MovieLabs Data Distribution Framework (MDDF) Services API

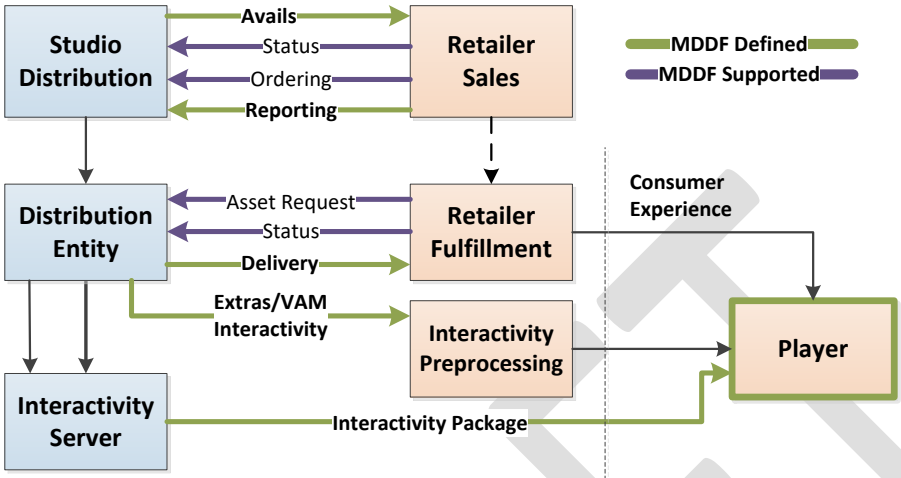
CONTENTS

REVISION HISTORY

Version	Date	Description
1.0	TBD	First Release

1 INTRODUCTION

The MovieLabs Data Distribution Framework (MDDF) Services API is a secure RESTful-style interface for exchange of distribution metadata.



1.1 Document Organization

This document is organized as follows:

1. Introduction—Background, scope and conventions
2. API Overview
3. HTTP and TLS – Protocol specification for HTTP and TLS
4. Authentication and Authorization – Specification of authentication and OAuth 2.0-based authorization
5. Endpoints – Definition endpoints (URLs) for REST and Atom
6. REST Interface – Definition of RESTful interface
7. Atom Interface – Definition of how Atom is used for notifications
8. Avails API – API specifics for EMA Avails
9. MEC API – API specifics for Media Entertainment Core (MEC) metadata

1.2 Document Notation and Conventions

The document uses the conventions of Common Metadata [CM].

1.3 Normative References

[Manifest] TR-META-MMM MovieLabs Media Manifest Metadata, v1.6, http://www.movielabs.com/md/manifest

[CM] TR-META-CM MovieLabs Common Metadata, v2.5, http://www.movielabs.com/md/md

[EIDR-TO] *EIDR Technical Overview*, November 2010. http://eidr.org/technology/#docs

[MMC] *Media Manifest Core (MMC)*

[MEC] *Media Entertainment Core (MEC) Metadata, version 2.5. www.movielabs.com/md/mec and http://entmerch.org/programsinitiatives/ema-metadata-structure.*

[Avails] *Entertainment Merchant's Association (EMA) Avails, Version 2.2. www.movielabs.com/md/avails*

[RFC3629] IETF RFC 3629, UTF-8, a transformation format of ISO 10646, https://tools.ietf.org/html/rfc3629

[HTTP] IETF RFCs 7230-7235
IETF RFC 7230, Hypertext Transfer Protocol —HTTP/1.1: Message Syntax and Routing, https://tools.ietf.org/html/rfc7230
IETF RFC 7231, Hypertext Transfer Protocol —HTTP/1.1: Semantics and Content, https://tools.ietf.org/html/rfc7231
IETF RFC 7232, Hypertext Transfer Protocol —HTTP/1.1: Conditional Requests, https://tools.ietf.org/html/rfc7232
IETF RFC 7233, Hypertext Transfer Protocol —HTTP/1.1: Range Requests, https://tools.ietf.org/html/rfc7233
IETF RFC 7234, Hypertext Transfer Protocol —HTTP/1.1: Caching, https://tools.ietf.org/html/rfc7234
IETF RFC 7235, Hypertext Transfer Protocol —HTTP/1.1: Authentication, https://tools.ietf.org/html/rfc7235

[RFC7617] IEFT RFC 7617, The 'Basic' HTTP Authentication Scheme, https://tools.ietf.org/html/rfc7617

[RFC3986] IETF RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, https://www.ietf.org/rfc/rfc3986.txt

[RFC4346] IETF RFC 4346. The Transport Layer Security (TLS) Protocol Version 1.1. https://www.ietf.org/rfc/rfc4346.txt

[RFC5246] IETF RFC 5246, The Transport Layer Security (TLS) Protocol, Version 1.2, https://tools.ietf.org/html/rfc5246 as updated by Errata.

[TLS13-draft] IETF DRAFT RFC, The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-19, https://tools.ietf.org/html/draft-ietf-tls-tls13-19

[RFC4287] IETF RFC 2460, The Atom Syndication Format, December 2005. https://tools.ietf.org/html/rfc4287

[RFC5023] IETF RFC 5023, The Atom Publishing Protocol, October 2007, https://tools.ietf.org/html/rfc5023 as modified by Errata 1304 and 3207

[RFC6749] IETF RFC 6749, The OAuth 2.0 Authorization Framework, https://tools.ietf.org/html/rfc6749, as modified by Errata.

All Common Metadata and Media Manifest references are included by reference.

1.4 Informative References

[MMDelivery] BP-META-MMMD, Using Media Manifest, File Manifest and Avails for file Delivery (Best Practice), v1.1, www.movielabs.com/md/manifest

[Atom] Atom Enabled web site, http://www.atomenabled.org/

[REST] Fielding, Roy, "Chapter 5, Representational State Transfer (REST)", *Architectural Styles and the Design of Network-based Software Architectures*, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

1.5 Identifiers

Identifiers must be universally unique. Recommended identifier schemes may be found in Common Metadata [CM] and in DECE Content Metadata [DECEMD].

The use of Entertainment Identifier Registry identifiers (www.eidr.org) is strongly encouraged. Please see [EIDR-TO].

Best practices for identifiers can be found in Best Practices for Delivery [MMDelivery].

1.6 Rationale

Every studio or service provider must communicate with every retailer service that offers their content. This creates the potential of *s* x *r* interfaces (*s* studios and *r* retailers).

To reduce the number of unique means of communication, the industry has created specifications such as EMA Avails for Avail, Media Entertainment Core (MEC) for metadata, Media Manifest Core (MMC) for delivery information, and Cross Platform Extras (CPE) for interactivity. Although these specifications are not universally implemented, they are the accepted direction for the industry.

Prior to this specification there was not standard means for communicating data. Common practices include email, ftp (and other file transfers), and phone conversations. Another method is for the retailer to host a portal for manual uploads. Each has its merits, but none offers all of automation, security, accuracy and auditability. Portals are particularly problematic because they require manual entry of data. With hundreds of partners, studios complain of being "portaled to death" and retailers complain about timeliness and inaccuracy.

The solution is to establish an API that supports automation. Under the coordination of MovieLabs, the retailers, studios, Entertainment Merchant's Association and the Digital Entertainment Group launched an effort to, "Define a standardized API is essential to maximize

automation: Reduce cost, reduce errors, reduce time to market, improve visibility, increase quality, and support business models that increase revenue."

This specification is the result of that effort.

Initial use cases defined for this activity include:

- Avails submission
- Content ordering
- Fulfillment status
- Ingest status & QC feedback
- Storefront status
- Asset delivery/asset updates
- Manifest updates
- POS/revenue reporting

Goals included

- Help justify business case for implementation
- Speed/efficiency
- Reduction in errors
- Security
- Reduction in costs
- Reduction in latency
- Ability to support new content and business models

1.7 Status

This specification is not completed and ready for implementation. Although tested, we anticipate that additional implementation experience will yield recommendation for changes. Implementers should anticipate one or more revisions. Reasonable measures will be taken to ensure changes are backwards compatible. See Backwards Compatibility Best Practices in [CM]

2 API OVERVIEW

2.1 Scope

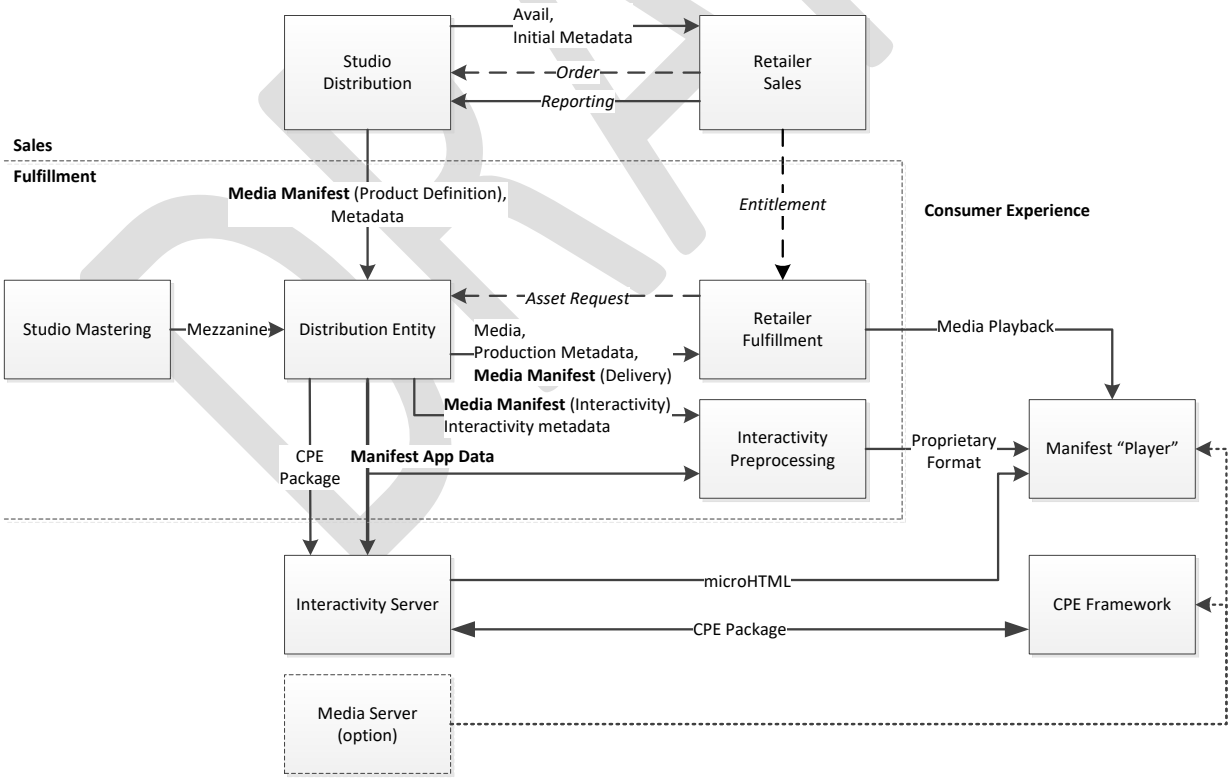
The scope of the API is the exchange of all distribution metadata interfaces associated with the MovieLabs Digital Distribution Framework (MDDF); such as Avails, Media Manifest, Common Metadata. This API is not intended for media (i.e., video, audio, subtitles, images, apps, games, etc.).

As a general rule, the data exchanged are the XML documents associated with MDDF specification such a EMA Avails [Avails], Media Entertainment Core [MEC], Media Manifest Core [MEC]. The base specifications for the data are Common Metadata [CM], Common Media Manifest [CMM].

Cross-Platform Extras (CPE) are a series of specifications covering interactivity. As of this authoring, it has not been determined whether CPE is included in the MDDF API.

There are several data formats that have not yet fully specified, most notably title acceptance, asset ordering and reporting. When available, it is anticipated that these will be included.

MDDF interfaces are shown in the following illustration (more detailed version of the diagram in Section 1).



3 HTTP AND TLS

3.1 HTTP

3.1.1 Specification Compliance

Servers and clients shall comply with HTTP/1.1 in accordance with [HTTP].

3.1.2 Character Encoding

XML elements shall use UTF-8 character encoding in accordance with [UTF-8].

3.1.3 XML and JSON

Clients must include "Accept: application/xml".

Servers must support XML.

Client may include "application/json".

Servers may support JSON.

3.1.4 Range-GET for multiple resources

Range-GET need not be supported by servers or clients.

Sorted GET requests not supported—Resources are assumed to be returned in an undefined order.

3.1.4.1 Pagination

Pagination is supported as described in this section. Note that bulk operations are not an immediate priority, so implementation of this functionality may be deferred. [CHS: We need a separate section for this kind of information.]

For pagination, all resources are assumed to a fixed but unspecified ordering that can be referenced with a numerical index into the array of resources. That is, if there are *n* resources, there can be referenced in the range 1-*n*[1]. The Clients request via GET one or more resources by referencing an index or limited range of indices.

To obtain a range of resources, the Client performs a GET using the endpoint with a query string including the following:

- `offset` – offset from first record (0 offset represents the first record)
- `limit` – Maximum number of records that can be returned (note that only on the last request will < limit resources be returned)

[1] Note that resource references are relative (using `offset`) and there is no actual recording numbering.

The Server returns all available records inclusive of the record associated with `offset` through the record associated with `offset` + `limit` records corresponding with `offset` (first record is `offset`=0). For example, if `offset`=10 and `limit`=5, the eleventh through fifteenth will be returned, assuming those records are available.

When a range is specified (`offset` + `limit`), all records that exist within that range (which may be less than the limit) are returned. That informs the Client that all records have been returned. If no resources are available in that range, HTTP response code 404 is returned, also informing the Client that no more records are available. That is, the Client continues to perform queries, increasing offset by limit until either fewer that limit resources are return, or a 404 response is returned.

For example, if 18 record are available, it might look something like this:

```
GET https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/?offset=0&limit=5
(5 avails returned)

GET https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/?offset=5&limit=5
(5 avails returned)

GET https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/?offset=10&limit=5
(5 avails returned)

GET https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/?offset=15&limit=5
(3 avails return)
```

3.1.4.2 Resource Count

To determine the number of resources, a Client may (optionally) perform a GET to the **getcount** endpoint (e.g., `https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/getcount`). The Server returns the number of Resources. For example:

```
<ResourceCount>
  <NumberOfResources>18<NumberOfResources>
</ResourceCount>
```

As resources can be added or deleted, the number of resources can change between resource GETs and is therefore unreliable.

3.1.5 Cache Support

Servers must include "ETag:" in GET responses. [CHS: I included this because it's commonly used and allows clients to know which mechanisms to use. I think this improves interoperability. However, it might be best to leave this to implementation.]

Clients include "If-None-Match:" in GET requests

Servers respond with 304 if matched

3.1.6 Error Response

Invalid client requests result in an error message being returned.

The error message is designed to provide some degree of automated processing at the client. The error message is also designed to provide useful information to operators and engineers who need to determine what went wrong. Consequently, the returned error information includes both error codes and human-readable text.

Server implementers are encouraged to provide useful guidance in error messages.

3.1.6.1 Schema and data definition

Errors are returned in an Error element as follows

Element	Attribute	Definition	Value	Card.
Error				
ErrorCode		A numeric or text label associated with the error. These codes must be documented.	xs:string	
ErrorMessage		A human-readable text description of the error.	xs:string	
Resource		Resource URL associated with Resource associated with request. This will be the same URL that was used in the request.	xs:anyURI	
MoreInfo		Any additional information the server implementer believe the client implementer will find useful.	xs:string	0..1
Ref		A transaction reference for debugging (e.g., retrieving logs).	xs:string	0..1

3.1.6.2 Error Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Error>
   <ErrorCode>XMLValidation</ErrorCode>
   <ErrorMessage>Avails XML Document does not comply with Avails v2.2 Schema</ErrorMessage>
   <Resource>api.craigsmovies.com/mddf/v1/craigs.movies.com/avails/md:alid:eidr-s:abcd-1234-abcd-
1234-abcd-m</Resource>
   <MoreInfo>Missing required field AvailsLicensor</MoreInfo>
   <Ref>1234abcde</Ref>
</Error>
```

3.1.7 Connection Reuse

To avoid connection establishment overhead, especially with respect to reestablishing TLS, persistent HTTP connections should be supported on servers and used by clients in accordance with [HTTP].

3.1.8 HTTP Response Codes

The following response codes must be generated by servers and supported by Clients. Other error codes may be generated by Servers and must be accepted by Clients.

Code	Interpretation
200 OK	Request received and processed.
201 Created	Object created (POST). Location information is returned
204 No Content	A DELETE is performed on a nonexistent resource
304 Not Modified	Content has not been modified since previous request with same ETag
400 Bad Request	Improperly formed request (e.g., bad XML)
401 Unauthorized	Client failed to authenticate properly and cannot access resource. Can be repeated after authentication.
403 Forbidden	Client authenticated properly, but is attempting to access a resources for which the client does not have rights. Do not repeat request
404 Not Found	Attempting to access a resource that does not exist
5xx Server Error	Try again

[CHS: One school of thought is to limit the error codes to a list (like above), and prohibit other response codes. One variation is to includes unlisted response codes in <error>. Thoughts?]

3.2 TLS

Connections between clients and servers shall use TLS 1.2 in accordance with [RFC5246].

Note that the intent is to adopt TLS 1.3 when available. Draft documentation can be found in [TLS13-draft] and successive drafts.

4 AUTHENTICATION AND AUTHORIZATION

4.1 Overview

4.1.1 Terminology used in this section

The following terms are used in this section. Many of these correspond with OAuth2 terms. In those cases, the formal definition is in [RFC6749].

Term	Meaning
User	A human being accessing some service
Resource Owner	User that provides/owns Resource data (e.g., Studio with Avails)
Resource Server	API server (e.g., Retailer providing Avail service)
Client	Application acting on behalf of a Resource Owner accessing a Resource Server (e.g., Application at Studio supporting Avails)
Authorization Server	Server that delivers appropriate access tokens to the Client after authenticating Resource Owner
Portal	A web user interface provided in conjunction with the Resource Server and Authorization Server to provide authorization information to a user (e.g., Portal at Retailer that supports Avails).

4.1.2 Requirements

It is necessary to authenticate both Clients and Resources.

For purposes of Authorization, Resource Owners grant permissions to Clients. Clients can be same organization as Resource Owner, or distinct. There may be multiple Clients authorized to act on behalf of Resource Owner.

4.1.3 Guiding Principles

We are striving for strong security. In that spirit, the following principles apply:

- Protect long-lived secrets by using them sparingly (infrequent use of passwords, relatively short lived access tokens)
- Authentication secrets not reused between organizations (no “password” sharing)
- Secrets always passed through secure channels

4.2 Authentication

Authentication is required to register clients with OAuth2 Authorization Servers and and to obtain OAuth2 Authorization Codes.

4.2.1 Authentication Methods

A service (e.g., Portal or OAuth2 Authentication Server) can authenticate a User via any mechanism is feels is appropriate. Common methods include username and password and biometrics (fingerprints).

When practical, Two Factor Authentication is recommended.

4.2.2 Client Authentication

Client operators must register clients with OAuth2 Authentication Server. Part of this registration process requires a user to be authenticated. This process is defined in Section 4.3.2.

Note that a client can act on behalf of multiple Resource Owners (e.g., service provider servicing multiple studios).

4.2.3 Resource Owner User Authentication

User authentication is needed for a Resource Owner's Client User to obtain access to resources on a Resource Server. More specifically, User authentication is a necessary part of the process to obtain an Authentication Code. The process of obtaining an Authentication Code is described in Section 4.3.3

4.3 Authorization

Authorization is based on OAuth 2.0 (OAuth2) as defined in [RFC6749]. Access Tokens are obtained in accordance with *Authorization Code Grant* as defined in Section 4.1 of [RFC6749].

4.3.1 Authorization Overview

4.3.1.1 Terminology

The following terms are provided here for convenience, but the normative definitions are in the RFC:

- Authorization Grant – a secret presented to obtain an Authentication Token and, optionally, a Refresh Token
- Authentication Tokens – A secret presented to access Resources
    - Includes 'scope' of what can be accessed
    - Short lifespan reduces impact of compromise
    - Can be revoked
- Refresh Tokens – A secret presented to obtain Authentication Token
    - Facilitate short-life Authentication tokens (can always refresh)
    - More secret than Authentication token

- Can be revoked

4.3.1.2 Note on Methods for Obtaining Access Tokens

OAuth2 provides several methods for obtaining Access Tokens. The MDDF API currently supports only Authorization Code Grant as defined in Section 4.1 of [RFC6749].

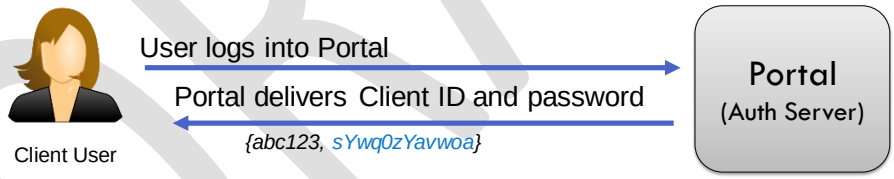
The following is provided for information purposes in the spirit of providing rationale for reviewers, and to provide data for future modifications of the specification: Implicit Grant (Section 4.2) is considered insufficiently secure (access tokens too long-lived). Resource Owner Password Credentials Grant (Section 4.3) is would require sharing passwords between organizations, a weak security practice that violates one of our guiding principles. Client Credentials Grant (Section 4.4) is not supported because Client credentials do not necessarily identify Resource Owner

4.3.2 Client Registration

To obtain access to an OAuth2 Authentication Server, clients must be registered with that server ([RFC4769], Section 2. In this case, the server is operating by the Resource Owner (e.g., in the case of Avails, a retailer). The registration process provides the client with a Client Identifier and Client Secret.

First, a user associate with the client must authenticate to the Authentication Server. The assumed model is that client user authenticates (e.g., uses username and password) to log into a Portal capable of generating Client Identifiers and Client Password. The Portal performs the registration process and delivers the Client Identifier and Client Password to the user.

This is illustrated in the following picture:



Note that a Client can act on behalf of multiple Resource Owners (e.g., service provider servicing multiple studios).

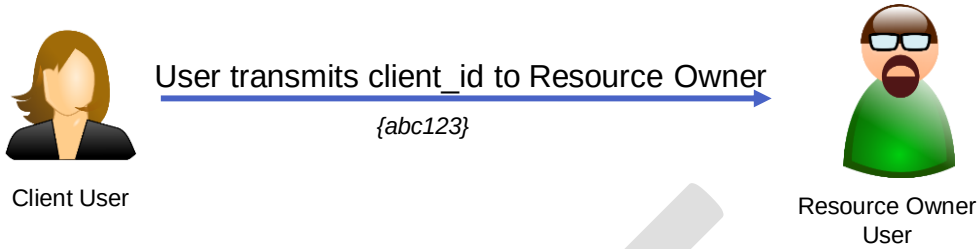
4.3.3 Obtaining Authentication Code

The Authentication Code is information needed to obtaining Access Tokens, and optional Refresh Tokens.

The Authorization Grant process follows the Authorization Code Grant approach. However, to support studio, service provider and retailer workflows that method of obtaining an Authentication Code may appear slightly different.

4.3.3.1 Client Identifier

The Client Identifier is needed to obtain an Authorization Code. The Client User must transmit the Client Identifier to the Resource Owner User.



4.3.3.2 Access Token Scope

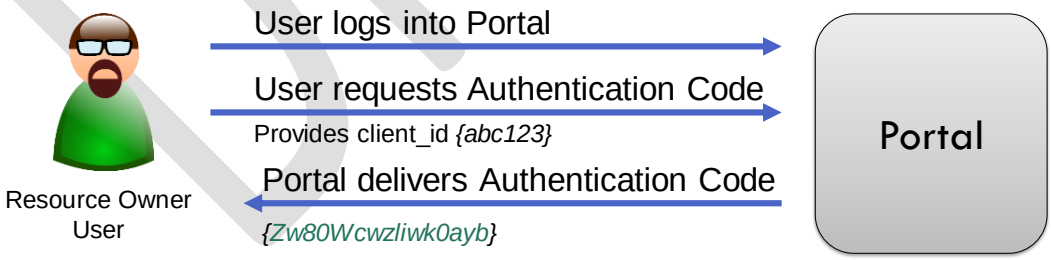
Scope defines which resources the Client can access as defined in [RFC6749], Section 3.3. The following terms are used to define scope

Scope Token	Scope
avail	Avail API as defined in Section 8
mec	MEC API as defined in Section 9
TBD	

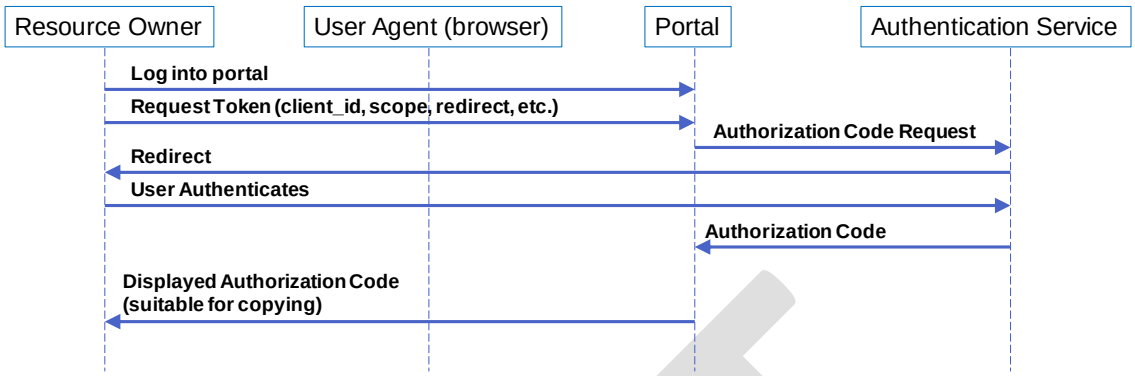
[CHS: Do we want to define scope more exactly (e.g., avail:rest, avail:atom, or avail:read, avail:write)]

4.3.3.3 Portal Methods

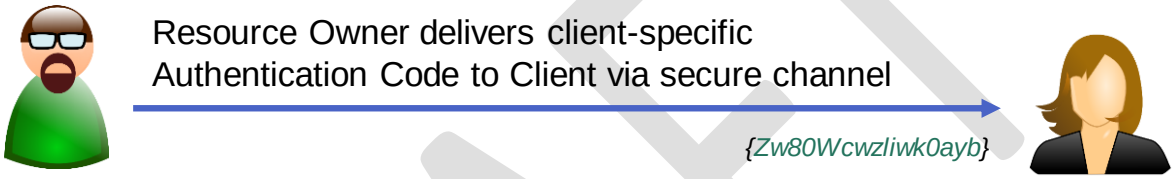
The simplest method for obtaining an Authentication Code is via a Portal. First, the Resource Owner User (e.g., studio user), logs into a Portal, indicates what information will be accessed (i.e., scope), and retrieves the Authentication Code (e.g., copies it from a screen provided by the Portal).



This flow is illustrated here:

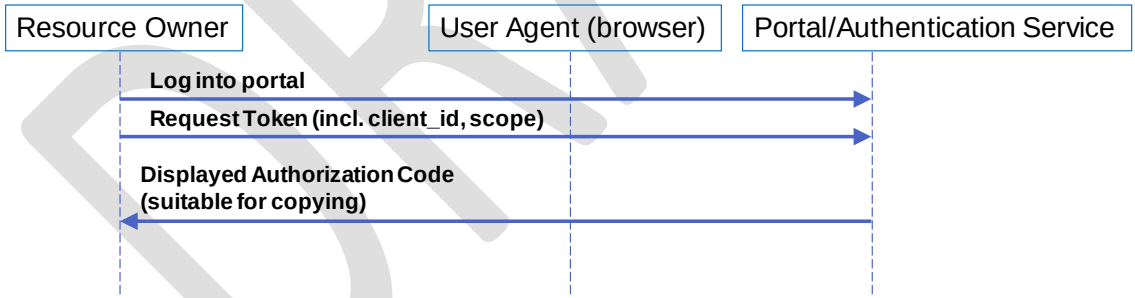


Once the Authentication Code is retrieved, it is passed to the Client user via secure method.



Although this specification does not define the method, good security practices are required. Studios, service providers and retailers have secure channels used for confidential data. These channels can be used. Unsecured email is an unacceptable method.

In some cases, the Portal and Authentication Server are integrated. In this model, the OAuth2 protocol is unnecessary and the exchange would look like this:

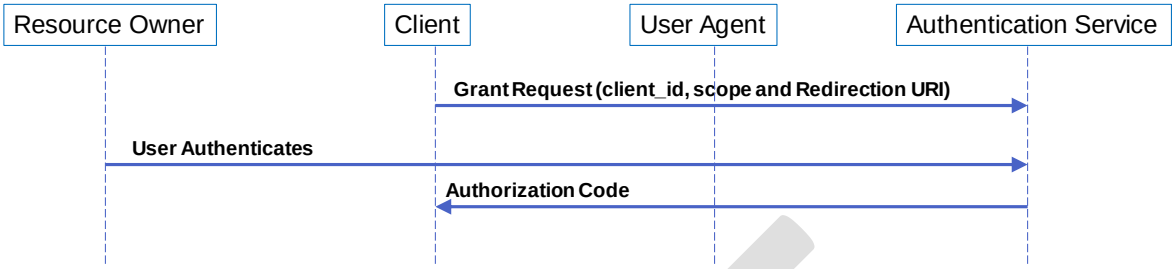


This flow is not strictly OAuth2 flows, but might make sense for some service implementers. Note that because it is non-standard it might be difficult to integrate OAuth2 libraries into this model.

4.3.3.4 Client-based methods

Following is the most basic method for obtaining an Authorization Code. This does not require a Portal. It is no clear whether implementers will build a client service for the rare operation of obtaining an Authorization Code.

The flow is illustrated here:

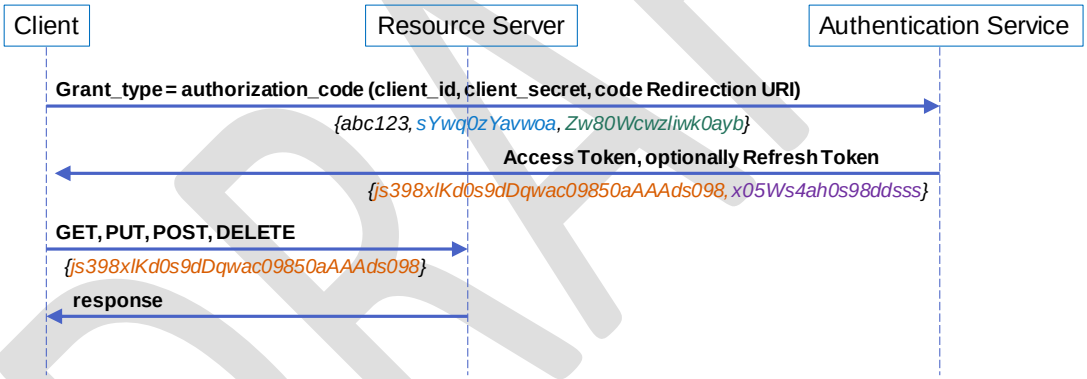


4.3.4 Authorization Process

The Authorization process involves obtaining Access Tokens, and optionally Refresh Tokens; and using those tokens to access Resources. This process is defined in RFC 6749.

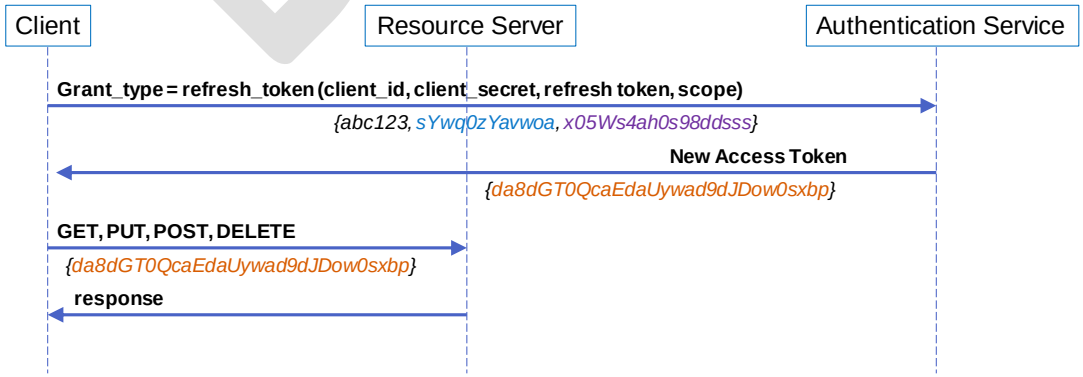
The process has the prerequisite that the Client hold a valid API Key and Authentication Code.

The flow for obtaining and using an Access Token is as follows:



Clients must support Refresh Tokens, as defined in[RFC6749] Section 1.5. Servers may support Refresh Tokens.

The flow for accessing a resource is as follows:



5 ENDPOINTS

5.1 Approach

Endpoints are defined by URLs constructed as follows

"https://" <Base URL> "/" <version> "/" <partner> "/" <function-specific path>

For example,
https://api.craigsmovies.com/mddf/v1/sofaspudfilms.com/avails/md:apid:eidr-s:B65C-7EC9-1F9F-D611-F84F-0

5.2 Base URL

Each Service Provider provides a Base URL from which REST endpoints are constructed. Base URLs include the domain where the service is hosted.

An example of a Base URL that might be provided by Sofa Spuds Studio is https://api.craigsmovies.com/mddf.

5.3 Version

Version indicates the version of the API. Within a version, the API should robustly accept content. Common Metadata [CM], Section 1.6 provides best practices for maximizing compatibility.

An example of a version is v1. Note that versions like v1.1 are discouraged as this implies the API is changing too frequently.

When a Service Provider begins supporting a new API version, it must continue to support the old API for some period, unless partners can transition immediately. That is, the Service Provider will support two or more versions of the API during that transition period. Recommended period is 6 months.

5.4 Function-specific Path

The function-specific path uniquely identifiers the resource (for REST API), Atom endpoint or service function.

The function-specific path begins with the service, unless the function spans all services. An example of a service is, "avails/".

5.4.1 REST API Path

When the function-specific path contains only the service, REST requests apply to all resources in that service. Note that bulk services have been prioritized lower than individual operations and might not be implemented immediately.

For example,
`https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails` refers to all Avails resources for that Service Provider and that Partner.

When the function-specific path contains the service plus a resource identifier, REST requests apply to the identified resource.

For example,
`https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/md:alid:eidr-s:B65C-7EC9-1F9F-D611-F84F-0` refers to the associated with that ALID.

5.4.2 Atom API Path

The only defined Atom endpoint is the location of the Service Document. It can be found with a Function-Specific Path of "avails_atom".

For example, a GET of
`https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails_atom` returns the service document.

The Service document returns URLs for the Feed documents. These URLs are at the discretion of the Service Provider.

5.4.3 Other Paths

Special paths are defined for specific purposes. These use neither REST nor Atom semantics.

Special paths include the following.

Path	Function
getcount	Returns ResourceCount indicating the number of resources associated with an endpoint. For example, `https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/getcount`
getstatus	Returns a status object associated with the resource. These are defined for the specific service. For example, of obtaining status on multiple resources is as follows: `https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/getcount`, and obtaining status on a single resource is as follow: `https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/md:alid:eidr-s: B65C-7EC9-1F9F-D611-F84F-0/getstatus`
TBD	

This list may be expanded.

5.4.3.1 ResourceCount

The ResourceCount object is defined as follows:

Element	Attribute	Definition	Value	Card.
ResourceCount				
NumberOfResources		The number of resources currently associated with the endpoint.	xs:nonNegativeInteger	

6 RESTFUL WEB SERVICE

This API defines a RESTful web service, based on Roy Fielding's doctoral dissertation [REST].

The API uses four basic HTTP methods: GET, POST, PUT and DELETE. These correspond with the CRUD function of Read, Create, Update and Delete respectively.

- GET is used to retrieve one or more resources
- POST creates one or more resources
- PUT updates an existing resource or replaces resources
- DELETE removes one or more resources

These methods use the endpoints defined in Section 5.4.1**Error! Reference source not found.**.

[CHS: I think this should be enough. I'm assuming people know what REST is or can figure it out. If not, what should I include here?]

7 ATOM INTERFACE

The Atom interface provides a means for clients to determine which objects have been updated by the server and to status progress in the processing of those objects.

Feeds are provided to optimize around the urgency of status with exceptions being highest priority and general status being the lowest.

7.1 Why Atom

Only a small percentage of resources are updated each week (e.g., in Avails on the order of fifty updates against hundreds of thousands of entries). Although polling all resources would function, even with caching it would not be efficient. A better solution is for the server to post a list of updates which the client could status.

There are numerous proprietary solutions, however since these solution favor one company over another, the decision was made to use a standard interface. Atom was the best candidate identified.

7.2 Implementation Requirement

Servers are required to implement Atom in accordance with this specification.

Clients are not required to implement Atom, although highly recommended with the number of resources becomes large. We don't have a specific number for 'large' but, when there are more than several hundred resources, implementation should be considered.

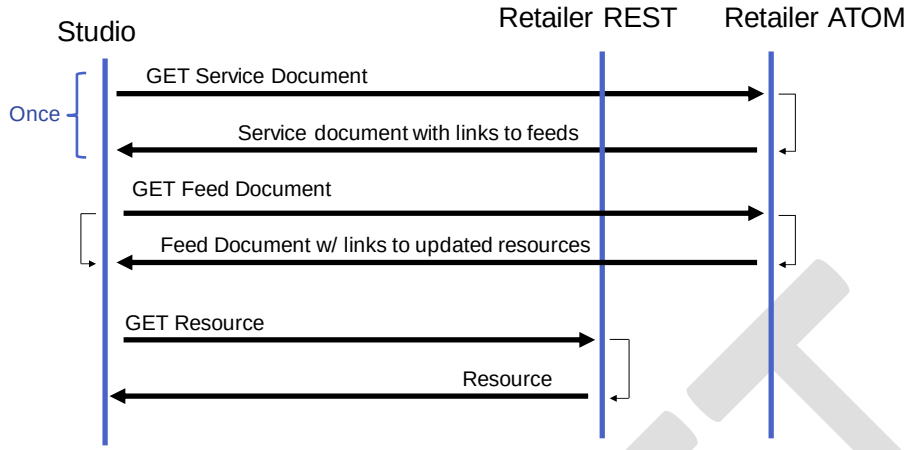
7.3 Approach

The information provided by the RESTful interface is the authoritative source. The Atom feed provides pointers to the RESTful interface when updates of interest occur.

7.3.1 Protocol

Protocols used shall be in compliance with Protocol and Endpoint requirements of Sections 3, 4 and 5.

An Atom exchange is illustrated here:



7.3.2 Atom Compliance

Atom will be implemented in accordance with the Atom Publishing Specification (AtomPub) [RFC5023] and the Atom Syndication Format [RFC4287] as constrained by this specification.

7.3.3 Service Document

The Atom Service Document defines the feeds.

7.3.3.1 Obtaining the Service Document

The Service Document is retrieve with a GET from the well-known address provided by the Service as found in Section 5.4.2.

Each application has its own Service Document. For example, there are unique Avails Service Documents, MMC Service Documents, and MEC Service Documents [CHS: Everyone agree?].

7.3.3.2 Service Document Format

The Service Document shall conform with RFC 5023 [RFC5023] and contain the following

- A service element
- A workspace element with the title defined by the practice for the particular application. For example, an Avails Service Document has type="Avails".
- A collection element for each feed as defined in the Feeds Section (Section 7.3.4).

An example service document is here:

```
<?xml version="1.0" encoding='utf-8'?>
<service xmlns="http://purl.org/atom/app#">
```

```
   <workspace title="Avails" >
      <collection title="Exception"
href="https://api.craigsmovies.com/mddf/v1.0/avails_atom/exception.atom" />
      <collection title="Status" href="
https://api.craigsmovies.com/mddf/v1.0/avails_atom/status.atom" />
      <collection title="Progress" href="
https://api.craigsmovies.com/mddf/v1.0/avails_atom/progress.atom" />
   </workspace>
</service>
```

7.3.4 Feeds

A Feed Document provides links to Resources that merit attention.

7.3.4.1 Obtaining a Feed Document

The Feed Document of a given type is retrieved with a GET from address provided in the Service Document with @title matching the label for the feed type.

Each application has its own Service Document. For example, there are unique Avails Service Documents, MMC Service Documents, and MEC Service Documents [CHS: Everyone agree?].

7.3.4.2 Feed Document Format

A Feed Document shall conform with RFC 4287 [RFC4287] and have the following

Element	Usage
title	Title of feed ("Exception", "Status", "Progress")
link	Link to this feed
id	Unique ID for this entry
updated	Date and time when feed was updated
entry	One entry for each resource
entry/title	Optional: Title. e.g., ShortDescription from Service
entry/link	Link with href attribute referring to resource
entry/id	ID for resource (e.g., ALID for Avail)
entry/updated	Date and time resource was updated
entry/category	TBD

Following is a sample Exception Feed document:

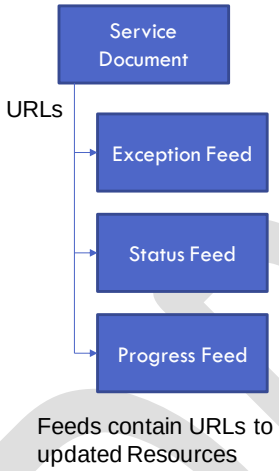
```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>Avails Exception: craigsmovies.com</title>
    <link href="https://api.craigsmovies.com/mddf/v1.0/avails_atom/exception.atom" />
    <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa61</id>
```

```
    <updated>2017-03-16T03:43:02Z</updated>
    <entry>
        <title>Unbelievable Example Movie</title>
        <link href="https://api.craigsmovies.com/mddf/v1.0/avails_atom/md:alid:eidr-s:1234-5432-
abcd-efgh-abcd-l”>
        <id>md:alid:eidr-s:1234-5432-abcd-efgh-abcd-l</id>
        <updated>2017-03-10T17:46:02Z</updated>
    </entry>
    <entry>
        <title>Sequel to Unbelievable Movie</title>
        <link href="https://api.craigsmovies.com/mddf/v1.0/avails_atom/md:alid:eidr-s:abcd-5432-
abcd-efgh-abcd-l”>
        <id>md:alid:eidr-s:abcd-5432-abcd-efgh-abcd-l</id>
        <updated>2017-03-16T03:43:02Z</updated>
    </entry>
</feed>
```

7.3.4.3 Feed Types

There are three types of Feeds required.

These feeds are referenced by the Service Document as illustrated here:



Feeds are defined as follows:

Feed Type Label	Purpose	Update Latency	Recommended Polling Frequency
Exception	Contains records associated with events that require intervention	ASAP	5 minutes or faster
Status	A change of status that likely requires non-urgent review or action.	1 hour or less	2 hours or faster
Progress	Any other update, such as an update of status that is purely informational	24 hours or less	48 hours or faster

Feed Type Label is the value assigned to `/service/collection/@title` in the Service Document

7.3.4.4 Adding and removing Feed Entries

Entries need not be maintained in a Feed indefinitely. However, it should not be removed before all the clients who are monitoring the data have retrieved it.

Entries can be removed when it becomes stale. For example, if an exception has been resolved or if an entry has been superseded with more current data. With respect to superseded entries, it is not necessary to maintain multiple references to the same resource in a feed.

Exception data should [CHS: shall?] be maintained in the feed until the exception has been resolved. Other constraints on data retention should be defined in Best Practices.

8 AVAILS API

This section provides details on using the MDDF API to implement Avails.

[CHS: Note, some of these can be moved to general rules making it easier to add other services.]

8.1 General Requirements

Data objects associated with Avails are documented in [Avails].

Avails are subject to the rules and conventions in Sections 1-7 of this document.

In addition to errors listed, additional errors can be returned in accordance with [HTTP].

8.2 Single Avails

Single Avails provides the means to access a single Avails associated with a studio.

8.2.1 Operations

Operations include

- GET – GET a specific Avail
- POST – Create an Avail
- DELETE – Remove a specific Avail
- PUT – Update a specific Avail

8.2.1.1 Endpoint

Avail endpoints are of the form:

<BaseURL>+"`avails/`"+<ALID>

Where <BaseURL> is the Base URL as defined in Section 5, and <ALID> is the ALID associated with the Avail.

For example,
`https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/avails/md:alid:eidr-s: B65C-7EC9-1F9F-D611-F84F-0`

8.2.1.2 GET

When a GET is performed from the Avail endpoint, the Resource Server returns the Avail associated with that endpoint represented by an AvailsList containing on Avail element.

If successful, a 200 (OK) status code is returned.

If the Avail is not present, 404 (Not Found) status code is returned.

8.2.1.3 POST

When a GET is performed to an Avail endpoint with a valid AvailsList as data, the Resource Server creates an Avail associated with that endpoint.

If successful, a 201 (Created) status code is returned.

If the Avail associated with that endpoint already exists, a 409 (Conflict) status code is returned.

If /AvailList/Avails/ALID does not identically match <ALID> in the path (URL-save encoding not withstanding), a 400 (Bad Request) status code is returned.

8.2.1.4 DELETE

When a DELETE is performed from the Avail endpoint, the Resource Server removes the Avail associated with that endpoint.

If the Avail exists, 200 (OK) is returned.

If the Avail did not exist, 404 (Not Found) is returned. Note that by returning 404 rather than 204 (No Content) the server signals an error if the resource does not exist. Although this results in a non-idempotent DELETE, it is more likely to catch ID errors and is therefore safer. [CHS: agree?]

8.2.1.5 PUT

When a PUT is performed to an Avail endpoint with a valid AvailsList as data, the Resource Server updates an Avail associated with that endpoint.

If the PUT is successful, 200 (OK) is returned.

If the Avail does not exist, 204 (No Content) is returned.

If /AvailList/Avails/ALID does not identically match <ALID> in the path (URL-save encoding not withstanding), a 400 (Bad Request) status code is returned.

8.2.2 Error Codes

TBD

8.3 Bulk Avails

Bulk Avails provides the means to access multiple Avails associated with a studio.

Bulk Avails implementation has been deprioritized and may not be present in many implementations.

8.3.1 Operations

Bulk Operations include

- GET – GET all Avails

- POST – Create or update multiple new Avails
- DELETE – Remove all Avails
- PUT – Update multiple Avails. Note that does not remove Avails not listed.

8.3.1.1 GET

A GET from the Bulk Avails endpoint returns all Avails as return data in the form of AvailsList.

Individual Avails are returned in /AvailsList/Avail where /AvailsList/Avail/ALID corresponds with <ALID> in the Single Avails path.

If successful, a 200 (OK) status code is returned.

If no Avails are present, 404 (Not Found) status code is returned.

8.3.1.2 POST

For each Avail in /AvailsList/Avail, if that Avail does not already exist on the Resource Server, the Avail will be created; and, if the Avail exists it is updated.

The resource name is /AvailsList/Avail/ALID and the path for the new Avail is the Single Avails path where <ALID> corresponds with the resource name.

POST is equivalent to of performing a POST on each new Avail, and a PUT on each existing Avail.

[CHS: I don't love the semantics of mixing creation and update.]

[CHS: Note that this is NOT a Master Avail because it does not delete old Avails.]

8.3.1.3 DELETE

A DELETE to the Bulk Avails endpoint removes all Avails associated with that endpoint.

If there is a desire to remove several Avails, but not all of them, the Client must DELETE the Avails individually.

There are no DELETE-specific errors.

8.3.1.4 PUT

A PUT to the Bulk Avails endpoint is the same as POST, but individual Avails cannot be created.

If the PUT is successful, 200 (OK) is returned.

Errors are returned in the error object.

8.3.2 Error Return

TBD: Errors for multiple avails (add multiple instances to error object?)

8.4 Avails Status

Avails processing status can be obtained using the 'getstatus' endpoint as defined in Section 5.4.3.

An AvailsStatus object is returned. [CHS: This is a placeholder for a better definition. The big question is whether we define fields or just create a structure of name/value pairs. If the latter, it should used across services.]

Element	Attribute	Definition	Value	Card.
AvailsStatus				
Resource		Resource URL associated with Resource associated with request. This will be the same URL that was used in the request.	xs:anyURI	
ProcessingState		TBD		
LastUpdated		Date and time of last update		
History		TBD – record of changes		
TBD…				

8.5 Avails Atom

Atom for Avails is in accordance with Section 7.

The endpoint for Avails is in accordance with Section 5.4.2.

9 MEDIA ENTERTAINMENT CORE (MEC) METADATA API

This section provides details on using the MDDF API to implement Media Entertainment Core metadata [MEC].

9.1 General Requirements

Data objects associated with MEC are documented in [MEC].

The MEC API are subject to the rules and conventions in Sections 1-7 of this document.

In addition to errors listed, additional errors can be returned in accordance with [HTTP].

A MEC resource consists of a XML CoreMetadata object as defined in [MEC] containing at least a Basic element.

Note that MEC resources are tied to studios. This allows multiple distributors to each have their own metadata objects associated with the same content.

9.2 Single MEC

Single MEC functions provides the means to access a single MEC resource.

9.2.1 Operations

Operations include

- GET – GET a specific Basic Metadata resource
- POST – Create Basic Metadata resource
- DELETE – Remove a specific Basic Metadata resource
- PUT – Update a specific Basic Metadata resource

9.2.1.1 Endpoint

MEC endpoints are of the form:

<BaseURL>+"`mec/`"+<ContentID>

Where <BaseURL> is the Base URL as defined in Section 5, and <ContentID> is the ContentID associated with the BasicMetadata.

For example,
`https://api.craigsmovies.com/mddf/sofaspudfilms.com/v1/mec/md:cid:eidr-s: B65C-7EC9-1F9F-D611-F84F-0`

9.2.1.2 GET

When a GET is performed from the MEC endpoint, the Resource Server returns the metadata associated with that endpoint represented by a CoreMetdata document containing a Basic element.

If successful, a 200 (OK) status code is returned.

If the MEC resource is not present, 404 (Not Found) status code is returned.

9.2.1.3 POST

When a GET is performed to an MEC endpoint with a valid CoreMetadata as data, the Resource Server creates an MEC resource associated with that endpoint.

If successful, a 201 (Created) status code is returned.

If the metadata associated with that endpoint already exists, a 409 (Conflict) status code is returned.

If /CoreMetadata/Basic/@ContentID does not identically match <ContentID> in the path (URL-save encoding not withstanding), a 400 (Bad Request) status code is returned.

9.2.1.4 DELETE

When a DELETE is performed from the MEC endpoint, the Resource Server removes the MEC resource associated with that endpoint.

If the MEC resource exists, 200 (OK) is returned.

If the MEC resource did not exist, 404 (Not Found) is returned. Note that by returning 404 rather than 204 (No Content) the server signals an error if the resource does not exist. Although this results in a non-idempotent DELETE, it is more likely to catch ID errors and is therefore safer. [CHS: see note in Avails]

9.2.1.5 PUT

When a PUT is performed to an MEC endpoint with a valid CoreMetadata as data, the Resource Server updates MEC metadata associated with that endpoint.

If the PUT is successful, 200 (OK) is returned.

If the MEC resource does not exist, 204 (No Content) is returned.

If /CoreMetadata/Basic/@ContentID does not identically match <ContentID> in the path (URL-save encoding not withstanding), a 400 (Bad Request) status code is returned.

9.2.2 Error Codes

TBD

9.2.3 Image Delivery

Images are referenced in /CoreMetdata/Basic/LocalizedInfo/ArtReference in the form of a URL. These URLs can be

- local file references (i.e., file scheme), with location agreed upon between client and server
- Internet references (e.g., ftp or http)

- Manifest references in the form of imageid (i.e., md:imageid:…)

9.3 Bulk MEC

TBD whether this is supported. [CHS: ?]

9.4 MEC Status

MEC processing generally takes some time because images must be ingested, either from files sent separately or from URLs provided in ArtReference. Status provides the means to communicate the status of any offline processing.

MEC processing status can be obtained using the 'getstatus' endpoint as defined in Section 5.4.3.

An MECStatus object is returned. [CHS: TBD. This should match pattern in Avails, etc..]

Element	Attribute	Definition	Value	Card.
AvailsStatus				
Resource		Resource URL associated with Resource associated with request. This will be the same URL that was used in the request.	xs:anyURI	
ProcessingState		TBD		
LastUpdated		Date and time of last update		
History		TBD – record of changes		
TBD…				

9.5 MEC Atom

Atom for MEC is in accordance with Section 7.

The endpoint for MEC is in accordance with Section 5.4.2.

10 MEDIA MANIFEST CORE (MMC) API

[CHS: it will follow the same pattern.]