FILE COMPILATION PROFILE

OVERVIEW

This purpose of this document is to describe how to create, deploy and use a File Compilation Profile (FCP). A File Compilation Profile is a text file containing instructions that define how to handle different types of files and folder structures found during the ingestion of assets.

A profile is created using a special syntax that makes it easy to define what happens as the Ingest process walks through a list of files and folders.

CREATING A FILE COMPILATION PROFILE

A File Compilation Profile is capable of including or excluding files and folders based on rules defined in the profile syntax. The rules can be as simple as locating files with a given file extension or as complex as grouping files into an archive where a given binary value is found as a certain offset in a file.

BASIC PROFILE

The following section will build a simple profile and explain the available rules and how to employ them.

A Profile file contains at least one profile section and at least one construct. Each Construct is evaluated in its order within the current Profile. If the condition is not met then next Construct is evaluated until the end of the Profile is reached.

```
profile example.one {
      construct bad.file {
            match {
                  file extension "bad"
            }
            logical type encapsulation type
            consume no
      }
      construct default {
```

```
            match {
                  file is file
            }
            logical type encapsulation type
            consume yes
      }
}
```

This example will ignore any file with the extension “bad” and will ingest any other file found.

1. The example begins by defining a container Profile “example.one” within which two Constructs are found. We have named the first Construct “bad.file” and the second “default”.
2. As the ingest engine walks through the selected files and folders it tests each file against the “bad.file” Construct. If the file has the extension “bad” then it is ignored using the “consume no” statement.
3. If the file does not match the “bad.file” Construct then the “default” Construct is evaluated which matches all files. We ingest the file using the “consume yes” statement.

PACKAGING GROUPS OF FILES

We can expand this simple profile to additionally package a folder of related files into an archive. This example locates folders called “data-files” and compresses each folder into an archive.

```
profile example.one {
      construct bad.file { .. }
      construct grouping.example {
            match {
                  file is directory and file name "data-files"
            }
            encapsulate as archive level 6
            logical type encapsulation type
            consume yes
      }
      construct default { .. }
}
```

ADDING COMMENTS

It is good practice to include comments in your FCP so that you can explain the intent of a Construct to someone reading the Profile at a later date. Comments begin with “/*” and end with “*/”.

```
/* defines a file with a .bad extension */
construct bad.file {
      match {
            file extension "bad"
      }
```

```
      logical type encapsulation type
      /* we don't want to consume this file */
      consume no
}
```

NESTED PROFILES

Sometimes a File Compilation Profile can become quite lengthy and the file/folder rules complex. When this happens it helps to break the rules into sub-profiles.

Each sub-profile could contain Constructs specific to a file format or business unit.

```
profile main.profile {
      profile data.profile {
            construct data.one { .. }
            construct data.two { .. }
      }
      profile accounts.profile {
            construct accounts.one { .. }
            construct accounts.two { .. }
      }
      construct main.one {
            match {
                  file is directory and file name “accounts”
            }
            switch to profile accounts.profile
      }
      construct main.two{
            match {
                  file is directory and file name “data”
            }
            switch to profile data.profile
      }
}
```

PROFILE SYNTAX

The Profile must begin with a “profile” statement followed by curly braces to encapsulate its Constructs. Each Profile name must be unique and can contain alpha-numeric characters as well as colon, underscore, period or hyphen.

A Profile maintains an ordered list of its Constructs and any nested Profiles so that they are evaluated in the order they are laid out in the profile.

```
profile profile.one {
      profile profile.two {
            construct construct.one { .. }
            construct construct.two { .. }
      }
      construct construct.one { .. }
      construct construct.two { .. }
```

```
}
```

CONSTRUCT SYNTAX

The Construct must begin with a “construct” statement followed by curly braces to encapsulate its statements. Each Construct name must be unique within its parent Profile and can contain alpha-numeric characters as well as colon, underscore, period or hyphen.

A Construct must contain a “match” block with at least one filtering statement.

```
match {
      file extension "pdf"
}
```

Optionally, the Construct can package the matching file(s) in an archive with an optional compression level.

```
encapsulate as archive level 6
```

The Construct must also set the Logical MIME Type for the evaluated files or folder. This is often achieved simply by setting the Logical MIME Type to be the Encapsulated MIME Type determined by Mediaflux. For example, if the file extension is “.pdf” then the Encapsulated MIME Type is resolved as “application/pdf”. However in this instance the file is being packaged in an Arcitecta Archive files (.aar) which will have an Encapsulated MIME type of “application/arc-archive”

```
logical type "application/pdf"
```

A matched Construct will, by default, consume the referenced file. This can be declared explicitly by adding the “consume yes” statement or overridden using “consume no”.

```
consume yes
```

These Construct settings must appear in the correct order for the Construct to be recognized correctly.

```
construct construct.one {
      match {
            file extension "pdf"
      }
      encapsulate as archive level 6
      logical type "application/pdf"
      consume yes
}
```

If the Construct defines that the profile must be switched then this statement must come directly after the “match” statement.

```
construct construct.two {
      match {
            file is directory and file name "source"
      }
      switch to profile profile.two
}
```

MATCH SYNTAX

Each Construct must begin contain a “MATCH” block with one or more selection criteria. The File Compilation Profile provides a number of useful functions to help define your rules. Each statement can be written in upper or lower case.

```
MATCH { .. }
MATCH NOT { .. }
```

FILE TYPE MATCHING

There are several statements to assist with matching specific files. The “FILE IS” statement provides a way to determine if the current file being evaluated is a file or a directory.

```
FILE IS FILE
FILE IS DIRECTORY
```

FILE AND DIRCTORY NAME MATCHING

The “FILE EXTENSION” statement provides a way to test the file extension of the file being evaluated.

```
FILE EXTENSION "doc"
```

The “FILE NAME” statement includes both basic and advanced features to test the name of the file or folder being evaluated. The last two statements employ a Regular Expression pattern to match the filename.

```
FILE NAME "TestFile.pdf"
FILE NAME IGNORE CASE "testfile.pdf"
FILE NAME PATTERN "^T[a-z]{3}F[a-z]{3}\.pdf"
FILE NAME PATTERN IGNORE CASE "[a-z]{5,8}\.pdf"
```

FILE CONTENT

The “FILE CONTAINS” statement provides a mechanism to examine the content of text or binary files.

```
FILE CONTAINS TEXT "Mediaflux"
FILE CONTAINS TEXT "Mediaflux" BETWEEN 10 AND 18
FILE CONTAINS BINARY "4d65646961666c7578"
FILE CONTAINS BINARY "4d65646961666c7578" BETWEEN 10 AND 18
FILE CONTAINS XML "/person[@department='Accounts']"
```

The “TEXT” switch searches for the string value provided anywhere within the file, or at a certain byte offset if start and end bytes have been provided.

The “BINARY” switch searches for the hexadecimal value provided anywhere within the file, or at a certain byte offset if start and end bytes have been provided.

The “XML” switch attempts to parse the file as an XML Document. If the XPath provided evaluates to a Node in the XML then it matches.

COMBINING RULES

Statements within the “MATCH” block can be chained together using the “AND” and “OR” statements.

```
FILE IS FILE AND FILE CONTAINS TEXT "Mediaflux"
FILE EXTENSION "dat" OR FILE NAME "database.db"
```

RULES FOR EXAMING DIRECTORIES

Other statements assist with matching directories and their content. The “DIRECTORY CONTAINS” statement can match if a directory contains a file or sub-directory with a given name. It is also able to select another Construct to provide the matching rules.

```
DIRECTORY CONTAINS FILE "readme.txt"
DIRECTORY CONTAINS DIRECTORY "library"
DIRECTORY CONTAINS CONSTRUCT construct.three
```

FILE GROUPING

The “GROUP” statement is a block that declares a series of FILE statements upon which to match.

The default behaviour of the “GROUP” statements matching process is to match files with the declared extensions and the same filename prefix. The following will match files (“sample.abc”, “sample.def” and “sample.xyz”).

```
GROUP {
```

```
      FILE EXTENSION "abc"
      FILE EXTENSION "def"
      OPTIONAL FILE EXTENSION "XYZ"
}
```

With the addition of the “UNNAMED” option the grouping ignores the filename prefix. The following will match files (“fileone.abc”, “filetwo.def” and “filethree.xyz”)

```
GROUP UNNAMED {
      FILE EXTENSION "abc"
      FILE EXTENSION "def"
      OPTIONAL FILE EXTENSION "XYZ"
}
```

GROUPING BEHAVIOUR

The “GROUP” statement can operate in two distinct ways depending on how it is invoked.

- When used within a Construct that consumes the matching files, only the files declared in the Group are added to the package.
- Grouping can also be used as the criteria for matching a directory, when this is matched all the files in the folder matching the Construct will be packaged together.

As an example, assume we have the following folder:

```
      folder-a /
            sample-file.abc
            sample-file.def
            sample-file.ghi
            sample-file.txt
```

The following profile will walk through the folder hierarchy. It will evaluate “folder-a” and not find a match. It will then evaluate “sample-file.abc” and match on the “GROUP” statement as it has the other files.

```
profile grouping.one {
      construct grouping.rule {
            match {
                  group {
                        file extension "abc"
                        file extension "def"
                        optional file extension "ghi"
                  }
            }
            encapsulate as archive level 6
            logical type encapsulation type
            consume yes
      }
      construct default {
```

```
            match {
                  file is file
            }
            logical type encapsulation type
            consume yes
      }
}
```

The three declared files are compressed into an archive asset called “sample-file”, but the archive does not include the “sample-file.txt” file. It is also ingested though thanks the “default’ Construct.

The next profile will evaluate “folder-a” using the “group.folder” Construct that in turn uses the grouping rule in the “grouping.rule” Construct.

```
profile grouping.two {
      construct grouping.rule {
            match {
                  group {
                        file extension "abc"
                        file extension "def"
                        optional file extension "ghi"
                  }
            }
            logical type encapsulation type
            consume no
      }
      construct group.folder {
            match {
                  directory contains construct grouping.rule
            }
            encapsulate as archive level 6
            logical type encapsulation type
            consume yes
      }
      construct default {
            match {
                  file is file
            }
            logical type encapsulation type
            consume yes
      }
}
```

The result will be an asset called “folder-a” being created with a compressed archive containing all the files within “folder-a”, including “sample-file.txt”. This is because the “GROUP” statement is being used only to provide a match for the calling Construct.

METADATA SYNTAX

Occasionally you may need to merge metadata into an asset created from a matched file.

The “METADATA” block allows you to specify the location of the XML file that the metadata is captured from, as well as how it is processed.

```
construct default {
      match {
            file is file
      }
      metadata {
            content {
                  extension "xml"
            }
      }
      logical type encapsulation type
      consume yes
}
```

The simple example above matches all files and additionally merges XML where it finds an adjacent file with the same filename prefix and an “xml” extension. In this instance we do not consume the adjacent XML file.

XML METADATA IN FILES

The “CONTENT” statement attempts to extract metadata XML from a file adjacent to the file or folder currently being matched. .

```
content {
      extension "xml"
}
```

The “EXTENSION” option looks for an adjacent file with the same filename but with the supplied extension with a period between. Using the example above for *filename.jpg* the metadata extractor will look for metadata in *filename.xml*

Adding a period between the filename and the supplied extension to match on will not always suit all scenarios. To accommodate this the LITERAL option can be applied as follows.

```
content {
      extension "-metadata.xml"
      {
            literal
      }
}
```

Using the same example file (*filename.jpg*) this would not include the period when compiling the matching filename and will now match on *filename-metadata.xml* instead

XML found in these files will likely not conform to any document type in the server. The default behavior in Mediaflux is to block metadata that is not known. To allow Mediaflux to import these documents, by default, we wrap them in a special metadata document type called “raw-xml”. This is an unbounded document type that can contain any XML.

Occasionally the metadata will be in a known format, either by agreement with the sender who will conform to a known schema or by modeling the incoming metadata within

Mediaflux. To allow for this situation the “KNOWN TYPE” option informs the metadata extract that we know the format and will accept it as is without wrapping it in the “raw-xml” container.

```
content {
      extension "xml"
      name "invoice.xml"
      {
            known type
      }
      pattern "^invoice-[0-9]{3,4}\.xml$"
      {
            known type
      }
}
```

METADATA FROM FILE OR DIRCTORY NAMES

The “FILE NAME” statement attempts to extract metadata from the name of the file or folder currently being matched. The following example uses a Regular Expression to segment the file name into name and date metadata. The brackets around the parts of the regular expression are made available to the string in the “XML” statement if the pattern matches the current file name. The “{2}” value in the string represents the value in the second group. The fragment supplied is prefix by a format option that defines the format of the fragment. Currently XML formatted strings are supported.

```
file name {
      pattern "([A-Z]{4})_(\d{2})(\d{2})(\d{4})" {
            xml "<a><b>{1}</b><c>{2}-{3}-{4}</c></a>"
      }
      pattern "([A-Z]{4})_(\d{2})(\d{2})(\d{4})" {
            xml "<y><z>{2}-{3}-{4}</z><x>{1}</x></y>"
      }
      pattern "(\d{2})(\d{2})(\d{4})_([A-Z]{4})" {
            xml "<f><g>{4}</g><h>{1}-{2}-{3}</h></f>"
            known type
      }
}
```

By default the XML metadata generated will be wrapped in the “raw-xml” container document type. To notify the generator that the document we are creating exists in Mediaflux we can add the “KNOWN TYPE” option that will pass the metadata document through without the wrapper.

LITERAL METADATA

Literal XML can be added to an asset with the optional “XML” statement. The fragment supplied is prefix by a format option that defines the format of the fragment. Currently XML formatted strings are supported.

```
Literal {
      xml "<a><b>200</b></a>"
      xml "<f><g>200</f></f>"
      {
            known type
      }
}
```

As with the other generators XML metadata generated will, by default, be wrapped in the "raw-xml" container document type. To notify the generator that the document we are creating exists in Mediaflux we can add the "KNOWN TYPE" option that will pass the metadata document through without the wrapper.

TRANSFORMING METADATA

Most metadata located in adjacent XML files will not conform to document types within Mediaflux. To help with converting these structures the "TRANSFORM" operation provides the ability to convert the document definition of the metadata using XSLT documents stored on the server.

The following example that locates a metadata file called "invoice.xml" adjacent to the matched file, imported the XML content and then transforms it prior to it being added to the asset.

The transformation process begins by locating documents that are capable of being transformed. The "XML CONTAINS" option provides an XPath to match against a document. If it matches then the document is transformed using the named XSLT file identified in the "USE" statement. The named XSLT is returned from the server providing it has a MIME Type of "application/arc-fcp-xslt".

If the XML does not match either transform statement then the result is that the original raw XML is merged with the asset.

```
construct default {
      match {
            file is file
      }
      metadata {
            content {
                  file name "invoice.xml"
            }
            transform {
                  xml contains "/root/invoice" {
                        use "ClientA-Xslt"
                  }
                  xml contains "/company.com/invoices" {
                        use "ClientB-Xslt"
                        keep original
                  }
            }
      }
      logical type encapsulation type
      consume yes
```

```
}
```

USING A PROFILE IN THE DESKTOP

A File Compilation Profile can be used when ingesting a file or folder structure from Desktop but first it needs to be registered within Mediaflux.

1. Create your File Compilation Profile. In this demonstration it will be called “example.fcp”
2. Create a new asset from the example.fcp file and give it a MIME Type of “application/arc-fcp”.
3. Once the asset has been created, open Desktop and ingest a local folder. At this point you will be able to select the “example” File Compilation Profile from the “Packaging” drop-down list.

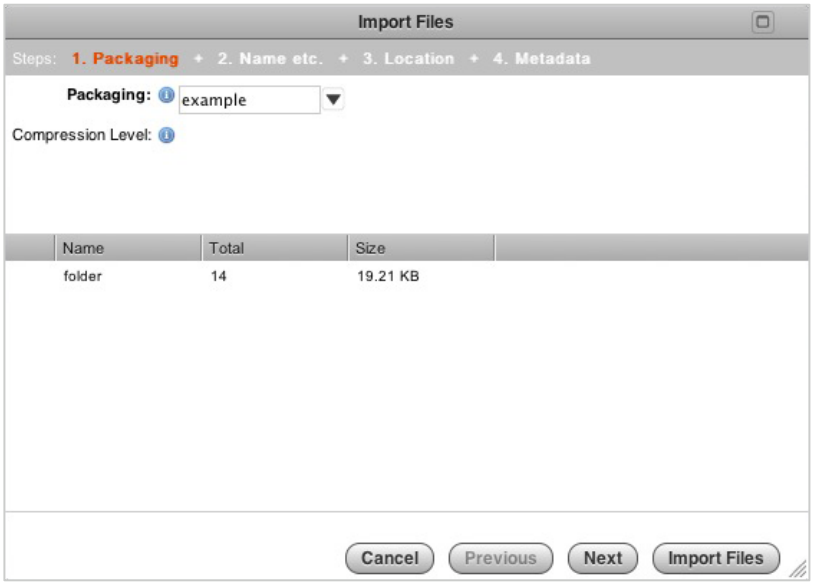

Figure 1 – Desktop Import Files Dialog

USING A PROFILE IN ATERM

A File Compilation Profile can also be used in the ATERM Command Line. The following command will ingest a folder using the Profile of your choice.

```
import -lp /path/to/example.fcp /path/to/folder/to/ingest
```

ATERM can be a great tool to test your Profiles prior to release. Adding the “-mode test” and “-verbose true” attributes will cause the profile to be evaluated against the folder and will output detailed information about the matching files, but will not ingest the files.

```
import -mode test -verbose true -lp /path/to/example.fcp
/path/to/folder/to/ingest
```

Arcitecta Pty Ltd
ABN: 83 081 599 608 ACN: 081 599 608
Postal Address: Suite 5/26-36 High Street Northcote Victoria Australia 3070
Telephone: + 61 3 8683 8523
Fax: + 61 3 9005 2876
Website: www.arcitecta.com
Email: info@arcitecta.com

ARCITECTA™
OPERATING SYSTEMS FOR META+DATA