MediaSock Specification

version 1.4 | 060707

This is now depreciated and will be replaced by a new specification and set of guidelines to be made available first quarter 2009. Please follow or subscribe to the Profile Connect Twitter account for updates.

A lightweight service discovery protocol and programmatic interface for web-services handling a user's personal media assets.

Service Discovery

MediaSock's service discovery mechanism is designed to promote the adoption of applications using web-service APIs through a common method discovery and selection process. Users with a MediaSock compatible client application simply enter a server domain and the client discovers the available methods without the need for the user to select from a confusing list of APIs or URLs.

- Outline and Implementation

Programmatic Interface

MediaSock's methods have been conceived with the goal of simplifying the exchange of an authorised user's media (images and movies, initially) with disparate online services whilst accommodating their distinct architecture and functionality.

- Quick Implementation

- The **Resource**
- The **Methods**
- The Mechanisms

Client Framework

A framework supporting the integration of disparate photo-sharing and media web-services with desktop and server-based client applications is currently under development. Scheduled for initial commercial availability by the end of year, the framework will be open-sourced under a free-use licence in the future.

Architectural Model

MediaSock draws on the workings of existing private APIs and related public APIs in use by web-service providers, and it is intended to work alongside the methods provided by them. It is not a wide-ranging specification but a guideline for minimum interoperability, with ease of implementation by services and ease of use for consumers as core considerations.

MediaSock has been conceived primarily around HTTP, REST and XML in a client-server model, however it may be implemented in any protocol and data-format combination that might be suitable.

The XML output has been structured for ease of parsing such that substrings and tokens may be used, whilst service methods may exist at disparate URLs and responses may be embedded within existing output (e.g. HTML). MediaSock may therefore be used by providers not already having an API simply by adding a few lines to their existing pages, or equally it may be implemented for enhanced web-services functionality.

Current Status

This is a work-in-progress, MediaSock support is currently implemented in the following **desktop applications**:

- PictureSync – Mac and Windows

And **web applications**:

- Dripbook
- More soon!

Feedback, suggestions and greater participation are welcomed! Please visit the MediaSock Google group. This specification was authored, and is maintained by Jacob Jay <jjay[at]verse.org>.

Service Discovery

Quick Implementation

Create an index.xml or index.html file with the following content and place it in a directory named /mediasock/ on your webserver, replacing the method names and URLs as appropriate.

```
<mediasock value="com.example" title="Example Service" version="1.4">
<handshake>
<stat value="1" message="" resource="" serial="1"/>
<apis value="flickr.*:http://example.com/services/rest/"/>
</handshake>
</mediasock>
```

Endpoint

http://example.com/mediasock/

> If necessary an alias, filesystem symbolic link or index script should be used in preference to a HTTP redirect to reduce transaction costs, however clients should transparently follow HTTP redirects. The MediaSock path is lower-case.

The common endpoint provides the ability for a client to efficiently discover the MediaSock resource and the particular characteristics of a service.

> The discovery URL may be the common endpoint, or simply one manually entered into a client in place of a server domain. A client will construct a fully-qualified URL.

Request

> **GET /mediasock/ HTTP/1.0**
>
> The use of HTTP HEAD is recommended when the service characteristics are not required, such as by bots and crawlers.

Response

If the service supports discovery, retrieval of the resource will return an HTTP 200 (or redirect) status and the handshake response (also see methods), if the service does not support MediaSock, it will return HTTP 404.

Payload

The content is a data structure (XML by default), when used solely for service discovery this may be placed in a static index.xml or index.html file.

```
<mediasock value="com.example" title="Example Service" version="1.4">
<handshake>
<stat value="1" message="" resource=""/>
<apis
value="mediasock.taxa.append;flickr.photosets.create:http://example.com/services/rest/"/>
</handshake>
</mediasock>
```

> The XML response block may be embedded in HTML and other compataible formats. The HTTP Content-Type header should represent the container format (e.g. xml or html).

The supported API methods are returned in the apis element contained by the handshake block as above. The value attribute of which is a list (delimited with the semi-colon and no white-space, see values) of method names supported by the service. A client application will parse this list to determine if its required methods are supported, or to select its preferred methods from equivalents (by its own definition).

Items in the apis value may be namespace-qualified method names or namespace wildcards, and may optionally specify a specific URL for access, following a delimiting colon. Items are declared in order of preference and items delcared after a wildcard inherit that item's URL (if any). The colon, semi-colon and double-quote characters are illegal in method names, and must be encoded in URL paths.

Examples

1. value="mediasock.*" denotes full support for the mediasock methods at the current endpoint
2. value="mediasock.taxa.append" denotes support for a single method at the current endpoint
3. value="mediasock.taxa.append;flickr.photosets.create" denotes support for two methods at the current endpoint and a preference for the first
4. value="flickr.*:http://example.com/services/rest/" denotes full support for flickr methods at the specified endpoint
5. value="mediasock.*:http://example.com/services/rest/;mediasock.media.append:http://upload.example.com/" denotes full support for mediasock methods at the specified endpoint; but that the mediasock.media.append method must use the specified endpoint in place of the prior

API Implementation Guide

A full implementors guide for client and server implementations will be added in the future, in the mean time please read through the rest of the documentation on this site for the specification details. Also see service discovery. You can download a sample HTTP conversation for an upload using auth.hsc (session cookies) and feat.amp (media must belong to a taxa container).

Services

The MediaSock API may be implemented:

- as an extension of existing web pages
- as a standalone service
- within an existing framework of API methods

Extending existing webpages

Carrying out the following simple steps will allow MediaSock compatible client applications to login and upload items using your existing website employing session cookies.

1. Aliase /mediasock/ to your login page. Your server must return a 302 redirect to the URL or map it it transparently (e.g. via a directive or symbolic link).

2. If necessary rename your login field parameter names to comply with the MediaSock names (username and password).

3. Supplement the output of your login page to include the MediaSock handshake response (preferably within <head>); specify the mediasock.media.append URL in the apis element as the URL of your upload page; and dynamically set a *stat* value of "2" for invalid credentials and "0" for logged in (optionally "1" for not logged in, e.g. from a GET).

```
<!--
<mediasock value="com.example" title="Example Service" version="1.4">
<handshake>
<stat value="0" message="" resource="" serial="1"/>
<apis value="mediasock.handshake;mediasock.media.append:http://example.com/upload.php"/>
<sock value="auth.hsc;type.img;type.image/jpeg"/>
</handshake>
</medisock>
-->
```

4. Rename your upload field parameter names to comply with the media.append method query parameter names (e.g. media.data, media.title, media.caption, media.keywords). Note that keywords are delimited with the semi-colon ; (ASCII 59) character (and no qualification), you may replace these upon submission and before processing as appropriate for your datastore.

5. Supplement the output of your upload page to include the media.append method response; specify the value of the *value* attribute as the ID of the newly created item (or leave empty):

```
<!--
<mediasock value="com.example" title="ExampleService" version="1.4">
<media.append>
<stat value="0"/>
<item value="123456"/>
</media.append>
</medisock>
-->
```

Additionally you may refer to media types to restrict upload to specific formats, and/or add a message to the *stat* element of the handshake to report a quota.

> Because you can declare specific URLs for each method, you may also add support for as many methods as correspond to existing pages, e.g. album lists and creation of new albums. See the methods documentation to implement support for taxa (albums).
>
> If your service can carry out additional functions on uploaded items, or requires

additional attributes that are not specifically supported within a client, consider having users enter special keywords before upload.

Clients

MediaSock ideally exists as a translation layer between protocols and service-specific functions (if any). Developers should consider that services may support one protocol only, and that therefore for full compatibility with MediaSock services, they should implement both REST and XML-RPC protocol/format support, in addition to supporting any other protocols and behaviours that might be employed by specific services.

If you'd like to implemented MediaSock support in your application get in touch for some example processes.

An API library for client implementations will be developed at some point in the future.

The API

Endpoint and Discovery

See the Service Discovery outline.

Methods

- mediasock.handshake

- mediasock.auth.signout
- mediasock.media.append
- mediasock.media.update
- mediasock.taxa.list
- mediasock.taxa.append

Protocols

A service is not obliged to support any particular data-exchange protocol and it is recommended that clients support multiple protocols, however this version of the API only documents REST. The following protocol identifiers are specified in the sock element of the handshake method.

- prot.res – RESTian HTTP (default, assumed if not present)
- prot.xrp – XML-RPC
- prot.jrp – JSON-RPC

Requests

Requests are made in the format appropriate to the protocol used. A service will determine the protocol and format used for a request internally (e.g. from path, or Content-Type).

- prot.res – HTTP GET; HTTP POST with Content-Type multipart/form-data
- prot.xrp – HTTP POST with Content-Type text/xml
- prot.jrp – HTTP POST with Content-Type text/javascript

Responses

A service may announce the response formats it is able to return from those below by listing the respective value in the sock element. A client may in turn request a specific response format by sending that value in the sock parameter (as a list).

- resp.xml – XML (default with prot.res, prot.xrp)

  ```
  <?xml version="1.0"?>
  <mediasock value="com.example" title="Example Service" version="1.4">
  <!-- response(s) -->
  </mediasock>
  ```

  The value is unique identifier for the service and should never change once specified. The title is a human-readable label (i.e. brand name) that may be displayed to the user and may be changed.

  Where the response is returned within another document (such as an HTML page) the entire response may be placed in a comment or other container (ideally in the head of the document) without the XML declaration.

  Every element of the method response(s) must appear on individual lines with no preceeding identation or whitespace, and no changes made to the structure or order of attributes (elements may be reordered). This structure is used to simplify parsing with substrings and tokens in non-XML capable clients (see implementation, if supported JSON is preferable).

  The HTTP response Content-Type header should represent the container format (e.g. xml or html).

- resp.jso — JSON (default with prot.jrp)

  This format is not considered in the current version. The response may not be embedded in HTML or XML and must be returned with the Content-Type text/javascript.

Errors

In responses all methods return a stat element with the value attribute corresponding to a state code integer.

Where the state is non-zero an optional message attribute may be included to further explain the result, and/or a resource attribute with a URL to further details. If stat is returned with a resource attribute the client should give the user an option to open it (and show at least the URL domain).

```
<stat value="1" message="" resource=""/>
```

General state values are:

- 0 – sucess (continue)
- 1 – authentication required (handshake, retry)
- 2 – invalid authentication (stop, warn user)
- 3 – renegotiate (handshake, retry)
- 4 – [reserved]
- 5 – interrupted (retry immediately)
- 6 – temporary fault (stop or retry later)
- 7 – unsupported (continue, warn user)
- 8 – permanent failure (stop, warn user)
- 9 – service error (custom behaviours)

System and transport errors should be reported using the mechanisms provided by the transport protocol (such as through an HTTP status). A service may employ its own codes prefixed "9" to define custom states, clients should by default handle these as a generic 'Service error' and warn the the user with an option to stop, retry or continue.

Values

Text

Should be UTF-8 encoded. Linebreaks should be UNIX-style using the LF character (ASCII 10) only.

Lists

Multiple text values must be delimited with the semi-colon character (ASCII 59) and without white-space. A list must not be terminated with the delimiter, and qualification of values containing spaces or other special values is not required. The ; (semi-colon) character in values must be replaced or encoded (not considered in this version).

Attribute lists

A list of types and values (used as media attributes), each delimited with a colon (e.g. ":country:wibble;:location:wobble" specifies two values with country and location types respectively. The ;: (semi-colon, colon) characters in values must be replaced.

> Attributes are composed of three parts, the attribute-domain, the attribute-identifier (collectively the 'type'), and the attribute-value, delimited with colons. An empty attribute-domain assumes a plain text string. Recommended attribute types are:
>
> - :product:*text*
> - :category:*text*
> - :event:*text*
> - :author:*text*
> - :person:*text*
> - :country:*text*
> - :state:*text*
> - :city:*text*
> - :location:*text*
> - date:lastused:*date*
> - date:created:*date*
> - date:updated:*date*
> - geo:location:*lat,long*
>
> Dates are specified as YYYYMMDDHHMMSS and adjusted to UTC time (GMT) from the original time zone (if known). An original timezone may optionally be specified by suffixing .ZZZZX where ZZZZ is the +/- prefixed and zero-padded offset from UTC in minutes representing the local timezone, and where X is a value indicating the hour offset of local daylight savings time (0 for none, 1 for 1h). E.g. the string 20051005094503.+0601 represents the 5th of October 2005 at 9:45:03 UTC, but where the local time is +1h00 (060 minutes) and daylight savings is in effect (1h).
>
> Geographic coordinates are sepecified decimally using a WGS84 projection as lattitude and longitude delimited with the comma e.g. 0.050,-0.050 and negative prefixed values as appropriate.

API Methods

See formats for formatting and structure.

> Custom parameters may be submitted with requests, but should not alter the core functionality provided by the method; parameter names starting 'media.' or 'sock.' are reserved. Custom elements may be returned within the <mediasock> response block but the names of these may only start with 'x-', all other element names are reserved; custom attributes may not be included.
>
> All MediaSock method and parameter names are lower-case.

Standard requests

All queries utilise the following standard parameters.

- method – a list of methods being called (see also combined calls)
- sock – a list of the characteristics for the request (optional)
- apikey – a unique client identifier or the product name (required)
- sock.key – a session token for authentication (optional)

> sock is used when a characteristic other than a default is used with a request or wanted with the response, such as response data-format, or cipher used with login credentials: e.g. sock=ciph.md5;resp.jso

Standard responses

One common element is returned with all responses, the stat element—this indicates a response status code.

```
<stat value="0" message="" resource=""/>
```

Combined method calls

The API optionally allows a list of multiple method names to be specified in a single query by the client, with all parameters submitted simultaneously, and for their responses to be concatenated by the service and returned in a single transaction.

Such combined calls reduce network overhead and latency in client applications. This is generally more efficient and provides better interactivity in client implementations [and more context in service implementations]. Such support is optional in both service and client, but if implemented should support all the methods returned in the sock element of the handshake. Methods should be processed in the order specified by the method parameter.

Support for combined method calls is identified in the api element of the handshake with the value: feat.cmc

```
<sock value="feat.cmc"/>
```

mediasock.handshake

This **required** method is multi-purpose, providing compatability, login authentication, and account information (all only where applicable). A handshake should be made at the start of each session.

Query

The handshake method is used for authentication by negotiating with the parameters associated with a corresponding authentication mechanism value declared in the sock element.

> When used for authentication with REST (prot.res) credential parameters must be POSTed (get is not allowable).

Response

```
<handshake>
<stat value="0" message="2048 MB available" resource="" serial="1"/>
<user value="1234" home="http://user.example.com/"
conf="http://example.com/myaccount/"/>
<apis
value="mediasock.handshake;mediasock.taxa.list;mediasock.media.append:http://example.com/mediasock.pl
<sock value="prot.res;auth.hpo;ciph.txt;type.img;type.image/jpeg"/>
<akey value="T63YV3Y3JOJ9JB663VVSTE33G" expiry="0"/>
```

</handshake>

- stat – state: this is the only required element; *value* must be 0 when authorised or an error state; the *serial* attribute must be any number, to be incremented when any of the apis or sock values are changed; see state for details of *message* and *resource* attributes
- user – user details: *value* is an ID; *home* is a URL to a users account; *conf* is a URL to configure their account (optional)
- apis – api methods: a list of methods supported by the service (required, see service discovery); this list must not contain whitespace
- sock – characteristics: a list of protocols, authentication mechanisms, ciphers, and media-types supported by the MediaSock methods (optional)
- akey – authentication key: an authentication token for subsequent method calls with an expiry time (optional)

> A client may cache the values for apis, sock and user until the serial attribute of *stat* changes. A client must parse the apis list for URLs after login (as well as before) so that a service may return user-specific method URLs, and must not cache these values beyond the duration of a session. Format of the *akey* expiry is 0 (zero) for a permanent token, minutes since last use (e.g. 25) or seconds from now (e.g. 3600).

Function

A service should return a stat value of 0 when authentication is valid, or an error code as appropriate. Under special circumstances (outages, disabled account) a service may return an act of 8 with a specific URL for the user to visit for more details.

A client application is not required to perform an initial request on the resource and may carry out an initial login request with its preferred options (e.g. encrypted credentials), and only parse the apis and sock values to retry the login appropriately if the state is a failure.

> If a client attempts to login with an unsupported mechanism an error state of 3 should be returned.

mediasock.auth.signout

Closes a session and should be used by the service to invalidate or delete the associated session token, cookies, or signing key. There are no parameters (except sock-key when using auth.hst or auth.hsk).

mediasock.media.append

Accepts file data and metadata for a new or existing media item (when used to update an existing item it must be addressed as media.update). The behaviour of a client when using this function may be altered using the following feature flags in the sock element of the handshake:

- feat.amp — a new media item must be created within a taxanomic structure by specifying its parent taxon, a null parent (album/set) is not allowed; clients should display a list of taxon identified with the append attribute and value containing media, within which to add the new item.

Query

- media.data – file data of any type, text, images or video (required)
- media.iid – empty for append, or the id of an existing item for update
- media.parents – a list, the ids of taxa in which to insert the item (optional, required with feat.amp)

- media.title – text (optional)
- media.caption – text, a public comment (optional)
- media.note – text, a private comment (optional)
- media.keywords – a list (should be optional)
- media.attributes – an attribute list, must contain type:value pairs (optional)

Response

```
<media.append>
<stat value="0"/>
<item value="123456" resource="http://example.com/user/media/123456"/>
</media.append>
```

- *item* – value is a unique string which may be used to identify the new item in subsequent method calls; resource a URL with which the user may access the item outside the API
- *stat* – status result code:
  - 0 – sucess (return new id and url)
  - 5/6/7 – failed (return message)
  - 20 – exists (return existing id and url) (not for update)

Equivalents

-

mediasock.media.update

Synonym for media.append and must be used when updating an existing item.

> If a service does not explicitly state support for this method in the apis element, the service does not allow the modification of existing items.

mediasock.taxa.list

Returns a list of taxonomy names and identifiers for a structured classification system (categories, albums, sets, etcetera). It would generally be used in a combined call with the handshake.

Response

The item element is to be repeated for every parent and child.

```
<taxa.list>
<stat value="0"/>
<item value="" append="media,taxa" title="" parents="" caption="" resource="" note="" icon=""/>
</taxa.list>
```

- value – a taxon identifier (required)
- append – used in combination with feat.atp and/or feat.amp, value may be empty or a list
- title – (optional)
- caption – (optional)
- note – (optional)
- parents – a list of corresponding taxon identifier (optional)
- resource – a URL for the user to access the item (optional)
- icon – a media identifier or URL to a preview image in any format (optional)

> Where a client does not support hierarchical taxonomy it may flatten the taxa.

Equivalents

- metaWeblog.getCategories

mediasock.taxa.append

Accepts a name and properties for a new element in a structured classification system (categories, albums, sets, etcetera). The behaviour of a client when using this function may be altered using the following feature flags in the sock element of the handshake:

- feat.atp — in a heirarchical taxa, a new taxon must be created inside an existing taxon by specifing its parent; clients should display a list of taxon identified with the append attribute and value containing taxon, within which to add the new item

Query

- taxon.title – (required)
- taxon.caption – (optional)
- taxon.note – (optional)
- taxon.parents – a list (optional)
- taxon.icon – a media identifier (optional)
- taxon.attributes – an attribute list (optional)

Response

```
<taxa.append>
<stat value="0"/>
<item value="" append="media" title=""/>
</taxa.append>
```

Equivalents

-

API Mechanisms

Authentication

Most methods require authentication. The MediaSock API methods supports several schemes for HTTP each with distinct requirements. A service can return one or more of these identifiers in the sock element during handshake to specify its preferred schemes (if not the default). Support for any of these authentication mechanism must be declared in the apis element by including the mediasock.handshake value.

```
<sock value="auth.hsc"/>
```

- auth.hpo – **POST**; user authentication occurs with every call and credentials passed as username and password parameters. (default) (it is recommended that these values are encrypted)
- auth.hsc – **session cookies**; user authentication occurs during handshake by passing username and password parameters, and an HTTP cookie is returned, this cookie must be stored and returned in the HTTP headers with subsequent calls. (suggested)
- auth.hsk – **key signing**; user authentication occurs once during first handshake using one of the other mechanisms (identified in the sock element of the handshake), and a permanent authentication key is returned, this token is then

combined with every method-specific parameter and returned as an md5 digest as the sock-key parameter. (recommended)
- auth.hst – **session token**; user authentication occurs during handshake using one of the other mechanisms (identified in the sock element of the handshake), and a session authentication token is returned, this token must be returned with subsequent calls as the sock-key parameter.
- auth.hdi – **digest**; user authentication occurs with every call and credentials passed in the HTTP headers.
- auth.hba – **basic**; user authentication occurs with every call with credentials passed in the HTTP headers.
- auth.hws – **WSSE**; user authentication occurs with every call and credentials passed in the HTTP headers (X-Wsse).
- auth.pub – **none**; user authentication is not required to use the service (e.g. it is public, or authenticated by IP).
- auth.exa – **external**; a client must signin with one of the methods listed in the sock element of the handshake response (i.e. login may not be performed with the handshake) and then authenticate using one of the other auth mechanisms listed in the sock element.

> Session cookies or tokens should be valid for a minimum of fifteen minutes after last use. With hsc the service must set all necessary cookies upon the first handshake request and **should not** assume that they are present during the handshake.
>
> Services may require an apikey parameter containing an individually issued application ID for sucessful authentication.
>
> These mechanisms apply when used with MediaSock API methods. When using MediaSock for discovery of other API methods (listed in the apis element), those API methods will have their own distinct authentication requirements.

Encryption

When using the hpo, stk, or sck authentication schemes, values for login credentials (both username and password) may be encrypted (independant from the transport), a service can return one or more of these identifiers in the sock element during handshake to specify required ciphers.

```
<sock value="auth.hpo;ciph.md5"/>
```

- ciph.txt – **Plain**; unencrypted (default)
- ciph.md5 – **MD5**
- ciph.cry – **UNIX Crypt**
- ciph.ssl – **SSL HTTP**

> If SSL requires a domain and path other than that of the common endpoint it should be specified in the sock attribute of the handshake.

Media types

These optional values may be specified in the sock element of the handshake to allow a service to indicate what generic media/object types a user or client may upload. If no type is specified a default of type.any is assumed. Mime types may be listed to declare specific file-formats that are accepted (a client should reject or reformat as appropriate).

```
<sock value="type.img;type.image/jpeg;type.image/tiff"/>
```

- type.img – images
- type.mov – movies (multimedia)
- type.aud – audio

- type.txt – plain text documents
- type.rtf – formatted text documents
- type.doc – documents of other types (e.g. page layouts)
- type.any – any other, or undefined (default)
- type.mimetype/subtype – a specific mime type pair