Guidance for
File Delivery using File Manifest

This practice defines how to proper use of File Manifest (FileManifest from Common Media Manifest spec and schema)

This practice was originally part of *Using Media Manifest, File Manifest and Avails for file Delivery.*

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

NOTE: No effort is being made by the Motion Picture Laboratories to in any way obligate any market participant to adhere to this specification. Whether to adopt this specification in whole or in part is left entirely to the individual discretion of individual market participants, using their own independent business judgment. Moreover, Motion Picture Laboratories disclaims any warranty or representation as to the suitability of this specification for any purpose, and any liability for any damages or other harm you may incur as a result of subscribing to this specification.

REVISION HISTORY

Version	Date	Description
1.0	May 25, 2017	Initial publication

1 FILE DELIVERY

1.1 File Manifest

The File Manifest is an XML document that describes the structure of files delivered from one party to another. Common Media Manifest (www.movielabs.com/md/manifest) describes how to encode the FileManifest element. This section provides additional guidance on how to use the File Manifest in various applications.

1.2 Package Concept and Identifier

A set of files is called a Package and is identified with a PackageID.

Avails must be identified in a globally unique manner. An ALID might be used by the studio or various service providers partnering with the studio.

1.2.1 What an Package Identifier Identifies

A Package ID represents a collection of files associated with a delivery.

There are two use cases here

- PackageID applies to a single delivery of files. Any files subsequent delivery would require a new PackageID. Versioning is handled outside the scope of File Manifest.
- Package ID applies to a set of files, regardless of when and how they are delivered. Subsequent versions of the File Manifest, as distinguished by FileManfiest/PackageDateTime, represent the complete set of files that the sender expects the recipient to obtain.

1.2.2 Constructing an PackageID

A Package ID SHALL be globally unique. The same PackageID SHALL NOT be used for distinct sets of files.

A PackageID SHOULD comply with the PackageID format above. This is not a strict requirement, but it will make global uniqueness much easier and avoid us IDs in the wrong context. Note that UUIDs avoid the first issue, but not the second.

An PackageID SHOULD be based on an EIDR ID.

1.2.3 EIDR IDs and PackageIDs

A PackageID using an EIDR IDwould be of one the two following forms:

“`md:alid:eidr-s:` ” <Short EIDR>

“`md:alid:eidr-x:`”<Short EIDR>“:”<extension>

In one usage, an EIDR ID is created for the Package as a Compilation. In this case, the EIDR-s form would be used. Alternatively, the EIDR ID could be constructed as an EIDR-x. See Section **Error! Reference source not found.** for instructions how to use an EIDR ID for ALID.

If an ALID is further extended, <extension> part would include both the Avail unique information and the package information. For example, let's assume a single title: Do the Right Thing, EIDR Edit = 5EE7-A973-819A-DC1A-CDD8-H. ALID might be:

```
md:alid:eidr-x:5EE7-A973-819A-DC1A-CDD8-H:craigsmovies.com_july_NorthAmerica
```

The Package ID could be:

```
md:package:eidr-x:5EE7-A973-819A-DC1A-CDD8-H:craigsmovies.com_july_NorthAmerica_pkg1
```

Note that <type> changed to 'package' and '_pkg1' was added to indicate that this Package ID is for the first package (package 1) associated with that Avail.

Identifiers are to be processed in their entirety as an opaque object. Naming conventions in extensions are intended to support uniqueness human readability and SHOULD NOT be parsed automatically to extract information.

1.3 File Identification and Versioning

1.3.1 Identifying Files

Files can be identified in several ways. The most important distinction when identification refers to a specific version (e.g., a particular encoding) or to the contents (e.g., an encoding of a particular language track). For versioning files, it is important to know both.

The most definitive identification of a given file at the bit level is the file Hash (FileInfo/Hash). It is effectively guaranteed to be unique for any file.

The most flexible identification is an identifier (i.e., FileInfo/Identifier). An identifier can identify version and contents. Multiple identifiers can be included.

1.3.2 Versioning

Files versioning requires knowing two files are the same except for the version. They must have some level of identification, but also be different in some manner.

Currently the following methods are available for versioning.

- FileInfo/Version element
- Information inherent in the file
- Determine from Media Manifest which files are relevant

1.3.2.1 FileInfo/Version FileInfo/FileDate versioning

The simplest means to determine the most recent version is the use of the Version element in FileVersion. The initial has Version either absent or '0'. These are semantically equivalent. A Version>0 means there has been an update. The file with the largest Version is the most current.

As this is the simplest and most unambiguous, this method is recommended.

FileInfo/FileDate can also be used to determine file version.

1.3.2.2 Information in the File

Some files contain version information in the file. This is not the easiest method to determine version and it does not apply to all files, so it is not the preferred method.

Version information in files include:

File Type	Means to determine version
Metadata	Basic Metadata such as found in Media Entertainment Core (MEC) can be versioned using CoreMetadata/Basic/UpdateNum
Media Manifest	Media Manifest is versioned using MediaManifest/@updateNum. ExtraVersionReference can be used as an additional versioning identifier.
Avails	Avails can be versioned using Avail/Disposition/IssueDate. Note that IssueDate can include time.
Common File Format (CFF)	Common File Format files can be versioned by APID with additional information in MetadataMovie/ContainerVersionReference and MetadataMovie/@MetadataVersionReference.

1.3.2.3 Media Manifest

The Media Manifest can link Avails to Experience and ultimately to media files. In general, if a file appears in the Media Manifest it is required.

If the Manifest is complete, there is no ambiguity about what file is what or how it is used in the system. Media file information is in the Inventory. Metadata is referenced by ContentID.

1.4 File Delivery

The File Manifest design recognizes that not all files are necessarily delivered at once and may be updated once delivered. Methods are provided for generating and accepting incremental deliveries.

1.4.1 Delivery Methods

The File Manifest supports most common means of delivering files: FTP, email, HTTP GET, etc. These methods are defined in [Manifest]. This document has no further recommendations regarding delivery methods.

1.4.2 Delivering Sets of Files

The File Manifest supports delivering multiple files together. To do so, include multiple instances of FileInfo, one for each file.

The basic types of files delivered are as follows, although any type of file can be delivered.

- Manifest file – A file with data described in this section
- Metadata files
- Media files – audio, video, subtitle, apps, images, text, interactivity/apps)
- Avails files
- Ancillary files (any other files). Images are Ancillary files.

All files could be delivered in a single manifest or they could be spread across multiple manifests. It is reasonable to have one of files in one manifest and another set in another manifest. One should consider timing, including updates, when deciding how files should be grouped. Using the New Package Model described below (a new FileManifest for each drop), this is less critical. However, if the Single Package Model is used, the initial choice of grouping must be maintained for updates.

1.4.3 Incremental Delivery

There are two ways to handle incremental delivery:

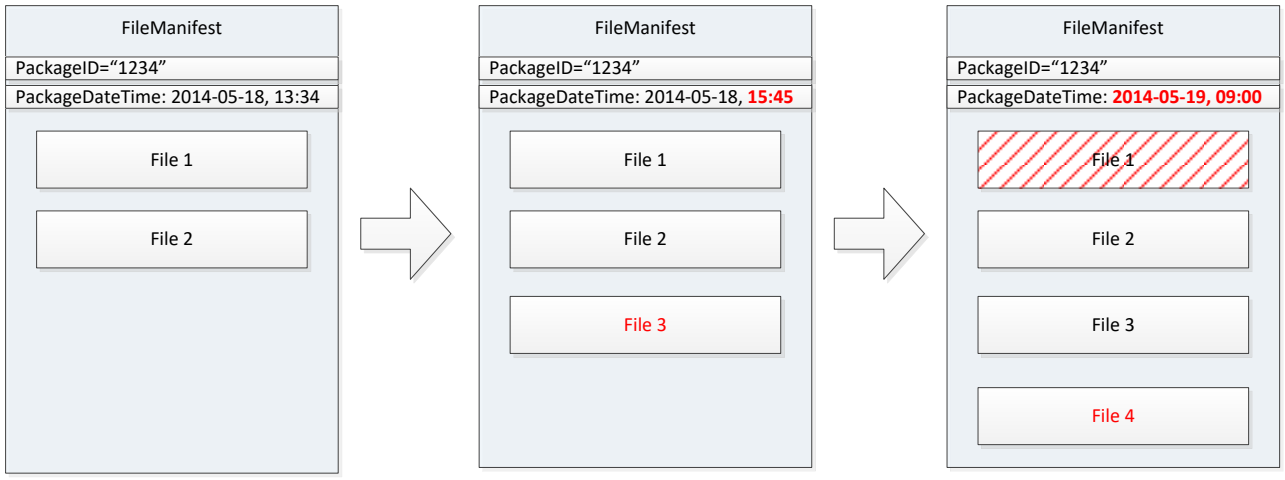
- A single Package definition is used, but updated to reflect changes.
- A new Package is generated for each update.

These are described in the following sections.

1.4.3.1 Single Package Model

The Single Package Model uses the same Package and PackageID for all deliveries. Each Package represents a complete snapshot of the delivery. The goal is for the recipient to have the same files represented in the File Manifest. If a file is added to the File Manifest, that file should be obtained. If a file is removed from the File Manifest, that file can be removed.

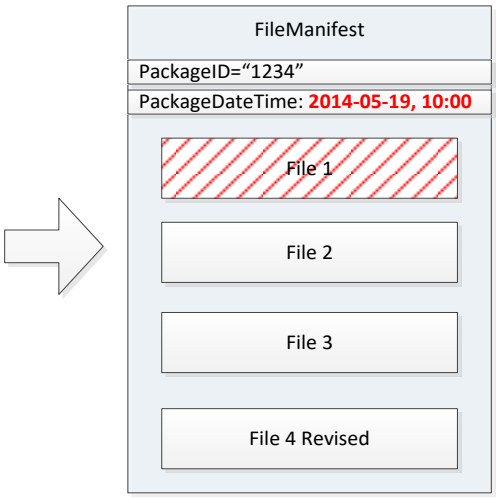
Consider the following. There are three deliveries of the File Manifest. They all have the same PackageID. The date-time increases with each delivery. The second drop adds File 3. The third drop adds File 4 and removes File 1. As the 2014-05-19 09:00 delivery is the last, the recipient should have Files 2, 3 and 4.



The removal of File 1 is indicated by FileInfo/Delivery/DeliveryMethod='removed'.

In the second delivery, File 1 and File 2 have already been delivered. The same is true for File 3 in the third delivery. This can be determined by the recipient by looking at the identification and the Hash. If the file is still deliverable, it should be noted in DeliveryMethod. However, if the file is presumed delivered and is no longer available, it should be noted by setting DeliveryMethod='delivered'.

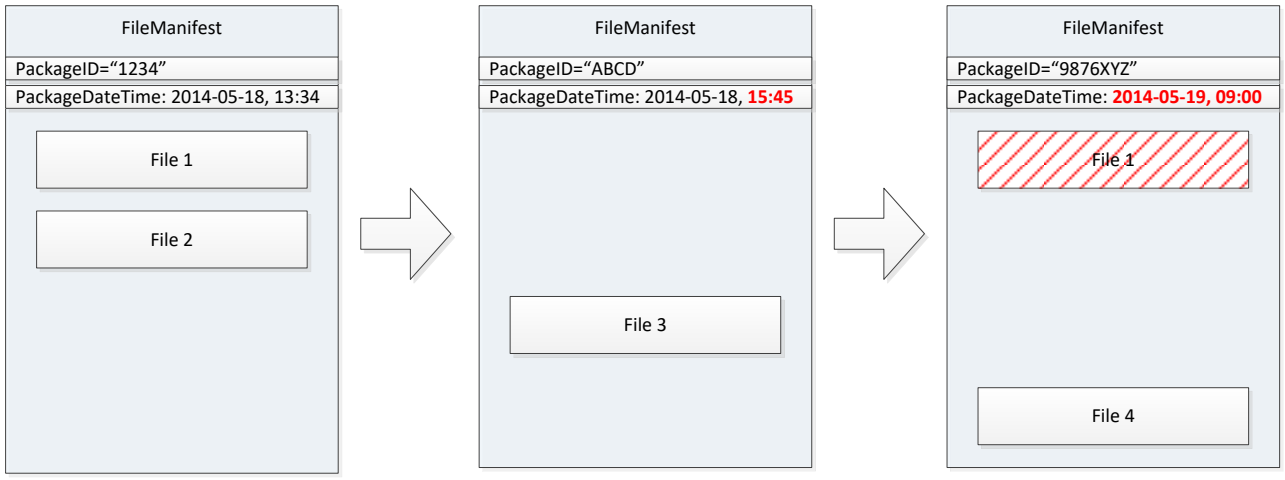
The following diagram that follows the example above illustrates a File update:.



In this example, File 4 is revised. To recognize the File 4 is updated is must have the same Identifier as the previous version. Version is updated to be a higher value than the previous Version.

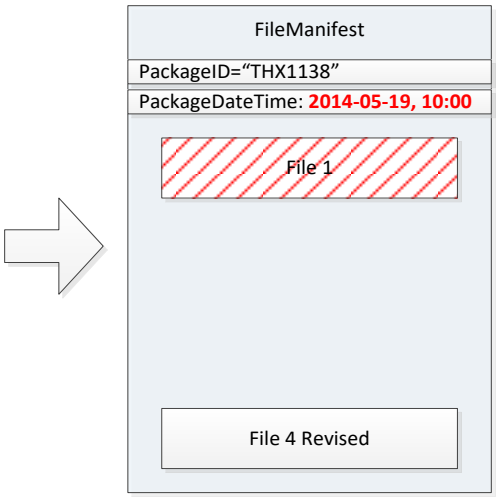
1.4.3.2 New Package Model

In the New Package Model, each File Manifest has a unique PackageID. Only new or removed files are included. The following example has the same net effect as the example in the Single Package Model.



This model is requires every update to processed, and processed in order.

Update is similar in the model.



Like the previous model, Identifier remains constant and the Version is revised.

1.5 Verifying File Correctness

It is strongly suggested that Hash be included and verified. Hash is required in cases where Hash is used to differentiate versions.

Software is subject to manipulation and should be treated as sensitive. All files that include any code, whether source or executable, must include Hash in the File Manifest and must be checked.

XML canonicalization is not required. Two XML documents might be functionally equivalent, but have slightly different encoding (e.g., different white space). There is no attempt to match such equivalences so XML Canonical Form [XML-C] is not required.

Hash algorithms (Hash/@method as defined in [CM], should be either 'MD5', 'SHA-1' or 'SHA-256'. It is strongly recommended that implementations ingesting data from a File Manifest be capable of verifying using these algorithms.

If Hash is used, it is strongly recommended that the Length element be used to ensure that the sender and the receiver are checking the correct number of bytes. If Length differs more than a few bytes from the actual length of the file, be sure to verify that the hash was properly generated (think Heartbleed).