Proposal for Document Serialization Interfaces

Introduction

- We define a generalized "data operation" from which all specific data operations can be derived.
- Serialization operators can be specified as
    - A transformation to or from a `Document` to another format. (e.g. JSON, SLX, glTF)
    - An extraction of packaging to or from another format (e.g. ZIP, USD)
    - The data in the other format may reside in an arbitrary location and is not restricted to residing on local file systems.
- The interfaces for serialization should be designed to be thread-safe.
- The interfaces should also not preclude usage by request-based / asynchronous APIs, though this will be a separate future consideration.
    - It is possible to have "read" operations not return any data, but instead provide a callback or future/promise mechanism for the caller to receive the data when it becomes available. "write" operations could also use a similar mechanisms to indicate completion status.
    - We note that the existing XML serialization and deserialization is not thread-safe and synchronous and will not be addressing this here.

Implementation Details

- All code interfaces are available in C++ and will provide bindings for other supported languages. This currently includes Javascript and Python.
- Interfaces must include an API entry point which can take as input or output a `URI`. (This matches with the current MaterialX specification).
- Not all input / output types can support string / stream output these entry points are optional. It is up to the implementation to decide what makes sense. e.g. it is possible to encode / decode binary data to / from strings for transport but could be very inefficient.
- Interfaces will support optional user options / arguments to be specified. In general these may be anything from value arguments to function pointers.
- Most command line and user interfaces define options in a key-value format. As such, it is useful to provide a reusable interface for this.
- For life-time management all interfaces will be shared pointers, which is consistent with all other classes in the code base.
- Options many or may not be reference counted.

Operation Class

- This is the base class for all operations.
- It does not define storage or access points.
- The interface class simply specifies a unique name identifier.

```
using Operation  = std::shared_ptr<Operation>;
class Operation
{
  public:
    Operation(const string& name) : _name(name) {}
```

```
    virtual ~Operation() = default;

    const string& name() const { return _name; }

  private:
    string _name;
}
```

Options Structure

- An operation may have one or more options / arguments associated with it.
- These options can be used to customize the behavior of the operation.

```
class OperationOptions
{
  public:
    OperationOptions() = 0;
    virtual ~OperationOptions() = 0;
};
```

- A common way to specify arguments is with key-value pairs.
- As such, we introduce a reusable interface for this.
- This will allow for options to be extended without modifying the original class and can be used for creating command line or interactive user interfaces.

```
using KeyValueMap = std::unordered_map<string, ValuePtr>;
class KeyValueOptions : public OperationOptions
{
  public:
    KeyValueOptions() = default;

    void setOption(const std::string& key, ValuePtr value)
    {
        _options[key] = value;
    }

    ValuePtr getOption(const std::string& key) const
    {
        auto it = _options.find(key);
        return (it != _options.end()) ? it->second : nullptr;
    }

  private:
    KeyValueMap _options;
};
```

File Paths

Note that we do not require usage of file search via the existing MaterialX search paths as they only work on local file systems on desktop and are thus not generally applicable. We do *not* assume that integrations will support frameworks such as virtual file systems, nor do we assume that all assets are statically packaged with the integration (e.g. web-pack for Javascript deployment).

Currently a file search paths option is only used for supporting "include" directives which are only supported for some formats such as XML, but not in other formats such as JSON.

Interface Class for Document Readers

For a document reader interface we add in the existing API methods provided by the current XML reader to make the transition to the new interface easier.
As noted, support for string and stream data inputs is optional.

Optional arguments are passed in via the `OperationOptions` interface.

A list of format extensions may be returned to indicate the supported output formats. There are of the form: ".".
For example for `glTF` this could be "[".gltf", ".glb"].

```
using DocumentReaderPtr = std::shared_ptr<DocumentReader>;
class DocumentReader : public Operation
{
  public:

    virtual DocumentPtr read(const FilePath& uri, OperationOptions* options =
nullptr) = 0;

    virtual DocumentPtr read(const std::string& data, OperationOptions* options =
nullptr)
    {
        return nullptr;
    }

    virtual DocumentPtr read(std::istream& stream, OperationOptions* options =
nullptr)
    {
        return nullptr;
    }

    virtual StringVec supportedExtensions() const = 0;
};
```

Interface Class for Document Writers

For a document writer, we add in the existing API methods provided by the current XML writer to make the transition to the new interface easier.
As noted, support for string and stream data outputs is optional.

Optional arguments are passed in via the `OperationOptions` interface.

A list of format extensions may be returned to indicate the supported output formats. There are of the form: ".".
For a USD writer this could be "[".usda", ".usdz", ".usd"].

```
using DocumentWriterPtr = std::shared_ptr<DocumentWriter>;
class DocumentWriter : public Operation
{
  public:
    virtual bool write(DocumentPtr, const FilePath& uri, OperationOptions* options
= nullptr) = 0;
    virtual bool write(DocumentPtr, const std::string& data, OperationOptions*
options = nullptr)
    {
        return false;
    }
    virtual void write(DocumentPtr, std::ostream& stream, OperationOptions*
options = nullptr)
    {
        return false;
    }

    virtual StringVec supportedExtensions() const = 0;
}
```

XML Reader Wrapper Class

- Due to all the global references used, the easiest way to make this work without code changes is to add interfaces in-place into `Xmlio.cpp`
- Global functions which are not currently part of the public API, can be made into local statics to decrease exposed surface area.
- Current global public functions can be made local if / when this new interface replaces the existing interface.
- For readability, error handling has been left out.

XML Reader Options

- We consolidate all of the existing options into a single class which derives from the `OperationOptions` interface.
- A example small variation is shown here to support automatic library inclusion of standard libraries.

```
class XmlDocumentReadOptions : public OperationOptions, XmlReadOptions
{
    // Include search path option from existing interface
    FileSearchPath _searchPath = FileSearchPath();
```

```
    // Extend to include standard library
    DocumentPtr _standardLibrary = nullptr;
};
```

XML Reader

```
// XML Reader
using XMLDocumentPtr = std::shared_ptr<XMLDocument>;
class XMLDocumentReader : public DocumentReader
{
  public:
    XMLDocumentReader() = default;

    DocumentPtr read(const FilePath& uri, OperationOptions* options = nullptr)
override
    {
        DocumentPtr doc = createDocument()
        if (options && options->_standardLibrary)
        {
            doc->setDataLibrary(options->_standardLibrary);
        }
        readFromXmlFile(uri,
                        options->_searchPath ? options->_searchPath :
FileSearchPath(),
                        options);
    }

    DocumentPtr read(const std::string& data, OperationOptions* options = nullptr)
override
    {
        DocumentPtr doc = createDocument()
        if (options && options->_standardLibrary)
        {
            doc->setDataLibrary(options->_standardLibrary);
        }
        readFromXmlString(doc, data, options->_searchPath ? options->_searchPath :
FileSearchPath(), options);
    }

    DocumentPtr read(std::istream& stream, OperationOptions* options = nullptr)
override
    {
        DocumentPtr doc = createDocument()
        if (options && options->_standardLibrary)
        {
            doc->setDataLibrary(options->_standardLibrary);
        }
        readFromXmlStream(doc, stream, options->_searchPath ? options->_searchPath
: FileSearchPath(), options);
    }
```

```
    StringVec supportedExtensions() const override
    {
        return _supportedExtensions;
    }

  private:
    StringVec _supportedExtensions = { ".mtlx" };
}
```

XML Writer Wrapper Class

- As with read options we add in a new interface for write options

```
class XMLDocumentWriteOptions : public OperationOptions, XmlWriteOptions
{
    // Extend to include current search path
    FileSearchPath _searchPath = FileSearchPath();
};
```

- The writer class is similar to the reader class.
- Note: There is a `prependXInclude()` global function that is unused and if kept should belong with `Document` class.

```
using XMLDocumentWriterPtr = std::shared_ptr<XMLDocumentWriter>;
class XMLDocumentWriter : public DocumentWriter
{
  public:
    XMLDocumentWriter() = default;

    bool write(const DocumentPtr doc, const FilePath& uri, OperationOptions*
options = nullptr)
    {
        writeToXmlFile(doc, uri, options ? options->_searchPath :
FileSearchPath(), options);
        return true;
    }

    bool write(const DocumentPtr doc, const std::string& data, OperationOptions*
options = nullptr)
    {
        writeToXmlString(doc, data, options ? options->_searchPath :
FileSearchPath(), options);
        return true;
    }

    bool write(const DocumentPtr doc, std::ostream& stream, OperationOptions*
options = nullptr)
    {
        writeToXmlStream(doc, stream, options ? options->_searchPath :
```

```
FileSearchPath(), options);
        return true;
    }

    StringVec supportedExtensions() const override
    {
        return _supportedExtensions;
    }

  private:
    StringVec _supportedExtensions = { ".mtlx" };
}
```

Examples

The following examples demonstrate other possible serializer extensions.

C++ HTTP Reader Class

- The following example shows extensions for both the options and reader interface classes.
- We add both MaterialX and ZIP formats as possible data formats, though only the MaterialX logic is shown.

Use Case

- This could be used to read MaterialX documents or packaged zip assets from remote material repositories such as `ambientCG`, `PolyHaven`, `GPUOpen`, `PhysicallyBased` etc.
- One issue with embedding this code into the core is that there is a need to add an explicit build time dependency -- in this case in the form of the C++ `CURL` library.
- An extension can separate this dependency and only include it as needed.

```
// C++ http loader.
#if defined(CURL_INSTALLED)
#include <curl/curl.h>
#endif

class HTTPXMLOptions : public OperationOptions
{
  public:
    HTTPXMLOptions() = default;

    void setHttpOptions(const HttpRequestOptions& options)
    {
        _timeout = options.timeout();
        _connectTimeout = options.connectTimeout();
        _headers = options.headers();
    }

    void setTimeout(int timeout)
```

```
    {
        _timeout = timeout;
    }

    void setHeaders(const StringMap& headers)
    {
        _headers = headers;
    }

    int timeout() const
    {
        return _timeout;
    }

    int connectTimeout() const
    {
        return _connectTimeout;
    }

  private:
    int _timeout = 30;
    int _connectTimeout = 10;
    StringMap _headers;
};

using HTTPXMLReaderPtr = std::shared_ptr<HTTPXMLReader>;
class HTTPXMLReader : public DocumentReader
{
  public:
    HTTPXMLReader()
    {
        curl_global_init(CURL_GLOBAL_DEFAULT);
    }

    virtual ~HTTPXMLReader()
    {
        curl_global_cleanup();
    }

    DocumentPtr read(const FilePath& uri, OperationOptions* options = nullptr)
override
    {
        HTTPXMLOptions httpOptions;
        if (options)
        {
            httpOptions.setHttpOptions(*static_cast<HttpRequestOptions*>
(options));
        }}

        std::string materialString;

        CURL* curl;
        CURLcode res;
        curl = curl_easy_init();
```

```
        if (curl)
        {
            curl_easy_setopt(curl, CURLOPT_URL, uri.c_str());
            curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
            curl_easy_setopt(curl, CURLOPT_WRITEDATA, &materialString);

            // For HTTPS
            curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
            curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L);

            // Set timeout settings for network requests
            curl_easy_setopt(curl, CURLOPT_TIMEOUT, httpOptions.timeout());
            curl_easy_setopt(curl, CURLOPT_CONNECTTIMEOUT,
httpOptions.connectTimeout());

            res = curl_easy_perform(curl);
            if (res != CURLE_OK)
                return nullptr;

            curl_easy_cleanup(curl);

            if (!materialString.empty())
            {
                XMLDocumentReader reader;
                DocumentPtr doc = reader.read(materialString);
                return doc;
            }
        }
        return nullptr;
    }

    DocumentPtr read(const std::string& data, OperationOptions* options = nullptr)
override
    {
        return nullptr;
    }

    DocumentPtr read(std::istream& stream, OperationOptions* options = nullptr)
override
    {
        return nullptr;
    }

    StringVec supportedExtensions() const override
    {
        return _supportedExtensions;
    }

  private:
    StringVec _supportedExtensions = { ".mtlx", ".zip" };
}
```

Python Zip File Writer

- This is a Python example of a custom `DocumentWriter` that saves a MaterialX document and its referenced textures into a ZIP file.
- Dependencies are part of the module / package.
- Integrations would handle any error exceptions including package imports

```
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Dependency check
have_zip = False
try:
    import zipfile
    import tempfile
    import MaterialX as mx
    logger.info("MaterialX and zip modules loaded successfully")
    have_zip = True
except ImportError:
    raise ImportError("Please ensure MaterialX and zipfile modules are
installed.")

class ZipWriter(mx.DocumentWriter):
    _plugin_name = "ZipWriter"
    _ui_name = "Save to Zip..."

    def name(self):
       return self._plugin_name

    def uiName(self):
       return self._ui_name

    def supportedExtensions(self):
        return [".zip"]

    # Override "write"
    def write(self, doc, path, options=None):
        if not have_zip:
            return None
        if not path:
            return None

        # Determine the .mtlx filename based on the .zip filename
        zip_basename = os.path.basename(path)
        zip_folder = os.path.splitext(zip_basename)[0]
        mtlx_name = zip_folder + ".mtlx"  # .mtlx at root of zip
        logger.info(f"zip base name: {zip_basename}, mtlx name: {mtlx_name}")

        # Write the document to a temporary file in the temp directory (not in a
subfolder)
```

```
        with tempfile.TemporaryDirectory() as tmpdir:
            mtlx_path = os.path.join(tmpdir, mtlx_name)

            # Determine the base directory for resolving relative texture paths
            # Use the directory of the source .mtlx file if available, else
current working dir
            mtlx_source_dir = None
            if hasattr(doc, 'getSourceUri'):
                source_uri = doc.getSourceUri()
                if source_uri and os.path.isfile(source_uri):
                    mtlx_source_dir = os.path.dirname(os.path.abspath(source_uri))
            if not mtlx_source_dir:
                mtlx_source_dir = os.getcwd()

            with zipfile.ZipFile(path, 'w') as z:

                # Save all texture files under 'textures/'
                texture_file_list = resolve_all_image_paths(doc)
                for element_path, texture in texture_file_list.items():
                    # If texture path is not absolute, resolve it relative to the
document's path
                    abs_texture = texture
                    if not os.path.isabs(texture):
                        logger.info(f"Texture path is relative: {texture},
resolving against {mtlx_source_dir}")
                        abs_texture =
os.path.normpath(os.path.join(mtlx_source_dir, texture))
                    if os.path.isfile(abs_texture):
                        arcname = os.path.join("textures",
os.path.basename(texture))
                        logger.info(f"Adding texture to ZIP: {abs_texture} as
{arcname}")
                        z.write(abs_texture, arcname=arcname)

                        # Replace the references in the materialx
                        logger.info(f"Updating texture path on element
{element_path} from {texture} to {arcname}")
                        doc.getDescendant(element_path).setValueString(arcname)
                    else:
                        logger.warning(f"Texture file not found: {abs_texture}")

                mx.writeToXmlFile(doc, mtlx_path)
                logger.info(f"Write MaterialX document to temp file: {mtlx_path}")
                # Add the .mtlx file at the root of the zip
                z.write(mtlx_path, arcname=mtlx_name)
                logger.info(f"Added MaterialX document to ZIP as: {mtlx_name}")

                logger.info(f"MaterialX document and textures saved to ZIP:
{path}")
        return True
```

Management

- A `DocumentHandler` class provides similar functionality to the existing `ImageHandler` and `GeometryHandler` classes.
- If source and destination formats are both exposed in the API then it is possible to use the `DocumentHandler` as a
  hub-and-spoke mechanism to support additional conversions by chaining the serializers.

```
class DocumentHandler;
using DocumentHandlerPtr = std::shared_ptr<DocumentHandler>;

class DocumentHandler
{
  public:
    static DocumentHandlerPtr create()
    {
        return std::make_shared<DocumentHandler>();
    }
    virtual ~DocumentHandler() = default;

    void addDocumentReader(DocumentReaderPtr reader)
    {
        documentReaders[reader->getIdentifier()] = reader;
    }

    void addDocumentWriter(DocumentWriterPtr writer)
    {
        documentWriters[writer->getIdentifier()] = writer;
    }

private:
    std::unordered_map<std::string, DocumentReaderPtr> documentReaders;
    std::unordered_map<std::string, DocumentWriterPtr> documentWriters;
};
```

Python SLX Reader Example

This example specified a reader which compiles SLX (.mxsl) files to MaterialX (.mtlx) documents.
It includes sample code showing how to add the reader to the document handler.

```
import MaterialX as mx
import logging
import os

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger('slxPlugin')

have_slx = False
try:
    from pathlib import Path
```

```
    from mxslc.Decompiler.decompile import Decompiler
    from mxslc.compile_file import compile_file
    logger.info("SLX module loaded successfully")
    have_slx = True
except ImportError:
    raise ImportError("SLX module not found. Please ensure it is installed.")

class SLXLoader(mx.DocumentReader):
    _name = "ShadingLanguageX"
    _ui_name = "Load from SLX..."

    def name(self):
        return self._name

    def uiName(self):
        return self._ui_name

    def supportedExtensions(self):
        return [".mxsl"]

    def run(self, path):
        doc = None
        mtlx_path = path.replace('.mxsl', '.mtlx')
        try:
            logger.info(f"Compiling from SLX to MaterialX: {path}...")
            compile_file(Path(path), mtlx_path)
            # Check if the MaterialX file was created
            if not os.path.exists(mtlx_path):
                logger.error("Failed to compile SLX file to MaterialX: " + path)
            else:
                doc = mx.createDocument()
                logger.info(f"Compiled SLX file to MaterialX: {mtlx_path}")
                mx.readFromXmlFile(doc, mtlx_path)
        except Exception as e:
            logger.error(f"Failed to compile SLX file: {e}")
        return doc

# ------------------------------------
# Register loader with document handler
# ------------------------------------
loader = SLXLoader()
manager = mx.getDocumentHandler()
manager.addDocumentReader(loader)
```

Language Wrappers

Python

- The existing `pybind11` mechanism is used to expose the new interface and XML serialization classes.
- "Trampoline" classes are added to allow calling of Python interfaces from C++.

- All interfaces are exposed as part of the package within the `MaterialXCore` module.
- XML serialization and deserialization will be exposed as part of package in the `MaterialXFormat` module.

Javascript

- Create wrappers using `emscripten` to expose the C++ interfaces as `WASM` implementations with `JavaScript` wrappers.
- Both based interfaces and XML wrappers would be packaged with the `JsMaterialXCore` Javascript and WASM modules which includes `MaterialXFormat` wrappers.