MovieLabs Best Practices for Simple API Communications

Framework for Unmediated Network Communication

CONTENTS

REVISION HISTORY

Version	Date	Description
1.0	December 8, 2020	First Release

1 INTRODUCTION

To promote interoperability, this document defines how certain network communications mechanisms should be used. The specific focus of this document is the exchange of non-media data directly between two parties. Other specifications will address other types of communication.

The communications addressed in this specification address what is found in most RESTful or simple message-based APIs. These are, at the application layer, point-to-point. That is, other than traditional network functions like proxies, firewalls and load balancers which are effectively invisible, there are no intermediate functions managing the communication. This would be in contrast to, for example, a notification service, which is mediating the communications. To contrast these functions from functions with intermediaries, we call this class of service “unmediated”.

Data expressed using this API is typically expressed in XML or JSON. All sorts of data, but generally not media (i.e., video, audio, subtitles, images, apps, games, etc.) which tends to be transferred via mechanisms specifically designed for large data objects. There are other ways to exchange these data, such as email, proprietary file exchange, proprietary cloud-based exchange mechanisms, and standardized cloud-based mechanisms.

For convenience, we are calling it an “API”, although more correctly these are underlying mechanisms for building APIs. Some mechanisms, such as channel encryption, apply to all APIs. Others, such as OAuth2 or Webhooks might apply only to certain classes of APIs. Any given API must choose which of these mechanisms are required.\

This document should be considered a menu of options. Generally, each section is self-contained with a set of practices that should all be followed if any are used. Although some sections (e.g., TLS) apply to all APIs, it would never make sense to use everything in this document in one API. Select the features that apply.

1.1 Background and Motivation

MovieLabs has envisioned the future of ‘film’ and television in *MovieLabs Evolution of Media Creation*, *The Evolution of Production Security*, and *The Evolution of Production Workflows* which we refer to collectively as MovieLabs’ 2030 Vision. These papers can be found here: https://movielabs.com/production-technology.

1.1.1 Goals

The vision papers describe principles for building what we call software-defined workflows, which are built from interoperable components. There are many aspects of interoperability, including communications, some classes of which are addressed here.

Through MovieLabs collaboration with organizations working from across the spectrum from concept through archive, we have determined that the unmediated communications addressed here are used almost ubiquitously. Unfortunately, each underlying mechanism provide numerous options, resulting in countless permutations and combinations that have worked their way into practice. So, two parties using the exact same mechanisms may be unable to communicate because of sometimes arbitrary differences in implementation options.

Interfaces are an *n x m* problem. That is every studio, platform, or service provider (n) communication with multiple partners (m). This creates the potential of *n x m* interfaces. The goal of standardization is to reduce *n x m* to as close to *1* as possible. This leads to a reduction in implementation and maintenance costs, and a greater ability to adapt to requirement changes. Furthermore, since there are fewer implementations, they can be better optimized. Finally, security is hard. Implementing and validating a single security model meaningfully increases security.

This document seeks to specify choices that will promote interoperability. For those implementing from scratch, the methods in this document represents best practices.

1.1.2 Portals vs APIs

One approach to avoiding APIs is to offer a user interface. commonly referred to as a "portal". Although portals solve the automation problem for one side of the interface, they preclude full automation. This is problematic because manual entry results in errors, and the need to deal with portals for each partner. If you have hundreds of partners, you could have hundreds of portals. We call this "portalitis".

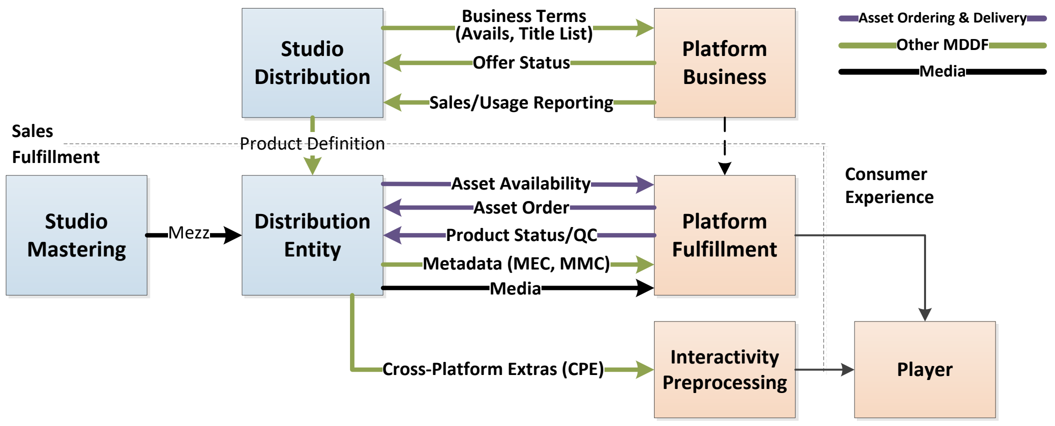
Portals can still be useful in the right circumstances. They provide easy access to smaller partners who are unlikely to automate. One of the OAuth2 models described below uses portals.

If you are implementing a portal, our recommendation is that you implement an API to support full automation, and develop your portal using the API.

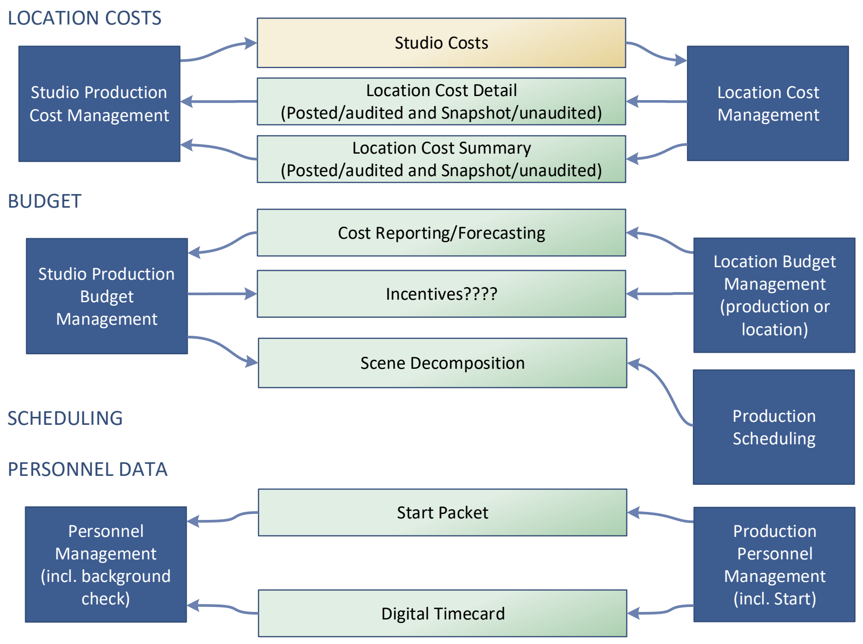
1.1.3 Representative Use Cases

This specification is motivated by two initial use cases. Both have immediate needs for API mechanisms. By addressing these and other use cases not described here, we ensure that this framework addresses a broad range of requirements.

The first use case is the MovieLabs Digital Distribution Framework (MDDF) illustrated below. Details can be found at www.movielabs.com/md



The second use case is Production Finance, which initial focus on location cost management. This is still being evolved, but the following illustrates the initial scope.


LOCATION COSTS
Studio Production Cost Management
Studio Costs
Location Cost Detail (Posted/audited and Snapshot/unaudited)
Location Cost Summary (Posted/audited and Snapshot/unaudited)
Location Cost Management
BUDGET
Studio Production Budget Management
Cost Reporting/Forecasting
Incentives????
Scene Decomposition
Location Budget Management (production or location)
SCHEDULING
Production Scheduling
PERSONNEL DATA
Personnel Management (incl. background check)
Start Packet
Digital Timecard
Production Personnel Management (incl. Start)

1.2 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. RESTful Web Service – Definition of RESTful interface
7. Reverse Channel – Methods for server to reply back to clients
8. Signing Payloads – Methods for signing JSON and XML payloads

1.3 Document Notation and Conventions

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

1.4 Normative References

[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
[IANAJWT]	IANA JSON Web Token (JWT) Registry, https://www.iana.org/assignments/jwt/jwt.xhtml
[NIST800-52-2]	McKay, Kerry A., and David A. Cooper, NIST Special Publication 800-52, Revision 2, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations., August 2019, https://csrc.nist.gov/publications/detail/sp/800-52/rev-2/fin
[RFC2104]	IETF RFC 2104, MAC: Keyed-Hashing for Message Authentication, February 1997, https://tools.ietf.org/html/rfc2104
[RFC3629]	IETF RFC 3629, UTF-8, a transformation format of ISO 10646, https://tools.ietf.org/html/rfc3629
[RFC3986]	IETF RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, https://www.ietf.org/rfc/rfc3986.txt
[RFC4287]	IETF RFC 2460, The Atom Syndication Format, December 2005. https://tools.ietf.org/html/rfc4287
[RFC4346]	IETF RFC 4346. The Transport Layer Security (TLS) Protocol Version 1.1. https://www.ietf.org/rfc/rfc4346.txt
[RFC4686]	IETF RFC 4648, The Base16, Base32, and Base64 Data Encodings, October 2006, https://tools.ietf.org/html/rfc4648
[RFC5023]	IETF RFC 5023, The Atom Publishing Protocol, October 2007, https://tools.ietf.org/html/rfc5023 as modified by Errata 1304 and 3207
[RFC5246]	IETF RFC 5246, The Transport Layer Security (TLS) Protocol, Version 1.2, https://tools.ietf.org/html/rfc5246 as updated by Errata.

[RFC6151] IETF RFC 6151, Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, March 2011, https://tools.ietf.org/html/rfc6151

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

[RFC7165] IETF RFC 7165, Use Cases and Requirements for JSON Object Signing and Encryption (JOSE), April 2014, https://tools.ietf.org/html/rfc7165

[RFC7515] IETF RFC 7515, JSON Web Signature (JWS), May 2015, https://tools.ietf.org/html/rfc7515

[RFC7516] IETF RFC 7516, JSON Web Encryption (JWE), May 2015, https://tools.ietf.org/html/rfc7516

[RFC7518] IETF RFC 7518, JSON Web Algorithms (JWA), May 2015, https://tools.ietf.org/html/rfc7518

[RFC7519] IETF RFC 7519, JSON Web Token (JWT), May 2015, https://tools.ietf.org/html/rfc7519

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

[RFC7797] IETF RFC 7797, JSON Web Signature (JWS) Unencoded Payload, February 2016, https://tools.ietf.org/html/rfc7797

[RFC8446] IETF RFC, The Transport Layer Security (TLS) Protocol Version 1.3, https://tools.ietf.org/html/rfc8446

[SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[XMLC1.1] Canonical XML Version 1.1, W3C Recommendation 2, May 2008, https://www.w3.org/TR/xml-c14n11/

[XMLDSIG] XML Signature Syntax and Processing Version 1.1, W3C Recommendation, April 2013, *https://www.w3.org/TR/xmldsig-core/*

1.5 Informative References

[Vision] *The Evolution of Media Creation, A 10-Year Vision for the Future of Media Production, Post and Creative Technologies*, MovieLabs, December 2019, https://movielabs.com/production-technology/

[Security] *The Evolution of Production Security, Securing the 10-Year Vision for the Future of Media Production, Post and Creative Technologies*, MovieLabs, December 2019, https://movielabs.com/production-technology/

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

[CM] TR-META-CM MovieLabs Common Metadata, v2.8 (or later), http://www.movielabs.com/md/md

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

[GitWebhook] GitHub Developer, Webhooks, https://developer.github.com/webhooks/

[Manifest] TR-META-MMM MovieLabs Media Manifest Metadata, v1.9 (or later), http://www.movielabs.com/md/manifest

[SOAP] Simple Object Access Protocol (SOAP), https://www.w3.org/TR/soap/

[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

1.6 Status

This specification is 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.

2 API OVERVIEW

There are several parts to this API.

This first part of the API is communications protocols, described in Section 3. These are HTTP and TLS. This specification identifies specific features that must be supported by a compliant implementation.

Security is an essential feature of the API Security doesn't just *happen*—it must be designed into any secure system. While the API is only one piece of the overall system, it must contain the necessary security elements to ensure data are protected. Some security features are addressed in the protocols section (i.e., TLS). Authentication and Authorization is addressed in Section 4. The security approach is based on the most common security protocols. We offer solutions using API Keys, JSON Web Tokens (JWT), and OAuth2., particularly OAuth 2. These are protocols that support practical and robust implementations.

NOTE that this document addresses specific usage of several security mechanisms. However, it relies on the use of security practices that are beyond the scope of this document. Principles of media security our outlined in MovieLabs' white paper, *Securing the 2030 Vision*. [Security]. This document expects that security best practices are adopted. The practices specified here are only as good as the overall security approaches adopted by the involved parties involved.

Our API is based on the RESTful approach[1]. When sending or retrieving data or status in an essential aspect of a RESTful interface is endpoints (i.e., the URL construction that facilitates access to data). Section 5 established the rules for endpoints. Specific endpoints still need to be defined for each application and will be defined as part of those specifications. Section 6 fills the gaps in the RESTful definition.

The REST model does not support return data. We provide several models for return channel in Section 7. There are proprietary notification services, particularly in cloud implementations. Webhooks are commonly used for simple return data, or as part of a Pub/Sub (Publisher/Subscriber) model.

APIs carry data where proof of origin (who originated data) or non-repudiation (did they really say that) is required. Section 8 provides methods for signing payloads or portions of payloads.

[1] Like most RESTful API definitions, we do not strictly comply with RESTful principles. However, for convenience we refer to our interface as RESTful. Please forgive us for this indiscretion.

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

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

3.1.3 XML, JSON and other formats

Clients and servers may use XML, JSON or both.

When clients use XML, they must include "`Accept: application/xml`".

When clients use JSON they must include "`Accept: application/json`".

Many specifications will define both XML and JSON encoding. MovieLabs will strive to create schemas that work suitably in either model.

Note that this does not preclude the use of other formats. For example, "`Accept: application/vnd.usd+zip`".

The goal is to reduce implementation cost and complexity as much as possible. Most implementers would prefer to pick their favorite and let their partners implement it. However, that means *somebody* must create and support multiple implementations. As a general rule, when it is "one-to-many" (e.g., a single server services many clients, or a single client accesses many servers), the "one" should support both JSON and XML to ease the burden on the "many". When the situation is many-to-many, there is no clear strategy. Consequently, certain APIs may opt for one or the other to avoid complexity.

3.1.4 Range-GET for multiple resources

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

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

3.1.4.1 Pagination

Pagination is the retrieval of a full set of data by requesting subsets in separate requests. Pagination is supported as described in this section.

Pagination is complex because the state of the server can change between requests for pages of data. If data are added or deleted between requests, the range of a request can refer to different data. How pagination works depends on the desired semantics. Generally, the assumption is that once the requests are completed the client has the set of data as it existed when the first query was made. This, unfortunately, requires the server to maintain state. Besides the obvious issues for the server, there are also issues in caching. The approach defined here (pagination tokens) does not solve the problem, but it has some advantages over the other methods (e.g., range gets). The pagination token method is sometimes called cursor-based paging or paging tokens.

For pagination, all resources are assumed to be fixed in order, but that order unspecified (i.e., not sorted). For example, the third item remains the third item for the period during which the GETs are performed.

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

- `limit` – Maximum number of records that can be returned (note that only on the last request can fewer resources than the limit be returned)

The Server returns all available records, but no more than `limit` records. If there are more records to be retrieved, the server also returns in the HTTP header:

- `nextToken` – A token used in the next request to indicate where to continue the request (i.e., next page)

When the client makes the next request, it includes in the query string `limit` in the query string and also in the query string:

- `next` – The value from `nextToken` as returned from the previous request

To retrieve all records the Client continues to perform requests, until `nextToken` is not returned.

For example, if 13 record are available, it might look something like this:

```
GET https://api.craigsmovies.com/mddf/v1/avails/?limit=5
nextToken: xa93amdf0
(5 avails returned)

GET https://api.craigsmovies.com/mddf/v1/avails/?next=xa93amdf0&limit=5
nextToken: qza92dars
(5 avails returned)

GET https://api.craigsmovies.com/mddf/v1/avails/?next=qza92dars&limit=5
(3 avails returned, no nextToken)
```

3.1.4.2 Resource Count

To determine the number of resources of a given resource type, a Client optionally may perform a GET to a **getcount** endpoint as defined for that resource type as defined within a specific API definition (e.g., `https://api.craigsmovies.com/mddf/v1/avails/getcount`). The Server returns the number of Resources. For example, in XML:

```
<ResourceCount>
  <NumberOfResources>13<NumberOfResources>
</ResourceCount>
```

Or, in JSON:

```
{
   "NumberOfResources": 13
}
```

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

3.1.5 Cache Support

Servers must include "ETag:" in GET responses.

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/Property	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 believes the client implementer will find useful.	xs:string	0..1
Ref	A transaction reference for debugging (e.g., retrieving logs).	xs:string	0..1

This might be extended in the future to support localization of error messages. This is important if errors might be user-visible.

3.1.6.2 Error Example

Following is an example of an XML error return:

```
<?xml version="1.0" encoding="UTF-8"?>
<Error>
   <ErrorCode>XMLValidation</ErrorCode>
   <ErrorMessage>Avails XML Document does not comply with Avails v2.4 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>
```

And, in JSON:

```
{
   "Error": {
      "ErrorCode": "XMLValidation",
      "ErrorMessage": "Avails XML Document does not comply with Avails v2.4 Schema",
      "Resource": "api.craigsmovies.com/mddf/v1/craigs.movies.com/avails/md:alid:eidr-s:abcd-
1234-abcd-1234-abcd-m",
   "MoreInfo": "Missing required field AvailsLicensor",
   "Ref": "1234abcde"
   }
}
```

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. Additional 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 in a Location: header
204 No Content	A DELETE is performed on a nonexistent resource
301 Moved Permanently	If resource has been renamed, access to old resource should be redirected
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 resource 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

3.2 TLS

Connections between clients and servers shall use TLS in accordance with NIST SP 800-52, Rev. 2 [NIST800-52-2].

Note that TLS 1.3 should be used when practical. Software should be updated when vulnerabilities are found. TLS is not in itself sufficient for Zero Trust Security—measures must be taken on both sides of the interface to limit vulnerabilities.

The establishment of Certificate Authorities (CAs) are outside of the scope of this document. For TLS it is necessary to have a CA. So, it will be necessary to select suitable CA vendors and/or create CAs.

4 AUTHENTICATION AND AUTHORIZATION

4.1 Overview

This section identifies several mechanisms that are useful in providing authentication and authorization. Often, they can be used together.

As different applications have different requirements, this section does not recommend specific use cases. Each application API must be specific about which models apply, and certain specifics within those models (identified in the text).

The methods we describe here are:

- API Keys
- JSON Web Tokens (JWT)
- OAuth2

4.2 API Keys

API Keys are secrets passed in API calls that identify a consumer of an API. They can also be used to limit the volume of requests from a particular application, control scope of access and other purposes.

4.2.1 API Key Model

Typically, a service will provide one or more API keys when an application developer registers. Consider a server that provides some service. An application developer is developing an application that will use the service. The developer registers an application the service and receives an API Key. Whenever the application accesses the service it provides the API Key in the call.

4.2.2 Managing API Keys

This document does not specify how API Keys are obtained. This is generally a process of logging into a service and requesting a key. Keys may also be issued manually. Care must be taken to ensure that the key is not compromised.

API Keys are persistent with lifespans of months, years, or even indefinitely. The duration of validity for an API Key should be specified as part of individual API definitions. For example, a production might issue keys only for the duration of the production while other services allow longer terms of validity. Generally, the renewal process is performed in a manner similar to the original request. Some services extend the validity of a given key, and others issue new keys.

There should be a method to revoke API keys. This may be due to a security compromise, or for business reasons (e.g., end of a project or license).

Keys should be carefully managed in the code base. For example, API Keys should not be included in source code or checked into GitHub repositories. An example of best practices for API Keys can be found on Google's developer's site here: https://developers.google.com/maps/api-key-best-practices.

4.2.3 API Key in HTTP requests

Individual APIs will define where API Keys are passed.

4.2.3.1 API Key in query string

An API Key can be included in the query string of a request. For example:

```
?api_key=abc123def456
```

4.2.3.2 API Key in header

API Keys can be included in the HTTP header as X-API-KEY:

```
X-API-Key: abc123def456
```

Note that API Keys can hypothetically be included in `Authorization: Basic`, but the `X-API-KEY` is preferred because it is unambiguous.

4.2.3.3 Accepted or rejected API Keys

If an API Key is not accepted, the HTTP response should use the 401 Unauthorized response code.

Otherwise, there is no API Key indication in the response.

4.3 JSON Web Tokens (JWT)

JSON Web Tokens (JWT) is a method for asserting claims. JWT is primarily useful for server-to-server authorization. When appropriate security measures are applied, the recipient of a token can trust that the bearer can access specific functionality or data.

4.3.1 JWT Overview

JWT[2] is part of the JavaScript Object Signing and Encryption (JOSE) Framework as described in [RFC7165]. JOSE is a collection of RFCs that includes JWT as well JWT's underlying mechanisms. JWT is defined by [RFC7519].

The advantage of a JWT token over simple authentication or API Keys is that JWT can carry information. Because the JWT is signed, the information can be trusted as long as the signature is trusted.

[2] The term JWT is used in literature to refer both the specification and the token itself. We will sometimes use the term JWT token to refer to the tokens defined by the JWT spec.

Generally, a server will generate a JWT, signing it with a suitable algorithm. If the server sees the token again, it knows it is valid because it knows it signed it. Since it is valid, it can trust the information contained in the token.

In the context of this specification, the payload of the token is the authorizations granted to the bearer. Like an OAuth2 Bearer Token, the JWT token can open access to a service.

Other than stating that tokens must always be passed over secure channels, this specification does not state how tokens are generated or how they are passed from party to party.

4.3.2 Signing and Encrypting Tokens

Tokens must be signed using JSON Web Signatures (JWS) and documented in [RFC7515] and [RFC7519]. Note that RFC 7519 disallows the use of the unencoded payload option.

The required algorithm is HMAC using SHA-256. This is defined in [RFC7518], Section 3.2. This uses symmetric keys. It is expressed in the JOSE header with

```
 "alg" : "HS256"
```

The recommended algorithm is RSASSA-PKCS1-v1_5 using SHA-256. This is defined in [RFC7518], Section 3.3. This uses public key encryption. It is expressed in the JOSE header with

```
 "alg" : "RS256"
```

Token encryption is covered by JSON Web Encryption (JWE) defined in [RFC7516]. JWT tokens need not be encrypted as they are presumed to always be sent over encrypted channels.

Unsecured JWTs (i.e., JWTs with “alg” : “none”) are not allowed.

4.3.3 Using a Token

JWT tokens can be passed using various methods such as query strings (JWT is URL-safe), cookies, and in HTTP headers. The method appropriate for APIs defined by this specification is that they be included in the HTTP headers.

JWT tokens are passed in HTTP requests using

```
Authorization: Bearer  <JWT token>
```

4.3.4 Claims

The following claims should be included

Claim	RFC 7519 Section	Notes
‘iss’ Issuer	4.1.1	Included if issuer is not the audience
‘sub’ Subject	4.1.2	

'aud' Audience	4.1.3	
'exp' Expiration Time	4.1.4	It is strongly recommended tokens have limited lifespan.

If a Claim exists in the IANA JWT Registry [IANAJWT], the registered name must be used for that data object. For example, if one is referring a birthday, then the Claim Name of "birthdate" must be used. Otherwise, as stated in [RFC7519], Section 4.2, use a "Collision-Resistant Name".

4.4 OAuth 2.0

4.4.1 OAuth2 Overview

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].

4.4.1.1 Terminology used in this section

When discussing *authentication* there are two parties, one authenticating to the other. When discussing *authorization* there are three parties: Party 1 authorizes Party 2 to access information from Party 3. Generally, Party 2 is the client and Party 3 is the server. The client and server are working together to perform some function for Party 1. Authentication is an essential element of an authorization process.

Role Terminology in this section is based OAuth2 [RFC6749] Section 1.1. We sometimes append the term "User" to indicate the human associated with that function.

Term	Meaning
Resource Owner	"An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user." [RFC6749]
Resource Server	"The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens." [RFC6749]
Client	"An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices)." [RFC6749]
Authorization Server	"The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization." [RFC6749]

The following terms are also used:

Term	Meaning
User	A human being accessing some service
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).

RFC6749 calls the Resource Owner and "end-user" when it is a person. To remind people the End-user is the Resource Owner, we say: End-user (Resource Owner)

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

- Authorization Code – a secret presented to obtain an Authentication Token and, optionally, a Refresh Token
- Access 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)
  - Can be revoked

4.4.1.2 Outline of OAuth2 Participants

As noted above, there are three parties involved in authentication: The Resource Owner authorizes a Client to access protected resources on the Resource Server. An additional party, the Authorization Server facilitates this process.

In the context of media production and distribution, the Resource Owner is often a studio, the Resource Server is a platform involved in production (e.g., a location accounting system) or distribution (e.g., a retailer's Avails system). The Client might be a system within the studio's IT department or a system run be a 3rd party service provider.

Note that OAuth2 can be used in either direction, so in other situations the studio might be providing Resource Server with the other party being the Client and Resource Owner.

4.4.1.3 Authorization Scope

Scope defines which resources the Client can access as defined in [RFC6749], Section 3.3. Specific scope is defined by the API.

Care must be taken to avoid collisions between scopes. For example, the scope might be a concatenation of the API and the specific access (e.g., "`md:scope:org.sofaspudfilms.com:avails-read`").

4.4.2 OAuth2 Client Registration

To obtain access to an OAuth2 Authorization Server, clients must be registered with that server ([RFC4769], Section 2.

The registration process provides the Client with a Client Identifier (i.e., 'client_id') and Client Secret (i.e., 'client_secret'). Unregistered Clients ([RFC6749] Section 2.4) are not supported.

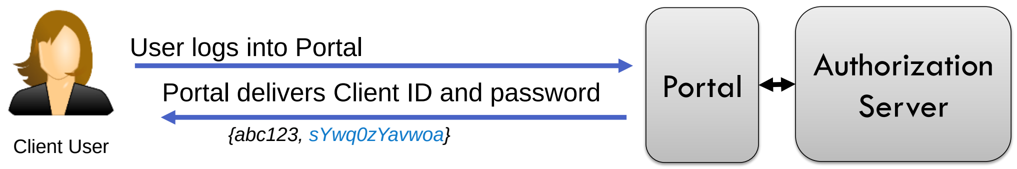
The registration process is outside of the scope of RFC 6749, and this document. However, many systems, especially in distribution, have Portals, the following section illustrates how such a Portal can be used for Client Registration.

4.4.2.1 Portal-based registration

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.

First, a user associate with the client ("Client User") must authenticate to the Authentication Server, using the Portal's authentication method. Once that user has authenticated, the Portal accesses the Authorization Server to generate client_id and client_secret.

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).

4.4.3 OAuth2 Authorization Code

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

The Client needs Client ID (i.e., client_id), Client Secret (i.e., client_secret) and Authorization Code to obtain the Access Token. To get the Authorization Code, one need the Client ID and Resource Owner authorization.

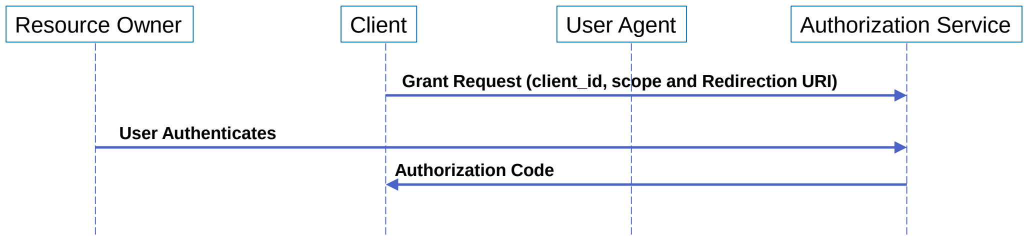
This section defines two methods for the Client to obtain and Authorization Code

- Synchronously – This is the model typically used when a user who is logged into a service or is using an application authorizes that service or app to access data from a 3rd party. The entire operation happens in one set of transactions.
- Asynchronously – This model more typically reflects B2B interfaces found in media and entertainment. The act of obtaining a authorization code occurs asynchronously.

4.4.3.1 Obtaining Authorization Code Synchronously

Following is the most basic method for obtaining an Authorization Code. Generally, a User Agent (e.g., web browser) is used to provide a user interface and redirection as necessitated by the protocol.

The flow is illustrated here:



4.4.3.2 Obtaining Authorization Code Asynchronously

This section addresses most B2B architectures for which this document is intended. The methods here allow the Resource Owner to obtain the Authorization Code and pass it to the Client. The assumed systems architectures do not include a User Agent that can redirect authentication requests as defined in the previous section.

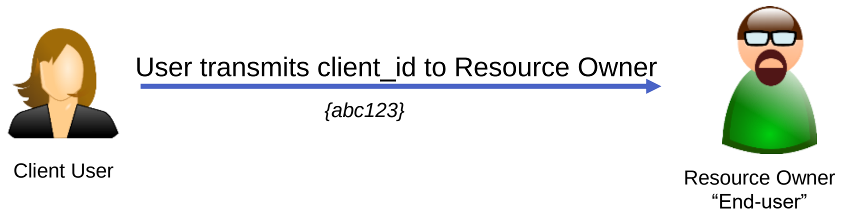
In this model, the Resource Owner obtains the Authorization Code on behalf of the Client. Note that the Authorization Code is useless without the Client Secret known only to the Client. That is, only the Client can use the Authorization Code to obtain Access Tokens.

The steps are as follows

- The Client shares the Client ID with the Resource Owner
- The Resource Owner obtains the Authorization Code from the Authorization Server
- The Resource Owner transmits the Authorization Code to the Client

4.4.3.3 Transmitting Client Identifier to Resource Owner

The Client Identifier is needed to obtain an Authorization Code. In the following illustration, a Client User transmits the Client ID to the End-user (Resource Owner).

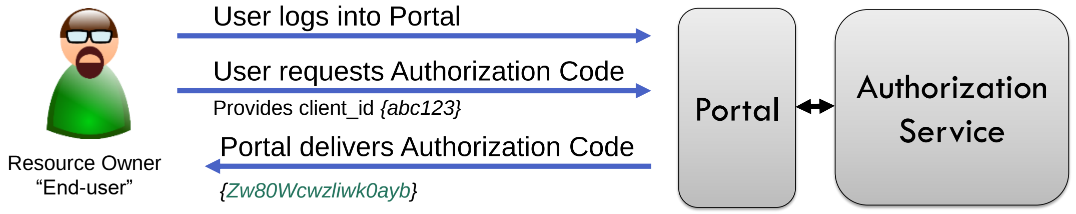


It is not essential that the client_id remain secret. However, it is essential that it be unmodified. If a surreptitious client_id is substituted, authorization will be granted to the wrong client_id; denying service to the original Client and potentially granting access to a 3rd party unauthorized access to data. The integrity of the client_id must be protected.

4.4.3.3.1 Obtaining Authorization Code

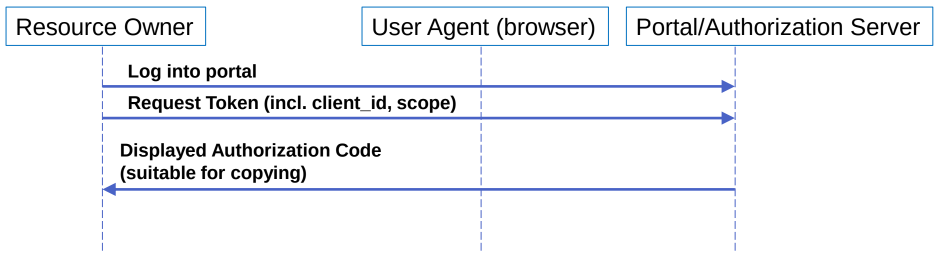
Once the Resource Owner has the Client ID, it can obtain the Authorization Code. In this case, the Portal acts as a proxy for the Client.

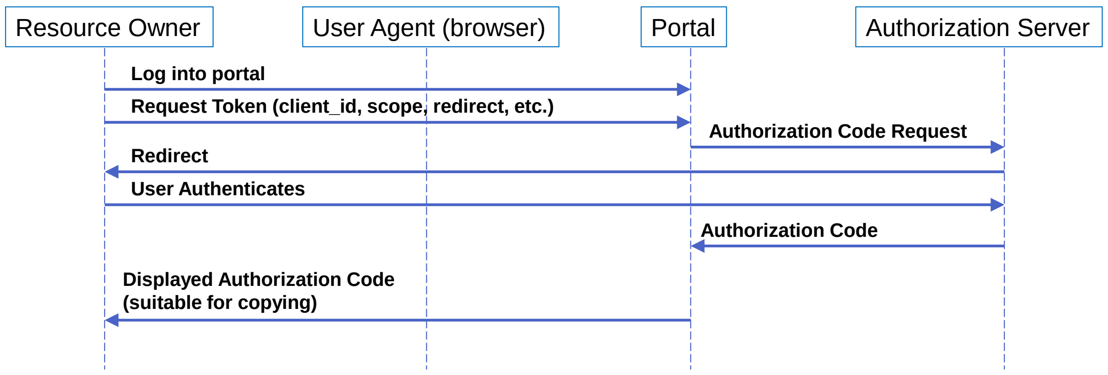
First, the End-user (Resource Owner)—for example, studio user—logs into a Portal, indicates what information will be accessed (i.e., scope), and retrieves the Authorization Code (e.g., copies it from a screen provided by the Portal).


User logs into Portal
User requests Authorization Code
Provides client_id {abc123}
Portal delivers Authorization Code
{Zw80Wcwzliwk0ayb}
Resource Owner
"End-user"
Portal
Authorization Service

The Authorization Service must be aware that although the Portal is not the Client (i.e., is not presenting the Client Secret), it still may be granted an Authorization Code.

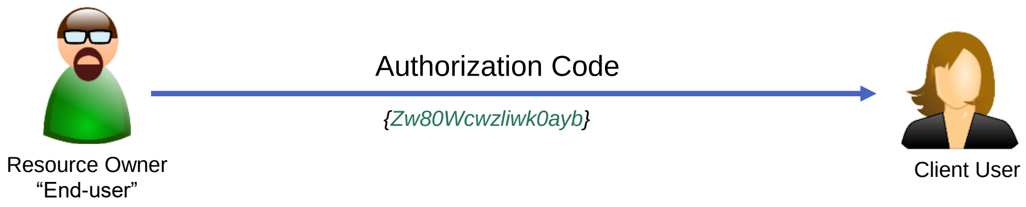
Below are two models for implementing this exchange, one with a combined Authorization Server and Portal, one with distinct entities.


Resource Owner
User Agent (browser)
Portal/Authorization Server
Log into portal
Request Token (incl. client_id, scope)
Displayed Authorization Code (suitable for copying)


Resource Owner
User Agent (browser)
Portal
Authorization Server
Log into portal
Request Token (client_id, scope, redirect, etc.)
Authorization Code Request
Redirect
User Authenticates
Authorization Code
Displayed Authorization Code (suitable for copying)

4.4.3.4 Transferring Authorization Code to Client

Once the Authorization Code is retrieved, it is passed to the Client via a secure method.



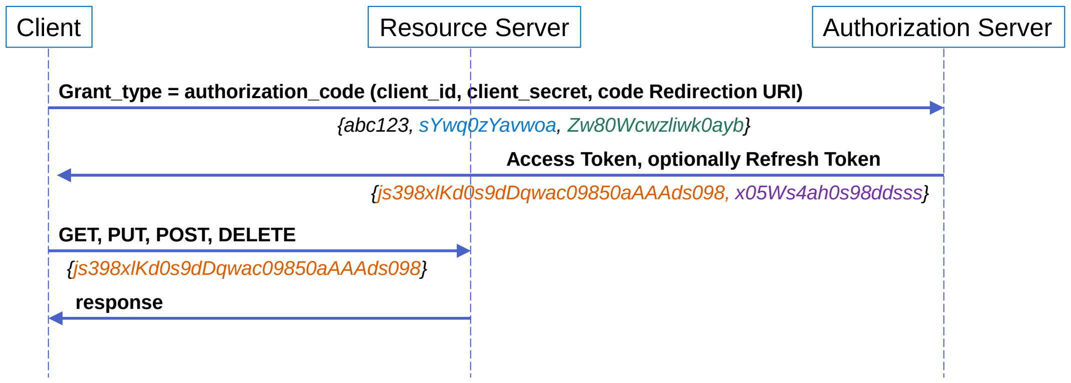
The Authorization Code is useless without the Client Secret, so it can be transferred in the clear. However, it is always recommended that secure channels be used whenever practical.

4.4.4 Access and Refresh Tokens

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.

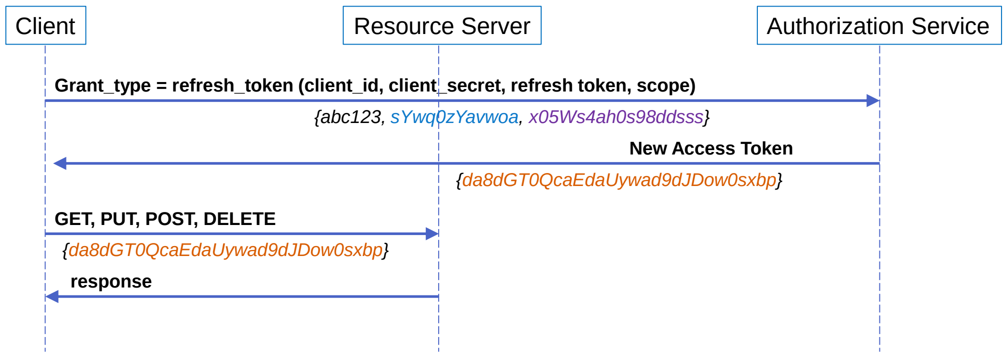
The process has the prerequisite that the Client hold a valid API Key and Authentication Code.

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


Client
Resource Server
Authorization Server
Grant_type = authorization_code (client_id, client_secret, code Redirection URI)
{abc123, sYwq0zYavwoa, Zw80Wcwzliwk0ayb}
Access Token, optionally Refresh Token
{js398xlKd0s9dDqwac09850aAAAds098, x05Ws4ah0s98ddsss}
GET, PUT, POST, DELETE
{js398xlKd0s9dDqwac09850aAAAds098}
response

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:


Client
Resource Server
Authorization Service
Grant_type = refresh_token (client_id, client_secret, refresh token, scope)
{abc123, sYwq0zYavwoa, x05Ws4ah0s98ddsss}
New Access Token
{da8dGT0QcaEdaUywad9dJDow0sxbp}
GET, PUT, POST, DELETE
{da8dGT0QcaEdaUywad9dJDow0sxbp}
response

5 ENDPOINTS

5.1 Approach

Endpoints are defined by URLs constructed as follows

“`https://`” <Base URL> “/” <version> “/” <function-specific path>

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

`https://api.sofaspudfilms.com/prodacct/v1/gl/md:prodidn: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 Craig’s Movies (streaming and retail service) is `https://api.craigsmovies.com/mddf` or `https://api.sofaspudfilms.com/prodacct`.

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.

If the version portion of the path is missing, the endpoint is associated with the latest version of the API.

5.4 Function-specific Path

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

The function-specific path begins with the service, unless the function spans all services. An example of a service is, “`gl/`” (i.e., general ledger).

5.4.1 REST API Path

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/v1/avails/md:alid:eidr-s:B65C-7EC9-1F9F-D611-F84F-0` refers to the Avail associated with that ALID.

When the function-specific path contains only the service followed by '`/getall`', requests apply to all resources within the scope of that service. For example, `https://api.craigsmovies.com/mddf/v1/avails/getall` refers to all Avails resources for Sofa Spud Films on the Craig's Movies platform. This should be used with pagination.

Note that multi-resource endpoints are often not supported in initial implementations of APIs; with an offline onboarding service sometimes provided instead. Note also that implementations may require `getall` queries to use pagination to avoid accidental requests for large volumes of data.

5.4.2 Atom API Path

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

For example, for Avails a GET of `https://api.craigsmovies.com/mddf/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 endpoints use neither REST nor Atom semantics.

Special paths include the following.

Path	Function
getall	Returns all resources associated that path. See Section 5.4.1.
getcount	Returns ResourceCount indicating the number of resources associated with an endpoint. For example, `https://api.craigsmovies.com/mddf/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/v1/avails/getcount` and obtaining status on a single resource is as follow: `https://api.craigsmovies.com/mddf/v1/avails/md:alid:eidr-s: B65C-7EC9-1F9F-D611-F84F-0/getstatus`

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.

Although not strictly RESTful, it is acceptable to optimize by using POST to request multiple objects (resources) be returned. That is, a POST contains a payload that includes the list of objects that are required. The payload returned would have the requested records.

7 REVERSE CHANNEL

It is often necessary to asynchronously send data from server to client outside of a request-response. This section defines reverse channel interfaces.

7.1 Considerations in selecting a return channel model

7.1.1 Polling, Callback, and Pub/Sub

There are fundamentally two approaches to sending data asynchronously: Either the client repeatedly polls a server to see if state has changed (polling), or the server initiates a response when there is a change. There is no clear delineation between callbacks and pub/sub (Publisher/Subscriber). The term 'callback' generally refers to a response to a specific request; often using a Webhook. Pub/Sub mechanisms are generally services where subscriber asks a publisher to inform it when state has changed.

Polling has the advantage of maintaining the client-server relationship. It is always the client contacting the server. The most obvious benefit is security—polling piggybacks on the authentication established for the regular services. However, polling requires a tradeoff between polling resources and latency. The more frequently one polls, the timelier the response, but the more resources wasted for polling.

Pub/Sub and other forms of callbacks have the advantage of being targeted and purposeful. Messages are only generated when there is data to send or events to signal. One of the most common callback mechanisms is Webhooks; described below. Callback and Pub/Sub require more complexity because of the state that must be maintained by all parties (i.e., what subscriptions are active, what if a subscriber is offline, etc.).

7.1.2 Security Implications

Polling works under the same security model as the forward channel. It is just a different way of using the same channel.

A reverse channel can be configured just like a forward channel, just in the other direction: TLS, OAuth2 etc. That is, an interface is established where the server becomes the client and the client becomes the server. Let's say Party A is providing a service to Party B. For the reverse channel, Party B is the server and Party A is the client.

The forward and reverse channels must be established separately because the authorization protocol is designed for one direction. The server has no means to authenticate itself to the client.

However, this does not guarantee that the reverse channel is secure. Callbacks introduce potential attack surfaces unless care is taken to ensure the return channel is fully trusted. Consider Webhooks. In this model the requestor provides an endpoint for the return channel. If someone slips in an alternate endpoint, the server will not necessarily know it is not legitimate. Simply put, this model complicates the security model with the proportional risk of compromising data or denial of service.

7.1.3 Open vs. Proprietary

In this section we focus on open mechanisms. This is primarily to give implementers options that are not tied to a particular vendor (e.g., depending a cloud vendor's proprietary notification mechanisms). Some of these proprietary mechanisms[3] in some circumstances are superior to the approaches outlined here, except for them being proprietary. Implementers must trade off the advantages of each approach and make their decisions accordingly.

7.2 Bidirectional Message Passing

An asymmetric (client/server) model such as a RESTful model is a design choice. Protocols can be defined such that messages go in both directions without prejudice. Protocols like the Simple Object Access Protocol (SOAP) [SOAP] are based on this model.

This model is not currently very popular for a variety of reasons. We do not address it further.

7.3 Pub/Sub and Callback

7.3.1 Pub/Sub and Callback Considerations

Pub/sub and callbacks require a server/publisher to maintain state on clients/subscribers. They must make a best effort to deliver return channel information, even if the client/subscriber is offline. These characteristics do not (exactly) exist in polling.

For convenience we will also refer to callback clients and servers as subscribers and publishers.

Any mechanism must address the following

- Unsubscribing – How does a subscriber indicate it is no longer interested in receiving data. Related to this, a subscription might end because a subscriber: 1) Ceases to exist, 2) is removed because of business partnership changes, 3) is removed for security reasons, or 4) any other policy, business or technical reason.
- Retry frequency and duration – Situations will exist where the subscriber is offline or unreachable. The publisher should attempt to reach the subscriber, but within practical constraints. Policies must be established for how frequently the publisher tries, back-off strategies, how long it tries before giving up, and what happens to the messages that were undeliverable.

There is no generic policy that would work in all conditions. Therefore, these behaviors must be determined in the context of specific APIs.

[3] Some examples of proprietary mechanisms for notifications and pub/sub are Amazon Simple Notification Service (SNS), Azure Service Bus, and Google Firebase Cloud Messaging.

7.3.2 Webhooks

A Webhook is a mechanism that supports callbacks. A Webhook is like reply-to in an email message. As part of the HTTP request, response conditions and a return web address (hook) are provided to the server. When a condition is met, the server sends a response to the provided web address.

This is a very powerful and flexible mechanism. It is especially flexible in that the server does not need a priori knowledge of the client endpoints. And those endpoints can change over time, or even from request to request.

There are no standards for Webhooks, although there is considerable practice. An example of a Webhook definition can be found in GitHub Developer Webhooks [GitWebhooks].

7.3.2.1 Webhook Semantics

The expected response(s) to a Webhook in a request is specific to each API and should be defined as part of that API.

7.3.2.2 Webhook Return Channel

For the return channel, communications must be in accordance with Section 3. That is, the HTTP and TLS requirements apply.

Note that some Webhooks use challenge-response mechanisms. At this time, this document does not recommend challenge-response.

7.3.2.3 Authenticating a response

A webhook response (from server to client) must be authenticated to ensure it did not come from a third party. That is, the client sends a message with response information to the server. This is always done on a secure, encrypted, authenticated channel. When the client receives data, it must know that it is coming from the server, not someone else. This is done by adding information to the response that can only be added by the server.

We offer two methods for authentication

7.3.2.4 Method 1 OAuth2

Method 1 is to use OAuth2 Access Tokens or JWT tokens as described elsewhere in this document. If the server (now the client) has been authenticated to the client (now the server), the identity of the sender is known.

7.3.2.5 Method 2 Hashed response

Method 2 is more specific to Webhooks and is often more practical. Method 2 requires the server to hash a combination of the payload and shared secret. The client validates the message by performing the hash function with the payload and secret. Due to the extreme difficulty of generating a hash collision without knowing the secret, the payload can be assumed to be from the server (compromised secrets notwithstanding).

First a secret must be negotiated and shared between the Client and Server. This document does not specify the method for this.

We recommend the use of protocols defined in GitHub Webhooks [GitWebhooks], Securing Your Webhooks.

7.3.3 Proprietary subscription models

When working within a particular cloud environment, it is often beneficial to using pub/sub model made available within the cloud, even if they are proprietary. We are not making recommendations whether or not to use any of the following services, but are listed them to illustrate services that might be used in this context.

Following are some examples:

- Amazon Simple Notification Service (SNS). https://aws.amazon.com/pub-sub-messaging/. To illustrate a potential benefit of using a proprietary system, SNS can invoke a Lambda instance.
- Google Cloud Pub/Sub. https://cloud.google.com/pubsub/
- Azure messaging services, https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services

7.4 Polling

7.4.1 RESTful Polling

The simplest mechanism is to poll using RESTful GETs for resources of interest. This model works well if 1) the number of entities to be polled is relatively small, 2) the acceptable latency for noticing and update is long, and/or 2) caching is fully implemented and the ETag mechanism avoids requests hitting servers.

Note that polling can be made more efficient with proper API design and implementation. For example, out of hundreds of thousands of Avails, generally only a small percentage of records change in any week; and most of those changes are in a small number of titles. It would be inefficient to poll hundreds of thousands of records many times an hour. However, querying on title (or ALID) can reduce queries by two orders of magnitude. Querying only specific titles of interest can reduce queries even further. A strategy might be to query active titles frequently, and less active titles infrequently.

7.4.2 Atom-based Polling

Atom is a polling method defined by the Atom Publishing Specification (AtomPub) [RFC5023] and the Atom Syndication Format [RFC4287]. Atom 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. Clients poll the feeds.

Atom is appropriate when only a small percentage of resources are updated in a given period (e.g., in Avails on the order of fifty updates against hundreds of thousands of entries). Polling each resource would be prohibitively expensive, but polling a feed document is not.

7.4.2.1 Implementation Requirement

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

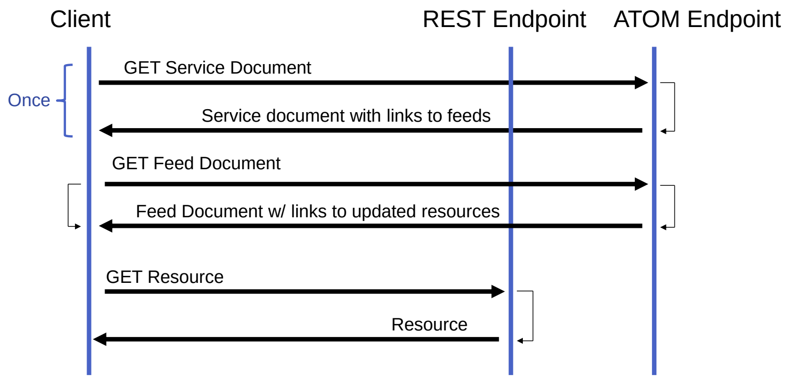
Servers are required to implement Atom in accordance with this specification.

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.

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.

7.4.2.2 Protocol

Protocols used shall be in compliance with Protocol and Endpoint requirements of Sections 3, 4 and 5. An Atom exchange is illustrated here:



7.4.2.3 Service and Feed Documents

The Atom Service Document defines the feeds.

The Service Document is retrieved with a GET from the well-known address provided by the Service.

Each application has its own Service Document. These are defined by specification associated with the application.

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 application.

- A collection element for each feed as defined in the Feeds Section (Section **Error! Reference source not found.**).

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>
```

A Feed Document provides links to Resources that merit attention.

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.

A Feed Document shall conform with RFC 4287 [RFC4287]. The following is generally the minimal subset, although applications can modify as appropriate.

Element	Usage
title	Title of feed as defined by the application
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., EIDR)
entry/updated	Date and time resource was updated
entry/category	TBD

Following is a sample 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>
```

Entries need not be maintained in a Feed indefinitely. However, the goal is not to remove data 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.

Best Practices should be established for each application.

8 SIGNING PAYLOADS

In some circumstances it may be appropriate to sign data to ensure integrity and or for authentication/nonrepudiation. This section addresses signing JSON and XML payloads.

8.1 Signature Keys

The methods for creating and distributing keys is outside the scope of this document. Within the type of keys and their distribution will drive the algorithm used in signing.

For most applications, we anticipate that public key infrastructure (PKI) will be the most practical solution.

8.2 Signature Validation

When data is signed, the signature should be validated. If the validation process fails, the data must be rejected.

8.3 JSON Web Signature

JSON document are signed using JSON Web Signature (JWS) as defined in [RFC7515]. JWS is a JSON Web Token (JWT) as per RFC 7159 [RFC7159] that contains signature information covering a JSON document.

8.3.1 Algorithm

APIs using JWS will support the same algorithms as JSON Web Tokens as described in Section 4.3.2.

8.3.2 Serialization Method

JWS contains two serializations: Compact Serialization, described in RFC 7159, Section 3.1 and 7.1; and JWS JSON Serialization, described in Section 3.2 and 7.2. The former is used in URLs and in headers; the latter in the JSON document, or as a companion document.

While APIs can choose the specific mechanism used, generally the JWS JSON Serialization method will be used. This adds the signature data to the JSON object. Note that this has the advantage of carrying the signature with the data, regardless of the data transfer method.

Applications should support the general and flattened JWS JSON Serializations as described in RFC 7159 sections 7.2.1 and 7.2.2 respective. The general serialization supports multiple signatures. This is useful when more than one algorithm is used, or if different audiences have different keys.

From RFC 7159, the simpler flattened serialization shows how to provide signature data for a payload of <payload contents>:

```
{
 "payload":"<payload contents>",
 "protected":"<integrity-protected header contents>",
 "header":<non-integrity-protected header contents>,
 "signature":"<signature contents>"
}
```

From RFC 7159, general serialization would look as follows.

```
{
 "payload":"<payload contents>",
 "signatures":[
  {"protected":"<integrity-protected header 1 contents>",
    "header":<non-integrity-protected header 1 contents>,
    "signature":"<signature 1 contents>"},
    ...
  {"protected":"<integrity-protected header N contents>",
    "header":<non-integrity-protected header N contents>,
    "signature":"<signature N contents>"}]
}
```

8.4 XMLDSIG

The method for signing XML documents or sections of documents is XML is W3C's XML Digital Signature (XMLDSIG) as defined [XMLDSIG].

For message-level authentication, the general process is that the sender generates unsigned messages (based on the appropriate specification for the message), generates a digital signature for that message, and then packages the message with the signature. This package is then sent to the recipient. The signed message contains enough information to validate the sender of the message, and includes both the unsigned message as well as the digital signature of the unsigned message XMLDSIG Signature.

XML Digital Signatures can be used to sign and validate messages across any delivery structure. These shall be in conformance with [XMLDSIG]. Note that later versions may be adopted as defined here: http://www.w3.org/TR/xmldsig-core/.

The following constraints shall apply when generating digital signatures:

- For CanonicalizationMethod
  - `Algorithm=`http://www.w3.org/2006/12/xml-c14n11#WithComments
- For SignatureMethod,
  - `Algorithm=`http://www.w3.org/2000/09/xmldsig#rsa-sha1
- For DigestMethod,
  - `Algorithm=`http://www.w3.org/2000/09/xmldsig#sha1

Note that senders must use the same certificate, as defined in the KeyInfo element of the XMLDSig, for all messages using web services. This Key will serve as a unique identifier for the sender, and will be used to describe configuration information (such as URIs) associated with the sender. Note that the Reference element's URI attribute will always be set to the value "#Body".

The following constraints shall apply when generating digital signatures:

- Data will be transmitted in accordance with section 6.6.4 of that document, “Envelope Transform”. XML for encoding may be found here: http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/xmldsig-core-schema.xsd#enveloped-signature

All web-based delivery mechanisms will support Signed Messages as defined above as a mechanism to sign and validate messages. Email-based delivery will not use XMLDSIG to sign messages.

All recipients of messages should validate Signed Messages before processing them.

Note that all messages require the use of Canonical XML, Version 1.1 (With Comments), [XMLC1.1], which is necessary for proper signing.

Note that when using W3C schemas it is best to copy schemas to a local directory. http://www.w3.org/Help/Webmaster.html#slowdtd.