSize*   numReadObjects,
             MtSize*   numCreatedObjects,
             MtOID*    readObjs,
             MtOID*    createdObjs,
             MtString  xmlDocument,
             MtBoolean pkUpdate)
          MtXMLSTS MtXMLFImportXML
             (MtSize*   numReadObjects,
             MtSize*   numCreatedObjects,
             MtOID*    readObjs,
             MtOID*    createdObjs,
             FILE*     xmlFile,
             MtBoolean pkUpdate)
          MtXMLSTS MtXMLMImportXML
             (MtSize*   numReadObjects,
             MtSize*   numCreatedObjects,
             MtOID**   readObjs,
             MtOID**   createdObjs,
             MtString  xmlDocument,
             MtBoolean pkUpdate)
          MtXMLSTS MtXMLMFImportXML
             (MtSize*   numReadObjects,
             MtSize*   numCreatedObjects,
             MtOID**   readObjs,
             MtOID**   createdObjs,
             FILE*     xmlFile,
             MtBoolean pkUpdate)
```

Purpose These functions read an XML document and store it as objects in a database.

Arguments `numReadObjects  OUTPUT`

The number of objects which are parsed in the xmlDocument. Can be set to NULL, in which case the function simply does not return this number.

```
numCreatedObjects  OUTPUT
```

The number of objects which are parsed in the xmlDocument and newly created in a database.

Can be set to NULL, in which case the function simply does not return this number.

```
readObjs        OUTPUT
```

An array containing all the OIDs of parsed objects. They include both newly created objects and existing objects. (Existing object: object found through entry-point from a given MtPrimaryKey value.)

```
createdObjs     OUTPUT
```

An array containing all the OIDs of new objects created.

```
xmlDocument     INPUT
```

A string containing an XML document.

```
xmlFile         INPUT
```

A file containing an XML document.

```
pkUpdate        INPUT
```

This parameter indicates whether the values of the object have to be updated if the object already exists in a database.

Result

```
MTXML_SUCCESS
MTXML_INVALXML
MTXML_NULLPOINTER
MTXML_TYPENOTSUPPORTED
MTXML_MATISSE_ERROR
   MATISSE_NOSUCHCLASS
```

Description

The numbers numReadObjects and numCreatedObjects returned by the functions count only top-level objects, not including nested objects. For example, the following XML document contains two top-level objects of class person. The person object named Brian Watts is not counted.

```
<person>
   <name>John Smith</name>
   <person>
        <name>Brian Watts</name>
   </person>
</person>
<person>
   <name>Tom Lehman</name>
</person>
```

An XML document can specify an object in a database by using the MtPrimaryKey attribute in the XML document (see section 2.4, Using the MtPrimaryKey Keyword). When an object is found in a database according to the MtPrimaryKey, the values of the object are updated if the argument `pkUpdate` is set to MT_TRUE. If the argument `pkUpdate` is MT_FALSE, the values of the object are not updated.

This function can be called only inside a transaction.

MATISSEError

Synopsis

```
#include "matisseXML.h"
MtSTS  MtXMLMATISSEError()
```

Purpose — When one of the Matisse XML functions returns the error status MTXML_MATISSE_ERROR, this function returns the status of the last generated Matisse error.

Result — A Matisse error status.

Description — The Matisse XML functions use the Matisse C API functions to access a database. When one of these Matisse C API functions returns an error, the Matisse XML function returns the error MTXML_MATISSE_ERROR. To get the Matisse error status, use this function.

Example — If the function MtXML_ExportObjects is called without opening a transaction or starting a version access, it returns the error MTXML_MATISSE_ERROR because it can not access the database. In this case, the function MtXMLMATISSEError returns the Matisse error MATISSE_NOTRANORVERSION, which indicates "Attempt to access objects without a transaction or version access."

Success

Synopsis

```
#include "matisseXML.h"
int MtXMLSuccess (MtXMLSTS status)
```

Purpose — This macro indicates if a Matisse XML function has executed successfully (see also, `Failure`).

Arguments

`status` `INPUT`

The status returned by a Matisse XML function.

Result — Zero (0) if the status corresponds to a failure; a non-null integer otherwise.

4.3 Types and Errors

An enumeration type MtXMLSTS is defined for the Matisse XML error status. This section lists the errors that may result from the use of the Matisse XML functions:

MTXML_INVALXML

Description The given XML document is not a valid XML document. This error occurs when calling one of the following functions:

```
MtXMLImportXML
MtXMLImportXMLFile
```

Solution Correct a syntax error in the XML document.

MTXML_MATISSE_ERROR

Description There is an error related to Matisse functions. This error occurs when calling one of the following functions:

```
MtXML_ExportObjects
MtXML_ExportNumObjects
MtXML_MExportObjects
MtXML_MExportNumObjects
MtXMLImportXML
MtXMLImportXMLFile
```

MTXML_NOTENOUGHSPACE

Description There is not enough space to copy data. This error occurs when calling one of the following functions:

```
MtXML_ExportObjects
MtXML_ExportNumObjects
```

Matisse XML attempts to copy the data into the space allocated by the user. The pointer and the size are specified in the arguments. Matisse XML has insufficient space to copy the data.

Solution Increase the memory space passed to the function until there is a sufficient amount for the data being exported.

MTXML_NULLPOINTER

Description Null pointer: A null pointer is specified as an argument, while this pointer should not be null.

MTXML_SYSTEMERROR

Description This error should never happen, but it might occur after a call to a Matisse XML function.

Solution Contact your software support center.

MTXML_TYPENOTSUPPORTED

Description The type of Matisse attribute is not supported. This error could occur when calling one of the following functions:

```
MtXML_ExportObjects
MtXML_ExportNumObjects
MtXML_MExportObjects
MtXML_MExportNumObjects
MtXMLImportXML
MtXMLImportXMLFile
```

Solution Contact your software support center.

5 Programming API for Internal Objects

The functions listed in this section provide the interface to access the internal object representation of a parsed XML document. You will find an example program to enumerate all objects in an XML document in section 5.3.

5.1 Environment

Your program needs to include the C header file matisseXMLinternal.h in the directory $MATISSE_HOME/include. The shared library is $MATISSE_HOME/lib/libmatisseXML.so.

5.2 API References

CloseInputStream

Synopsis

```
#include "matisseXMLinternal.h"
MtXMLSTS MtXMLCloseInputStream
    (MtXMLStream xmlStream)
```

Purpose This function closes the stream that is pointed at by xmlStream.

Arguments

xmlStream	INPUT	

An XML stream.

Result

```
MTXML_SUCCESS
MTXML_INVALSTREAM
```

FreeObjectRep

Synopsis

```
#include "matisseXMLinternal.h"
MtXMLSTS MtXMLFreeObjectRep
    (MtXMLObjElement* objectRep)
```

Purpose This function frees a previously allocated internal object structure.

Arguments

objectRep	INPUT	

An object representation to be freed.

Result `MTXML_SUCCESS`

Description An object structure allocated by the function MtXMLNextObjectRep must be freed using this function when the object is not needed any more.

NextObjectRep

Synopsis

```
#include "matisseXMLinternal.h"
MtXMLSTS MtXMLNextObjectRep
            (MtXMLStream      xmlStream,
             MtXMLObjElement* objectRep)
```

Purpose This function returns the next internal object representation in the stream (see also, `FreeObjectRep`).

Arguments

`xmlStream` `INPUT`

An XML stream.

`objectRep` `OUTPUT`

The subsequent object in the stream, or undefined if there is no subsequent element.

Result

`MTXML_SUCCESS`
`MTXML_ENDOFSTREAM`
`MTXML_INVALSTREAM`

Description The following XML document, for example, contains two objects of the class person. The first and second call of this function on the stream of the XML document return a person object named John Smith who has a friend, Brian Watts, and a person object named Tom Lehman respectively. The third call of this function returns the error status MTXML_ENDOFSTREAM since there is no subsequent element.

```
<?xml version="1.0"?>
<person>
   <name>John Smith</name>
   <person MtRelationship="friend">
      <name>Brian Watts</name>
   </person>
</person>
<person>
   <name>Tom Lehman</name>
</person>
```

The content of the object representation is allocated by the Matisse XML. When you do not need the object any more, you need to free the object using the function MtXMLFreeObjectRep.

OpenInputStream

Synopsis

```
#include "matisseXMLinternal.h"
MtXMLSTS MtXMLOpenInputFileStream
           (MtXMLStream* xmlStream,
            FILE*        file)
MtXMLSTS MtXMLOpenInputStringStream
           (MtXMLStream* xmlStream,
            MtString     string)
```

Purpose

These functions open an XML stream, xmlStream, on a file or a string. The function MtXMLNextObjectRep uses the stream to provide the user with the internal representation of objects that are created by parsing an XML document.

Arguments

```
xmlStream       OUTPUT
```

The stream of internal object representation.

```
file            INPUT
```

A file containing an XML document.

```
string          INPUT
```

A string containing an XML document.

Result

```
MTXML_SUCCESS
MTXML_NULLPOINTER
```

5.3 Classes and Types

The type MtXMLStream represents a stream used to manipulate objects. The object representation returned by the function MtXMLNextObjectRep is constructed using the following file classes:

```
class MtXMLElement
class MtXMLAttElement
class MtXMLRelElement
class MtXMLObjElement
class MtXMLAttribute
```

MtXMLElement

Synopsis

```
#matisseXMLinternal.h
class MtXMLElement
```

Description

This class is a pure abstract base class for the other three classes, MtXMLAttElement, MtXMLRelElement, and MtXMLObjElement.

Members

```
MtString tagName
```

Name of the XML element's start-tag.

```
MtSize numXmlAttributes
```

Number of XML attributes in the XML element.

```
        MtXMLAttribute** xmlAttributes
Array of XML attributes
```

MtXMLObjElement

Synopsis

```
#matisseXMLinternal.h
class MtXMLObjElement : public MtXMLElement
```

Description

This object represents an object holding attributes and relationships. The tagName of the object indicates its class name.

Example

To enumerate all objects in an XML file:

```
    MtXMLObjElement* oRep;
    MtXMLStream      stream;
    MtXMLSTS         xsts;
    FILE*            file;
// A file is opened and assigned to 'file'.
    // CHECK_XMLSTS is a macro to check the return
    // status of Matisse XML functions.
    CHECK_XMLSTS(MtXMLOpenInputFileStream(&stream, file));
oRep = new MtXMLObjElement;
    for(xsts = MtXMLNextObjectRep(stream, oRep);
        MtXMLSuccess(xsts);
        xsts = MtXMLNextObjectRep(stream, oRep)){
// Do something on 'oRep'
CHECK_XMLSTS(MtXMLFreeObjectRep(oRep));
        oRep = new MtXMLObjElement;
    }
    CHECK_XMLSTS(MtXMLFreeObjectRep(oRep));
    if(xsts != MTXML_ENDOFSTREAM){
```

```
        // If the last error status is not
           // MTXML_ENDOFSTREAM, you need to check
        // this error status.
        CHECK_XMLSTS(xsts);
    }
```

Members

```
MtSize numAttributes
```

Number of attributes that the object has.

```
MtXMLAttElement** attributes
```

Array of attributes.

```
MtSize numRelationships
```

Number of relationships that the object has.

```
MtXMLRelElement** relationships
```

Array of relationships.

Methods

```
MtXMLObjElement()
```

The constructor.

```
MtString GetElementValue(MtString tag)
```

This method returns a copy of the string value of the XML element named tag which can be found first. If the element has no value, it returns an empty string (""). If such an element is not found, it returns NULL.

```
MtString GetPrimaryKey()
```

This method returns a copy of the string value of the primary key element of the object. If the primary key element is found but has no value, an empty string ("") is returned. If the object has no primary key element, NULL is returned.

```
MtString GetPrimaryKeyElement()
```

This method returns a copy of the element's tag name, which is the primary key of the object. If the object has no primary key element, NULL is returned.

MtXMLAttElement

Synopsis

```
#matisseXMLinternal.h
class MtXMLAttElement : public MtXMLElement
```

Description

This object represents a Matisse attribute holding a value. The tagName of this object indicates a Matisse attribute name.

Members

```
MtString value
```

Value of the Matisse attribute.

```
MtBoolean isPrimaryKey
```

This member indicates whether the value of the object is considered as a primary key to specify an object in a database.

Refer to section 2.4 for more information about the PrimaryKey.

MtXMLRelElement

Synopsis

```
#matisseXMLinternal.h
class MtXMLRelElement : public MtXMLElement
```

Description

This object represents a Matisse relationship holding its relationship name and a successor object. The tagName of the object indicates the class name of its successor object.

Members

```
MtString relationshipName
```

Name of the Matisse relationship.

```
MtXMLObjElement* successor
```

A successor object.

MtXMLAttribute

Synopsis

```
#matisseXMLinternal.h
class MtXMLAttribute
```

Description

This object represents an XML attribute that can be held by instances of the classes MtXMLAttElement, MtXMLRelElement, or MtXMLObjElement.

Members

```
MtString name
```

Name of the XML attribute.

```
MtString value
```

Value of the XML attribute.

5.4 Errors

MTXML_ENDOFSTREAM

Description

```
End of stream - all values enumerated
```

This error, which can occur when there is a stream enumeration (function MtXMLNextObjectRep), indicates that the enumeration is over: all the elements of the stream have been visited.

Solution Close the stream.

MTXML_INVALSTREAM

Description `Stream is not a valid stream`

This error occurs when calling one of the following functions if the stream specified as an argument does not correspond to a valid opened stream (the stream may have been already closed):

```
MtXMLNextObjectRep
MtXMLCloseInputStream
```

Index

Symbols

$MATISSE_HOME/bin 8
$MATISSE_HOME/include 30, 38
$MATISSE_HOME/lib/libmatisseXML.so 30, 38

A

API for internal objects 38
  environment 38
API for MATISSE XML C
  memory allocation 32
  MtXML prefix 30
  MtXML_M prefix 30
attribute types
  Boolean 12
  date 12
  integer 10
  interval 13
  MT_CHAR 12
  MT_STRING 11
  real number 11
  timestamp 12

D

database schema 9
document type definition (DTD) 9

E

exporting XML documents 27, 30–32

I

importing XML documents 9, 33–35

L

lists
  Boolean 24
  numerical 24
  other 25
  string 24

M

matisseXML.h 30
matisseXMLinternal.h 38
MT_NULL 25
mt_xml utility
  -commit <n> option 25, 26
  definition of 7
  importing multiple objects 13
    processing instruction 13
  standard input 8
  standard output 8
  status messages 8
  status report 8
  using 8
mt_xml_replication utility
  location 8
MtAction 26
MtPrimaryKey 17–20, 26
MtRelationship 16, 26
MtXML_ExportNumObjects 31
MtXML_ExportObjects 31
MtXML_MExportNumObjects 31
MtXML_MExportObjects 31
MtXMLMFree 32
MtXMLStream 40
MtXMLSTS 36
MtXMLSuccess 35

N

null pointer 37

O

objects
  updating attributes of 18, 21

updating relationships of 19, 22, 23, 24
ODL document
example 2.1 9
example 2.2 14, 21
OIDs 27

P

pkUpdate 35

R

relationship successor operations
append 20
appendIfNotExist 20
forceAppend 20
remove 20
removeIfExist 20
replace 20
relationships, updating 14

S

-sql 27
SQL statement to specify objects to be exported 27
storing in a database 9

U

-update 18, 21

X

XML 30
XML C Programming API 30
XML document
example 2.1 9
example 2.2 13
example 2.3 15
example 2.4 16
example 2.5 17
exporting 27, 30–32
by OID 27
by SQL 27
importing 9, 33–35
current limitations 26
example 9
parsing 33
reading and storing 33
internal object representation 38
parsing 40
XML document file
exporting from database 8
importing to database 8