Amara Documentation

Release 1.2.0

Participatory Culture Foundation

Oct 02, 2017

Contents

Contents:

CHAPTER 1

Running Amara

Check out the Quick Start on our github page (http://github.com/pculture/unisubs/).

CHAPTER 2

Reporting bugs

When you are reporting a bug, please look over the following suggestions. The more information you can provide, the faster the bug can be fixed. And you will life easier for developers.

- In what environment is the bug happening?

CHAPTER 3

API Documentation

Amara provides a REST API to interactive with the site. Please contact us if you'd like to use the Amara API for commercial purposes.

Overview

Authentication

Before interacting with the API, you must have an API key. In order to get one, create a user on the Amara website, then go to the edit profile page. At the bottom of the page you will find a "Generate new key" button . Clicking on it will fetch your user the needed API key.

Every request must have the username and the API keys as headers. For example:

```
X-api-username: my_username_here
X-api-key: my_api_key_here
```

Note: You can also use the deprecated X-apikey header to specify your key

So a sample request would look like this:

GET https://amara.org/api/videos/

Request Headers

- *X-api-username* – <Username>
- *X-api-key* – <API key>

Data Formats

The API accepts request data in the several formats. Use the `Content-Type` HTTP header to specify the format of your request:

Format	Content-Type
JSON *(recommended)*	application/json
XML	application/xml
YAML	application/yaml

In this documentation, we use the term "Request JSON Object" to specify the fields of the objects sent as the request body. Replace "JSON" with "XML" or "YAML" if you are using one of those input formats.

Here's an example of request data formated as JSON:

```
{"field1": "value1", "field2": "value2", ... }
```

By default we will return JSON output. You can the `Accept` header to select a different output format. You can also use the `format` query param to select the output formats. The value is the format name in lower case (for example `format=json`).

We also support text/html as an output format and application/x-www-form-urlencoded and multipart/form-data as input formats. However, this is only to support browser friendly endpoints. It should not be used in API client code.

Paginated Responses

Many listing API endpoints are paginated to prevent too much data from being fetched and returned at one time (for example the video listing API). These endpoints are marked with `paginated` in their descriptions. Paginated responses only return limited number of results per request, alongside links to the next/previous page.

Here's an example paginated response from the Teams listing:

```
{
    "meta": {
        "previous": null,
        "next": "http://amara.org/api/teams?limit=20&offset=20",
        "offset": 0,
        "limit": 20,
        "total_count": 151
    },
    "objects": [
        {
            "name": "",
            "slug": "tedx-import",
            "description": "",
            "is_visible": true,
            "membership_policy": "Open",
            "video_policy": "Any team member"
        },
        ...
    ]
}
```

- The `meta` field contains pagination information, including next/previous links, the total number of results, and how many results are listed per page
- The `objects` field contains the objects for this particular page

Browser Friendly Endpoints

All our API endpoints can be viewed in a browser. This can be very nice for exploring the API and debugging issues. To view API endpoints in your browser simply log in to amara as usual then paste the API URL into your address bar.

Value Formats

- Dates/times use ISO 8601 formatting
- Language codes use BCP-47 formatting

Use HTTPS

All API requests should go through https. This is important since an HTTP request will send your API key over the wire in plaintext.

The only exception is when exploring the API in a browser. In this case you will be using the same session-based authentication as when browsing the site.

API interaction overview

All resources share a common structure when it comes to the basic data operations.

- `GET` request is used to viewing data
- `POST` request is used for creating new items
- `PUT` request is used for updating existing items
- `DELETE` request is used for deleting existing items

To view a list of videos on the site you can use

`GET https://amara.org/api/videos/`

To get info about the video with id "foo" you can use

`GET https://amara.org/api/videos/foo`

Many of the available resources will allow you to filter the response by a certain field. Filters are specified as GET parameters on the request. For example, if you wanted to view all videos belong to a team called "butterfly-club", you could do:

`GET https://amara.org/api/videos/?team=butterfly-club`

In addition to filters, you can request that the response is ordered in some way. To order videos by title, you would do

`GET https://amara.org/api/videos/?order_by=title`

To create a video you can use

`POST https://amara.org/api/videos/`

To update the video with video id *foo* use:

`PUT https://amara.org/api/videos/foo`

API Changes / Versioning

Sometimes we need to make backwards incompatible changes to the API. Here's our system for allowing our partners and other API consumers to deal with them:

- All changes are announced on the Amara Development Blog and the API Changes mailing list.
- When we make a change, we give clients between six weeks and three months of transition time, depending on the complexity of the changes, to update their code to work with the new system.
- During the transition time, we return an HTTP header to indicate that the API will be changing. The name is `X-API-DEPRECATED` and the value is the date the API will change in `YYYYMMDD` format.
- Clients can start using the new API during the transition time by sending the `X-API-FUTURE` header. The value should be the date of the API that you want to use, also in `YYYYMMDD` format. If the `X-API-FUTURE` date is >= the switchover date then the new API code will be used.
- You can use `X-API-FUTURE` to test changes to your API client code and to deploy new code that works with the updated API. Using this method you can ensure your integration works seamlessly through the API change.
- If you aren't able to change your request headers, then you can also use the api-future query parameter (for example `/api/videos/?api-future=20151021`)

Languages

Languages Resource

API endpoint that lists all available languages on the Amara platform.

GET /api/languages/

Response JSON Object

- **languages** – maps language codes to language names

Videos

Videos Resource

List/Search/Lookup videos on Amara

Listing Videos

GET /api/videos/

List videos. You probably want to specify a query filter parameter to limit the results

List results are paginated.

Query Parameters

- **video_url** (*url*) – Filter by video URL
- **team** (*slug*) – Filter by team
- **project** (*slug*) – Filter by team project. Passing in *null* will return only videos that don't belong to a project

- **order_by** (*string*) – Change the list ordering. Possible values:
  - *title*: ascending
  - *-title*: descending
  - *created*: older videos first
  - *-created*: newer videos

Note:

- If no query parameter is given, the last 10 public videos are listed.
- If you pass in the project filter, you need to pass in a team

---

Get info on a specific video

GET /api/videos/(video-id)/

Response JSON Object

- **id** (*video-id*) – Amara video id
- **primary_audio_language_code** (*bcp-47*) – Audio language code
- **title** (*string*) – Video title
- **description** (*string*) – Video description
- **duration** (*integer*) – Video duration in seconds (or null if not known)
- **thumbnail** (*url*) – URL to the video thumbnail
- **created** (*iso-8601*) – Video creation date/time
- **team** (*slug*) – Slug of the Video's team (or null)
- **metadata** (*dict*) – Dict mapping metadata names to values
- **languages** (*list*) – List of languages that have subtitles started. See below for a a description.
- **video_type** (*char*) – Video type identifier
- **all_urls** (*list*) – List of URLs for the video (the first one is the primary video URL)
- **activity_uri** (*uri*) – Video Activity Resource
- **urls_uri** (*url*) – Video URL Resource
- **subtitle_languages_uri** (*uri*) – Subtitle languages Resource
- **resource_uri** (*uri*) – Video Resource
- **original_language** (*string*) – contains a copy of primary_audio_language_code **(deprecated)**

Language data:

Response JSON Object

- **code** (*string*) – Language code
- **name** (*string*) – Human readable label for the language

- **published** (*boolean*) – Are the subtitles publicly viewable?
- **dir** (*string*) – Language direction ("ltr" or "rtl")
- **subtitles_uri** (*url*) – Subtitles Resource
- **resource_uri** (*url*) – Subtitles Language Resource

Adding a video

POST /api/videos/

Request JSON Object

- **video_url** (*url*) – The url for the video. Any url that Amara accepts will work here. You can send the URL for a file (e.g. http:///www.example.com/my-video.ogv), or a link to one of our accepted providers (youtube, vimeo, dailymotion)
- **title** (*string*) – title of the video
- **description** (*string*) – About this video
- **duration** (*integer*) – Duration in seconds, in case it can not be retrieved automatically by Amara
- **primary_audio_language_code** (*string*) – language code for the main language spoken in the video.
- **thumbnail** (*url*) – URL to the video thumbnail
- **metadata** (*dict*) – Dictionary of metadata key/value pairs. These handle extra information about the video. Right now the type keys supported are *speaker-name* and *location*. Values can be any string.
- **team** (*string*) – team slug for the video or null to remove it from its team.
- **project** (*string*) – project slug for the video or null to put it in the default project.

Note:

- When submitting URLs of external providers (i.e. youtube, vimeo), the metadata (title, description, duration) can be fetched from them. If you're submitting a link to a file (mp4, flv) then you can make sure those attributes are set with these parameters. Note that these parameters (except the video duration) override any information from the original provider or the file header.
- For all fields, if you pass an empty string, we will treat it as if the field was not present in the input (**deprecated**).
- For slug and project, You can use the string "null" as a synonym for the null object (**deprecated**).

Update an existing video

PUT /api/videos/(video-id)/

This uses the same fields as video creation, excluding *video_url*.

As with creating video, an update can not override the duration received from the provider or specified in the file header.

Delete an existing video

DELETE /api/videos/(video-id)/
This deletes a video.

If there are any subtitles/collaborations on the video, they will also be deleted.

Moving videos between teams and projects

- To move a video from one team to another, you can make a put request with a `team` value.
- Similarly, you can move a video to a different project using the `project` field. *team* must also be given in this case.
- The user making the change must have permission to remove a video from the originating team and permission to add a video to the target team.

Video URL Resource

Each video has at least 1 URL associated with it, but some can have more. This allows you to associate subtitles with the video on multiple video providers (e.g. a youtube version, a vimeo version, etc).

One video URL is flagged the *primary URL*. This is what will gets used in the embedder and editor.

List URLs for a video

GET /api/videos/(video-id)/urls/
List results are paginated.

Response JSON Object

- **video-id** (*string*) – Amara video ID
- **created** (*iso-8601*) – creation date/time
- **url** (*url*) – URL string
- **primary** (*boolean*) – is this the primary URL for the video?
- **original** (*boolean*) – was this the URL that was created with the video?
- **resource_uri** (*uri*) – Video URL Resource
- **videoid** (*string*) – ID on the Hosting platform
- **type** (*string*) – Video type (Youtube, Vimeo, HTML5, etc.)
- **id** (*integer*) – Internal ID for the object **(deprecated, use resource_uri rather than trying to construct API URLs yourself)**.

Get details on a specific URL

GET (video-url-endpoint)
The response fields are the same as for the list endpoint

Use the *resource_uri* from the listing to find the video URL endpoint

Add a URL for a video

POST /api/videos/(video-id)/urls/

Request JSON Object

- **url** (*url*) – Video URL. This can be any URL that works in the add video form for the site (mp4 files, youtube, vimeo, etc). Note: The URL cannot be in use by another video.
- **primary** (*boolean*) – If True, this URL will be made the primary URL
- **original** (*boolean*) – Is this is the first url for the video?

Making a URL the primary URL for a video

PUT (video-url-endpoint)

Request JSON Object

- **primary** – Pass in true to make a video URL the primary for a video

Use the *resource_uri* from the listing to find the video URL endpoint

Deleting Video URLs

DELETE (video-url-endpoint)

Remove a video URL from a video

Use the *resource_uri* from the listing to find the video URL endpoint

Subtitles

Subtitle Language Resource

Container for subtitles in one language for a video. Subtitle languages are typically created when the first editing session is started.

To see all possible languages see *Supported languages*.

Listing languages for a video

GET /api/videos/(video-id)/languages/

Get a list of subtitle languages for a video

List results are paginated.

Response JSON Object

- **language_code** (*bcp-47*) – Subtitle language
- **name** (*string*) – Human-readable name for this language
- **is_primary_audio_language** (*boolean*) – Is this language the primary language spoken in the video?
- **is_rtl** (*boolean*) – Is this language RTL?
- **resource_uri** (*uri*) – Subtitle Language Resource

- **created** (*iso-8601*) – when the language was created
- **title** (*string*) – Video title, translated into this language
- **description** (*string*) – Video description, translated into this language
- **metadata** (*dict*) – Video metadata, translated into this language
- **subtitles_complete** (*boolean*) – Are the subtitles complete for this language?
- **subtitle_count** (*integer*) – Number of subtitles for this language
- **reviewer** (*string*) – Username of the reviewer for task-based teams
- **approver** (*string*) – Username of the approver for task-based teams
- **is_translation** (*boolean*) – Is this language translated from other languages? **(deprecated)**
- **published** (*boolean*) – Are the subtitles publicly viewable?
- **original_language_code** (*string*) – Source translation language **(deprecated)**
- **num_versions** (*integer*) – Number of subtitle versions, the length of the versions array should be used instead of this **(deprecated)**
- **id** (*integer*) – Internal ID for the language **(deprecated)**
- **is_original** (*boolean*) – alias for is_primary_audio_language **(deprecated)**
- **versions** (*list*) – List of subtitle version data. See below for details.

Subtitle version data:

Response JSON Object

- **author** (*user-data*) – Subtitle author (see *User fields*)
- **version_no** (*integer*) – number of the version
- **published** (*boolean*) – is this version publicly viewable?

Note: *original_language_code* and *is_translation* fields are remnants from the old subtitle system. With the new editor, users can use multiple languages as a translation source. These fields are should not be relied on.

Getting details on a specific language

GET /api/videos/(video-id)/languages/(language-code)/

The response data is the same as the listing

Creating subtitle languages

POST /api/videos/(video-id)/languages/

Request JSON Object

- **language_code** (*string*) – bcp-47 code for the language
- **is_primary_audio_language** (*boolean*) – Is this the primary spoken language of the video? *(optional)*.
- **subtitles_complete** (*boolean*) – Are the subtitles for this language complete? *(optional)*.

- **is_original** (*boolean*) – Alias for is_primary_audio_language **(deprecated)**
- **is_complete** (*boolean*) – Alias for subtitles_complete **(deprecated)**

Subtitles Resource

Subtitle data in one language for a video.

Fetching subtitles for a given language

GET /api/videos/(video-id)/languages/(language-code)/subtitles/

Query Parameters

- **sub_format** – The format to return the subtitles in. This can be any format that amara supports including dfxp, srt, vtt, and sbv. The default is json, which returns subtitle data encoded list of json dicts.
- **version_number** – version number to fetch. Versions are listed in the VideoLanguageResouce request. If none is specified, the latest public version will be returned. If you want the latest private version (and have access to it) use "last".
- **version** – Alias for version_number **(deprecated)**

Response JSON Object

- **version_number** (*integer*) – version number for the subtitles
- **subtitles** (*object*) – Subtitle data. The format depends on the sub_format param
- **author** (*user-data*) – Subtitle author (see *User fields*)
- **sub_format** (*string*) – Format of the subtitles
- **language** (*object*) – Language data
- **title** (*string*) – Video title, translated into the subtitle's language
- **description** (*string*) – Video description, translated into the subtitle's language
- **metadata** (*string*) – Video metadata, translated into the subtitle's language
- **video_title** (*string*) – Video title, translated into the video's language
- **video_description** (*string*) – Video description, translated into the video's language
- **notes_uri** (*uri*) – Subtitle notes resource
- **actions_uri** (*uri*) – Subtitle actions resource
- **resource_uri** (*uri*) – Subtitles resource
- **site_uri** (*url*) – URL to view the subtitles on site
- **video** (*string*) – Copy of video_title **(deprecated)**
- **version_no** (*integer*) – Copy of version_number **(deprecated)**

Language data:

Response JSON Object

- **code** (*bcp-47*) – Language of the subtitles

- **name** (*string*) – Human-readable name for the language
- **dir** (*string*) – Language direction ("ltr" or "rtl")

Getting subtitle data only

Sometimes you want just subtitles data without the rest of the data. This is possible using a special Accept header or the *format* query parameter. This can be used to download a DFXP, SRT, or any other subtitle format that Amara supports. If one of these is used, then the sub_format param will be ignored.

Format	Accept header	format query param
DFXP	application/ttml+xml	dfxp
SBV	text/sbv	sbv
SRT	text/srt	srt
SSA	text/ssa	ssa
WEBVTT	text/vtt	vtt

Examples:

```
GET /api/videos/(video-id)/languages/(language-code)/subtitles/?format=dfxp
```

```
GET /api/videos/(video-id)/languages/(language-code)/subtitles/
Accept: application/ttml+xml
```

Creating new subtitles

POST /api/videos/(video-id)/languages/(language-code)/subtitles/

Request JSON Object

- **subtitles** (*object*) – The subtitles to submit, as a string. The format depends on the sub_format param.
- **subtitles_url** (*object*) – Alternatively, subtitles can be given as a text file URL. The format depends on the sub_format param.
- **sub_format** (*string*) – The format used to parse the subs. The same formats as for fetching subtitles are accepted. Optional - defaults to "dfxp".
- **title** (*string*) – Give a title to the new revision
- **description** (*string*) – Give a description to the new revision
- **action** (*string*) – Name of the action to perform - optional, but recommended. If given, the is_complete param will be ignored. For more details, see the subtitles action documentation by following the *actions_uri* field.
- **is_complete** (*boolean*) – Boolean indicating if the complete subtitling set is available for this language - optional, defaults to false. **(deprecated, use action instead)**

Deleting subtitles

DELETE /api/videos/(video-id)/languages/(language-code)/subtitles/

This will delete all subtitle versions for a language. It's only allowed if the video is part of a team and the API user is an admin for that team.

Subtitle Actions Resource

Subtitle actions are operations on subtitles. Actions correspond to the buttons in the upper-right hand corner of the subtitle editor (save, save a draft, approve, reject, etc). This resource is used to list and perform actions on the subtitle set.

Note: You can also perform an action together with adding new subtitles using the action field of the subtitles resource.

Listing actions

GET /api/videos/(video-id)/languages/(language-code)/subtitles/actions/
Get a list of possible actions for a subtitle set.

Response JSON Object

- **action** (*string*) – Action name
- **label** (*string*) – Human-friendly string for the action
- **complete** (*boolean*) – Does this action complete the subtitles? If true, then when the action is performed, we will mark the subtitles complete. If false, we will mark them incomplete. If null, then we will not change the subtitles_complete flag.

Performing actions

POST /api/videos/(video-id)/languages/(language-code)/subtitles/actions/
Perform an action on a subtitle set. This is like opening the subtitles in the editor, not changing anything, and clicking an action button (Publish, Save Draft, etc.)

Request JSON Object

- **action** (*string*) – name of the action to perform

Subtitle Notes Resource

Subtitle notes saved in the editor.

Note: Subtitle notes are currently only supported for team videos

Fetching notes

GET /api/videos/(video-id)/languages/(language-code)/subtitles/notes

Response JSON Object

- **user** (*username*) – Username of the note author
- **datetime** (*iso-8601*) – when the note was created
- **body** (*string*) – text of the note.

Adding notes

POST /api/videos/(video-id)/languages/(language-code)/subtitles/notes/

Request JSON Object

- **body** (*string*) – note body

Users

Users Resource

Fetching user data

GET /api/users/[identifier]/

Parameters

- **identifier** (*user-identifier*) – See *User Identifiers*

Response JSON Object

- **username** (*username*) – username
- **id** (*string*) – user ID
- **first_name** (*string*) – First name
- **last_name** (*string*) – Last name
- **homepage** (*url*) – Homepage URL
- **biography** (*string*) – Bio text
- **num_videos** (*integer*) – Number of videos followed by the user
- **languages** (*list*) – List of language codes for languages the user speaks.
- **avatar** (*url*) – URL to the user's avatar image
- **activity_uri** (*uri*) – User Activity resource
- **resource_uri** (*uri*) – User resource
- **full_name** (*string*) – Full name of the user.

Note:

- Many of these fields will be blank if the user hasn't set them from their profile page
- The `full_name` field is not used in the amara interface and there is no requirement that it needs to be first_name + last_name. This field is for API consumers that want to create users to match their internal users and use the full names internally instead of first + last.

Creating Users

POST /api/users/

Request JSON Object

- **username** (*username*) – 30 chars or fewer alphanumeric chars, @, _ and are accepted.
- **email** (*email*) – A valid email address
- **password** (*string*) – any number of chars, all chars allowed.
- **first_name** (*string*) – Any chars, max 30 chars. **(optional)**
- **last_name** (*string*) – Any chars, max 30 chars. **(optional)**
- **allow_3rd_party_login** (*boolean*) – If set, account can be automatically linked to 3rd party account at first login. **(optional)**
- **create_login_token** (*boolean*) – If sent the response will also include a url that when visited will login the created user. Use this to allow users to login without explicitly setting their passwords. This URL expires in 2 hours. **(optional)**
- **find_unique_username** (*boolean*) – If username is taken, we will find a similar, unused, username for the new user. If passed, make sure you check the username returned since it might not be the same one that you passed in. If set, usernames can only be a maximum of 24 characters to make room for potential extra characters. **(optional)**

Note: The response includes the `email` and `api_key`, which aren't included in the normal GET response. If clients wish to make requests on behalf of this newly created user through the api, they must hold on to this data.

Updating user accounts

PUT /api/users/[username]

Parameters

- **username** (*username*) – must match the username of the auth credentials sent

Inputs the same fields as POST, except *username* and *find_unique_username*.

User Identifiers

There are a couple ways to specify users:

- Username
- User ID prefixed with "id$" (`id$abcdef123`)

The user ID method is preferred since it's possible for users to change their username.

User fields

Users are often contained in other resources, for example the team members, subtitle authors, etc. When those users are represented, we use a dict with the following fields:

- username – Username
- id – User ID
- uri – Link to the user API endpoint

Example JSON:

```
{
  "user": {
    "username": "alice",
    "id": "abcdef",
    "uri": "https://amara.org/api/users/id$abcdef/"
  }
}
```

When you post data to an endpoint with a userfield, you can specify the user using any of the identifiers listed abose. For example, to create a team member you can send this data:

```
{
  "user": "id$abcdef",
  "role": "manager"
}
```

Activity

Video Activity Resource

GET /api/videos/(video-id)/activity/

Query Parameters

- **type** (*string*) – Filter by activity type (*Activity Types*)
- **user** (*user-identifier*) – Filter by user who performed the action (see *User Identifiers*)
- **language** (*bcp-47*) – Filter by the subtitle language
- **before** (*iso-8601*) – Only include activity before this date/time
- **after** (*iso-8601*) – Only include activity after

Response JSON Object

- **type** (*string*) – Activity type (*Activity Types*)
- **date** (*iso-8601*) – Date/time of the activity
- **user** (*user-data*) – User who performed the activity (see *User fields*)
- **video** (*video-id*) – Video related to the activity (or null)
- **language** (*bcp-47*) – Language of the subtitles related to the activity (or null)
- **video_uri** (*uri*) – Link to the video resource endpoint
- **language_uri** (*uri*) – Link to the subtitle language resource endpoint

Depending on the activity type, extra fields may be present in the response data (*Activity Types*).

Team Activity Resource

GET /api/teams/(*slug*)**/activity/**

Query Parameters

- **type** (*string*) – Filter by activity type (*Activity Types*)

- **user** (*user-identifier*) – Filter by user who performed the action (see *User Identifiers*)
- **video** (*video-id*) – Filter by video
- **video_language** (*bcp-47*) – Filter by video language
- **language** (*bcp-47*) – Filter by the subtitle language
- **before** (*iso-8601*) – Only include activity before this date/time
- **after** (*iso-8601*) – Only include activity after

Response data is the same as the video activity resource.

User Activity Resource

GET /api/users/(*username*)**/activity/**

Query Parameters

- **type** (*string*) – Filter by activity type (*Activity Types*)
- **video** (*video-id*) – Filter by video
- **video_language** (*bcp-47*) – Filter by video language
- **language** (*bcp-47*) – Filter by the subtitle language
- **team** (*slug*) – Filter by team
- **before** (*iso-8601*) – Only include activity before this date/time
- **after** (*iso-8601*) – Only include activity after

Response data is the same as the video activity resource.

Activity Types

An activity type classifies the activity. Some types have extra data that is associated with them

Type	Created When | Notes/Extra Fields	
video-added	Video added to amara	
comment-added	Comment posted	`language` will be null for video comments and set for subtitle comments
version-added	Subtitle version added	
video-url-added	URL added to video	`url` will contain the new URL
video-url-edited	Primary video URL change	`old_url`/`new_url` will contain the old/new primary URL
video-url-deleted	URL removed from video	`url` will contain the deleted URL
video-deleted	Video deleted from amara	`title` will contain the deleted video's title
Team Related Activity		
member-joined	User joined team	
member-left	User left team	
Task Related Activity		
version-approved	Subtitles approved	
version-rejected	Subtitles sent back by approver	
version-accepted	Subtitles approved by reviewer	
version-declined	Subtitles sent back by reviewer	

Legacy Activity Resource

Deprecated API endpoint that lists contains all amara activity. You should use the team/video/user query param to find the activity you want. New code should use the Video, Team, or User, resources (see above).

List activity

GET /api/activity/

Query Parameters

- **team** (*slug*) – Show only items related to a given team
- **team-activity** (*boolean*) – If team is given, we normally return activity on the team's videos. If you want to see activity for the team itself (members joining/leaving and team video deletions, then add team-activity=1)
- **video** (*video-id*) – Show only items related to a given video
- **type** (*integer*) – Show only items with a given activity type. Possible values:
  1. Add video
  2. Change title
  3. Comment
  4. Add version
  5. Add video URL

6. Add translation
7. Subtitle request
8. Approve version
9. Member joined
10. Reject version
11. Member left
12. Review version
13. Accept version
14. Decline version
15. Delete video

- **language** (*bcp-47*) – Show only items with a given language code
- **before** (*timestamp*) – Only include items before this time
- **after** (*timestamp*) – Only include items after this time

---

Note: If both team and video are given as GET params, then team will be used and video will be ignored.

Get details on one activity item

GET /api/activity/[activity-id]/

Response JSON Object

- **type** (*integer*) – activity type. The values are listed above
- **date** (*datetime*) – date/time of the activity
- **video** (*video-id*) – ID of the video
- **video_uri** (*uri*) – Video Resource
- **language** (*bcp-47*) – language for the activity
- **language_url** (*uri*) – Subtile Language Resource
- **resource_uri** (*uri*) – Activity Resource
- **user** (*username*) – username of the user user associated with the activity, or null
- **comment** (*string*) – comment body for comment activity, null for other types
- **new_video_title** (*string*) – new title for the title-change activity, null for other types
- **id** (*integer*) – object id **(deprecated use resource_uri if you need to get details on a particular activity)**

Messages

Message Resource

POST /api/message/
Send a message to a user/team

Request JSON Object

- **user** (*user-identifier*) – Recipient (see *User Identifiers*)
- **team** (*slug*) – Recipient team's slug
- **subject** (*string*) – Subject of the message
- **content** (*string*) – Content of the message

Note: You can only send either `user` or `team`, not both.

Teams

Team Resource

Get a list of teams

GET /api/teams/
Get a paginated list of all teams

Response JSON Object

- **name** (*string*) – Name of the team
- **slug** (*slug*) – Machine name for the team slug (used in URLs)
- **type** (*string*) – Team type. Possible values:
  - `default` – default team type
  - `simple` – simplified workflow team
  - `collaboration` – collaboration team
- **description** (*string*) – Team description
- **is_visible** (*boolean*) – Should this team's videos be publicly visible (True by default)?
- **membership_policy** (*string*) – Team membership policy. One of:
  - `Open`
  - `Application`
  - `Invitation by any team member`
  - `Invitation by manager`
  - `Invitation by admin`
- **video_policy** (*string*) – Team video policy. One of:

  - `Any team member`
  - `Managers and admins`
  - `Admins only`
- **`activity_uri`** (*`uri`*) – Team activity resource
- **`members_uri`** (*`uri`*) – Team member list resource
- **`projects_uri`** (*`uri`*) – Team projects resource
- **`applications_uri`** (*`uri`*) – Team applications resource (or null if the membership policy is not by application)
- **`languages_uri`** (*`uri`*) – Team preferred/blacklisted languages resource
- **`tasks_uri`** (*`uri`*) – Team tasks resource (or null if tasks are not enabled)
- **`resource_uri`** (*`uri`*) – Team resource

`GET /api/teams/(team-slug)/`
Get details on a single team

The data is the same as the list endpoint

Updating team settings

`PUT /api/teams/(team-slug)`

Request JSON Object

- **`name`** (*`string`*) – (required) Name of the team
- **`slug`** (*`slug`*) – (required) Manchine name for the team (used in URLs)
- **`description`** (*`string`*) – Team description
- **`is_visible`** (*`boolean`*) – Should this team be publicly visible?
- **`membership_policy`** (*`string`*) – Team membership policy. One of:
  - `Open`
  - `Application`
  - `Invitation by any team member`
  - `Invitation by manager`
  - `Invitation by admin`
- **`video_policy`** (*`string`*) – Team video policy. One of:
  - `Any team member`
  - `Managers and admins`
  - `Admins only`

Creating a team

Amara partners can create teams via the API.

`POST /api/teams/`

Request JSON Object

- **name** (*string*) – (required) Name of the team
- **slug** (*slug*) – (required) Manchine name for the team (used in URLs)
- **type** (*string*) – Team type. Possible values:
  - `default` – default team type
  - `simple` – simplified workflow team
  - `collaboration` – collaboration team
- **description** (*string*) – Team description
- **is_visible** (*boolean*) – Should this team be publicly visible?
- **membership_policy** (*string*) – Team membership policy. Possible values:
  - `Open`
  - `Application`
  - `Invitation by any team member`
  - `Invitation by manager`
  - `Invitation by admin`
- **video_policy** (*string*) – Team video policy. Possible values:
  - `Any team member`
  - `Managers and admins`
  - `Admins only`

Members Resource

API endpoint for team memberships

Listing members of a team

GET /api/teams/(team-slug)/members/

Response JSON Object

- **user** (*user*) – User associated with the membership (see *User fields*)
- **role** (*string*) – One of: `owner`, `admin`, `manager`, or `contributor`

Get info on a team member

GET /api/teams/(team-slug)/members/(user-identifier)/

The data is in the same format as the listing endpoint.

See *User Identifiers* for possible values for `user-identifier`

Adding a member to the team

POST /api/teams/(team-slug)/members/

Request JSON Object

- **user** (*user-identifier*) – User to add (see *User Identifiers*)
- **role** (*string*) – One of: `owner`, `admin`, `manager`, or `contributor`

Change a team member's role

PUT /api/teams/(team-slug)/members/(*username*)**/**

Request JSON Object

- **role** (*string*) – One of: `owner`, `admin`, `manager`, or `contributor`

Removing a user from a team

DELETE /api/teams/(team-slug)/members/(*username*)**/**

Projects Resource

List a team's projects

GET /api/teams/(team-slug)/projects/

Response JSON Object

- **name** (*string*) – project name
- **slug** (*slug*) – slug for the project
- **description** (*string*) – project description
- **guidelines** (*string*) – Project guidelines for users working on it
- **created** (*datetime*) – datetime when the project was created
- **modified** (*datetime*) – datetime when the project was last changed
- **workflow_enabled** (*boolean*) – Are tasks enabled for this project?
- **resource_uri** (*uri*) – Project details resource

Get details on a project

GET /api/teams/(team-slug)/projects/(project-slug)/
The data is the same as the listing endpoint

Creating a project

POST /api/teams/(team-slug)/projects/

Request JSON Object

- **name** (*string*) – project name

- **slug** (*slug*) – slug for the project
- **description** (*string*) – project description **(optional)**
- **guidelines** (*string*) – Project guidelines for users working on it **(optional)**

Updating a project

PUT /api/teams/(team-slug)/projects/(project-slug)/
Uses the same data as the POST method

Delete a project

DELETE /api/teams/(team-slug)/projects/(project-slug)/

Tasks Resource

List all tasks for a team

GET /api/teams/(team-slug)/tasks/

Query Parameters

- **assignee** (*user-identifier*) – Show only tasks assigned to a username (see *User Identifiers*)
- **priority** (*integer*) – Show only tasks with a given priority
- **type** (*string*) – Show only tasks of a given type
- **video_id** (*video-id*) – Show only tasks that pertain to a given video
- **order_by** (*string*) – Apply sorting to the task list. Possible values:
  - `created` Creation date
  - `-created` Creation date (descending)
  - `modified` Last update date
  - `-modified` Last update date (descending)
  - `priority` Priority
  - `-priority` Priority (descending)
  - `type` Task type (details below)
  - `-type` Task type (descending)
- **completed** (*boolean*) – Show only complete tasks
- **completed-before** (*integer*) – Show only tasks completed before a given date (as a unix timestamp)
- **completed-after** (*integer*) – Show only tasks completed before a given date (as a unix timestamp)
- **open** (*boolean*) – Show only incomplete tasks

Get details on a specific task

GET /api/teams/(team-slug)/tasks/(task-id)/

Response JSON Object

- **video_id** (*video-id*) – ID of the video being worked on
- **language** (*bcp-47*) – Language code being worked on
- **id** (*integer*) – ID for the task
- **type** (*string*) – type of task. One of `Subtitle`, `Translate`, `Review`, or `Approve`
- **assignee** (*user-data*) – Task assignee (see *User fields*)
- **priority** (*integer*) – Priority for the task
- **created** (*datetime*) – Date/time when the task was created
- **modified** (*datetime*) – Date/time when the task was last updated
- **completed** (*datetime*) – Date/time when the task was completed (or null)
- **approved** (*string*) – Approval status of the task. One of `In Progress`, `Approved`, or `Rejected`
- **resource_uri** – Task resource

Create a new task

POST /api/teams/(team-slug)/tasks/

Request JSON Object

- **video_id** (*video-id*) – Video ID
- **language** (*bcp-47*) – language code
- **type** (*string*) – task type to create. Must be `Subtitle` or `Translate`
- **assignee** (*user-identifier*) – Task assignee (*User Identifiers*)
- **priority** (*integer*) – Priority for the task **(optional)**

Update an existing task

PUT /api/teams/(team-slug)/tasks/(task-id)/

Request JSON Object

- **assignee** (*user-identifier*) – Task assignee (*User Identifiers*)
- **priority** (*integer*) – priority of the task
- **send_back** (*boolean*) – send a truthy value to send the back back **(optional)**
- **complete** (*boolean*) – send a truthy value to complete/approve the task **(optional)**
- **version_number** (*integer*) – Specify the version number of the subtitles that were created for this task **(optional)**

Note: If both send_back and approved are specified, then send_back will take preference.

Delete an existing task

DELETE /api/teams/(team-slug)/tasks/(task-id)/

Notifications Resource

This endpoint can be used to view notifications sent to your team. See *HTTP Callbacks for Teams* for details on how to set up notifications.

List notifications

GET /api/teams/(team-slug)/notifications/

Response JSON Object

- **number** (*integer*) – Notification number
- **url** (*url*) – URL of the POST request
- **data** (*object*) – Data that we posted to the URL.
- **timestamp** (*iso-8601*) – date/time the notification was sent
- **in_progress** (*boolean*) – Is the request still in progress?
- **response_status** (*integer*) – HTTP response status code (or null)
- **error_message** (*string*) – String describing any errors that occured

List results are paginated

Get details for a notification

GET /api/teams/(team-slug)/notifications/(*number*)**/**
This returns information on a single notification. The data has the same format as in the listing endpoint.

Applications Resource

This endpoint only works for teams with membership by application.

List applications

GET /api/teams/(team-slug)/applications

Query Parameters

- **status** (*string*) – Include only applications with this status
- **before** (*integer*) – Include only applications submitted before this time (as a unix timestamp)
- **after** (*integer*) – Include only applications submitted after this time (as a unix timestamp)
- **user** (*user-identifier*) – Include only applications from this user (see *User Identifiers*)

List results are paginated

Get details on a single application

GET /api/teams/(team-slug)/applications/(application-id)/:

Response JSON Object

- **user** (*user-data*) – Applicant user data (see *User fields*)
- **note** (*string*) – note given by the applicant
- **status** (*string*) – status value. Possible values are `Denied`, `Approved`, `Pending`, `Member Removed` and `Member Left`
- **id** (*integer*) – application ID
- **created** (*datetime*) – creation date/time
- **modified** (*datetime*) – last modified date/time
- **resource_uri** (*uri*) – Application resource

Approve/Deny an application

PUT /api/teams/(team-slug)/applications/(application-id)/

Request JSON Object

- **status** (*string*) – `Denied` to deny the application and `Approved` to approve it.

Preferred Languages Resource

Preferred languages will have tasks auto-created for each video.

PUT /api/teams/(team-slug)/languages/preferred/
Send a list of language codes as data.

Blacklisted Languages Resource

Subtitles for blacklisted languages will not be allowed.

PUT /api/teams/(team-slug)/languages/blacklisted/
Send a list of language codes as data.

Subtitle Request Resource

This API endpoint is used for subtitle requests for teams new team that use the collaboration model. At this point it's only used by a limited number of teams, but we hope to make it available to everyone in the near future.

Terminology

Several teams can be involved in a subtitle request. We use these terms to distinguish the teams:

- *source team* – Team that the video is part of.
- *work team* – Team that is creating the subtitles. We call this the *initial work* to differentiate it from the evaluations.

- *evaulation teams* – Teams that are evaluating/checking the subtitles. Each request can have 0-3 evaluation teams.

All subtitle request endpoints are contained inside the main endpoint for a team. When the docs say something like "source team only", it refers to that team.

Listing Requests

GET /api/teams/(team-slug)/subtitle-requests/

Query Parameters

- **type** (*string*) – Type of request. Possible values:
  - `local` *(default)* – The simple case, requests where the team is both the source and work team.
  - `outgoing` – Request where the team is the source team, but not the work team.
  - `incoming` – Request where the team is the work team, but not the source team.
  - `evaluations` – Request where the team is the evaluation team.
- **status** (*string*) – *(source team only)* Filter by overall status. Possible values:
  - `in-progress` – initial work in progress
  - `in-evaluation` – evaluation in progress
  - `in-evaluation1` – 1st evaluation in progress
  - `in-evaluation2` – 2nd evaluation in progress
  - `in-evaluation[N]` – Nth evaluation in progress
  - `complete` – all work complete
- **work_status** (*string*) – *(work team only)* Filter by status of the intial work (not including evaluations).
  - `in-progress` – initial work in progress
  - `available` – assignment currently available
  - `assigned` – assignment in progress
  - `needs-subtitler` – transcribe/translate assignment available
  - `being-subtitled` – transcribe/translate assignment in progress
  - `needs-reviewer` – review assignment available
  - `being-reviewed` – review assignment in progress
  - `needs-approver` – approval assignment available
  - `being-approved` – approval assignment in progress
  - `complete` – initial work complete
- **evaluation_status** (*string*) – *(evaluation teams only)* Filter by status of the evaluation. Possible values:
  - `upcoming` – evaluation not ready to be started
  - `in-progress` – evaluation in progress
  - `available` – evaluation assignment currently available

  - `assigned` – evaluation assignment in progress
  - `complete` – evaluation complete
- **video** (*video-id*) – Filter by video ID
- **video_title** (*string*) – Filter by video title
- **video_language** (*bcp-47*) – Filter by video language
- **language** (*bcp-47*) – Filter by subtitle request language
- **project** (*slug*) – Filter by team project
- **assignee** (*username*) – Filter by assignee
- **sort** (*string*) – Sort order. Possible values:
  - `-creation` *(default)* – creation date/time (latest first)
  - `creation` – creation date/time (earliest first)
  - `-due` – due date/time (latest first)
  - `due` – due date/time (earliest first)
  - `-completion` – latest completion date/time (latest first)
  - `completion` – latest completion date/time (earliest first)

Response JSON Object

- **job_id** (*job-id*) – ID for the request
- **video** (*video-id*) – Video for the request
- **language** (*bcp-47*) – Language code of the subtitle request
- **source_team** (*slug*) – Team that the video is part of
- **team** (*slug*) – Team handling the request (source team only)
- **evaluation_teams** (*slug-list*) – Team evaluating the request. Only present for requests for the team's videos.
- **status** (*string*) – *(source team only)* Overall request status. Possible values:
  - `in-progress` – initial work in progress
  - `in-evaluation1` – 1st evaluation in progress
  - `in-evaluation2` – 2nd evaluation in progress
  - `in-evaluation[N]` – Nth evaluation in progress
  - `complete` – all work complete
- **work_status** (*string*) – *(work team only)* Status of the intial work (not including evaluations). Possible values:
  - `needs-subtitler` – transcribe/translate assignment available
  - `being-subtitled` – transcribe/translate assignment in progress
  - `needs-reviewer` – review assignment available
  - `being-reviewed` – review assignment in progress
  - `needs-approver` – approval assignment available
  - `being-approved` – approval assignment in progress

  - `complete` – initial work complete
- **`evaluation_status`** (*`string`*) – *(evaluation teams only)* Status of the evaluation. Possible values:
  - `upcoming` – evaluation not ready to be started
  - `available` – evaluation assignment currently available
  - `assigned` – evaluation assignment in progress
  - `complete` – evaluation complete
- **`created`** (*`datetime`*) – when the request was created.
- **`completed`** (*`datetime`*) – *(source team only)* when the entire request was completed, or null.
- **`work_completed`** (*`datetime`*) – *(work team only)* when the initial work was completed, or null.
- **`evaluation_completed`** (*`datetime`*) – *(evaluation team only)* when the initial work was completed, or null.
- **`subtitler`** (*`user`*) – *(work team only)* user creating the subtitles (see *User fields*)
- **`reviewer`** (*`user`*) – *(work team only)* user reviewing the subtitles (see *User fields*).
- **`approver`** (*`user`*) – *(work team only)* user approving the subtitles (see *User fields*).
- **`evaluator`** (*`user`*) – *(evaluation teams only)* user evaluating the subtitles (see *User fields*).
- **`video_uri`** (*`uri`*) – Video API resource
- **`subtitles_uri`** (*`uri`*) – Subtitles resource
- **`actions_uri`** (*`uri`*) – Subtitle actions resource
- **`resource_uri`** (*`uri`*) – Subtitle request details resource

List results are paginated.

Request Details

`GET /api/teams/(team-slug)/subtitle-requests/(job-id)/`
Response data is the same as above.

Request Status Fields

We return several status fields for requests (`status`, `work_status`, `evaluation_status`). The status field that you see depends on the team's relationship to the request (where "the team" is identified by the `team-slug` param in the URL). This gives each team a different view of the request. Each team gets the information revelant to it's work, and doesn't get information about the work being done by other teams.

As an example, consider a request that has 2 evaluation teams. As the request progresses, here are the values that we will return for the various status fields:

status	work_status	evaluation_status (team1)	evaluation_status (team2)
in-progress	needs-subtitler	pending	pending
in-progress	being-subtitled	pending	pending
in-progress	needs-reviewer	pending	pending
in-progress	being-reviewed	pending	pending
in-progress	needs-approver	pending	pending
in-progress	being-approved	pending	pending
in-evaluation1	complete	available	pending
in-evaluation1	complete	assigned	pending
in-evaluation2	complete	complete	available
in-evaluation2	complete	complete	assigned
complete	complete	complete	complete

- `status` will be returned to the source team
- `work_status` will be returned to the work team
- `evaluation_status` will be returned to evaluation teams
- Multiple status fields can be returned if the team fits into more than one of the above categories

Creating Requests

POST /api/teams/(team-slug)/subtitle-requests/

Request JSON Object

- **video** (*video-id*) – Video ID. This must be part of the team identified in the URL path
- **language** (*bcp-47*) – language code for the subtitles
- **team** (*slug*) – Team to work on the subtitles. This can be any team you are an admin of. **(optional, defaults to the team the video is a part of.)**
- **evaluation_teams** (*list-of-slugs*) – Teams to evaluate the subtitles after the initial work is done. They can be any team you are an admin of. **(optional)**

The API user must be an admin of the source team to create a subtitle request.

Updating Requests

PUT /api/teams/(team-slug)/subtitle-requests/(job-id)/

Request JSON Object

- **subtitler** (*user*) – *(work team only)* User to assign as the subtitler, or null to unassign the current subtitler (see *User Identifiers*).
- **reviewer** (*user*) – *(work team only)* User to assign as the reviewer, or null to unassign the current reviewer (see *User Identifiers*).
- **approver** (*user*) – *(work team only)* User to assign as the approver, or null to unassign the current approver (see *User Identifiers*).
- **work_status** (*string*) – *(work team only)* Set to "complete" to mark the subtitles complete. This is the only value we currently support.
- **team** (*slug*) – *(source team only)* Change the team working on the subtitle request. This is only possible if work has not been started.

- **evaluation_teams** (*slug*) – *(source team only)* Change the teams evaluating on the subtitle request. This is only possible if evaluations have not been started.
- **evaluator** (*user*) – *(evaluation teams only)* User to assign as the evaluator, or null to unassign the current evaluator (see *User Identifiers*).

Deleting Requests

DELETE /api/teams/(team-slug)/subtitle-requests/(job-id)/

Assigning a user:

```
PUT /api/teams/my-team/subtitle-requests/abc123/

{
    "subtitler": "alice"
}
```

Unassigning a user:

```
PUT /api/teams/my-team/subtitle-requests/abc123/

{
    "subtitler": null
}
```

Marking a subtitle request complete

```
PUT /api/teams/my-team/subtitle-requests/abc123/

{
    "work_status": "complete"
}
```

Moving a subtitle request to another team

```
PUT /api/teams/my-team/subtitle-requests/abc123/

{
    "team": "other-team-slug"
}
```

Endorsing Subtitles

If you have an assignment, you can use the *Subtitles Resource* to submit subtitles. Use the `endorse` action to endorse them, moving the request to the next stage. You can also use the *Subtitle Actions Resource* to endorse the subtitles without submiting any changes.

CHAPTER 4

Supported languages

Abkhazian, Afar, Afrikaans, Akan, Albanian, American, Amharic, Arabic, Aragonese, Armenian, Assamese, Asturian, Avaric, Avestan, Aymara, Azerbaijani, Bambara, Bashkir, Basque, Belarusian, Bengali, Berber, Bihari, Bislama, Bosnian, Breton, Bulgarian, Burmese, Catalan, Cebuano, Chamorro, Chechen, Chewa, Chinese, Choctaw, Church, Chuvash, Cornish, Corsican, Cree, Creole, Croatian, Czech, Danish, Divehi, Dutch, Dzongkha, Efik, English, Esperanto, Estonian, Ewe, Faroese, Fijian, Filipino, Finnish, French, Frisian, Fula, Fulah, Galician, Ganda, Georgian, German, Gikuyu, Greek, Greenlandic, Guaran, Gujarati, Haida, Hausa, Hebrew, Herero, Hindi, Hiri, Hokkien, Hungarian, Hupa, Ibibio, Icelandic, Ido, Igbo, Ilocano, Indonesian, Ingush, Interlingua, Interlingue, Inuktitut, Inupia, Irish, Iroquoian, Italian, Japanese, Javanese, Kannada, Kanuri, Karen, Kashmiri, Kazakh, Khmer, Klingon, Komi, Kongo, Korean, Kuanyama, Kurdish, Kyrgyz, Lakota, Lao, Latin, Latvian, Limburgish, Lingala, Lithuanian, Luba-Kasai, Luba-Katagana, Luhya, Luo, Luxembourgish, Macedo, Macedonian, Madurese, Malagasy, Malay, Malayalam, Maltese, Mandinka, Manipuri, Manx, Maori, Marathi, Marshallese, Metadata:, Mohawk, Moldavian, Mongolian, Mossi, Naurunan, Navajo, Ndonga, Nepali, North, Northern, Norwegian, Occitan, Ojibwe, Oriya, Oromo, Ossetian, Pali, Pashto, Persian, Polish, Portuguese, Punjabi, Quechua, Romanian, Romansh, Rundi, Russian, Rusyn, Rwandi, Samoan, Sango, Sanskrit, Sardinian, Scottish, Serbian, Serbo-Croatian, Shona, Sichuan, Sindhi, Sinhala, Slovak, Slovenian, Somali, Sotho, Southern, Spanish, Sundanese, Swahili, Swati, Swedish, Tagalog, Tahitian, Tajik, Tamil, Tartar, Telugu, Tetum, Thai, Tibetan, Tigrinya, Tonga, Tsonga, Tswana, Turkish, Turkmen, Twi, Ukrainian, Umbundu, Urdu, Uyghur, Uzbek, Venda, Vietnamese, Volapuk, Walloon, Welsh, Wolof, Xhosa, Yiddish, Yoruba, Zhuang, Zulu.

CHAPTER 5

HTTP Callbacks for Teams

Enterprise customers can register a URL for http callbacks so that activity on their teams will fire an HTTP POST requests to that URL.

To register your Team to receive HTTP notifications, please send your request to us at enterprise@amara.org and we will set it up for you. You can also contact us with inquiry about any custom notifications that are not listed in our general offering below.

Please indicate a URL where you'd like to get notified. Each team can have their own URL, or a common URL can be used for several teams. We recommend that the selected URL uses HTTPS protocol for safer communication.

Notification Details

We currently send notifications for the following events related to team videos and team members.

Video notifications

Video notifications always include the following data:

- `event`
- `amara_video_id`
- `youtube_video_id` (null if the video is not hosted on YouTube)
- `team`
- `project`
- `primary_team` (used when the same callback URL is shared between multiple teams and the event that triggered callback happened on another team).

Supported events for videos:

video-added Sent when a video is added to your team, or moved to your team from another team.

Additional data: `old_team` (if video is moved from another team)

video-removed Sent when a video is removed from your team, or moved to another team.

Additional data: `new_team` (if video is moved to another team)

video-made-primary Sent when one of the multiple URLs for a video on your team is set as the primary URL.

Additional data: `url`

video-moved-project Sent when a video on your team is moved to a different project.

Additional data: `old_project`

subtitles-published Sent when a new subtitle version for a video on your team is published.

Additional data: `language_code`, `amara_version`

subtitle-unpublished Sent when subtitles are deleted for a video on your team.

Additional data: `language_code`

Team member notifications

Team member notifications always include the following data:

- `event`
- `username`
- `team`
- `primary_team` (used when the same callback URL is shared between multiple teams and the event that triggered callback happened on another team)

Supported events for team members:

member-added Sent when a user is added to your team.

member-removed Sent when a user is removed from your team.

member-profile-changed Sent when the information in a team member's profile is changed.

For each event we can customize the data that is sent with the notification.

Also, all notifications are numbered. You can use the number field in the notification to keep track of the events in your team(s).

To view previously sent notifications use the *Team Notifications API*.

CHAPTER 6

Babelsubs

We've split the subtitle handling into it's own separate project, Babelsubs. Anything that has do to with parsing, generating and formatting subtitles should be handled over there. The main unisubs repo should only make calls to babelsubs with the desired operations / data.

Storage

Internally, we're storing subtitles as the DFXP format. DFXP is the most complex, and most capable format of all. It's also the only one with a real spec. The advantage is that it lets us tell our users that they can input DFXP, process it throughout our system and get their data out correctly, even for features we don't currently support (like advanced styles).

Formatting

Formatting we **do** support:

- Bold text
- Italic text
- Underline
- Like breaks

Each format handles those different. On DFXP you have attributes on the xmlnodes (span, p and div) such as fontWeigh='bold' and textStyle='italic'. Line breaks are
 tags.

For SRT and friends, we have the 'b', 'i' and 'u' tags. Line breaks are displayed with the right line separator.

For HTML (which is not a download format, but it's displayed on the website), we have 'em', 'strong' and 'style' tags, and 'br' for line breaks.

Ideally, for testing a complete set of features we need to test:

- The forementioned formats (italics, bold, underline)
- Line breaks
- Single “>” and doubles “>>” . This is used to denote speaker changes and is widely used by our customers. They must come out correctly both when displayed on the website (subtitle view, the widget, the dialogs) and when downloaded. On DFXP those should use character entities.

For anything other than these tags, let’s say you have a video on web development, and they write a ‘<script>alert();</script> ‘ tag. Here’s what should happen:

- Should be stored with the tag chars escaped
- Should show up on the website (dialog, subtitle view and the widget) as is, but escaped (javascript shouldn’t run) , but it should be editable
- Non html / xml formats (such as srt) should display them as is

In general, here’s the intended workflow:

- On intake convert what we can to dfxp (such as a line break to
). Do not strip tags.
- On output (for the website only) escape anything other than the tags we expect (<script>, etc)

CHAPTER 7

Syncing and Importing

The externalsites app handles linking Amara users/teams to accounts on externalsites. This allows for:

- Syncing subtitles to the third party site when they are edited on Amara
- Importing new videos for the third party account

We support several sites, each works slightly differently

Youtube

Both user and team accounts can be linked to YouTube accounts, but they are handled slightly differently. The general idea here is that the use case is different for teams and users. In general, teams want to have finer grained control over what gets imported to Amara and what gets synced back to their YouTube channel. For users, we just import everything and sync everything.

User Accounts

- Users can link to YouTube from account section on their profile page
- A user can only link 1 YouTube account
- A YouTube account can only be linked to 1 user
- We create a video feed and import all videos for the YouTube channel.
- All subtitles for a video in that account will be synced

Team Accounts

- Teams can link to YouTube from their Settings -> Integrations page
- A team can link multiple YouTube accounts

- A YouTube account can only be linked to 1 team, but there is a way to share the account with other teams.
- Subtitles are normally only synced for the team's videos
- The linked team can add other teams to the syncing list, any of those team's videos will also be synced.
- We don't auto-import videos for the YouTube channel.
- A YouTube account can't be linked to both a team and a user

Kaltura

- Teams can link to Kaltura from their Settings -> Integrations page
- Once a team links to Kaltura, subtitles on their team videos with their Kaltura partner id will be synced back to Kaltura.

Brightcove

- Teams can link to Brightcove from their Settings -> Integrations page
- Once a team links to Brightcove, subtitles on their team videos with their Brightcove publisher id will be synced back to Brightcove.
- Teams can optionally choose to import videos from their Brightcove account.
- If importing, teams can either import all videos or videos matching certain tags.

CHAPTER 8

Static Media

Static media files are handled by the staticmedia app. This app has several goals:

- **Combine multiple files into a single "media bundle".** Linking to a single JS file results in faster page loads than linking to multiple files.
- **Compress JS/CSS code.**
- **Support preprocessors like SASS.**
- **Support media files served from the local server or S3**
- **Store media files on S3 in a unique location for each deploy.** This allows us to upload media for our next deploy without affecting our current one. It also allows us to the set the expire header to the far future which is good for caching.

Settings

Example

```
MEDIA_BUNDLES = {
    "base.css": {
        "files": (
            "css/v1.scss",
            "css/bootstrap.css",
        ),
    },
    "site.js": {
        "files": (
            "js/jquery-1.4.3.js",
            "js/unisubs.site.js",
        ),
    },
}
```

```
STATIC_MEDIA_COMPRESSED = True

STATIC_MEDIA_USES_S3 = True
AWS_ACCESS_KEY_ID = 'abcdef'
AWS_SECRET_ACCESS_KEY = 'abcdef
STATIC_MEDIA_S3_BUCKET = 'bucket.name'
STATIC_MEDIA_S3_URL_BASE = '//s3.amazonaws.com/bucket.name'
```

MEDIA_BUNDLES

`MEDIA_BUNDLES` defines our Javascript/CSS media bundles.

The keys are the filename that we will generate. The extension of the filename controls what type of media and should either by `js` or `css`.

The values are dicts that determine how we build the bundle. They can have these properties:

`files`
: list of files to bundle together (paths are relative to the media directory)

`add_amara_conf` (*optional*)

: If True, we will prepend javascript code to the source JS files. THis will create global object called `_amaraConf` with these properties:

- `baseURL`: base URL for the amara website
- `staticURL`: base URL to the static media

STATIC_MEDIA_COMPRESSED

Set to False to disable compressing/minifying Javascript and CSS

STATIC_MEDIA_USES_S3

If True we Will Serve media files from amazon S3. This will change the URLs that our template tags create for links to the media bundles. `STATIC_MEDIA_USES_S3` is usually True for production and False for development.

If `STATIC_MEDIA_USES_S3` is enabled, the following settings are available:

- `AWS_ACCESS_KEY_ID`: S3 access key.
- `AWS_SECRET_ACCESS_KEY`: S3 secret key.
- `STATIC_MEDIA_S3_BUCKET`: S3 bucket to store media in.
- `STATIC_MEDIA_S3_URL_BASE`: Base URL for S3 media.

Compilation & Minification

We use uglifyjs for Javascript files and SASS for CSS files. Using the SASS extensions is optional. If you just have regular CSS files that SASS will function simply as a CSS compressor.

Media Directory Structure

Regardless if media is uploaded to S3 or we are serving it from the local instance, we structure the files the same way:

- css/ - CSS bundles
- js/ - Javascript bundles
- images/ - Image files
- fonts/ - font files

When serving media from the local server, the root URL for media files will be `/media/`.

When serving media from S3, the root URL for media files will be `<STATIC_MEDIA_S3_URL_BASE><git-commit-id/`

Development, Media Bundles, and Caching

For development servers, STATIC_MEDIA_USES_S3 is usually False, which causes us to serve up the media bundles from the local server. It takes long enough to compile media bundles that we don't want to re-do it on every page request. So we cache the result and use that for subsequent requests. Before using a cached result, we check the mtime of all source files, and if any one is later than when the cache was created, we rebuild.

This works fine for most use cases, but there are a couple ways that it will fail. For example removing a file from the sources list won't trigger a rebuild. If you think this may be happening, just update the mtime on any source file to trigger the rebuild manually.

In Templates

To link to media files in templates load the `media_bundle` library. Then you can use these tags:

- `media_bundle` – include a CSS/JS media bundle (generates the entire script/link tag)
- `url_for` – Get the URL to a media bundle.
- `static_url` – Get the base URL for static media.

CHAPTER 9

Developer's Guide

This section contains information on the internal workings of the Amara codebase.

Development Workflow

This guide describes the development workflow for Amara.

Contents

Creating issues

Please follow these guidelines when creating issues, to ensure that they are easy to implement:

- Do a quick search to check for any existing issues before creating a new one.
- Make sure the title clearly and succinctly captures the issue at hand
- For bugs, describe the steps needed to reproduce the problem and what the correct behavior is.
- Try to describe the severity of the issue. Who is it affecting? How bad is the current behavior, etc.

Branches / Repositories

The `production` branch is what gets deployed to our production server. It's what gets deployed to production server. `staging` branch is what gets deployed to the staging server.

Commits should *never* be made directly to production and only trivial commits should be made to staging. Instead, Amara development tries to follow a "one branch per feature or bugfix" workflow (See *Workflow*)

As you work on your topic branch, other branches may have been merged into `staging` by other people. Make sure you merge staging back to your branch as often as possible to keep it up-to-date.

Other Git Repositories

Inside the unisubs repository, you may want to check out some other repositories.

If you have access to our private repository (https://github.com/pculture/amara-enterprise/). Check that out inside the root directory of the unisubs repository to add the extra functionality. See *Optional Apps* for details on how this works.

We also have a couple other repositories that integrate into unisubs:

- https://github.com/pculture/babelsubs/
- https://github.com/pculture/unilangs/

Both of these get installed inside your docker container. Normally you don't need to do anything to use them. However, if you want to test changes to those repositories you need to check out a local copy:

- Check out the git repository inside the root unisubs directory.
- Make a symlink from the root directory to the python package (for example: `ln -s babelsubs-git/babelsubs .`)
- After this the unisubs code will be using your local checkout rather than the default package. Make changes there, test them on your dev environment, then commit/push the changes back to a branch on the pculture repository, then open a PR to maste.
- When we deploy amara, we pick up the the latest commit in master for these libraries. So once your changes are merged to master, they will be live the next time we deploy.

Testing

At a minimum, make sure you *run the tests* after your changes and ensure that all tests pass.

If possible, use test driven development. Write new tests that cover the issue you're working on before you start any code. Write code that makes the test pass. Then consider refactoring code to fix the problem in a cleaner way.

Exception Logging

When catching exceptions, be sure to log these with a descriptive message and the stacktrace. Exceptions should be caught whenever it's necessary for flow control, an exception is expected, or where user input may cause unexpected behavior (such as forms). In the case where a caught exception is an expected part of flow control, such as making an invalid choice in a form, logging isn't necessary and doesn't need to be included.

As an example, here is a function that logs exceptions:

```
def foo(self, a, b):
    try:
        self.do_something(a, b)
    except InvalidChoiceError:
        self.invalid_choice_count += 1
    except ValueError:
        logger.error("Invalid input type in Class.foo()", exc_info=True)
    except Exception:
        logger.error("General exception in foo()", exc_info=True)
```

Workflow

We use zenhub for project management. It's basically a chrome extension that adds a kanban-like board to github. You can get it from https://www.zenhub.com/.

Zenhub adds a pipeline field to github issues. We use this field to track the current status of work on the issue. We use the following pipelines:

- `Icebox` – Issues that have been deprioritized, or are inside an Epic to be scheduled later
- `Discovery` – Issues that need to be triaged further and/or prioritized
- `Waiting for Design` – Issues that need design decisions, mockups, or css before back-end implementation
- `To Do` – Scheduled issues that a developer hasn't started yet
- `In Progress` – Issues that a developer is currently working on
- `Testing` – Issue that a developer believes to be handled and needs testing to verify the fix
- `Waiting for Deploy` – Issue that has been fixed in the staging branch and we need to deploy the change to production

Here's the workflow for a typical issue:

- **Prep work**
  - Someone creates a github issue that captures the bug/feature and puts it in the `Discovery` pipeline
  - The issue is prioritized and scheduled into a sprint
  - Developer reviews issue Friday before the sprint begins, adds story points to the issue
- **Initial development**
  - A developer creates topic branches for both the `unisubs` and `amara-enterprise` repositories to handle the issue. The branches should be named after its repository and issue number (e.g. `gh-enterprise-1234` or `gh-unisubs-5678` would be branches for github issue 1234 in the amara-enterprise repo and github issue 5678 in the unisubs repo, respectively). Changes for the issue get commited to these branches.
  - Once development on the issue is complete, developer moves the issue to the `Testing` pipeline and adds any relevant notes for testing to the issue.
- **Testing**
  - Tester tests the changes.
  - If there are problems, tester notes them on the issue and moves it back to `In progress`.
  - Developer fixes the problems, adds a note to the issue, moves it back to `Testing`, and we start testing again

  - Finally, tester approves the changes, then hands it back to developer to do a pull request

- **Review**
  - Developer merges any new code from staging/master back into the topic branches
  - Developer creates a pull request for unisubs and/or amara-enterprise depending on which repositories were changed for the issue
  - A second developer reviews the code
  - If there are issues, the developer #2 adds comments to the PR and works with developer #1 to resolve them
  - Once developer #2 thinks the code is ready, they merge the PR
  - Once we decide that staging is ready to be deployed to production, we will merge the staging branch to production then deploy andnd moves the issue to the `Waiting for deploy` pipeline
- **Deploy** - At some point we will deploy the code. - Usually this happens on a monday. - We first deploy staging, do a check to see if things are okay, then deploy production - Once production is deployed, tester closes all issues in `Waiting for deploy`

Testing

The Amara project uses the Nose testing framework.

Running tests

You should always run your tests inside the Vagrant VM because the test suite depends on a running Solr instance.

To run all unittests:

```
$ dev test
```

To run tests for a specific Django app:

```
$ dev test videos
```

To run a specific test class:

```
$ dev test videos.tests:ViewsTest
```

To run a specific test case within a test class:

```
$ dev test videos.tests:ViewsTest.test_index
```

Database Migrations

We use a MySQL database for Amara, which means every so often we need to update the database schema. This is tricky to do because we don't want to take down the site to do this. This means that we need to write migrations in a way that allows both the old and new code to run at the same time. This section has some advice on how to do this.

The main idea is to split the migration into parts and gradually change the schema using multiple deploys in a way that is compatible with the previous deploy.

As an example, let's suppose we want to replace the Video.duration field, which is an integer column, with Video.duration_string which is stored in "hh:mm:ss" form. Let's ignore the fact that this would be pretty silly and focus on how this would be done. We would split it into 4 stages:

- **Stage 1:** Add the new field and start writing values to it:
  - Create a migration that adds the duration_string field. Note that the field should have null=True, even if durations are always required, otherwise we'll have a database error when the old code writes rows to the videos table without duration_string set.
  - Add code to set the duration_string field whenever we set video durations.
  - Add a management command to update all videos and set duration_string. This command should be run after the old code has stopped running and before the stage2 code starts.
  - Continue to write to Video.duration, since the old code is relying on it.
  - Don't use the duration_string field when displaying video duration, since it will not be always be set.
- **Stage 2:** Start using the new field
  - Start using duration_string for displaying the duration
  - Remove the null=True clause from the field, if necessary
- **Stage 3:** Stop writing to the new field
  - Remove code that writes to Video.duration
  - Remove Video.duration from the Video model
  - Don't create a DB migration to remove Video.duration yet, since that would case an error when the code from stage2 tried to write to that field
- **Stage 4:** Drop the old field
  - Create a DB migration that drops the Video.duration column

---

Note: The same process works if you were just adding a new field, or dropping an old field. In those cases you can just skip some steps.

Each stage should be in a separarate branch, typically named `gh-[issue#]-stageX`. We only deploy 1 stage at a time.

This process adds extra complexity when developing, but reduces the complexity of deployment, since there's never a window when the site is down and we're waiting for a migration to run.

It's possible that we have migrations that can't be split up like this. If so, that's fine, we just need to take the site down for some time period.

Varnish

We use Varnish (http://varnish-cache.org/) as a reverse proxy/caching server in front of our app. The code is at github.com/pculture/amara-cache. Our general system is:

- Most pages use *cache-control: private* and are not cached by Varnish
- Pages that are mostly static like the homepage and watch pages are cached.
- We vary by Accept-Language and Cookie
- In the Varnish VCL, we try to normalize/reduce those headers to improve cache hits. We make it so:

  - Cookie only contains the sessionid
  - Accept-Language is normalized, so it only contains the language that we want to display the page in.

Caching App

Amara uses a couple tricks for caching things.

Cache Groups

Cache groups are used to manage a group of related cache values. They add some extra functionality to the regular django caching system:

- **Key prefixing**: cache keys are prefixed with a string to avoid name collisions
- **Invalidation**: all values in the cache group can be invalidated together. Optionally, all values can be invalidated on server deploy
- **Optimized fetching**: we can remember cache usage patterns in order to use get_many() to fetch all needed keys at once (see *Cache Patterns*)
- **Protection against race conditions**: (see *Race condition prevention*)

Typically cache groups are associated with objects. For example we create a cache group for each user and each video. The user cache group stores things like the user menu HTML and message HTML. The video cache group stores the language list and other sections of the video/language pages.

Overview

- A CacheGroup is a group of cache values that can all be invalidated together
- You can automatically create a CacheGroup for each model instance
- CacheGroups can be used with a cache pattern. This makes it so we remember which cache keys are requested and fetch them all using get_many()

Let's take the video page caching as an example. To implement caching, we create cache groups for Team, Video, and User instances. Here's a few examples of how we use those cache groups:

- Language list: we store the rendered HTML in the video cache
- User menu: we store the rendered HTML in the user cache (and we actually use that for all pages on the site)
- Add subtitles form: we store the list of existing languages in the video cache (needed to set up the selectbox)
- Follow video button: we store a list of user ids that are following the videos in the video cache. To the user is currently following we search that list for their user ID.
- Add subtitles permissions: we store a list of member user ids in the team cache. To check if the user can view the tasks/collaboration page we search that list of the user ID

When we create the cache groups, we use the video-page cache pattern. This makes it so we can render the page with 3 cache requests. One get_many fetches the Video instance and all cache values related to the video, and similarly for the Team and User.

Cache invalidation is always tricky. We use a simple system where if a change could affect any cache value, we invalidate the entire group of values. For example if we add/remove a team member then we invalidate the cache for the team.

Cache Patterns

Cache patterns help optimize cache access. When a cache pattern is set for a CacheGroup we will do a couple things:

- Remember which keys were fetched from cache.
- On subsequent runs, we will try to use get_many() to fetch all cache values at once.

This speeds things up by reducing the number of round trips to memcached.

Behind the scenes

The main trick that CacheGroup uses is to store a "version" value in the cache, which is simply a random string. We also pack the version value together with all of our cache values. If a cache value's version doesn't match the version for the cache group, then it's considered invalid. This allows us to invalidate the entire cache group by changing the version value to a different string.

Here's some example data to show how it works.

key	value in cache	computed value
version	abc	N/A
X	abc:foo	foo
Y	abc:bar	bar
Z	def:bar	*invalid*

Note: We also will prefix the all cache keys with the "<prefix>:" using the prefix passed into the CacheGroup constructor.

Note: If invalidate_on_deploy is True, then we will append ":<commit-id>" to the version key. This way the version key changes for each deploy, which will invalidate all values.

Race condition prevention

The typical cache usage pattern is:

1. Fetch from the cache
2. If there is a cache miss then:
   (a) calculate the value
   (b) store it to cache.

This pattern will often have a race condition if another process updates the DB between steps 2a and 2b. Even if the other process invalidates the cache, the step 2b will overwrite it, storing an outdated value.

This is not a problem with CacheGroup because of the way it handles the version key. When we get the value from cache, we also fetch the version value. If the version value isn't set, we set it right then. Then when we store the value, we also store the version key that we saw when we did the get. If the version changes between the get() and set() calls, then the value stored with set() will not be valid. This works somewhat similarly to the memcached GETS and CAS operations.

Cache Groups and DB Models

Cache groups can save and restore django models using get_model() and set_model(). There is a pretty conservative policy around this. Only the actual row data will be stored to cache – other attributes like cached related instances are not stored. Also, restored models can't be saved to the DB. All of this is to try to prevent overly aggressive caching from causing weird/wrong behavior.

To add caching support to your model, add `ModelCacheManager` as an attribute to your class definition.

class caching.cachegroup.**CacheGroup**(*prefix*, *cache_pattern=None*, *invalidate_on_deploy=True*)

Manage a group of cached values

Parameters

- **prefix** (*str*) – prefix keys with this
- **cache_pattern** (*str*) – *cache pattern* identifier
- **invalidate_on_deploy** (*bool*) – Invalidate values when we redeploy

get(*key*)

Get a value from the cache

This method also checks that the version of the value stored matches the version in our version key.

If there is no value set for our version key, we set it now.

get_many(*keys*)

Get multiple keys at once

If there is no value set for our version key, we set it now.

set(*key*, *value*, *timeout=None*)

Set a value in the cache

set_many(*values*, *timeout=None*)

Set multiple values in the cache

get_or_calc(*key*, *work_func*, **args*, ***kwargs*)

Shortcut for the typical cache usage pattern

get_or_calc() is used when a cache value stores the result of a function. The steps are:

•Try self.get(key)

•If there is a cache miss then

–call work_func() to calculate the value

–store it in the cache

get_model(*ModelClass*, *key*)

Get a model stored with set_model()

Note: To be catious, models fetched from the cache don't allow saving. If the cache data is out of date, we don't want to saave it to disk.

---

set_model(*key*, *instance*, *timeout=None*)

Store a model instance in the cache

Storing a model is a tricky thing. This method works by storing a tuple containing the values of the DB row. We store it like that for 2 reasons:

•It's space efficient

•It drops things like cached related objects. This is probably good since it makes it so we don't also cache those objects, which can lead to unexpected behavior and bugs.

Parameters

- **key** – key to store the instance with
- **instance** – Django model instance, or None to indicate the model does not exist in the DB. This will make get_model() raise a ObjectDoesNotExist exception.

invalidate()
Invalidate all values in this CacheGroup.

class caching.cachegroup.**ModelCacheManager**(*default_cache_pattern=None*)
Manage CacheGroups for a django model.

ModelCacheManager is meant to be added as an attribute to a class. It does 2 things: manages CacheGroups for the model class and implements the python descriptor protocol to create a CacheGroup for each instance. If you add `cache = ModelCacheManager()` to your class definition, then:

•At the class level, MyModel.cache will be the ModelCacheManager instance

•At the instance level, my_model.cache will be a *CacheGroup* specific to that instance

get_cache_group(*pk*, *cache_pattern=None*)
Create a CacheGroup for an instance of this model

Parameters

- **pk** – primary key value for the instance
- **cache_pattern** – cache pattern to use or None to use the default cache pattern for this ModelCacheManager

invalidate_by_pk(*pk*)
Invalidate a CacheGroup for an instance

This is a shortcut for get_cache_group(pk).invalidate() and can be used to invalidate without having to load the instance from the DB.

get_instance(*pk*, *cache_pattern=None*)
Get a cached instance from it's cache group

This will create a CacheGroup, get the instance from it or load it from the DB, then reuse the CacheGroup for the instance's cache. If a cache pattern is used this means we can load the instance and all of the needed cache values with one get_many() call.

Teams

Teams are a key concept in amara. A team is a group of users that work together to subtitle videos. Teams are typically made of members of a group that produces video and wants to add subtitles.

Team Workflows

Team workflows are ways for teams to get their subtitling work done. Team workflows compliment the *Subtitle Workflows* and add team-specific features.

Team workflows are responsible for:

- Providing a SubtitleWorkflow for team videos
- Handling the workflow settings page
- Handling the dashboard page
- Creating extra tabs or the teams section

class teams.workflows.teamworkflows.**TeamWorkflow**(*team*)

label = NotImplemented
Human-friendly name for this workflow. This is what appears on the team creation form.

workflow_settings_view = NotImplemented
view function for the workflow settings page.

Note: All workflows should allow the user to change membership_policy and video_policy in their workflow settings page.

---

setup_team()
Do any additional setup for newly created teams.

get_subtitle_workflow(*team_video*)
Get the SubtitleWorkflow for a video with this workflow.

extra_pages(*user*)
Get extra team pages to handle this workflow.

These pages will be listed as tabs in the team section. Workflows will typically use this for things like dashboard pages.

Parameters -- user viewing the page(*user*) –

Returns class:*TeamPage* objects

Return type list of

extra_settings_pages(*user*)
Get extra team settings pages to handle this workflow.

This works just like extra_pages(), but the pages will show up as tabs under the settings section.

Parameters -- user viewing the page(*user*) –

Returns class:*TeamPage* objects

Return type list of

class teams.workflows.teamworkflows.**TeamPage**(*name*, *title*, *url*)
Represents a page in the team's section

name
machine-name for this tuple. This is value to use for current in the _teams/tabs.html template

title
human friendly tab title

url
URL for the page

class teams.workflows.old.workflow.**OldTeamWorkflow**(*team*)
Workflow for old-style teams

We have tried to tackle the issue of team workflows in several ways. The most infamous has to be the tasks sytem. This class acts the glue between the new workflow components and the old systems.

The plan is to migrate all our teams from OldTeamWorkflow to newer workflow styles. At that point we can get rid of OldTeamWorkflow and also probably a bunch of other things like the tasks code, the Workflow table, several Team model fields, etc.

The Subtitle Editor

The subtitle editor is one of the larger features of amara. It's implemented using several components in a couple different areas:

- The view *subtitles.views.subtitle_editor* serves up the page
- The page runs javascript that lives in `media/src/js/subtitle-editor`
- We save subtitles using the API code (currently in a private repository, but we plan to merge it in to the main one soon)

See also:

Subtitle Workflows

Subtitle Workflows

Subtitle workflows control how subtitle sets get edited and published. In particular they control:

- Work Modes – Tweak the subtitle editor behavior (for example review mode)
- Actions – User actions that can be done to subtitle sets (Publish, Approve, Send back, etc).
- Permissions – Who can edit subtitles, who can view private subtitles

Overriding workflows

By default, we use a workflow that makes sense for public videos – Anyone can edit, the only action is Publish, etc.

To override the workflow by Video (for example for videos in a certain type of team):

- Create a *VideoWorkflow* subclass
- Create a *LanguageWorkflow* subclass (make this returned by VideoWorkflow.get_default_language_workflow())
- Override *get_workflow()* and return your custom VideoWorkflow

To override the workflow for by SubtitleLanguage (for example you can override the workflow for the SubtitleLanguage covered by professional service request):

- Create a *LanguageWorkflow* subclass
- Override *get_language_workflow()* and return your custom LanguageWorkflow

Workflow Classes

class subtitles.workflows.**VideoWorkflow**(*video*)

VideoWorkflow subclasses work with LanguageWorkflow subclasses to control the overall workflow for editing and publishing subtitles. Workflows control the work modes, actions, permissions, etc. for subtitle sets.

user_can_view_video(*user*)

Check if a user can view the video

Returns True/False

user_can_edit_video(*user*)

Check if a user can view the video

Returns True/False

get_add_language_mode(*user*)

Control the add new language section of the video page

Parameters **user** (*User*) – user viewing the page

Returns

Value that specifies how the section should appear

- None/False: Don't display anything
- "<standard>": Use the standard behavior a link that opens the create subtitles dialog.
- any other string: Render this in the section. You probably want to send the string through mark_safe() to avoid escaping HTML tags.

extra_tabs(*user*)

Get extra tabs for the videos page

Returns

list of (name, title) tuples. Name is used for the tab id, title is a human friendly title.

For each tab name you should create a video-<name>.html and video-<name>-tab.html templates. If you need to pass variables to those templates, create a setup_tab_<name> method that inputs the same args as the methods from VideoPageContext and returns a dict of variables for the template.

get_default_language_workflow(*language_code*)

Get the default LanguageWorkflow for this VideoWorkflow.

This will be used unless some other component overrides it with *get_language_workflow()*

class subtitles.workflows.**LanguageWorkflow**(*video*, *language_code*)

get_work_mode(*user*)

Get the work mode to use for an editing session

Parameters **user** (*User*) – user who is editing

Returns *WorkMode* object to use

Return type :class

get_actions(*user*)

Get available actions for a user

Parameters **user** (*User*) – user who is editing

Returns class:*Action* objects that are available to the user.

Return type list of

action_for_add_subtitles(*user*, *complete*)
Get an action to use for add_subtitles()

This is used when pipeline.add_subtitles() is called, but not passed an action. This happens for a couple reasons:

•User saves a draft (in which case complete will be None)

•User is adding subtitles via the API (complete can be True, False, or None)

Subclasses can override this method if they want to use different actions to handle this case.

Parameters

- **user** (*User*) – user adding subtitles
- **complete** (*bool or None*) – complete arg from add_subtitles()

Returns Action object or None.

get_editor_notes(*user*)
Get notes to display in the editor

Returns *EditorNotes* object

Return type :class

user_can_view_private_subtitles(*user*)
Check if a user can view private subtitles

Private subtitles are subtitles with visibility or visibility_override set to "private". A typical use is to limit viewing of the subtitles to members of a team.

Returns True/False

user_can_delete_subtitles(*user*, *language_code*)
Check if a user can delete a language

Returns True/False

user_can_edit_subtitles(*user*)
Check if a user can edit subtitles

Returns True/False

Behavior Functions

subtitles.workflows.**get_workflow**(*video*)
Get the workflow to use for a video.

By default this method returns the workflow for public, non-team videos. Other apps can override it to customize the behavior.

subtitles.workflows.**get_language_workflow**(*video*, *language_code*)
Override the default LanguageWorkflow for a subtitle set

Normally this method returns None, which means use the default for the VideoWorkflow. Other apps can override this and control the workflow for specific video languages.

See also:

behaviors module for how you can override these functions.

Editor Notes

class subtitles.workflows.**EditorNotes**(*video*, *language_code*)
Manage notes for the subtitle editor.

EditorNotes handles fetching notes for the editor and posting new ones.

heading
heading for the editor section

notes
list of SubtitleNotes for the editor (or any model that inherits from SubtitleNoteBase)

post(*user*, *body*)
Add a new note.

Parameters

- **user** (*CustomUser*) – user adding the note
- **body** (*unicode*) – note text

Work Modes

class subtitles.workflows.**WorkMode**
Work modes are used to change the workflow section of the editor and affect the overall feel of the editing session. Currently we only have 2 work modes:

•**class NormalWorkMode**
The usual work mode with typing/syncing/review steps.

•**class** subtitles.workflows.**ReviewWorkMode**(*heading*, *help_text=None*)
Review someone else's work (for example a review/approve task)
Parameters **heading** (*str*) – heading to display in the workflow area

Actions

Actions are things things that users can do to a subtitle set other than changing the actual subtitles. They correspond to the buttons in the editor at the bottom of the workflow session (publish, endorse, send back, etc). Actions can occur alongside changes to the subtitle lines or independent of them.

class subtitles.workflows.**Action**
Base class for actions

Other components can define new actions by subclassing Action, setting the class attributes, and optionally implementing perform().

name
Machine-friendly name

label
human-friendly label. Strings should be run through ugettext_lazy()

in_progress_text
text to display in the editor while this action is being performed. Strings should be run through ugettext_lazy()

visual_class
visual class to render the action with. This controls things like the icon we use in our editor button. Must be one of the *CLASS_* constants

complete
how to handle subtitles_complete. There are 3 options:

•True – this action sets subtitles_complete

•False – this action unsets subtitles_complete

•None (default) - this action doesn't change subtitles_complete

subtitle_visibility
Visibility value for newly created

SubtitleVersions (*"public" or "private"*)

CLASS_ENDORSE
visual class constant for endorse/approve buttons

CLASS_SEND_BACK
visual class constant for reject/send-back buttons

require_synced_subtitles()
Should we require that all subtitles have timings?

The default implementation uses the complete attribute

validate(*user*, *video*, *subtitle_language*, *saved_version*)
Check if we can perform this action.

Parameters

- **user** (*User*) – User performing the action
- **video** (*Video*) – Video being changed
- **subtitle_language** (*SubtitleLanguage*) – SubtitleLanguage being changed
- **saved_version** (*SubtitleVersion or None*) – new version that was created for subtitle changes that happened alongside this action. Will be None if no changes were made.

Raises ActionError – this action can't be performed –

perform(*user*, *video*, *subtitle_language*, *saved_version*)
Perform this action

Parameters

- **user** (*User*) – User performing the action
- **video** (*Video*) – Video being changed
- **subtitle_language** (*SubtitleLanguage*) – SubtitleLanguage being changed
- **saved_version** (*SubtitleVersion or None*) – new version that was created for subtitle changes that happened alongside this action. Will be None if no changes were made.

update_language(*user*, *video*, *subtitle_language*, *saved_version*)
Update the subtitle language after adding subtitles

Parameters

- **user** (*User*) – User performing the action
- **video** (*Video*) – Video being changed
- **subtitle_language** (*SubtitleLanguage*) – SubtitleLanguage being changed

- **saved_version** (*SubtitleVersion or None*) – new version that was created for subtitle changes that happened alongside this action. Will be None if no changes were made.

editor_data()
Get a dict of data to pass to the editor for this action.

class subtitles.workflows.**Publish**
Publish action

Publish sets the subtitles_complete flag to True

The permission system

The permission system in Amara subtitles is very flexible to allow for the needs of different teams. This document will give you a high level overview of what is possible. You should read this before trying to understand the source code.

Overview

Let's start with some language. In the simplest case, when a user is part of a team, they can have one of the following roles:

- **Contributor**
  - Transcribe
  - Translate
  - Assign tasks to themselves
- **Manager**
  - Review subtitles
  - Approve subtitles
  - Assign tasks to other people
  - Everything that a contributor can do
- **Admin**
  - Assign new managers
  - Delete subtitles
  - Everything that a manager can do
- **Owner**
  - Everything

Note: This is just an example to give you an idea of how this could work.

A user's role is stored in the teams.models.TeamMember model which stores a reference to the user and team objects.

Checking for required permissions

When you want to check if a certain user has the required privileges to perform a task, you should use one of the functions in `teams.permissions`. For example, if you'd like to check if a user can approve a video, you could do something like this:

```
from teams.permissions import can_approve

if can_approve(video, user):
    # Do something that requires the approval permission
```

Note: There is no middleware to attach the current user's privileges to the request instance. Instead, you have explicitly call the necessary function whenever you want to verify the user's privileges.

Workflows

A team can choose their own workflow to efficiently manage their videos, translations and volunteers. When you are setting up a workflow for your team, you can decide how certain actions will be performed. For example:

- Who can join the team?
- Who can and remove videos from the team?
- Who can assign tasks?
- How many tasks a user can have at a time?
- How many days should a user get to complete a task?
- Who can transcribe subtitles?
- Who can translate subtitles?
- Is there a review process?
- Is there an approval process?

So, why should you care? For example, you don't trust your contributors with transcription of new videos since it's somewhat difficult. Therefore, you can choose to only allow managers and above to transcribe videos and contributors to only translate videos to different languages. Or, the quality of the subtitles is crucial to you and you want to make sure that nothing less than that ever gets out. So, you would turn on both the review and approval process. This way three sets of eyes will look at the subtitles before it goes public.

Optional Apps

Amara.org uses several apps/packages that are stored in private github repositories that add extra functionality for paid partnerships. These apps are optional – the amara codebase runs fine without them.

The coding issue is how to make amara work without these repositories, but automatically pull them in if they are present. Here's how we do it:

- For each repository we create a file inside the optional/ directory:
  - The filename is the name of the repository
  - The contents are the git commit ID that we want to use

- To enable a repository, it must be checked out in the amara root directory, using the same name as the git repository.
- The optionalapps module handles figuring out which repositories are present and how we should modify things at runtime

optionalapps.**setup_path**()
Add optional repositories to the python path

optionalapps.**get_repository_paths**()
Get paths to optional repositories that are present

Returns list of paths to our optional repositories. We should add these to sys.path so that we can import the apps.

optionalapps.**get_apps**()
Get a list of optional apps

Returns list of app names from our optional repositories to add to INSTALLED_APPS.

optionalapps.**get_urlpatterns**()
Get Django urlpatterns for URLs from our optional apps.

This function finds urlpatterns inside the urls module for each optional app. In addition a module variable can define a variable called PREFIX to add a prefix to the urlpatterns.

Returns url patterns containing urls for our optional apps to add to our root urlpatterns

optionalapps.**exec_repository_scripts**(*filename*, *globals*, *locals*)
Add extra values to the settings module.

This function looks for files named filename in each optional repository. If that exists, then we call execfile() to run the code using the settings globals/locals. This simulates that code being inside the current scope.

Internationalization (i18n)

Unisubs has some complex requirements in terms of 18n. This is a rough guide of how things work.

Django's system is gnu's get text system. For example:

```
pt
```

The first two letters are the language code, according to ISO 639-1. In this case Portuguese.

If the locale has variation as to the country, for example Portugal's Portuguese vs Brazilian's portuguese then the locale name is appended an underscore + the country two digit code, which is ISO 3166. Therefore the locales for portuguese speaking countries are:

```
pt_BR -> Brazilian Portuguese
pt_PT -> Portugal's Portuguese
```

Some of the less common languages are not covered by ISO 639-1 but are by ISO 639-3.

Guidelines

Most of the heavy lifting is handled by our Unilangs library.

Steps for adding a new language:

- Figure Out the ISO code If ISO 639-1 covers it, it's the prefered way to handle this, as it would keep our code streamlined with Django's. If not, then we should prefer ISO 639-3. One can depend on the list of ISO 693-1 and the list of ISO 693-3 on Wikipedia.

  Also if it's more of an uknown language it's useful to look at the Wikipedia entry for it. Sometimes, we have a request to include a language that doesn't make sense. For example there is no Norwegian(no), but there are dialects (nb and nn) that are used. Usually the Wikipedia page will discuss similar languages and specific dialects.

- Update Unilangs INTERNAL_NAMES: Add the language code, the English name for the language, and the language in it's own name to the unilangs.py file .
- Update the unisubs standard: Add the language code to our standard .
- Update other codecs: If django supports that locale (as in we can i18n the site's UI) update the 'dango' standard. If other standards (such as ISO-693-1) support it, update them too.

Of course, once you've updated unlilangs, you'll need to update the virtual envs on all installations of the app.

Updating Django

One must be careful when updating the Django's version. As new locales are added between releases, we must check if the locale is already added on our end with a different encoding. If that happens, we'll have duplication . This has beaten us before.

Partners

Different partners might have different language requirements while mapping to their own internal systems. We should update this guide once we have more specifics on how we're implementing those mappings.

Behaviors

Extensible behavior functions

This module allows one app to define a "behavior function", which other apps can then override to change the behavior. This is the Chain of responsibility pattern which helps keep modules loosely coupled.

A typical use for this is the subtitle that we display for video cards, underneath the title. Normally, we don't show anything there, but for TED we show the speaker name. Putting the code that deals with this inside the videos app is bad practice because:

- It's adding complexity to the videos app. Handling team requirements is outside of its scope.
- It requires importing from the teams app, but the teams app needs to import from the videos app. So we now have a circular dependency.

Instead, videos defines the *get_video_subtitle* behavior, which can then be overriden by other apps. This allows us to change the behavior without having to add complexity/dependencies to the videos app. The code works something like this.

Example

```
>>> @behavior
... def get_video_subtitle(video)
...     return video.title
>>> @get_video_subtitle.override
... def get_video_subtitle_for_team_foo(video):
...     team_video = video.get_team_video()
...     if team_video and team_video.slug == 'foo'
...         return 'My Team: %s' % video.title
...     else:
...         return DONT_OVERRIDE
```

CHAPTER 10

Contributing

The source code to Amara is available on Github. Feel free to fork the project and open a pull request.

Before working on a sizeable feature, please do run it by us in our IRC channel. We don't want you to waste your time working on something we don't really want. We are at `#amara` on freenode.

CHAPTER 11

License

Amara is freely available under the terms of the GNU Affero General Public License. You can find the full text of the license here.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

/(video-url-endpoint)

/api

28
PUT /api/teams/(team-slug)/projects/(project-slug)/,
29
PUT /api/teams/(team-slug)/subtitle-requests/(job-id)/,
36
PUT /api/teams/(team-slug)/tasks/(task-id)/,
30
PUT /api/users/[username], 20
PUT /api/videos/(video-id)/, 12
DELETE /api/teams/(team-slug)/members/(username)/,
28
DELETE /api/teams/(team-slug)/projects/(project-slug)/,
29
DELETE /api/teams/(team-slug)/subtitle-requests/(job-id)/,
37
DELETE /api/teams/(team-slug)/tasks/(task-id)/,
31
DELETE /api/videos/(video-id)/, 13
DELETE /api/videos/(video-id)/languages/(language-code)/subtitles/,
17

/https:

GET https://amara.org/api/videos/, 9
GET https://amara.org/api/videos/?order_by=title,
9
GET https://amara.org/api/videos/?team=butterfly-club,
9
GET https://amara.org/api/videos/foo, 9
POST https://amara.org/api/videos/, 9
PUT https://amara.org/api/videos/foo, 9

Python Module Index

Index