

# **Universal Metadata Framework**

## **Software Architecture Document – version 0.2**

## Revision History

Revision	Description	Date	Author
0.1	Initial revision	13 <sup>th</sup> of November, 2013	Andrew Senin, Ivan Gubochkin

## Contents

Revision History.....	2
Contents .....	3
1. Introduction .....	5
Purpose .....	5
Scope .....	5
Definitions, Acronyms and Abbreviations.....	5
References.....	6
Overview .....	6
2. Architectural representation.....	7
3. Architectural Goals and Constraints.....	9
Technical Platform .....	9
Persistence .....	9
Performance.....	9
Internationalization.....	9
4. Logical View.....	10
5. Process View .....	11
6. Deployment view .....	12
7. Implementation view .....	13
Overview .....	13
Functionality Levels.....	13
UMF class diagrams.....	14
Metadata layer classes .....	14
Data source layer classes .....	14

Exception classes .....	14
Loading and saving objects .....	15
8. Data View .....	16
Basic storage principle .....	16
XMP headers .....	17
Next metadata id storage .....	17
Metadata storage.....	17
Schemas storage .....	19
Examples .....	21
Storing of structures.....	21
Storing of arrays .....	24
Storing of schemas .....	26
Storing of references .....	27
9. Quality .....	31

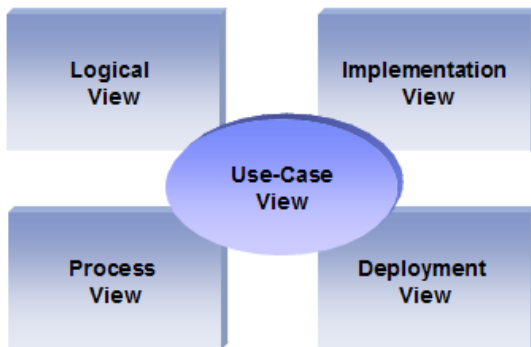
## 1. Introduction

Universal Metadata Framework (UMF) provides functionality for creating, editing and embedding of metadata into video files. The library lies on top of Adobe XMP Toolkit SDK and uses XMP format internally.

This document describes a high-level architecture of Universal Metadata Framework.

### Purpose

- The Software Architecture Document (SAD) provides a comprehensive architectural overview of the Universal Metadata Framework. It presents a number of different architectural views to depict different aspects of the framework. It is intended to capture and convey the significant architectural decisions which have been made on the framework.
- In order to depict the software as accurately as possible, the structure of this document is based on the “4+1” model view of architecture [KRU41] but without Use-Case view.



### Scope

The scope of this SAD is to depict the architecture of the Universal Metadata Framework.

### Definitions, Acronyms and Abbreviations

- **SAD:** Software Architecture Document
- **UMF:** Universal Metadata Framework

## References

- [KRU41]: The “4+1” view model of software architecture, Philippe Kruchten, November 1995, <http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/Pbk4p1.pdf>

## Overview

In order to fully document all the aspects of the architecture, the Software Architecture Document contains the following subsections.

Section 2: describes the use of each view

Section 3: describes the architectural constraints of the framework

Section 4: describes the functional requirements with a significant impact on the architecture

Section 5: describes design’s concurrency aspects

Section 6: describes how the framework will be deployed.

Section 7: describes the layers and subsystems of the framework

Section 8: describes any significant persistent element. Will contain the Data Model

Section 9: describes any performance issues and constraints

## 2. Architectural representation

This document details the architecture using the views defined in the “4+1” model [KRU41]. The views used to document the UMF are:

### Logical view

**Audience:** Designers.

**Area:** Functional Requirements: describes the design's object model. Also describes the most important use-case realizations.

**Related Artifacts:** Design model

### Process view

**Audience:** Integrators.

**Area:** Non-functional requirements: describes the design's concurrency and synchronization aspects.

**Related Artifacts:** (no specific artifact).

### Implementation view

**Audience:** Programmers.

**Area:** Software components: describes the layers and subsystems of the framework.

**Related Artifacts:** Implementation model, components

### Deployment view

**Audience:** Deployment managers.

**Area:** Topology: describes the mapping of the software onto the hardware and shows the distributed aspects.

**Related Artifacts:** Deployment model.

## **Data view**

**Audience:** Data specialists

**Area:** Persistence: describes the architecturally significant persistent elements in the data model.

**Related Artifacts:** Data model.



### **3. Architectural Goals and Constraints**

This section describes the software requirements and objectives that have some significant impact on the architecture

#### **Technical Platform**

The Universal Metadata Framework is a cross platform library. Windows (ver. 7, 8 on x86 and x64 platforms), Ubuntu Linux 12.04 x64, Android (armembi, armembi-v7a, x86) environments are supported.

#### **Persistence**

Metadata persistence will be addressed using a schema approach.

#### **Performance**

Estimations are not available.

#### **Internationalization**

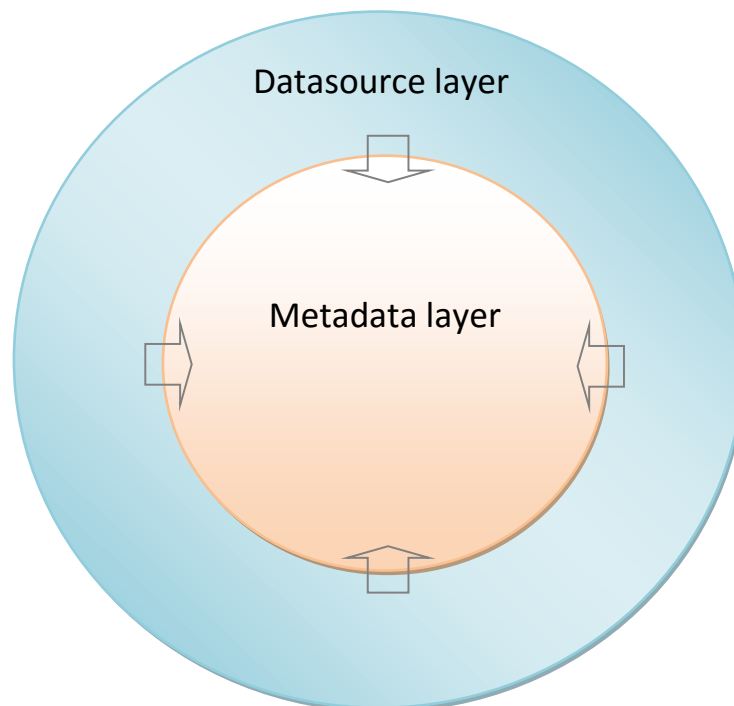
The UMF supports UTF-8 encoding to be able to deal with several languages.

#### 4. Logical View

The Universal Metadata Framework consists of 2 layers:

- **Metadata layer** – this is the layer which implements core functionality of the project such as adding, editing, deleting metadata, making relations between metadata and querying metadata.
- **Datasource layer** – this layer implements functionality for embedding metadata to media files and for serialization/deserialization of UMF structures into XMP format.

The following picture shows main abstract layers of the Universal Metadata Framework.



## 5. Process View

The Universal Metadata Framework is not thread safe. Therefore, user should implement thread safe way for UMF API calls in multithreaded programs.

## 6. Deployment view

On Windows platform the UMF can be deployed via Installation package. After install user can manually link UMF libraries to their own projects.

On Android platform the UMF is distributed and installed with user application packages.

## 7. Implementation view

### Overview

- The Implementation view depicts the physical composition of the implementation in terms of Implementation Subsystems, and Implementation Elements (directories and files, including source code, data, and executable files).

### Functionality Levels

- It is unnecessary to document the Implementation view in great details in this document. For further information, refer to the Online Catering Service 1.0 workspace in Rational Software Architect.

The metadata layer library has two functionality levels:

- User API level. This level provides user interfaces which are easy to use. Most user functions are implemented with a single line which calls for a corresponding function on the implementation level. The user level hides some complexities of the next level. For instance, user can create two instances of the MetadataStream class on the user level which use the same file. But internally both instances will use a single object from the implementation level.
- Implementation level. This level provides extended functionality which is not intended to be used by users but by other classes of the library.

The data source layer implements serialization, deserialization and embedding of metadata with using the XMP library.

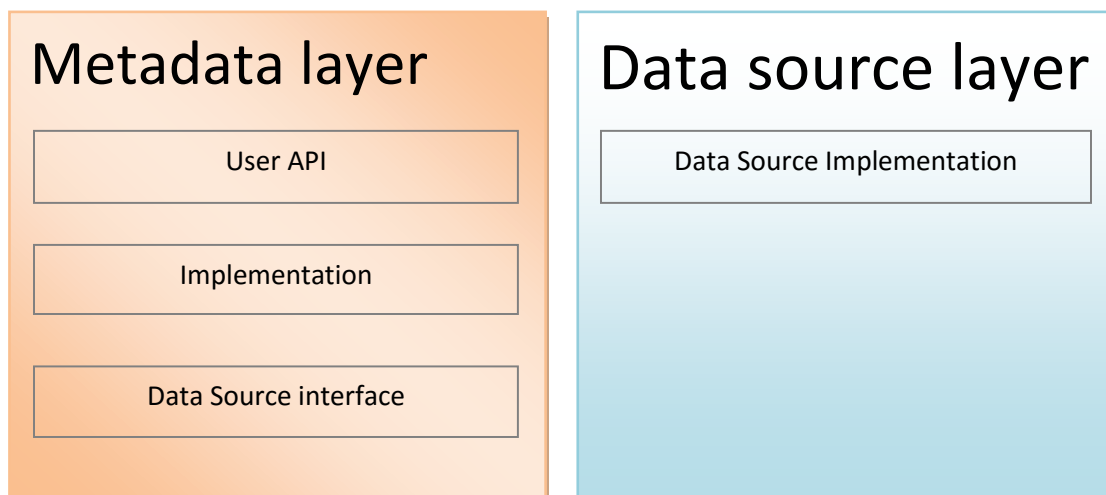
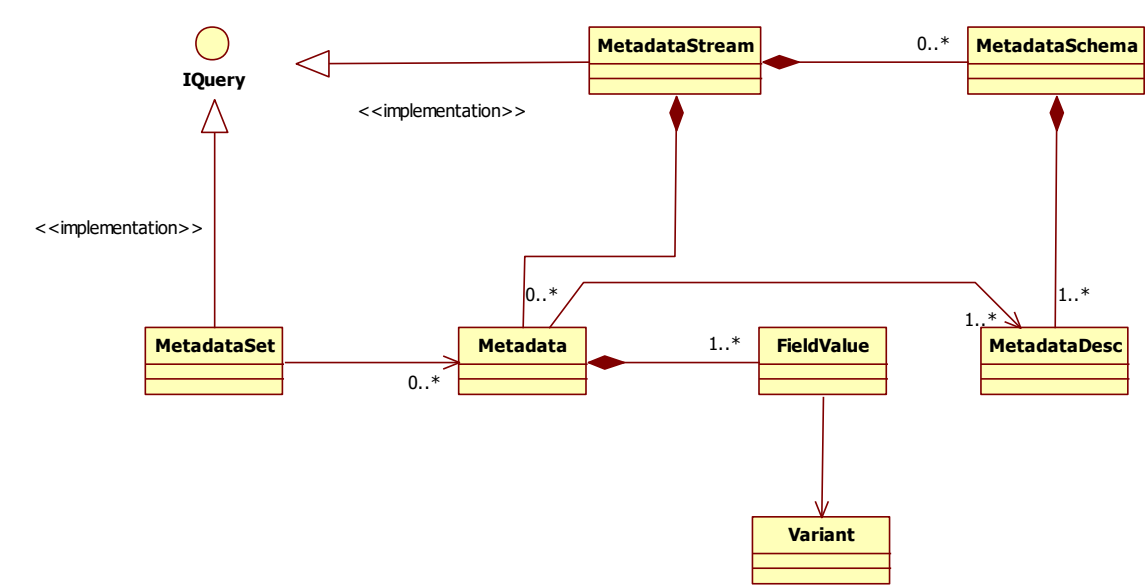


Fig. 2. UMF Layers

UMF class diagrams

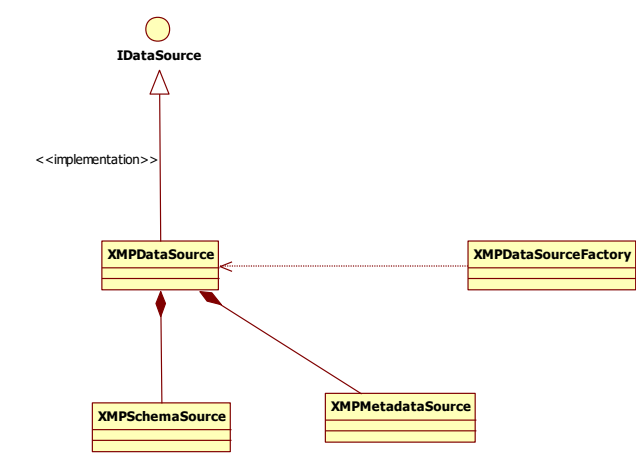
Metadata layer classes

The figure below shows core classes of UMF.



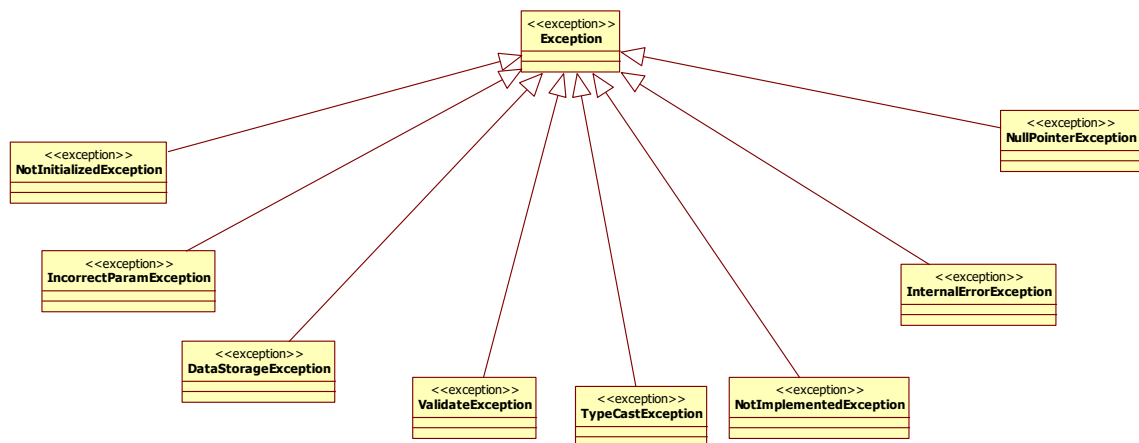
Data source layer classes

The figure below shows classes of data source layer



Exception classes

Exception class hierarchy of UMF



## Loading and saving objects

The framework implements loading in two stages. At the beginning user should create `MetadataStream` object and open specified file. After that, he can load metadata from specified metadata schema or description.

The save operation is performed in one step. On user level there is a function `MetadataStream::save`. Once it is called the call is redirected to `DataSource::save` method. The `DataSource::save` method can take as input parameter a reference to a `MetadataStream` object.

`DataSource` is responsible for querying of all nested objects. `DataSource` shall maintain a list of all objects it loads from files. This list is necessary for automatic detection of edited objects. Without such list each object will have to maintain a list of cached objects (the objects which were not loaded from the file).

The compression of data is provided by `DataSource` at saving stage. To enable it, the user should pass the string ID of the compression algorithm to the `MetadataStream::save` method. Then this ID is used by `DataSource` to create an instance of `Compressor` class that will compress all the data. The compressed binary data is then saved in base64 encoding in a `compressed_data` property. The ID of compression algorithm is saved in a `compression_algo` property.

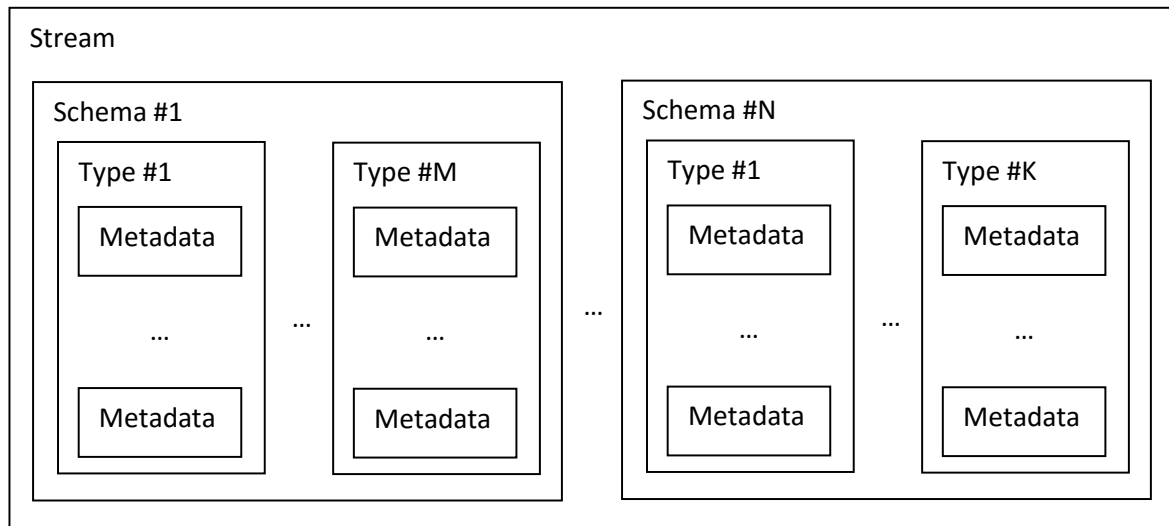
Loading of the compressed data is performed transparently for user.

It's possible to use built-in compression algorithms and the ones provided by user. Note that before using user-provided algorithms for both loading and saving they should be registered. See the details of the compression feature in user guide and in specification.

## 8. Data View

### Basic storage principle

All metadata embedded into a media file form a metastream. Each metastream can contain multiple metadata schemas. Each schema can contain metadata of different types. Each metadata type can contain multiple metadata items. This is illustrated on a picture below.



So it is natural to store metadata as a tree structure which is natively supported by XMP:

- XMP headers
  - Next free metadata id (umf:next-id tag)
  - Metadata storage
    - XMP array of schemas (umf:metadata tag)
      - XMP array of metadata names (umf:set tag)
        - XMP array of Metadata items (umf:set tag)
          - XMP array of Field/Values (umf:fields tag)
          - XMP array of references (umf:refs tag)
  - Schemas description storage
    - XMP arrays of schemas (umf:schemas tag)
      - XMP array of schema names (umf:descriptors tag)



- XMP array of fields/types (umf:fields tag)

Let's review all listed parts of XMP file in details in the remaining sections.

## XMP headers

All XMP metadata is stored inside of the following tags:

```
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 5.4.0">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
      xmlns:umf="http://ns.intel.com/umf/2.0">
    </rdf:Description>
  </rdf:RDF>
</x:xmpmeta>
```

Here rdf:Description tag contains a reference to the XMP namespace used by all UMF files:  
<http://ns.intel.com/umf/2.0>.

## Next metadata id storage

All metadata items inside of a single metafile have unique ids. A next id is generated by incrementing the last used id by 1. In order to optimize I/O operations there is a reserved field in the UMF XMP format which stores the next free id index (umf:next-id tag):

```
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 5.4.0">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
      xmlns:umf="http://ns.intel.com/umf/2.0">
      <umf:next-id>2</umf:next-id>
    </rdf:Description>
  </rdf:RDF>
</x:xmpmeta>
```

## Metadata storage

The metadata items are stored as a nested structure of schemas/names arrays. In the example below we have 4 metadata items stored inside of 2 schemas, each with 1 name:

```
<umf:metadata>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <umf:schema>Schema #1</umf:schema>
      <umf:set>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
```

```

    <umf:name>Name #1</umf:name>
  </umf:set>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <!-- Metadata item #1 -->
    </rdf:li>
    <rdf:li rdf:parseType="Resource">
      <!-- Metadata item #2 -->
    </rdf:li>
  </rdf:Bag>
</umf:set>
</rdf:li>
</rdf:Bag>
</umf:set>
</rdf:li>
<rdf:li rdf:parseType="Resource">
  <umf:schema>Schema #2</umf:schema>
  <umf:set>
    <rdf:Bag>
      <rdf:li rdf:parseType="Resource">
        <umf:name>Name #2</umf:name>
        <umf:set>
          <rdf:Bag>
            <rdf:li rdf:parseType="Resource">
              <!-- Metadata item #3 -->
            </rdf:li>
            <rdf:li rdf:parseType="Resource">
              <!-- Metadata item #4 -->
            </rdf:li>
          </rdf:Bag>
        </umf:set>
      </rdf:li>
      <rdf:li>
    </rdf:Bag>
  </umf:set>
</rdf:li>
</rdf:Bag>
</umf:metadata>

```

The metadata item structure consists of the following tags:

- umf:id – unique id of this metadata item
- umf:index – index of the first frame associated with this metadata item (-1 in case of global metadata)
- umf:nframes – number of sequential frames the metadata item is associated with (0 in case of global metadata)
- umf:fields – fields of metadata item as a list of value/name pairs (in case it is a structure) or an array of values (in case it is an array) or just a single value

- umf:refs – array of references to other metadata items. Each reference is stored as metadata id

Here is an example of metadata item. Here we have a global metadata item with id 0. The item contains a list of name/value pairs which describe 21 age old John Does. The item has 2 references to items with id #1 and id #2:

```
<rdf:li rdf:parseType="Resource">
  <umf:id>0</umf:id>
  <umf:index>-1</umf:index>
  <umf:nframes>0</umf:nframes>
  <umf:fields>
    <rdf:Bag>
      <rdf:li rdf:parseType="Resource">
        <rdf:value>John</rdf:value>
        <umf:name>name</umf:name>
      </rdf:li>
      <rdf:li rdf:parseType="Resource">
        <rdf:value>Doe</rdf:value>
        <umf:name>last name</umf:name>
      </rdf:li>
      <rdf:li rdf:parseType="Resource">
        <rdf:value>21</rdf:value>
        <umf:name>age</umf:name>
      </rdf:li>
    </rdf:Bag>
  </umf:fields>
  <umf:refs>
    <rdf:Bag>
      <rdf:li>1</rdf:li>
      <rdf:li>2</rdf:li>
    </rdf:Bag>
  </umf:refs>
</rdf:li>
```

Here is an example of an array item. Please note that now the rdf:li tags of the array items do not have the “rdf:parseType” attribute:

```
<rdf:li rdf:parseType="Resource">
  <umf:id>100</umf:id>
  <umf:index>-1</umf:index>
  <umf:nframes>0</umf:nframes>
  <umf:fields>
    <rdf:Bag>
      <rdf:li>1</rdf:li>
      <rdf:li>2</rdf:li>
      <rdf:li>3</rdf:li>
    </rdf:Bag>
  </umf:fields>
</rdf:li>
```

## Schemas storage

Metadata container consists of two parts: metadata storage and schema storage. Metadata storage does not contain any information about schemas. Each metadata object only has a reference to a specific schema from the schemas storage and metadata description (see the following fragment of xml).

```
<umf:metadata>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <!-- Schema name from the schema storage -->
      <umf:schema>Schema #1</umf:schema>
      <umf:set>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <!-- Name of metadata descriptor from specified schema -->
            <umf:name>numbers</umf:name>
            <umf:set>
              <rdf:Bag>
```

Schema storage is an array of schemas descriptions. It starts from the umf:schemas tag and looks as follows:

```
<umf:schemas>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <!-- Name of schema -->
      <umf:schema>Schema #1</umf:schema>
      <umf:descriptors>
        <!-- Description of schema #1 -->
        </umf:descriptors>
      </rdf:li>
    <rdf:li rdf:parseType="Resource">
      <!-- This is another schema "my schema" -->
      <umf:schema>Schema #2</umf:schema>
      <umf:descriptors>
        <!-- Description of schema #2 -->
        </umf:descriptors>
      </rdf:li>
    </rdf:Bag>
  </umf:schemas>
```

Each Metadata name description consists of the following tags:

- umf:name – name of metadata
- umf:fields – description of possible metadata fields

The `umf::fields` tag contains a list of all possible name/types pairs of metadata members as in the example below where we have 3 possible members:

- “name” of type string
- “last name” of type string
- “age” of type integer

```
<rdf:Bag>
  <rdf:li rdf:parseType="Resource">
    <!-- The new description "people" is a structure that contains info about some person -->
    <umf:name>people</umf:name>
    <umf:fields>
      <rdf:Bag>
        <!-- Each item of structure represents as "name"/"type" pair -->
        <rdf:li rdf:parseType="Resource">
          <umf:name>name</umf:name>
          <umf:type>string</umf:type>
        </rdf:li>
        <rdf:li rdf:parseType="Resource">
          <umf:name>last name</umf:name>
          <umf:type>string</umf:type>
        </rdf:li>
        <rdf:li rdf:parseType="Resource">
          <umf:name>age</umf:name>
          <umf:type>integer</umf:type>
        </rdf:li>
      </rdf:Bag>
    </umf:fields>
  </rdf:li>
</rdf:Bag>
```

In case of arrays there is only 1 field is allowed and its name is empty:

```
<rdf:Bag>
  <rdf:li rdf:parseType="Resource">
    <!-- This schema have only one descriptor "numbers" and this is an array of integers -->
    <umf:name>numbers</umf:name>
    <umf:fields>
      <rdf:Bag>
        <rdf:li rdf:parseType="Resource">
          <!-- Name tag is empty for arrays -->
          <umf:name/>
          <umf:type>integer</umf:type>
        </rdf:li>
      </rdf:Bag>
    </umf:fields>
  </rdf:li>
</rdf:Bag>
```

## Examples

### Storing of structures

Here is an example of XMP metadata storage where we have an array of structures described by specified schemas (umf:schema tag). Inside of the schema there is an array of metadata structures that represent data from Metadata class. So each metadata has a name (<umf:name>people</umf:name> for example) and zero or more items. An item has unique numeric identifier (umf:id) and list of fields. Each field describes name (<umf:name>name</umf:name> for example) and value.

A sample code uses of video metadata framework to store some metadata structures to media file.

```
#define TEST_FILE "C:/temp/umf/me1.avi"

//...

umf::initialize();

std::shared_ptr<umf::MetadataSchema> schema(new umf::MetadataSchema("my schema"));
std::vector<umf::FieldDesc> fields;
fields.push_back(umf::FieldDesc("name", umf::Variant::type_string));
fields.push_back(umf::FieldDesc("last name", umf::Variant::type_string));
fields.push_back(umf::FieldDesc("age", umf::Variant::type_integer));
std::shared_ptr<umf::MetadataDesc> desc(new umf::MetadataDesc("people", fields));
schema->add(desc);

umf::MetadataStream stream;
stream.open(INTEGRATION_TEST_FILE, umf::MetadataStream::ReadWrite);
stream.addSchema(schema);
std::shared_ptr<umf::Metadata> john(new umf::Metadata(desc)), richard(new umf::Metadata(desc));
john->setFieldValue("name", "John");
john->setFieldValue("last name", "Doe");
john->setFieldValue("age", (umf::umf_integer) 21);

richard->setFieldValue("name", "Richard");
richard->setFieldValue("last name", "Roe");
richard->setFieldValue("age", (umf::umf_integer) 18);

stream.add(john);
stream.add(richard);
stream.save();
stream.close();

umf::terminate();
```

This code generates an XMP format xml code like you can see below:

```
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 5.4.0">
```

```

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about=""
    xmlns:umf="http://ns.intel.com/umf/2.0">
    <umf:metadata>
      <rdf:Bag>
        <rdf:li rdf:parseType="Resource">
          <umf:schema>my schema</umf:schema>
          <umf:set>
            <rdf:Bag>
              <rdf:li rdf:parseType="Resource">
                <umf:name>people</umf:name>
                <umf:set>
                  <rdf:Bag>
                    <rdf:li rdf:parseType="Resource">
                      <umf:id>0</umf:id>
                      <umf:index>-1</umf:index>
                      <umf:nframes>0</umf:nframes>
                      <umf:fields>
                        <rdf:Bag>
                          <rdf:li rdf:parseType="Resource">
                            <rdf:value>John</rdf:value>
                            <umf:name>name</umf:name>
                          </rdf:li>
                          <rdf:li rdf:parseType="Resource">
                            <rdf:value>Doe</rdf:value>
                            <umf:name>last name</umf:name>
                          </rdf:li>
                          <rdf:li rdf:parseType="Resource">
                            <rdf:value>21</rdf:value>
                            <umf:name>age</umf:name>
                          </rdf:li>
                        </rdf:Bag>
                      </umf:fields>
                    </rdf:li>
                  <rdf:li rdf:parseType="Resource">
                    <umf:id>1</umf:id>
                    <umf:index>-1</umf:index>
                    <umf:nframes>0</umf:nframes>
                    <umf:fields>
                      <rdf:Bag>
                        <rdf:li rdf:parseType="Resource">
                          <rdf:value>Richard</rdf:value>
                          <umf:name>name</umf:name>
                        </rdf:li>
                        <rdf:li rdf:parseType="Resource">
                          <rdf:value>Roe</rdf:value>
                          <umf:name>last name</umf:name>
                        </rdf:li>
                        <rdf:li rdf:parseType="Resource">
                          <rdf:value>18</rdf:value>
                          <umf:name>age</umf:name>
                        </rdf:li>
                      </rdf:Bag>
                    </umf:fields>
                  </rdf:li>
                </rdf:Bag>
              </umf:set>
            </rdf:Bag>
          </umf:set>
        </rdf:li>
      </rdf:Bag>
    </umf:metadata>
  </rdf:Description>
</rdf:RDF>

```

```

        </rdf:li>
      </rdf:Bag>
    </umf:set>
  </rdf:li>
</rdf:Bag>
</umf:set>
</rdf:li>
</rdf:Bag>
</umf:metadata>
<umf:next-id>2</umf:next-id>
<umf:schemas>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <umf:schema>my schema</umf:schema>
      <umf:descriptors>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:name>people</umf:name>
            <umf:fields>
              <rdf:Bag>
                <rdf:li rdf:parseType="Resource">
                  <umf:name>name</umf:name>
                  <umf:type>string</umf:type>
                </rdf:li>
                <rdf:li rdf:parseType="Resource">
                  <umf:name>last name</umf:name>
                  <umf:type>string</umf:type>
                </rdf:li>
                <rdf:li rdf:parseType="Resource">
                  <umf:name>age</umf:name>
                  <umf:type>integer</umf:type>
                </rdf:li>
              </rdf:Bag>
            </umf:fields>
          </rdf:li>
        </rdf:Bag>
      </umf:descriptors>
    </rdf:li>
  </rdf:Bag>
</umf:schemas>
</rdf:Description>
</rdf:RDF>
</x:xmpmeta>

```

## Storing of arrays

An array is stored in a similar way as a structure. But instead description of fields are used an XMP list (<rdf:li> tag) of items. The code below creates and saves an array of integers into media file.

```

#define TEST_FILE "C:/temp/umf/video.avi"
#define TEST_SCHEMA_NAME "TEST_SCHEMA_NAME"
//...

```



```

umf::initialize();

umf::MetadataStream stream;
std::shared_ptr<umf::MetadataSchema> spSchema(new umf::MetadataSchema(TEST_SCHEMA_NAME));
stream.open(TEST_FILE, umf::MetadataStream::ReadWrite);
std::shared_ptr< umf::MetadataDesc > spNumbersDesc ( new umf::MetadataDesc( "numbers",
umf::Variant::type_integer ));
spSchema->add( spNumbersDesc );
std::shared_ptr< umf::Metadata > spNumbers( new umf::Metadata( spNumbersDesc ));
spNumbers->addValue( (umf::umf_integer) 1);
spNumbers->addValue( (umf::umf_integer) 2);
spNumbers->addValue( (umf::umf_integer) 3);
spNumbers->addValue( (umf::umf_integer) 4);
spNumbers->addValue( (umf::umf_integer) 5);

stream.addSchema(spSchema);
stream.add( spNumbers );
stream.save();
stream.close();

umf::terminate();

```

The typical xml markup of stored array looks like code below.

```

<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 5.4.0">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
      xmlns:umf="http://ns.intel.com/umf/2.0">
      <umf:metadata>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:schema>TEST_SCHEMA_NAME</umf:schema>
            <umf:set>
              <rdf:Bag>
                <rdf:li rdf:parseType="Resource">
                  <umf:name>numbers</umf:name>
                  <umf:set>
                    <rdf:Bag>
                      <rdf:li rdf:parseType="Resource">
                        <umf:id>0</umf:id>
                        <umf:index>-1</umf:index>
                        <umf:nframes>0</umf:nframes>
                        <umf:fields>
                          <rdf:Bag>
                            <rdf:li>1</rdf:li>
                            <rdf:li>2</rdf:li>
                            <rdf:li>3</rdf:li>
                            <rdf:li>4</rdf:li>
                            <rdf:li>5</rdf:li>
                          </rdf:Bag>
                        </umf:fields>
                      </rdf:li>
                    </rdf:Bag>
                  </umf:set>
                </rdf:li>
              </rdf:Bag>
            </umf:set>
          </rdf:li>
        </rdf:Bag>
      </umf:metadata>
    </rdf:Description>
  </rdf:RDF>
</x:xmpmeta>

```

```

        </rdf:li>
      </rdf:Bag>
    </umf:set>
  </rdf:li>
</rdf:Bag>
</umf:metadata>
<umf:next-id>1</umf:next-id>
<umf:schemas>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <umf:schema>TEST_SCHEMA_NAME</umf:schema>
      <umf:descriptors>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:name>numbers</umf:name>
            <umf:fields>
              <rdf:Bag>
                <rdf:li rdf:parseType="Resource">
                  <umf:name/>
                  <umf:type>integer</umf:type>
                </rdf:li>
              </rdf:Bag>
            </umf:fields>
          </rdf:li>
        </rdf:Bag>
      </umf:descriptors>
    </rdf:li>
  </rdf:Bag>
</umf:schemas>
</rdf:Description>
</rdf:RDF>
</x:xmpmeta>

```

## Storing of schemas

Here is an example of XMP metadata storage where we have an array of schemas (umf:schemas) with two schemas ("my schema" and "TEST\_SCHEMA\_NAME"). Inside of the schema there is an array of metadata descriptors (umf::descriptors). Each descriptor has a name (umf:name) and stores data from MetadataDesc class. Inside of the descriptor there is an array of field items (umf:fields). Each field consists from name and type. The code below creates and saves describes schemas.

```

#define TEST_FILE "C:/temp/umf/video.avi"
#define TEST_SCHEMA_NAME "TEST_SCHEMA_NAME"
//...
umf::initialize();

umf::MetadataStream stream;
std::shared_ptr<umf::MetadataSchema> schema(new umf::MetadataSchema("my schema"));
std::vector<umf::FieldDesc> fields;
fields.push_back(umf::FieldDesc("name", umf::Variant::type_string));

```

```

fields.push_back(umf::FieldDesc("last name", umf::Variant::type_string));
fields.push_back(umf::FieldDesc("age", umf::Variant::type_integer));
std::shared_ptr<umf::MetadataDesc> desc(new umf::MetadataDesc("people", fields));
schema->add(desc);

std::shared_ptr<umf::MetadataSchema> spSchema(new umf::MetadataSchema(TEST_SCHEMA_NAME));
stream.open(TEST_FILE, umf::MetadataStream::ReadWrite);
std::shared_ptr< umf::MetadataDesc > spNumbersDesc ( new umf::MetadataDesc( "numbers",
umf::Variant::type_integer ));
spSchema->add( spNumbersDesc );
std::shared_ptr< umf::Metadata > spNumbers( new umf::Metadata( spNumbersDesc ));
spNumbers->addValue( (umf::umf_integer) 1);
spNumbers->addValue( (umf::umf_integer) 2);
spNumbers->addValue( (umf::umf_integer) 3);
spNumbers->addValue( (umf::umf_integer) 4);
spNumbers->addValue( (umf::umf_integer) 5);

stream.addSchema(spSchema);
stream.addSchema(schema);
stream.add( spNumbers );
stream.save();
stream.close();

umf::terminate();

```

There is a sample xml code to illustrate how schema is stored into XMP metadata.

```

<umf:schemas>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <umf:schema>TEST_SCHEMA_NAME</umf:schema>
      <umf:descriptors>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:name>numbers</umf:name>
            <umf:fields>
              <rdf:Bag>
                <rdf:li rdf:parseType="Resource">
                  <umf:name/>
                  <umf:type>integer</umf:type>
                </rdf:li>
              </rdf:Bag>
            </umf:fields>
          </rdf:li>
        </rdf:Bag>
      </umf:descriptors>
    </rdf:li>
    <rdf:li rdf:parseType="Resource">
      <umf:schema>my schema</umf:schema>
      <umf:descriptors>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:name>people</umf:name>

```

```

    <umf:fields>
      <rdf:Bag>
        <rdf:li rdf:parseType="Resource">
          <umf:name>name</umf:name>
          <umf:type>string</umf:type>
        </rdf:li>
        <rdf:li rdf:parseType="Resource">
          <umf:name>last name</umf:name>
          <umf:type>string</umf:type>
        </rdf:li>
        <rdf:li rdf:parseType="Resource">
          <umf:name>age</umf:name>
          <umf:type>integer</umf:type>
        </rdf:li>
      </rdf:Bag>
    </umf:fields>
  </rdf:li>
</rdf:Bag>
</umf:descriptors>
</rdf:li>
</rdf:Bag>
</umf:schemas>

```

## Storing of references

References are stored for each metadata items in additional array (**umf:refs** tag). Each item of this array contains ID of referenced metadata item. The code below demonstrates how to use UMF library how to make references between metadata items.

```

#define TEST_FILE "C:/temp/umf/me1.avi"
#define TEST_SCHEMA_NAME "TEST_SCHEMA_NAME"
#define TEST_SCHEMA_NAME_0 "TEST_SCHEMA_NAME_0"
#define TEST_SCHEMA_NAME_1 "TEST_SCHEMA_NAME_1"
#define TEST_PROPERTY_NAME_0 "TEST_PROPERTY_NAME_0"
#define TEST_PROPERTY_NAME_1 "TEST_PROPERTY_NAME_1"
#define TEST_STRING_VAL "TEST_STRING_VAL"
#define TEST_INTEGER_VAL 0xDEADBEEF

//...

umf::initialize();

std::shared_ptr<umf::MetadataSchema> schema[2];
schema[0] = std::shared_ptr<umf::MetadataSchema>(new umf::MetadataSchema(TEST_SCHEMA_NAME_0));
schema[1] = std::shared_ptr<umf::MetadataSchema>(new umf::MetadataSchema(TEST_SCHEMA_NAME_1));

std::shared_ptr<umf::MetadataDesc> desc[2];
desc[0] = std::shared_ptr<umf::MetadataDesc>(new umf::MetadataDesc(TEST_PROPERTY_NAME_0,
umf::Variant::type_string));
desc[1] = std::shared_ptr<umf::MetadataDesc>(new umf::MetadataDesc(TEST_PROPERTY_NAME_1,
umf::Variant::type_integer));

```

```

schema[0]->add(desc[0]);
schema[1]->add(desc[1]);

std::shared_ptr<umf::Metadata> metadata[2];
metadata[0] = std::shared_ptr<umf::Metadata>(new umf::Metadata(desc[0]));
metadata[1] = std::shared_ptr<umf::Metadata>(new umf::Metadata(desc[1]));
metadata[0]->addValue(TEST_STRING_VAL);
metadata[1]->addValue((umf::umf_integer)TEST_INTEGER_VAL);

umf::MetadataStream stream;
stream.open(TEST_FILE, umf::MetadataStream::ReadWrite);
stream.addSchema(schema[0]);
stream.addSchema(schema[1]);
stream.add(metadata[0]);
stream.add(metadata[1]);

metadata[0]->addReference(metadata[0]);
metadata[0]->addReference(metadata[1]);

stream.save();
stream.close();

umf::terminate();

```

There is a sample xml code to illustrate how references is stored into XMP metadata.

```

<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 5.4.0">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
      xmlns:umf="http://ns.intel.com/umf/2.0">
      <umf:metadata>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:schema>TEST_SCHEMA_NAME_0</umf:schema>
            <umf:set>
              <rdf:Bag>
                <rdf:li rdf:parseType="Resource">
                  <umf:name>TEST_PROPERTY_NAME_0</umf:name>
                  <umf:set>
                    <rdf:Bag>
                      <rdf:li rdf:parseType="Resource">
                        <umf:id>0</umf:id>
                        <umf:index>-1</umf:index>
                        <umf:nframes>0</umf:nframes>
                        <umf:fields>
                          <rdf:Bag>
                            <rdf:li>TEST_STRING_VAL</rdf:li>
                          </rdf:Bag>
                        </umf:fields>
                        <umf:refs>
                          <rdf:Bag>
                            <rdf:li>0</rdf:li>
                            <rdf:li>1</rdf:li>
                          </rdf:Bag>

```

```

        </umf:refs>
      </rdf:li>
    </rdf:Bag>
  </umf:set>
</rdf:li>
</rdf:Bag>
</umf:set>
</rdf:li>
<rdf:li rdf:parseType="Resource">
  <umf:schema>TEST_SCHEMA_NAME_1</umf:schema>
  <umf:set>
    <rdf:Bag>
      <rdf:li rdf:parseType="Resource">
        <umf:name>TEST_PROPERTY_NAME_1</umf:name>
        <umf:set>
          <rdf:Bag>
            <rdf:li rdf:parseType="Resource">
              <umf:id>1</umf:id>
              <umf:index>-1</umf:index>
              <umf:nframes>0</umf:nframes>
              <umf:fields>
                <rdf:Bag>
                  <rdf:li>3735928559</rdf:li>
                </rdf:Bag>
              </umf:fields>
            </rdf:li>
          </rdf:Bag>
        </umf:set>
      </rdf:li>
    </rdf:Bag>
  </umf:set>
</rdf:li>
</rdf:Bag>
</umf:metadata>
<umf:next-id>2</umf:next-id>
<umf:schemas>
  <rdf:Bag>
    <rdf:li rdf:parseType="Resource">
      <umf:schema>TEST_SCHEMA_NAME_0</umf:schema>
      <umf:descriptors>
        <rdf:Bag>
          <rdf:li rdf:parseType="Resource">
            <umf:name>TEST_PROPERTY_NAME_0</umf:name>
            <umf:fields>
              <rdf:Bag>
                <rdf:li rdf:parseType="Resource">
                  <umf:type>string</umf:type>
                </rdf:li>
              </rdf:Bag>
            </umf:fields>
          </rdf:li>
        </rdf:Bag>
      </umf:descriptors>
    </rdf:li>
  </rdf:Bag>
</umf:schemas>
</rdf:li>

```

```

<rdf:li rdf:parseType="Resource">
  <umf:schema>TEST_SCHEMA_NAME_1</umf:schema>
  <umf:descriptors>
    <rdf:Bag>
      <rdf:li rdf:parseType="Resource">
        <umf:name>TEST_PROPERTY_NAME_1</umf:name>
        <umf:fields>
          <rdf:Bag>
            <rdf:li rdf:parseType="Resource">
              <umf:name/>
              <umf:type>integer</umf:type>
            </rdf:li>
          </rdf:Bag>
        </umf:fields>
      </rdf:li>
    </rdf:Bag>
  </umf:descriptors>
</rdf:li>
</rdf:Bag>
</umf:schemas>
</rdf:Description>
</rdf:RDF>
</x:xmpmeta>

```

## 9. Quality

As far as the UMF is concerned, the following quality goals have been identified:

### Portability:

- **Description** : Ability to be reused in another environment
- **Solution** : The framework me be used cross platform tools to compile on the various environments