Deliverable D2.6

Service Integration Draft Guidelines

Grant agreement nr	688382
Project full title	Audio Commons: An Ecosystem for Creative Reuse of Audio Content
Project acronym	AudioCommons
Project duration	36 Months (February 2016 - January 2019)
Work package	WP2
Due date	28 April 2017 (M15)
Submission date	28 April 2017 (M15)
Report availability	Public (X), Confidential ( )
Deliverable type	Report ( ), Demonstrator ( ), Other (X)
Task leader	QMUL/MTG-UPF
Authors	Frederic Font
Document status	Draft ( ), Final (X)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

Table of contents

Table of contents **2**

Executive Summary **3**

1 Guidelines for integrating services in the Audio Commons Ecosystem **4**

1.1 Architecture of the ecosystem and the Audio Commons Mediator 4

1.2 Adding new services to the ecosystem 5

1.2.1 Base component 6

1.2.2 Authentication component 7

1.2.3 Search component 9

1.2.4 Download component 13

1.2.5 Licensing component 14

1.2.6 Configuring a service 14

1.2.7 Service description 15

2 Conclusions and future directions **16**

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

Executive Summary

This deliverable presents the draft guidelines for integrating new services in the Audio Commons Ecosystem. The prototype of the Audio Commons Ecosystem which has been deployed at the time of this writing, allows the addition of new services by the implementation of service plugins developed using the Python programming language. In this deliverable we explain how to implement such services plugins and provide implementation examples of the already implemented plugins included in the Audio Commons Mediator codebase. At the end of this deliverable we describe the future improvements in service integration technologies that we expect to carry out as the Audio Commons project evolves.

This deliverable complements deliverables D2.4 and D2.5, which respectively describe the Audio Commons API specification and the technologies developed for service integration (the Audio Commons Mediator).

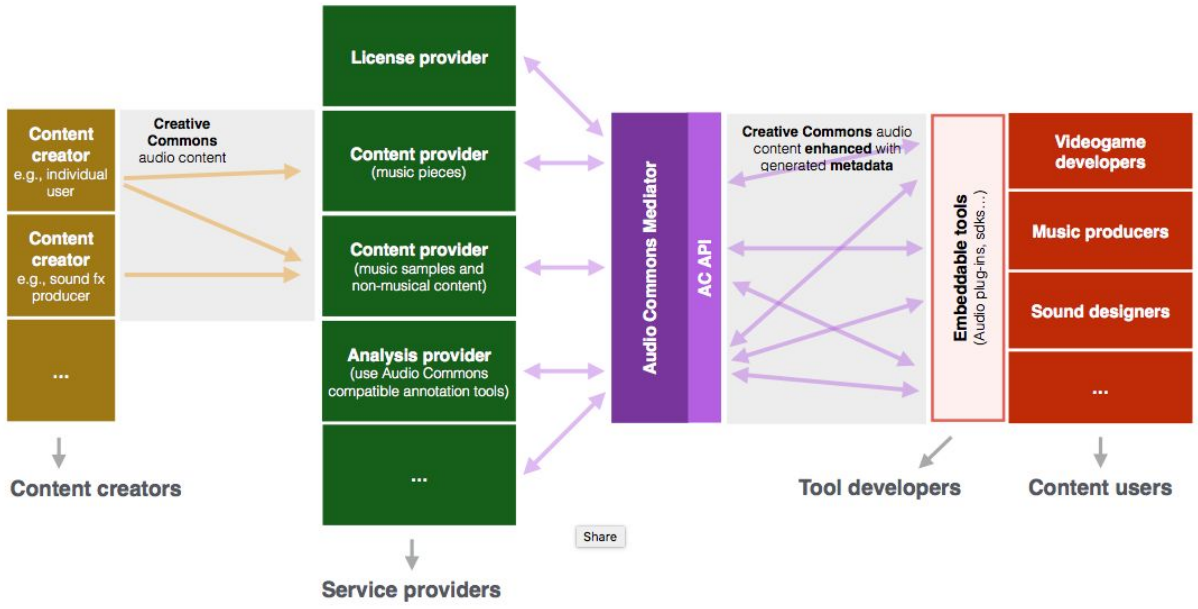
The version of the guidelines presented in this deliverable is expected to be a draft version. At the end of the project, a final version of this document is expected to be delivered (D2.7) which will include the final and full guidelines for service integration.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

1 Guidelines for integrating services in the Audio Commons Ecosystem

1.1 Architecture of the ecosystem and the Audio Commons Mediator

The Audio Commons Ecosystem (a diagram of which is depicted below) consists of a number of interconnected services and tools which are used by content creators and content users. In the core of the ecosystem there is the Audio Commons Mediator which is the component that allows the interoperation between the services and tools (applications) of the ecosystem. All communications between services and tools in the Audio Commons Ecosystem are done via HTTP requests and are *mediated* by the Audio Commons Mediator. The HTTP requests must follow the Audio Commons API Specification (see Deliverable D2.4 API Specification), which is implemented by the mediator (see Deliverable 2.5 Service Integration Technologies).



In order to integrate a service with the Audio Commons Ecosystem, the mediator needs to know what are the characteristics of that service and how to communicate with it. In essence, the mediator needs to know how to make requests to that service and how to interpret its responses. Therefore, a service that wants to join the Audio Commons Ecosystem **needs to expose an HTTP endpoint** through which the mediator will be able to communicate.

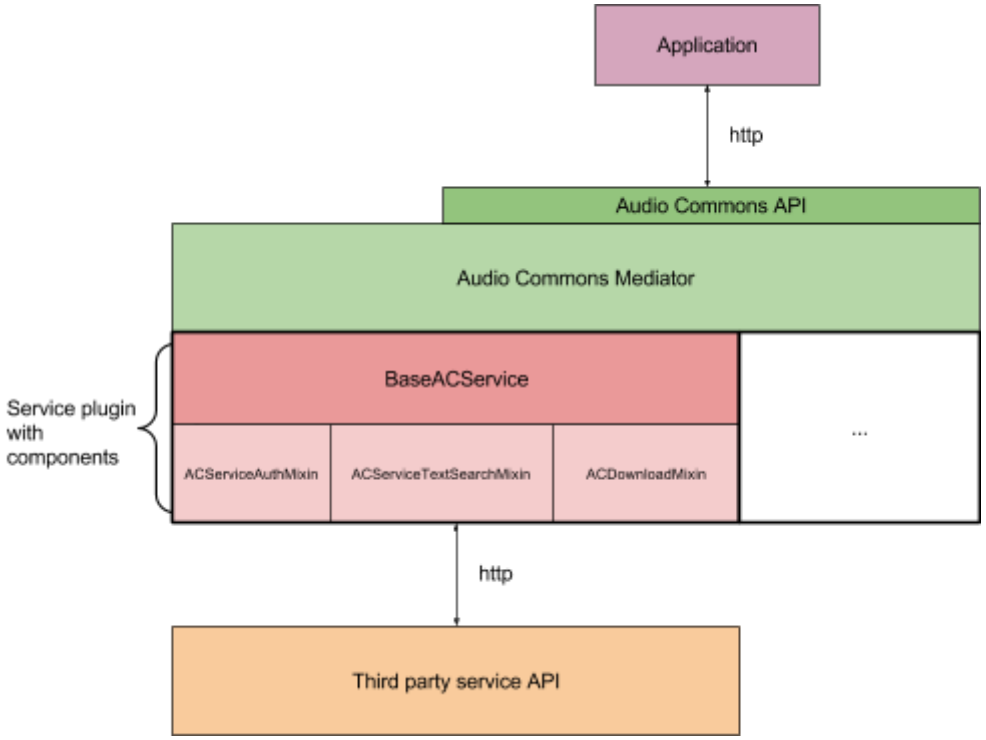
If this requirement is met, a new service can be added to the ecosystem by implementing a ***plugin* which can be loaded** by the Audio Commons Mediator and which tells the mediator what are the capabilities of the service and how to interact with it. This plugin needs to be implemented as a *class* using the Python programming language which includes a number of specific methods to communicate with the mediator. Once this class is implemented, it can be added into a Python module and included in the codebase of the Audio Commons Mediator so it can be loaded and

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

integrated with the ecosystem. The following section explains how to implement such Python class so that it is compatible with the Audio Commons Mediator.

1.2 Adding new services to the ecosystem

The codebase of the Audio Commons Mediator[1] provides a Python package names **acservice**[2] which needs to be used to implement the *service plugin* (a Python class) which communicates the Audio Commons Mediator and the third party service. The acservice package defines a base class **BaseACService** and a number of mixins **ACServiceXXXMixin** which represent the different functionalities that are described in the API and supported in the ecosystem. For example, one of this functionalities is "text search". The Python class representing a service is implemented by combined inheritance of BaseACService and a number of other mixins. For example, if a service supports text search and download features, its plugin will consist of a Python class which inherits from BaseACService, ACServiceTextSearchMixin and ACServiceDownloadMixin and implements a number of methods as required in each mixin. Each of this mixins are also called **components**, therefore a service inside the Audio Commons Mediator is defined as a composition of components. The figure below illustrates the architecture of service plugins inside the Audio Commons Mediator and how are these connected to other parts of the ecosystem.



[1] https://github.com/AudioCommons/ac-mediator
[2] https://github.com/AudioCommons/ac-mediator/tree/master/services/acservice

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

Currently, the acservice package defines the following available components/functionalities:

Base (BaseACService)	Needed for all services provides basic elements and features (such as storing service name and id) which are common and used by other components.
Authentication (ACServiceAuthMixin)	Adds support for authenticating requests with the third party service. Some service must implement the option of authenticating end users if an Audio Commons user account is linked with a third party service user account.
Text search (ACServiceTextSearchMixin)	Adds support for searching audio content based on an input textual query. The text search component includes also support for sorting results and deciding which metadata fields should be included in the search results response.
Download (ACDownloadMixin)	Adds support for downloading audio resources **directly from the third party content provider**. This component provides a way in which an application directly connects with a third party service to download an audio file.
Licensing (ACLicensingMixin)	Adds support for licensing audio resources from one or more content providers.

These components have an equivalent with the currently available API endpoints defined in the Audio Commons API specification (see Deliverable D2.4).

In the following subsections we describe the requirements for implementing each of the available types of components and show example implementations from the service plugins currently implemented in the Audio Commons Mediator. Details about very specific implementation aspects are given in the documentation of the acservice package[3] and as comments in the source code.

1.2.1 Base component

All service plugins must inherit from the base class BaseACService. The BaseACService provides methods which are common to the service plugins and which are used by the other components (i.e. mixins) of the plugin. BaseACService is the first class that should be inherited. When inheriting from BaseACService, and number of object level attributes must be defined which include:

- `NAME`: name of the service
- `URL`: web URL of the service (for informative reasons)
- `API_BASE_URL`: base URL of the API exposed by the service

[3] https://m.audiocommons.org/docs/acservice.html

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

Besides these properties, there is a class method that can optionally be overridden:

- `validate_response_status_code`: this method is called after a response is received from a request to the third party service API. This method is expected to check the status code of the received response and see if there are any errors. In case of errors, this method should *translate* the received error and raise an appropriate Audio Commons Mediator (ACException or ACAPIException objects[4]). If no specific ACException or ACAPIException object represents the received error, then a generic ACException object with a custom message should be raised.

  If `validate_response_status_code` is not overridden, then a default implementation is used in which a generic ACException is raised if the status code of a request response is not equal to 200 (HTTP response code for successful responses).

What follows is an implementation example of the base component for the Freesound service:

```
class FreesoundService(BaseACService, ...):

    NAME = 'Freesound'
    URL = 'http://www.freesound.org'
    API_BASE_URL = "https://www.freesound.org/apiv2/"

    def validate_response_status_code(self, response):
        if self.TEXT_SEARCH_ENDPOINT_URL in response.request.url:
            # If request was made to search endpoint, translate 404 to 'page
            # not found exception'
            if response.status_code == 404:
                raise ACAPIPageNotFound
        if 'download/link' in response.request.url:
            # If request was made to download link endpoint, translate 404 to
            # 'resource does not exist'
            if response.status_code == 404:
                raise ACAPIResourceDoesNotExist
        if response.status_code != 200:
            raise ACException(response.json()['detail'], response.status_code)
        return response.json()
```

1.2.2 Authentication component

The authentication component is mandatory to all service plugins as it specifies how should the mediator authenticate requests to third party services. The ACServiceAuthMixin package provides two types of authentication by default which are APIKEY_AUTH_METHOD and ENDUSER_AUTH_METHOD.

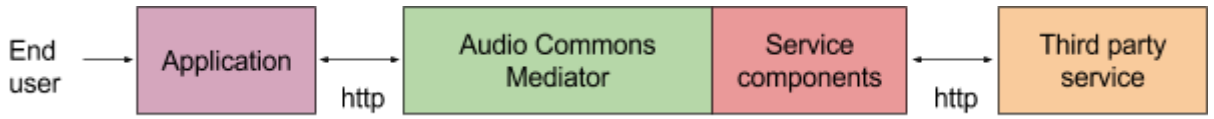
The first method (APIKEY_AUTH_METHOD) consists of the use of an application token which is added to every request to the third party service to identify the entity who's making the request. The token that's used is specified as a configuration parameter of the service (see section 1.2.6). The Audio Commons Mediator always uses the same token when communicates with a specific service. Developers of applications that connect to the Audio Commons Ecosystem do not need to request

[4] Available exceptions can be found in https://github.com/AudioCommons/ac-mediator/blob/master/ac_mediator/exceptions.py

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

API credentials for individual third party services as they are not directly going to communicate with them.

The second authentication method (ENDUSER_AUTH_METHOD) is used when the third party service wants to authenticate the end user who is accessing their service through an application that communicates with the Audio Commons mediator (see figure below).



Authentication between the application and the mediator is done through OAuth2 (see Deliverable D2.4), therefore the mediator always knows which end user is making a particular request and through which application. However, to authenticate an end user with a third party service, the mediator needs to know to which user account of the third party service a particular Audio Commons user account corresponds. This is done through linking of Audio Commons user accounts with third party services' user accounts (see Deliverable D2.5). When using ENDUSER_AUTH_METHOD, requests sent to the third party are authenticated with a valid third party service user account. This might be required in specific situations such as when a sound needs to be uploaded to the service. In this case it is quite probable that the sound needs to be linked to a specific user account and therefore ENDUSER_AUTH_METHOD is needed. The service providers can decide, for each supported component of the ecosystem which authentication method is required. ENDUSER_AUTH_METHOD is implemented following the OAuth2 specification.

By implementing the authentication component, service plugins tell the mediator how should requests by authenticated in one of the aforementioned two methods. If a service does not support any of these two methods, then custom code needs to be provided.

When inheriting from ACServiceAuthMixin, and number of object level attributes must be defined which include:

- `SUPPORTED_AUTH_METHODS`: a list of the supported auth methods
- `BASE_AUTHORIZE_URL`: authorize URL for the OAuth2 flow (needed for the Audio Commons mediator to provide the "link accounts" functionality. This is only needed if ENDUSER_AUTH_METHOD is supported.
- `ACCESS_TOKEN_URL`: URL where to retrieve access tokens as defined in the OAuth2 specification. This is only needed if ENDUSER_AUTH_METHOD is supported.
- `REFRESH_TOKEN_URL`: URL where to retrieve refresh tokens as defined in the OAuth2 specification. This is only needed if ENDUSER_AUTH_METHOD is supported.

When inheriting from ACServiceAuthMixin, the following class methods need to be implemented:

- `get_auth_info_for_request`: this method should return instructions about how should the authentication tokens be included in requests to the third party service API (e.g. in a header, as a request parameter, etc.). See the acservice documentation for specific details about how should these instructions be indicated. What follows is an example implementation of this method as taken from Freesound's service implementation:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

```
def get_auth_info_for_request(self, auth_method, account=None):
    header_content = 'Token {0}'.format(self.get_apikey())
    if auth_method == ENDUSER_AUTH_METHOD:
        header_content = 'Bearer {0}'.format(self.get_enduser_token(account))
        return {'headers': {'Authorization': header_content}}
```

- `get_access_token_from_credentials`: when a user links his Audio Commons account with a third party service account using the link functionality of the mediator, some credentials are returned and stored as a JSON object in the mediator database. This method takes as input an object with the stored credentials and returns the access token (which is part of the credentials). What follows is an example implementation of this method as taken from Freesound's service implementation:

```
def get_access_token_from_credentials(self, credentials):
    return credentials.credentials['access_token']
```

- `get_refresh_token_from_credentials`: similar to the previous method, but returning the refresh token instead (which can be used to renew a old access token).

- `check_credentials_are_valid`: this method also takes as input some previously stored credentials and tells the mediator whether this credentials are valid and therefore can be used (i.e. the access token is not expired yet). What follows is an example implementation of this method as taken from Freesound's service implementation:

```
def check_credentials_are_valid(self, credentials):
    date_expired = credentials.modified + \
        datetime.timedelta(seconds=credentials.credentials['expires_in'])
    if timezone.now() > date_expired:
        raise ACAPIInvalidCredentialsForService
```

1.2.3 Search component

The search component is the most complex of the components of a service plugin. Search component must be able to not only translate text search query parameters to the equivalent parameters in the third party API search endpoint, but it also must be able to interpret the complex response returned by the third party service. The search component in the acservice package is designed to provide i) a set of common search functionalities which mainly consist in interpreting and translating search results received from a service to a format that the mediator understands; and ii) a set of methods through which queries can be formulated in different ways (i.e. based on text, based on sound similarity, etc.). In this way, methods for interpreting search responses only need to be implemented once regardless of the number of methods available for formulating queries. The current implementation of search component in acservice package only supports text-based search

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

queries (where the input is some string with query terms), but the current design will allow the easy addition of other methods for specifying queries in the future.

When inheriting from ACServiceTextSearchMixin, and number of object level attributes must be defined which include:

- `TEXT_SEARCH_ENDPOINT_URL`: sets the URL of the third service API endpoint for search (or a suitable search endpoint that can take as input a string of query terms).

When inheriting from ACServiceTextSearchMixin, the following methods are expected to be implemented:

- `process_q_query_parameter`: this method takes as input a query parameter 'q' defined by the Audio Commons API and converts it into a dictionary of the required request parameters of third party search API endpoint. The following code example is taken from the Jamendo service plugin implementation. In Jamendo API, the query parameter is named 'search':

```
def process_q_query_parameter(self, q):
    return {'search': q}
```

- `process_s_query_parameter`: this method takes as input a query parameter 's' which indicates the sorting preference for the returned results. The Audio Commons Mediator defines a number of sorting options[5], and this method translates from the mediator sorting option to the corresponding third party service sorting option. If there is no equivalent, then a default value can be set but a warning can be raised. Similarly to the previous method, it returns a dictionary with the request parameters that will need to be set in the request to the third party service in order to return sounds with the specified ordering. An example of the implementation of this method can be seen online at https://github.com/AudioCommons/ac-mediator/blob/master/services/3rd_party/freesound.py#L104 (we don't include here to avoid very long sections of code).

ACServiceTextSearchMixin is a Python class which inherits from BaseACServiceSearchMixin, therefore, when implementing the search components there are also a number of methods from BaseACServiceSearchMixin that also need to be overridden:

- `get_results_list_from_response`: third party services are expected to return search results in some structured format that can be converted into a Python dictionary. The current version of acservice package expects a JSON response from third party services. Given that JSON response, this method returns all returned results in a single (sorted) list. The example implementation from the Jamendo service plugin is as follows:

```
def get_results_list_from_response(self, response):
   return response['results']
```

- `get_num_results_from_response`: similarly to the previous method, given the contents of a response as a Python dictionary this method extracts the information about the total number of found results (if present) and returns it.

- `process_size_query_parameter`: the Audio Commons API specifies a 'size' request parameter which can be used to decide how many search results per page should be received

[5] http://github.com/AudioCommons/ac-mediator/blob/master/services/acservice/constants.py#L71

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

from third party services. Similarly to the previously described "`process_X_query_parameter`" methods, this one returns the necessary query parameters that must be added to the request to the third party service to obtain a list of results of length 'size'. The following code shows an example implementation from Freesound:

```
def process_size_query_parameter(self, size, common_search_params):
    size = int(size)
    if size > 150:  # This is Freesound's maximum page size
        self.add_response_warning("Maximum 'size' is 150")
        size = 150
    return {'page_size': size}
```

See how in this example the service plugin checks if size is outside the bounds that are known for Freesound and adds a warning to the response using the `add_response_warning` class method. This method is defined in BaseACService class therefore can be used anywhere in the service plugin. When used, it stores a message which will be delivered to the original application (i.e. added to the aggregated response, see Deliverable D2.5).

- `process_page_query_parameter`: similarly to other methods, this one is used to request a specific page of results to the third party service. Example implementation from Freesound:

```
def process_page_query_parameter(self, page, common_search_params):
    return {'page': page}
```

- `add_extra_search_query_params`: this last method is used to add any other query parameters that are required by the third party service on its search API endpoint. This only needs to be implemented if there are any requirements of extra parameters. It should also return a dictionary with key and value pairs of additional query parameters to be added to the final request.

Besides all of the above methods, there is one last thing that search component needs to implement in order to be able to understand the returned search results. As mentioned above, search responses from third party services are expected to include a list of results (which is accessed by `get_results_list_from_response`). Each individual result in that list is expected to be a dictionary with key/value pairs as sound metadata. This is standard way of returning search results in RESTful APIs, see the following example response from Freesound:

```
{
    ...
    "results": [
        {
            "id": 320079,
            "name": "riesen_roboterarm.wav",
            "tags": [
                "marker",
                "industrial",
                "field-recording",
                "Servo",
                "trade-fair"
            ],
            "license": "http://creativecommons.org/licenses/by-nc/3.0/",
            "username": "system_fm"
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

```
        },
        {
            "id": 320078,
            "name": "gabelstabler.wav",
            "tags": [
                "marker",
                "industrial",
                "field-recording",
                "trade-fair"
            ],
            "license": "http://creativecommons.org/licenses/by-nc/3.0/",
            "username": "system_fm"
        },
        ...
    ],
    ...
}
```

Every service will include different information for each result and different services will typically use different names and have different ways to indicate specific metadata properties. The final version of the Audio Commons Ontology (not published yet) will include a section about "**sound schema**", which unifies metadata names for sound properties. The search component of a service plugin is in charge of translating from individual third party metadata "schemas" to the common names and types of values indicated in the Audio Commons Ontology. In order to do that, a number of methods can be implemented in the service plugin which take as input the Python dictionary corresponding of an individual search result coming from the third party service, and returns a specific Audio Commons compatible metadata field.

There are a number of metadata fields that can be easily translated from the service response to the Audio Commons compatible format. This can be defined using a method called `direct_fields_mapping` which in essence includes a dictionary of Audio Commons sound schema field names and their equivalent name in the third party service response. These can be translated by simply changing the metadata field name. What follows is an example implementation of this method for the case of Freesound:

```
def direct_fields_mapping(self):
   return {
       FIELD_URL: 'url',
       FIELD_NAME: 'name',
       FIELD_AUTHOR_NAME: 'username',
       FIELD_TAGS: 'tags',
       FIELD_DURATION: 'duration',
       FIELD_FILESIZE: 'filesize',
       FIELD_CHANNELS: 'channels',
       FIELD_BITRATE: 'bitrate',
       FIELD_BITDEPTH: 'bitdepth',
       FIELD_SAMPLERATE: 'samplerate',
       FIELD_FORMAT: 'type',
       FIELD_COLLECTION_URL: 'pack',
       FIELD_DESCRIPTION: 'description',
       FIELD_LICENSE_DEED_URL: 'license',
   }
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

Nevertheless, there are some metadata fields which can not be translated so easily. For each of these fields, an extra method must be provided with a decorator that tells for which Audio Commons metadata property it implements translation. The following examples (taken from the Freesound plugin) exemplify the definition of these functions:

```
@translates_field(FIELD_PREVIEW)
def translate_field_preview(self, result):
    # The URL of s preview is nested in a 'previews' dictionary instead of
    # at the root of the individual result dictionary
    return result['previews']['preview-hq-ogg']

@translates_field(FIELD_AUTHOR_URL)
def translate_field_author_url(self, result):
    # The URL to the author's page in the content provider site needs to be
    # manually constructed as it is not directly returned by Freesound
    return self.API_BASE_URL + 'users/{0}/'.format(result['username'])
```

This is all that is needed to add search support to a service of the Audio Commons Ecosystem. If done properly it only requires a few lines of code to integrate basic search functionality for content providers and make their content accessible in the Audio Commons Ecosystem.

1.2.4 Download component

An essential feature of the Audio Commons Ecosystem is that audio resources published in it should be available for download regardless of their later usage (as per Creative Commons licenses). However, when an application needs to download an audio resource it should download it from the content provider (the third party service) hosting the resource and not via the mediator. Because applications are always expected to interact with services through the mediator and don't know how to directly *talk* to third party services, the Audio Commons Mediator needs to provide a way through which applications can download from third party services without really knowing how to interact with them.

The solution that the Audio Commons Mediator proposes is the implementation of the download component which service plugins of content providers should include. This download component is supposed to provide download URLs that applications can use to download a resource from a content provider without the need of being authenticated or knowing any specifics of its API. The idea is that the Audio Commons Mediator requests a *direct download link* for a particular audio resource to a content provider and then hands the returned download link to the application so that it can download the content directly from the content provider.

In order to implement the download component, service plugins must inherit from ACDownloadMixin and define the following property:

- `DOWNLOAD_ACID_DOMAINS`: this should be a list of Audio Commons resource identifier (ACID) domains for which the service provides download links. ACID domains are service names. Typically a content provider will only provide download links for its content, therefore `DOWNLOAD_ACID_DOMAINS` will typically be a list of one single element which include the service name as defined in the base component (Section 1.2.1).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

Besides that property, the service plugin must also implement the following method:

- `get_download_url`: this method sends a request to the third party service and returns a direct download URL that can then be handed to an application form downloading a resource without the need of any authentication. What follows is an example implementation from the Freesound service plugin:

```
def get_download_url(self, context, acid, *args, **kwargs):

    # Translate ac resource id to Freesound resource id
    if not acid.startswith(self.id_prefix):
        raise ACAPIInvalidACID
    resource_id = acid[len(self.id_prefix):]
    try:
        int(resource_id)
    except ValueError:
        raise ACAPIInvalidACID

    response = self.send_request(
        self.API_BASE_URL + 'sounds/{0}/download/link/'.format(resource_id),
        use_authentication_method=ENDUSER_AUTH_METHOD,
        account=Account.objects.get(id=context['user_account_id']),
    )
    return response['download_link']
```

Note that to be able to implement the download component, third party services need to have a way to generate such direct download links built in their APIs or need to have some download URLs which serve the content without the need of authentication. If this requirement is not met, then the download component can not be implemented.

1.2.5 Licensing component

The Audio Commons API specifies that a service that implements licensing functionality should be able to return a URL that a content user can access to obtain a usage license of a particular Audio Commons audio resource. Therefore, the Audio Commons Mediator is expected to carry out any licensing operation besides telling a content user *where to go* to get a license of a particular Audio Commons resource.

The licensing component therefore works in a very similar way to the download component in the sense that it returns a URL given an Audio Commons identifier (ACID). In order to implement the licensing component, service plugins must inherit from ACLicensingMixin and define the following property:

- `LICENSING_ACID_DOMAINS`: this should be a list of Audio Commons resource identifier (ACID) domains for which the service provides licensing URLs. ACID domains are service names. Typically a content provider will only provide licensing for its content, therefore `LICENSING_ACID_DOMAINS` will typically be a list of one single element which include the service name as defined in the base component (Section 1.2.1).

Besides that property, the service plugin must also implement the following method:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

- `get_licensing_url`: this method sends a request to the third party service and returns a licensing URL that can then be handed to an application and shown to the content user. What follows is an example implementation from the Jamendo service plugin:

```
def get_licensing_url(self, context, acid, *args, **kwargs):
    if not acid.startswith(self.id_prefix):
        raise ACAPIInvalidACID
    resource_id = acid[len(self.id_prefix):]
    response = self.send_request(
        self.TEXT_SEARCH_ENDPOINT_URL,
        params={'id': resource_id, 'include': 'licenses'},
    )
    if response['headers']['results_count'] == 0:
        raise ACAPIResourceDoesNotExist
    return response['results'][0].get('prourl', None)
```

1.2.6 Configuring a service

In the sections above we have explained how to implement different components into a service plugin. However, service plugins need to be configured with some configuration parameters such as API credentials. These configuration parameters need to be defined in a configuration file and are automatically loaded by the service manager of the Audio Commons Mediator and made available to the service plugins.

Different third party services might require different configuration parameters. The different components of service plugins can implement a `conf_XXX` method which is given the configuration parameters from the configuration file and can store them as required. The typical use case is the loading of third party services' API credentials. These credentials (typically `CLIENT_ID` and `CLIENT_SECRET`), are stored in the configuration file, loaded by the mediator and passed to the authentication component by calling the `conf_auth` method with the configuration as parameter. The default implementation of the authentication component expects client ids and secrets to be in the configuration parameters, its conf_auth method is implemented as:

```
def conf_auth(self, config):
    if 'client_id' not in config:
        raise ImproperlyConfiguredACService('Missing item \'client_id\'')
    if 'client_secret' not in config:
        raise ImproperlyConfiguredACService('Missing item \'client_secret\'')
    self.set_credentials(config['client_id'], config['client_secret'])
```

The other typical use case is the loading of service IDs provided by Audio Commons. Each service is provided a unique ID which is loaded in the service plugin base component. In this file: https://github.com/AudioCommons/ac-mediator/blob/master/services/services_conf.example.cfg you can see an example configuration file which stores all configuration parameters from third party services.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

1.2.7 Service description

In order to carry out service orchestration, the Audio Commons Mediator needs to know the functionalities supported by each service and therefore decide how to forward requests based on that (see Deliverable D2.5). Also, the Audio Commons API defines an endpoint which applications can access to see which services are connected to the Audio Commons Ecosystem and what they provide.

The BaseACService class implements a method called `get_service_description` which inspects every individual component implemented in the service plugin and automatically returns a description of its capabilities. By implementing the components as described in the sections above, the service description can be done automatically without the need of any extra method to be implemented by developers. However, in future iterations of the mediator (see next section) we expect to use a declarative service description approach which can be interpreted both by the mediator and by third party applications and which use standardised service description frameworks like WSDL and OWL-S[6] . What follows is an example service description for the current implementation of the Freesound service plugin.

```
'Freesound': {
    'id': 'aaa099c0',
    'url': 'http://www.freesound.org',
           'components': ['download', 'text_search'],
    'description': {
        'download': {
            'acid_domains': ['Freesound']
        },
        'text_search': {
            'supported_fields': ['ac:duration', 'ac:bitdepth', 'ac:samplerate',
                                 'ac:channels', 'ac:license_deed_url',
                                 'ac:description', 'ac:format', 'ac:author',
                                 'ac:bitrate', 'ac:collection_url', 'ac:url',
                                 'ac:filesize', 'ac:tags', 'ac:name',
                                 'ac:timestamp', 'ac:license', 'ac:author_url',
                                 'ac:id', 'ac:preview_url'],
            'supported_sort_options': ['-relevance', '-popularity', 'popularity',
                                 '-duration', 'duration', '-downloads',
                                 'downloads', '-created', 'created']
        },
    },
}
```

[6] https://www.w3.org/Submission/OWL-S/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

2 Conclusions and future directions

In this deliverable we have provided draft guidelines about how to integrate new services to the Audio Commons Ecosystem. Integrating a new service is achieved by implementing a service plugin using the Python programming language and a Python package called acservice and included in the Audio Commons Mediator codebase.

We have seen that a service plugin is defined by the implementation of a number of components which add support for specific functionalities like 'text search' or 'download'. Service plugins need to inherit from specific class mixins and overwrite a number of methods and class properties to enable support for each component.

Examples of already implemented service plugins of the prototype version of the Audio Commons Ecosystem can be found in the Audio Commons Mediator source code repository: https://github.com/AudioCommons/ac-mediator/tree/master/services/3rd_party.

The current version of acservice package is functional and allows to easily add new services to the ecosystem. However, as the AudioCommons project advances we will improve the package and add new functionalities. In future updates we expect the following changes:

- Add new components and modify existing as required by updates in the Audio Commons API specification.
- Allow the service plugin to limit access quota (implement throttling) to specific API endpoints. Depending on how this functionality is designed, it could either be implemented in the Audio Commons Mediator itself or strictly as part of the acservice package.
- Provide integrated plugin testing in acservice package or as an external program. We will provide a tool that developers can use to test if their implementation of the service plugin is compatible with the Audio Commons Mediator and to diagnose potential problems or incompatibilities. For example, such a tool could test a search component and raise warnings if some query parameters or metadata fields are not supported by the component implementation. Such ideas have already been explored in the Audio Commons Mediator codebase[7] and deployed in the mediator, but have not been incorporated as part of the acservice package.
- The acservice is expected to be more integrated with the Audio Commons Ontology and take advantage of semantic-web technologies to refactor some of its current functionalities. This also applies to expected developments of the Audio Commons Mediator in general (see Deliverable D2.5). In particular we plan to explore a way to provide a single declarative service description which can be read and interpreted by the acservice package and which provides all needed information to incorporate a service into the ecosystem. In such case, to add a new service to the ecosystem developers should simply provide that service description file instead of implementing a Python class. To provide this service description we plan to take advantage of existing semantic web service description protocols such as WSDL[8] or OWL-S[9]
- Finally, we aim to make the AC API fully Linked Data compatible by tightly integrating the AC API responses with the the AC Ontology. This will be achieved using a semantic web and linked data compatible extension of the popular JSON messaging protocol currently used by the majority of Web APIs. This extension called JSON-LD is fully compatible with, and may be

[7] https://github.com/AudioCommons/ac-mediator/blob/master/services/views.py
[8] https://www.w3.org/TR/wsdl20/
[9] https://www.w3.org/Submission/OWL-S/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382

interpreted as conventional JSON data, however, it provides the additional benefit of consuming the API response as a Linked Data graph. This facilitates the inclusion of unique identifiers in all messages that expand into URIs providing globally unique names for metadata fields as well as audio items and other resources referred to in the messages. This provides the mechanism for tying responses directly to ontology-based schemata (effectively providing formal schema for otherwise ad-hoc JSON data) and allows for the interpretation of responses as semantic graphs which can be processed more effectively using graph search or transformation algorithms. This facilitates declarative interpretation and alleviates the need for hard-coding implementations against human readable API documentations that are amenable to constant change and evolution.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382