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.
- Path binding is required. Not all input / output types can support string / stream output. It is up to the implementation to decide what makes sense.
- 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. Note that element predicates don't really fit the Value model so TBD.

```
using OptionsMap = std::unordered_map<string, ValuePtr>;
```

Interface class for document reader

```
class DocumentReader
{
  public:

    virtual DocumentPtr read(const FilePath& uri) = 0;

    virtual DocumentPtr read(const std::string& data)
    {
        return nullptr;
    }

    virtual DocumentPtr read(std::istream& stream)
    {
        return nullptr;
    }

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

Interface class for document writer

```
class DocumentWriter
{
  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 globals used, the easiest way to make this work is to add this to `Xmlio.cpp`
- To "hide" the global functions they can be made into local statics.
- Small variation adds code to guarantee standard libraries are set if specified.

```
// XML Reader
class XMLDocumentReader : public DocumentReader
{
  public:
    XMLDocumentReader() = default;

    DocumentPtr read(const FilePath& uri) override
    {
        DocumentPtr doc = createDocument()
        if (_standardLibrary)
        {
            doc->setDataLibrary(_standardLibrary);
        }
        readFromXmlFile(uri,
                        _searchPath,
                        _readOptions);

        // Implementation for reading XML from a file path
    }

    DocumentPtr read(const std::string& data) override
    {
        DocumentPtr doc = createDocument()
        if (_standardLibrary)
        {
            doc->setDataLibrary(_standardLibrary);
        }
        readFromXmlString(doc, data, _searchPath, _readOptions );
    }

    DocumentPtr read(std::istream& stream) override
    {
        DocumentPtr doc = createDocument()
        if (_standardLibrary)
        {
            doc->setDataLibrary(_standardLibrary);
        }
        readFromXmlStream(doc, stream, _searchPath, _readOptions);
    }
    StringVec supportedExtensions() const override
    {
        return _supportedExtensions;
    }

    void setReadOptions(const XmlReadOptions& options)
    {
        _readOptions = options;
    }

    XmlReadOptions& getReadOptions() const
    {
        return _readOptions;
    }

    void setSearchPath(const FileSearchPath& searchPath)
    {
        _searchPath = searchPath;
    }

    FileSearchPath& getSearchPath() const
    {
        return _searchPath;
    }

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

  private:
    FileSearchPath _searchPath;
    XmlReadOptions _readOptions;
    StringVec _supportedExtensions = { ".mtlx" };
    DocumentPtr _standardLibrary = nullptr;
}
```

XML Writer Wrapper Class

```
// prependXInclude is unused and belongs with Document class
class XMLDocumentWriter : public DocmentDeserializer
{
  public:
    XMLDocumentWriter() = default;

    bool write(const DocumentPtr doc, const FilePath& uri)
    {
        writeToXmlFile(doc, uri, _searchPath, _writeOptions);
        return true;
    }
```

~~~
    bool write(const DocumentPtr doc, const std::string& data)
    {
        writeToXmlString(doc, data, _searchPath, _writeOptions);
        return true;
    }

    bool write(const DocumentPtr doc, std::ostream& stream)
    {
        writeToXmlStream(doc, stream, _searchPath, _writeOptions);
        return true;
    }

    StringVec supportedExtensions() const override
    {
        return _supportedExtensions;
    }

    void setWriteOptions(const XmlWriteOptions& options)
    {
        _writeOptions = options;
    }

    XmlWriteOptions& getWriteOptions() const
    {
        return _writeOptions;
    }

    void setSearchPath(const FileSearchPath& searchPath)
    {
        _searchPath = searchPath;
    }

    FileSearchPath& getSearchPath() const
    {
        return _searchPath;
    }

  private:
    FileSearchPath _searchPath;
    XmlWriteOptions _writeOptions;
    StringVec _supportedExtensions = { ".mtlx" };
}

```c++

### C++ http Reader Class

- 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, whereas an extension
can separate this dependency and only include it when needed.

```c++
// C++ http loader.
#if defined(CURL_INSTALLED)
~~~

```cpp
#include <curl/curl.h>
#endif

class HTTPXMLReader : public DocumentReader
{
  public:
    HTTPXMLReader()
    {
        curl_global_init(CURL_GLOBAL_DEFAULT);
    }

    virtual ~HTTPXMLReader()
    {
        curl_global_cleanup();
    }

    DocumentPtr read(const FilePath& uri) override
    {
        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, 30L);
            curl_easy_setopt(curl, CURLOPT_CONNECTTIMEOUT, 10L);

            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) override
    {
        return nullptr;
    }

    DocumentPtr read(std::istream& stream) override
    {
        return nullptr;
    }

    StringVec supportedExtensions() const override
    {
        return _supportedExtensions;
    }

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

Python Zip file writer

- Python example is a custom DocumentSaver 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
    import MaterialX.PyMaterialXRender as mx_render
    logger.info("MaterialX and zip modules loaded successfully")
    have_zip = True
except ImportError:
    raise ImportError("Please ensure MaterialX and zipfile modules are
installed.")

class ZipSaver(mx_render.DocumentSaver):
    _plugin_name = "ZipSaver"
    _ui_name = "Save to Zip..."

    def name(self):
       return self._plugin_name

    def uiName(self):
       return self._ui_name

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

```python
    # Overrride "write"
    def write(self, doc, path):
        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
```