AURA

Release 0.2.0

Mar 07, 2021

Contents

AURA is an API specification for music libraries. Music players—from HTML5 applications to mobile apps to embedded devices—use AURA to access servers that host catalogs of music. An AURA server can act as a personal alternative to centralized cloud services like Spotify or Rdio.

The AURA protocol is a lightweight and open alternative to DLNA or DAAP.

The API specification is organized as a *core API* that reflects the basic concepts and optional *extensions*. While not every sever will implement the same extensions, clients can assume that those that do will implement them in the same way.

Contents:

CHAPTER 1

Core API Specification

This document describes the core AURA protocol, which is a simple REST API built on JSON resources. The core protocol includes basic, read-only access to *tracks* and, optionally, organization into *albums* and *artists*. It exposes both metadata and audio.

The API adheres to the JSON API 1.0 specification.

This description uses words like "SHOULD" and "MUST" in all caps to invoke their meaning according to RFC 2119.

1.1 Organization

The API root **SHOULD** appear under a prefix named `/aura/`. This facilitates servers with multiple APIs, allows for human-readable content at the root on the same server, and provides for forward compatibility: future versions of this spec may recommend `/aura2/`, for example.

1.1.1 Response Format and Errors

The MIME type for all responses **MUST** be `application/vnd.api+json`. Every response is a JSON object. When a request is successful, the document has a top-level key `data` corresponding to the response's "primary data." When it fails, the document has an `errors` key, which maps to an array of JSON API error objects. Other keys may also be present, as described below.

1.2 Server Information

`GET /aura/server`
: The "root" endpoint exposes global information and status for the AURA server. The response's `data` key maps to a resource object dictionary representing the server. The object's `attributes` key **MUST** contain these keys:

  - **aura-version**, string: The version of the AURA spec implemented.

- **server**, string: The name of the server software.
- **server-version**, string: The version number of the server.
- **auth-required**, bool: Whether the user has access to the server. For unsecured servers, this may be true even before authenticating.

It **MAY** also contain these keys:

- **features**, string array: A list of optional features the server supports.

```
GET /aura/server HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "server",
    "id": "0",
    "attributes": {
      "aura-version": "0.2.0",
      "server": "aura-ref",
      "server-version": "0.2.1",
      "auth-required": false,
      "features": ["albums"]
    }
  }
}
```

1.3 Resources and Collections

The core resource in AURA is the *track*, which represents a single audio file and its associated metadata.

The server may also optionally group tracks into *albums* and *artists*. Since tracks represent the music itself, albums and artists are not required—clients **SHOULD** disable features that depend on browsing by album, for example, when the server only exposes individual tracks. Clients can still filter tracks by metadata that indicates the album or artist they belong to. AURA's optional concepts of *albums* and *artists* are appropriate when the server supports metadata that is independent of the constituent tracks: cover art for albums, for example, or home towns for artists.

Every resource is represented as a JSON object. Each resource type has a list of keys that are *required* on each object and a list of *optional* fields that the server may support. Servers may also provide other, non-standard fields not listed in this specification. The optional fields are included in an effort to standardize the name and format of common (albeit not universal) metadata.

Names of attributes, including non-standard attributes, **SHOULD** only contain characters matched by the regular expression `[a-zA-Z0-9_-]`.

1.3.1 Relationships

In AURA, there are *relationships* among resources. For example, a track can have a relationship to its containing album and, conversely, an album has relationships to its tracks.

JSON API relationships appear under the `relationships` key within the resource object, which maps to an object. Values in that object are JSON API "relationship objects": in AURA, these are wrappers for resource linkages, which indicate the ID of another resource. For example, a track object links to its album like this:

```
GET /aura/tracks/42 HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "track",
    "id": "42",
    "attributes": {
       // ...
    },
    "relationships": {
      "albums": {
        "data": [ { "type": "album", "id": "84" } ]
      }
    }
  }
}
```

This means that the client can get more information about the album at `/aura/albums/84`.

The client can request inclusion of related resources. The client provides an `include` request parameter containing a comma-separated list of resources. The response then **MUST** include any such objects referenced in `relationships` under an `included` key in the top-level response object. That `included` key maps to an array of resource objects. (This kind of response is called a compound document in JSON API.) For example:

```
GET /aura/tracks/42?include=album HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "track",
    "id": "42",
    "attributes": {
       // ...
    },
    "relationships": {
      "albums": {
        "data": [ { "type": "album", "id": "84" } ]
      }
    }
  },
  "included": [
    {
      "type": "album",
      "id": "84",
      // ...
    }
  ]
}
```

1.3.2 Filtering

Servers provide filtered lists of resources according to metadata. To request a subset of a collection, the client uses request parameters specifying the fields or links to filter on. If the client sends a parameter `filter[key]=value`, the server **MUST** respond with only those resources whose `key` field exactly matches `value`.

For example, the request `/aura/tracks?filter[title]=Blackbird` finds the track titled "Blackbird".

Filtering is by exact match only (i.e., no substring or case-insensitive matching is performed). More flexible queries may be eventually be specified in an AURA extension.

If there are no exact matches, or if the server does not support filtering by the given key, then the `data` key of the response should be an empty array.

1.3.3 Sorting

Sorting of collections and subsets of collections follows the JSON API sorting specification. Sort fields correspond to keys in a resource's `attributes` member.

This example shows albums sorted by descending release date (newest first):

```
GET /aura/albums?sort=-year,-month,-day HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "type": "album",
      "id": "42",
      "attributes": {
        // ...
        "year": 2019,
        "month": 3,
        "day": 24,
        // ...
      }
    },
    {
      "type": "album",
      "id": "39",
      "attributes": {
        // ...
        "year": 2018,
        "month": 12,
        "day": 6,
        // ...
      }
    },
    // ...
  ]
}
```

If not all resources in the collection have the attribute specified by the sort parameter, then the server **SHOULD** return only those resources with the attribute. For example, the request `/aura/tracks?sort=composer` should return only those tracks with a `composer` attribute.

1.3.4 Pagination

Collection endpoints can return truncated results to avoid potential performance issues on both the client and the server. Pagination works using a *pagination token* that describes how to retrieve the next chunk of results. (In practice, the token could be the offset in the collection, the id of the next item to return, or a reference to a database cursor.) Truncation can be requested by the client or unilaterally imposed by the server.

Pagination applies to the three collection endpoints (`/aura/tracks`, `/aura/albums`, and `/aura/artists`). A server **MAY** truncate its responses. If it does so, it **MUST** provide pagination information in the `links` object of its response. That object **MUST** have a `next` member with a URL to the next page if one is available—otherwise, the `next` member may be null or missing altogether. The URL for the next page **MUST** be the same as the original, except that the `page` request holds a different value.

A pagination token is not guaranteed to be useful indefinitely. If a token expires, the server **MAY** respond to subsequent requests with the same token with an HTTP 410 "Gone" error. (This is critical for servers that retain state for each in-progress pagination sequence.)

The client **MAY** include a `limit` parameter (an integer) with a collection `GET` request. The server **MUST** respond with *at most* that number of resources, although it may return fewer. (A `next` link must be supplied if there are more results, as above.)

For example, a client could request a "page" of results with a single result:

```
GET /aura/tracks?limit=1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [ ... ],
  "links": {
    "next": "http://example.org/aura/tracks?limit=1&page=sometoken"
  }
}
```

The client can then issue another request for the next chunk:

```
GET /aura/tracks?limit=1&page=sometoken
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [ ... ]
}
```

The absence of a `links.next` URL indicates that the sequence is finished (there are only two tracks in the library).

1.4 Tracks

An AURA server **MUST** expose a collection of tracks (i.e., individual songs). Information about a track is provided in a track resource object, which is in the form of a JSON API resource object. The top-level key `type` of all track resource objects **MUST** be the string `"track"`.

GET /aura/tracks
The collection of all tracks in the library. The reponse is a JSON object whose `data` key maps to an array of track resource objects.

GET /aura/tracks/(*id*)
An individual track resource. The response is a JSON object whose `data` key maps to a single track resource object.

1.4.1 Required Attributes

Track resource objects **MUST** have these attributes:

- `title`, string: The song's name.
- `artist`, string: The recording artist.

1.4.2 Optional Attributes

Tracks resource objects **MAY** have these attributes:

- `album`, string: The name of the release the track appears on.
- `track`, integer: The index of the track on its album.
- `tracktotal`, integer: The number of tracks on the album.
- `disc`, integer: The index of the medium in the album.
- `disctotal`, integer: The number of media in the album.
- `year`, integer: The year the track was released.
- `month`, integer: The release date's month.
- `day`, integer: The release date's day of the month.
- `bpm`, integer: Tempo, in beats per minute.
- `genre`, string: The track's musical genre.
- `recording-mbid`, string: A MusicBrainz recording id.
- `track-mbid`, string: A MusicBrainz track id.
- `composer`, string: The name of the music's composer.
- `albumartist`, string: The artist for the release the track appears on.
- `comments`, string: Free-form, user-specified information.

These optional attributes reflect audio metadata:

- `mimetype`, string: The MIME type of the associated audio file.
- `duration`, float: The (approximate) length of the audio in seconds.
- `framerate`, integer: The number of frames per second in the audio.
- `framecount`, integer: The total number of frames in the audio. (The exact length can be calculated as the product of the frame rate and frame count.)
- `channels`, integer: The number of audio channels. (A frame consists of one sample per channel.)
- `bitrate`, integer: The number of bits per second in the encoding.

- `bitdepth`, integer: The number of bits per sample.
- `size`, integer: The size of the audio file in bytes.

Support for multi-valued attributes like `artists` and `genres` may be specified in a future AURA extension.

1.4.3 Relationships

Track resources **MAY** have relationships to albums they appear on, their recording artists and any associated images using the `albums`, `artists` and `images` fields respectively. These keys are also the valid values for the `include` parameter (see *Relationships*).

1.5 Albums

Album resources are optional. If a server supports artists, it **MUST** indicate the support by including the string `"albums"` in its `features` list (see *Server Information*). If the server does not support albums, it **MUST** respond with an HTTP 404 error for all `/aura/albums` URLs. Information about an album is provided in an album resource object, which is in the form of a JSON API resource object. The top-level key `type` of all album resource objects **MUST** be the string `"album"`.

`GET /aura/albums`
The collection of all albums in the library. The response is a JSON object whose `data` key maps to an array of album resource objects.

`GET /aura/albums/`(*id*)
An individual album resource. The response is a JSON object whose `data` key maps to a single album resource object.

1.5.1 Required Attributes

Album resource objects **MUST** have these attributes:

- `title`, string: The album's name.
- `artist`, string: The names of the artist responsible for the release (or another indicator such as "Various Artists" when no specific artist is relevant).

1.5.2 Optional Attributes

Album resource objects **MAY** have these attributes:

- `tracktotal`, integer: The number of tracks on the album.
- `disctotal`, integer: The number of media in the album.
- `year`, integer: The year the album was released.
- `month`, integer: The release date's month.
- `day`, integer: The release date's day of the month.
- `genre`, string: The album's musical genres.
- `release-mbid`, string: A MusicBrainz release id.
- `release-group-mbid`, string: A MusicBrainz release group id.

Support for multi-valued attributes like `artists` and `genres` may be specified in a future AURA extension.

1.5.3 Relationships

Album resources **MUST** link to their constituent tracks via the `tracks` field. They **MAY** also link their performing artists and associated images under the `artists` and `images` fields. These keys are also the valid values for the `include` parameter (see *Relationships*).

1.6 Artists

Artist resources are optional. If a server supports artists, it **MUST** indicate the support by including the string `"artists"` in its `features` list (see *Server Information*). If the server does not support artists, it **MUST** respond with an HTTP 404 error for all `/aura/artists` URLs. Information about an artist is provided in an artist resource object, which is in the form of a JSON API resource object. The top-level key `type` of all artist resource objects **MUST** be the string `"artist"`.

`GET /aura/artists`
: The collection of all artists in the library. The response is a JSON object whose `data` key maps to an array of artist resource objects.

`GET /aura/artists/` (*id*)
: An individual artist resource. The response is a JSON object whose `data` key maps to a single artist resource object.

1.6.1 Required Attributes

Artist resource objects **MUST** have these attributes:

- `name`, string: The artist's name.

1.6.2 Optional Attributes

Artist resource objects **MAY** have these attributes:

- `artist-mbid`, string: A MusicBrainz artist id.

1.6.3 Relationships

Artist resources **MUST** have relationships to their associated tracks under the `tracks` field. They **MAY** also link to their albums and associated images under the `albums` and `images` fields. These keys are also the valid values for the `include` parameter (see *Relationships*).

1.7 Images

Image resources are optional. If a server supports images, it **MUST** indicate the support by including the string `"images"` in its `features` list (see *Server Information*). If the server does not support images, it **MUST** respond with an HTTP 404 error for all `/aura/images` URLs. Information about an image is provided in an image resource object, which is in the form of a JSON API resource object. The top-level key `type` of all image resource objects **MUST** be the string `"image"`.

Images can be associated with tracks, albums, and artists. Most pertinently, albums may have associated cover art.

In contrast to the other resource types, servers **SHOULD** respond with an HTTP 404 error for the URL `/aura/images`. This is because enumerating all images may be difficult for the server, and a large collection of image metadata is not generally useful to music browsers and players.

The flexible string nature of resources' `id` field can be used to easily give images globally unique ids. For example, `"album-3-cover.jpg"` could be used to identify the cover image of the album with id `"3"`. This type of id may be useful if image information is not stored in a database.

GET /aura/images/(*id*)
Get metadata about a specific image. The response is a JSON object where the `data` key maps to a single image resource object.

GET /aura/images/(*id*)**/file**
Download an image file. The response's `Content-Type` header **MUST** indicate the mimetype of the image file returned.

1.7.1 Required Attributes

Image resource objects have no required attributes.

1.7.2 Optional Attributes

These fields on image resource objects are optional:

- `role`, string: A description of the image's purpose: "cover" for primary album art, etc.
- `mimetype`, string: The MIME type of the image.
- `width`, integer: The image's width in pixels.
- `height`, integer: The image's height in pixels.
- `size`, integer: The size of the image data in bytes.

1.7.3 Relationships

Images **MAY** have relationships to any associated tracks, albums or artists using the `tracks`, `albums` and `artists` fields. These keys are also the valid values for the `include` parameter (see *Relationships*). Each image resource **MUST** have at least one relationship.

1.8 Audio

The server supplies audio files for each track.

GET /aura/tracks/(*id*)**/audio**
Download the audio file for a track.

The file is returned in an arbitrary audio file format. The server **MUST** set the `Content-Type` header to indicate the format.

The server **SHOULD** use the HTTP Content-Disposition header to supply a filename.

The server **SHOULD** support HTTP range requests to facilitate seeking in the file.

1.8.1 Audio Formats and Quality

The server can provide multiple encodings of the same audio—i.e., by transcoding the file. This can help when the client supports a limited range of audio codecs (e.g., in browser environments) and when bandwidth is limited (e.g., to avoid streaming lossless audio over a mobile connection).

The server decides which version of the file to send using HTTP content negotiation. Specifically, the client **MAY** specify the kinds of content it requests in the HTTP `Accept` header. The header is a comma-separated list of types, which consist of a MIME type and (optionally) some parameters. To request audio under a maximum bitrate, the client uses a `bitrate` parameter to specify the maximum bits per second it is willing to accept.

For example, the header `Accept: audio/ogg, audio/mpeg` requests audio in either MP3 or Ogg (Vorbis, Opus, etc.) format with no quality constraints. Similarly, `Accept: audio/ogg;bitrate=128000` requests Ogg audio at a bitrate of 128kbps or lower.

The server **SHOULD** respond with one of the requested types or a 406 Not Acceptable status (i.e., if it does not support transcoding). An omitted `Accept` header is considered equivalent to `audio/*`.

Also see the complete list of endpoints.

API Reference

/aura/albums

/aura/artists

/aura/images

/aura/server

/aura/tracks