Proposal for Document Serialization Interfaces

Criteria

- Code interfaces are available in C++ and can having bindings provided in other languages. This currently includes Javascript and Python.
- A API which is path / URI based required.
- As not all input / output types can support string / stream output these are optional.
  It is up to the implementation to decide what makes sense. e.g. it is possible to encode / decode binaries to / from strings for transport but could be very inefficient.
- It is possible to add a generic options structure which works well for non-programmed user options. The proposal it to use string (key), `Value` pairs.
- Interfaces should be designed to be thread-safe.
- Interfaces should be compatible with patterns found in request-based APIs.
    - It would be useful to consider the possibility of asynchronous "fetch" options but this is beyond the initial scope of the proposal.
    - Note that the existing XML serialization and deserialization is not thread-safe and synchronous.

Operation Class

- We provide interface class which does not define storage or access points. It can be used for more than just document serialization. We will call this an "operation" tentatively.
- The interface class simply specifies a unique name identifier.

```
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.
- We introduce this to provide a reusable class for handling options.
- This is useful to extend options 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 include usage of the existing MaterialX search paths as they only work on local file systems on desktop and are thus not generally applicable. The same is true for writers. 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. Also this option is only used for supporting "include" directives which are available in the XML format, but not in other formats.

Interface Class for Document Readers

For a document reader, add in the existing API methods provided by the current XML reader to make the transition to the new interface easier.

```
```

```
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, add in the existing API methods provided by the current XML writer to make the transition to the new interface easier.

```
class DocumentWriter : public Operation
{
  public:
    virtual bool write(DocumentPtr, const FilePath& uri) = 0;
    virtual bool write(DocumentPtr, const std::string& data)
    {
        return false;
    }
    virtual void write(DocumentPtr, std::ostream& stream)
    {
        return false;
    }
}
```

XML Reader Wrapper Class

- Due to all the global references used, the easiest way to make this work is to add this to `Xmlio.cpp`
- To "hide" the global functions which are not currently part of the public API, we can make these into local statics.
- If we decide to deprecate the global functions they can all be make local statics.

XML Reader Options

- We consolidate all of the existing options into a single class which derives from the `OperationOptions` interface.
- Small variation adds code to guarantee standard libraries are set if specified.

```
class XmlDocumentReadOptions : public OperationOptions, XmlReadOptions
{
    // Extend to include search path and standard library
    FileSearchPath _searchPath = FileSearchPath();
    DocumentPtr _standardLibrary = nullptr;
};
```

XML Reader

```
// XML Reader
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;
    }

    void setStandardLibrary(DocumentPtr &llib)
    {
        _standardLibrary = lib;
    }

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

XML Writer Wrapper Class

- As with read options we add in a new write options class

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

// prependXInclude is unused and belongs with Document class
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 are some example serializer derivations.

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.

Use Case

- One issue with embedding into code into core is that there is a need to add an explicit build time dependency on the C++ `CURL` library.
- An extension can separate this dependency and only include it when 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 std::map<std::string, std::string>& headers)
    {
        _headers = headers;
    }

    int timeout() const
    {
        return _timeout;
    }

    int connectTimeout() const
    {
        return _connectTimeout;
    }

  private:
    int _timeout = 30;
    int _connectTimeout = 10;
    std::map<std::string, std::string> _headers;
};

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

- Python example is a custom DocumentWriter that saves a MaterialX document and its referenced textures into a ZIP file.
- Dependencies are part of the module / package.

```
# 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"]

    # Overrride "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 can be added in the same that there are existing `ImageHandler` and `GeometryHandler` classes.
- For lifetime management we add in smart pointer support for readers, writers, and handler.
- If source and desintation 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;
};
```

Language Wrappers

Python

- Use the existing `pybind11` mechanism to add the new interface classes.
- "Trampoline" code is added to allow calling of Python classes from C++.
- AAll interfaces are exposed as part of the package in 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.

Asynchronous Transport Support

- It is possible to implement asynchronous loading and saving of documents using `std::async` and `std::future` or other suitable mechanisms.
- It could provide a callback mechanism for progress updates and completion notifications.
- The current thought is that this should not live at this level but instead at a higher integration level. For example an integration may be using Web sockets to achieve asynchronous loading and saving of documents.