GV File

FIMS REST API Guide

Version History

Date	Version	Release by	Reason for Changes
08/12/2012	0.01	Steve Dabner	Document originated (1st proposal)
18/02/2013	0.02	Steve Dabner	Added new commands
05/09/2014	1.0	Andy Gingell	Transferred into new template
09/09/2015	1.1	Andy Gingell	Rebranding
20/01/2016	1.2	Greg Emerson	Completed rebrand
10/10/2018	1.3	Jon Metcalf	GV rebrand
12/03/2021	1.4	Jon Metcalf	Rebrand

Table of Contents

List of Tables

1. Definitions, Acronyms and Abbreviations

Table 1: Table of Terminology

Term	Definition
FIMS	Framework for Interoperable Media Services
AMWA	Advanced Media Workflow Association
EBU	European Broadcasting Union
SOA	Service Oriented Architecture
MPP	Media Processing Platform
SOAP	Simple Object Access Protocol
WSDL	Web services description language
XSD	XML Schema Document

2. Introduction

2.1 Purpose

The GV File server supports the FIMs Transform SOAP interface API [Ref: i] for the addition and management of jobs.

In addition a simple RESTful API [Ref: i] is supported for the transfer of files and information between system components where this cannot be easily achieved using the FIMS protocol. This document describes the operation of this REST style interface.

2.2 References

FIMs Documentation

FIMS Implementation

3. RESTful API

3.1 Introduction

The GV File server REST API uses HTTP GET and HTTP PUT [Ref: i] methods to retrieve from and send data to the server. The URL that is used varies with the operation that is required but will always begin with the server listening address.

The URL that the GET and PUT requests are sent to can be split up as follows:-

$$\underbrace{\text{http}}_{\text{scheme}}\text{://}\underbrace{\text{172.3.4.5}}_{\text{host}}\text{:}\underbrace{\text{35061}}_{\text{port}}\underbrace{\text{/downloads/products}}_{\text{path}}$$

Or,

$$\underbrace{\text{http}}_{\text{scheme}}\text{://}\underbrace{\text{example.com}}_{\text{host}}\text{:}\underbrace{\text{35061}}_{\text{port}}\underbrace{\text{/downloads/products}}_{\text{path}}$$

GET and PUT requests to a single GV File server will always be sent to a URL that begins with the same scheme, host and port. The path element of the URL determines what operation will take place.

For example, if the GV File Server running on example.com and listening on port 35061, then the HTP GET and PUT requests will always be sent to a URL that begins:

http://example.com:35061/

HTP GET operations are used to retrieve data from the server. HTTP PUT operations are used to send data to the server.

3.2 File Operations

Files can be downloaded from the GV File server and uploaded to it. Files can only be transferred to and from a specific directory (and its sub-directories) in the GV File server installation area. When the server is installed a set of files describing the various GV products is placed in this area for use by clients. In addition, when clients create jobs any job specific files created by the client can be uploaded to this area. In order to retain the ability to re-run historic jobs files cannot be overwritten by clients. If a new file is to be sent to the server it must have a different name to all existing files.

Access to the file transfer area is obtained using the path ***downloads***.

Directory listings can be obtained so that clients can establish the files, products etc that are installed on the server.

If an HTTP GET operation is done on a directory in this area a plain text directory listing is returned by the server. For example, with our server listening on address example.com port 35061 if we issue an HTTP GET to address ***http://example.com:35061/downloads*** the server might return the following text:-

```
FimsConfig.xml

FimsConfig.xsd

products/

wsdl/
```

This is a directory listing of the root directory (downloads) available to clients. There are two files (FimsConfig.xml and FimsConfig.xsd) and two directories (*products* and *wsdl*). Directory names end with a forward slash (/).

If you do an HTTTP GET to address ***http://example.com:35061/downloads/products*** the server might return the following text:-

```
Alchemist_File/
```

This is a directory listing of the downloads/products directory and shows one directory (Alchemist_File).

To download a file from the server a client must do an HTTP GET on the path to the required file. So, following from our example directory listing above an HTTP GET to ***http://example.com:35061/downloads/FimsConfig.xml*** would return the file FimsConfig.xml in the contents of the response to the HTTP GET.

To upload a file to the server the client sends an HTTP PUT request to the server. The URL the request is sent to will be the server downloads path concatenated with the desired path the file is to be written to. For example to upload the file test.txt to the root of the client access area the client would issue an HTTP PUT to address ***http://example.com:35061/downloads/test.txt*** where the message contents are the data to be written to file test.txt. To write a file to another path in the file transfer area the write path is simply included in the request URL. So to write to downloads/upload/testdata.txt one would issue an HTTP PUT to http://example.com:35061/downloads/upload/testdata.txt

Note that the directory path must exist before a file can be uploaded. See 3.3 below for a mechanism to create server directories if files are to be uploaded to new directories. Note also that existing files cannot be overwritten.

If an HTTP GET operation is successful the server returns response code 200 (OK) or 204 (No Content). If an HTTP GET operation is unsuccessful the server returns error code 500 (Internal Server Error).

If an HTTP PUT operation is successful the server returns response code 201 (Created). If an HTTP PUT operation is unsuccessful the server returns error code 500 (Internal Server Error).

3.3 Command Operations

The command operations are intended to allow clients to manipulate server files and directories.

Currently the only supported command is ***mkdir*** which allows clients to create directories in the server file transfer area (see above). To create a directory the client issues an HTTP GET command to the server commands/mkdir path with the name of the directory appended.

For example,

In order to create directory “testDir” in the file transfer area, send an HTTP GET command to the server.

In the example the server name is “example.com” and listening port is 35061.

http://example.com:35061/commands/mkdir/testDir

3.4 WSDL Operations

The WSDL files for the FIMs services can be obtained using an HTTP GETs to the server wsdl/Fims directory. The only service currently supported is the transform service, the WSDL file for this can be obtained with an HTTP GET ***to wsdl/Fims/transform***. This will return a zip file containing all the wsdl and schema files for the FIMs transform service.

If a zip file is not desirable then the individual files are also accessible from this directory, just append the name of the required file to the URL. For example to download the transform service WSDL do an HTTP GET from

http://example.com:35061/wsdl/Fims/transformMedia-V1_0_7.wsdl

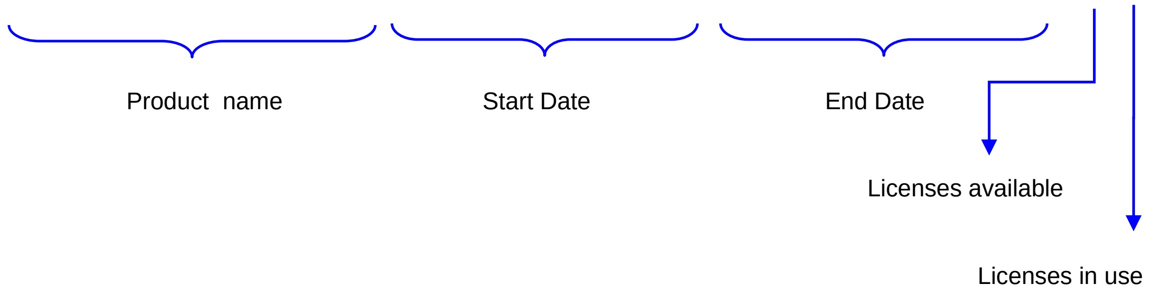
Note that you can find out the names of the files by doing an HTTP GET from ***downloads/wsdl/fFims***

3.5 License Queries

The xFile solutions are licensed. The licenses are managed by a separate license server. To find out what licenses are installed on the server do an HTTP GET from ***licenses.*** This will return a plain text listing of the available licenses.

For example, for our example server doing an HTTP GET from http://example.com:35061/licenses might return the following strings:-

```
Alchemist_File_Version_3.0.0.0, Tue 1. Jan 00:00:00 2013, Sat 31. Dec 23:59:59 2016, 2, 0,
Alchemist_File_Version_4.0.0.0, Tue 1. Jan 00:00:00 2013, Sat 31. Dec 23:59:59 2015, 4, 0,
```



This shows that there are two licenses for Alchemist File version 3.0.0.0 and four licenses for Alchemist Live 4.0.0.0 installed.

4. Product File Organisation

The files describing GV File products are in the ***products*** directory of the server file transfer area. For each installed product there will be a separate directory within the ***products*** directory, for each version of a given product there will be a separate directory within that products directory. For example, suppose we have the following directory structure:-

```
└───downloads
    └───products
    ├───Product_One
    │       ├───1.0.0.0
    │       └───2.0.0.0
    └───Product_Two
            ├───3.0.0.0
            └───4.0.0.0
```

We have two installed products Product_One and Product_Two.

Product_One has two installed versions – 1.0.0.0 and 2.0.0.0.

Product Two has two installed versions 3.0.0.0 and 4.0.0.0

Within this product version directories there will be a number of files that describe the product and how to control it. In addition there are default profiles that can be used to set up jobs for that product. When clients create profiles to be used with a product they will also be added to this area.

If we now look at the directory structure of Product One version 1.0.0.0 we get:-

```
└───downloads
      └───products
          ├───Product_One
          │   ├───1.0.0.0
          │   │     │   Product_OneSolution.xml
          │   │     │
          │   │     ├───client
          │   │     │       ApiModifiers.xml
          │   │     │       ApiModifiers.xsd
          │   │     │       BehaviourModifiers.xsl
          │   │     │       ControlDescriptionPreprocessor.xsl
          │   │     │
          │   │     ├───default_profiles
          │   │     │       default.1.xml
          │   │     │
          │   │     ├───icons
          │   │     │       audio.png
          │   │     │       input.png
          │   │     │       metadata.png
          │   │     │       output.png
          │   │     │       overview.png
          │   │     │       video.png
          │   │     │       video_conversion.png
          │   │     │       video_utilities.png
          │   │     │
          │   │     ├───sink_profiles
          │   │     │       mcSink.xml
          │   │     │
          │   │     ├───source_profiles
          │   │     │       mcSource.xml
          │   │     │
          │   │     |───transform_profiles
          │   │     │        transform.xml
          │   │     │
          │   │     └───user_profiles
          │   │           transform.xml
          │   └───2.0.0.0
          └───Product_Two
```

There will be one XML file in the root of the product version directory.

In our example above this file is called: **Product OneSolution.xml**

The path to this file is downloads/products/Product_One/1.0.0.0/Product_OneSolution.xml.

This file describes the organisation and control parameters that **Product_One** takes. This file is supplied by Grass Valley and should not be edited in any way as this may render it unusable.

Within the product version directory there are a number of directories supplying supporting files for the product solution XML file mentioned above. One of these directories is named user_profiles and this is where user generated profile files should be kept. There is also a directory named default_profiles that contains default profiles provided by Grass Valley. Profile filenames can be up to 64 characters long.

When a client wishes to send a job request to the GV File server, it must ensure that all the profile files necessary for that job are present on the server ***before*** the job request is sent. If profile files are not available at the time the job request is sent the job request may be rejected or the job may fail. Thus, clients should always use HTTP PUT operations to the server download area to place any missing profile files in the required locations before issuing job requests. The profile file locations specified in the job request should match the address given in those HTTP PUT operations.