Textinator

Release 1.0.0

Dmytro Kalpakchi

Nov 17, 2022

CONTENTS:

TEXTINATOR

Textinator is an open-source internationalized highly-customizable annotation and evaluation tool for Natural Language Processing (NLP) tasks. The tool offers a web interface with a user-friendly UI and supports a number of NLP tasks out-of-the-box:

- Question Answering
- Question Answering with Complexity Ranking
- Multiple Choice Question Answering
- Multiple Choice Question Answering with Complexity Ranking
- Named Entity Recognition
- Pronoun Resolution
- Co-reference Chain Resolution
- Machine Translation

Textinator is currently localized for 4 languages (presented in an alphabetical order):

- English
- Russian
- Swedish
- Ukrainian

We are constantly working to improve available localizations and extend them to new languages. If you are willing to help, please visit our Github repository.

CHAPTER

ONE

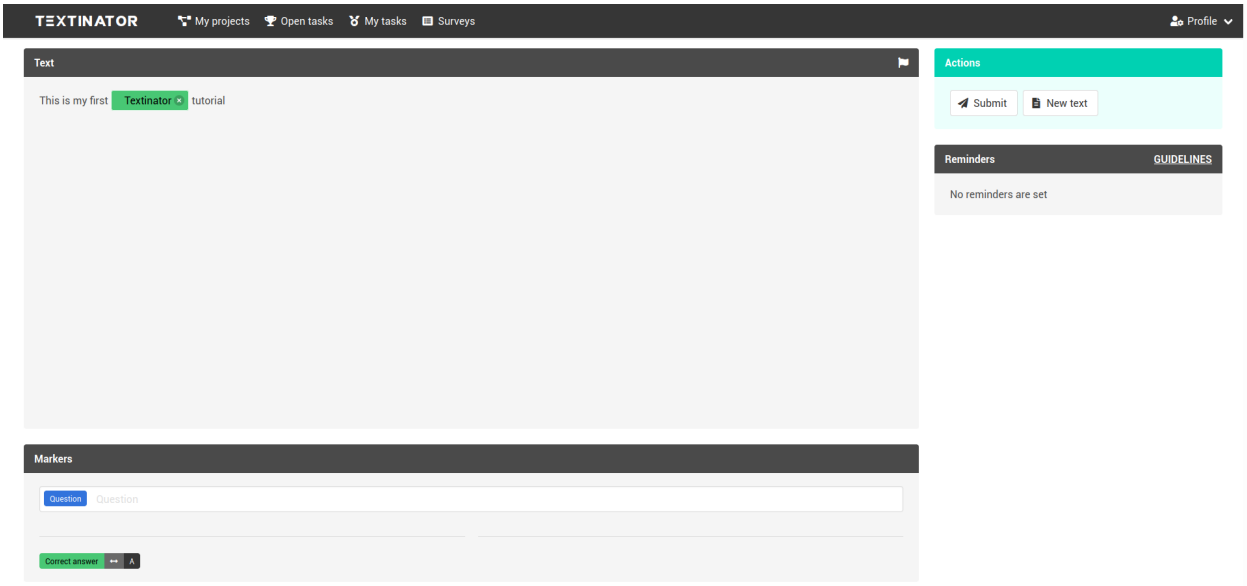
TUTORIAL

1.1 [Part 1] Exploring annotation tasks supported out-of-the-box

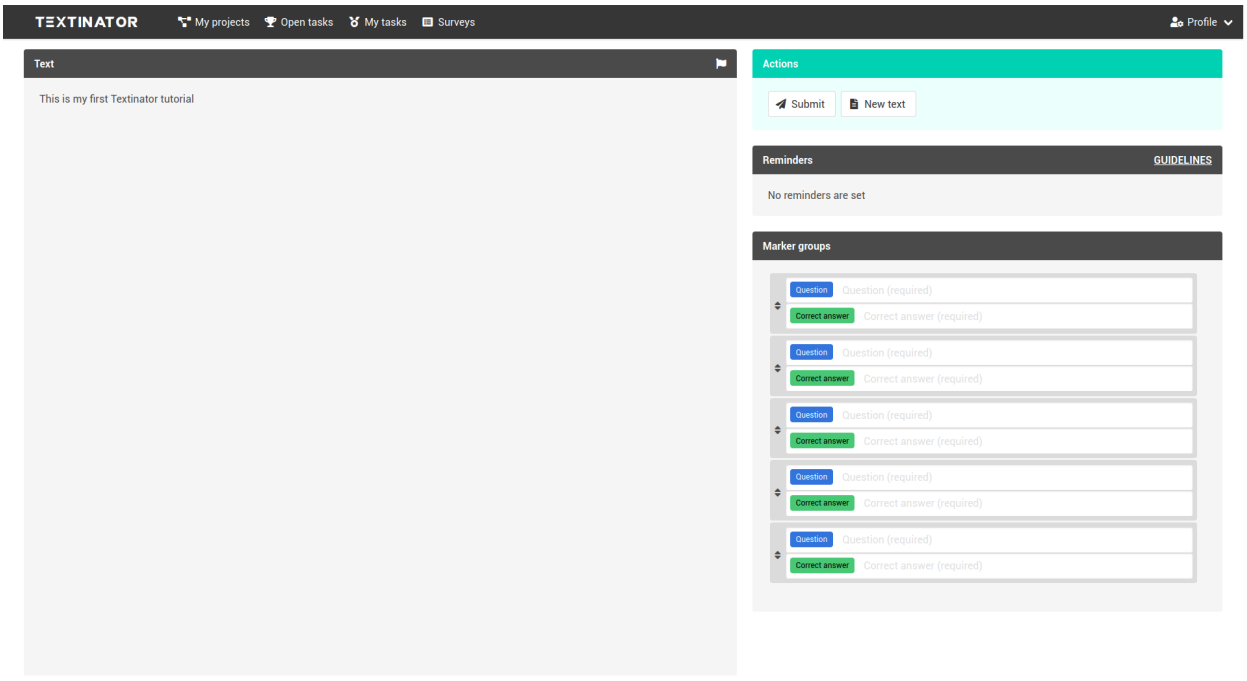
Currently Textinator supports 8 annotation tasks out-of-the-box. Note that although adding new markers to these default annotation tasks is possible, but data exporters might not support the added markers. Please consult the documentation for each annotation task separately on the *the page about custom tasks*. The information about how you can create a completely custom annotation task if you wish so, is available on the same page.

Table of Contents

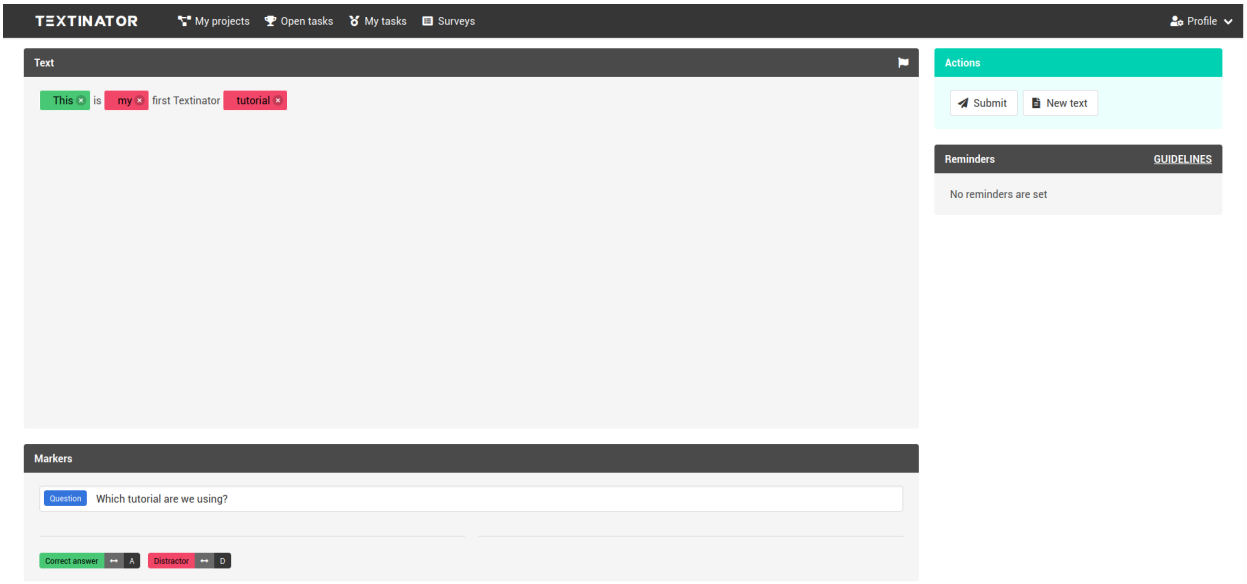
1.1.1 Question Answering



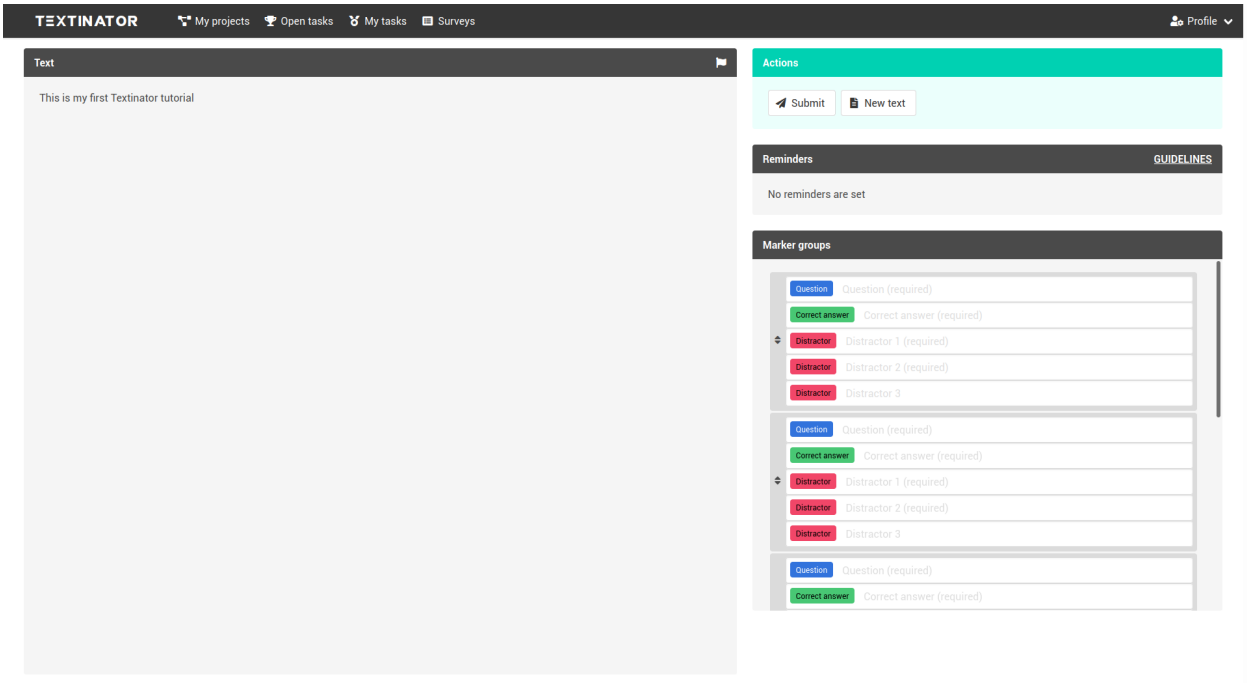
1.1.2 Question Answering with Ranking



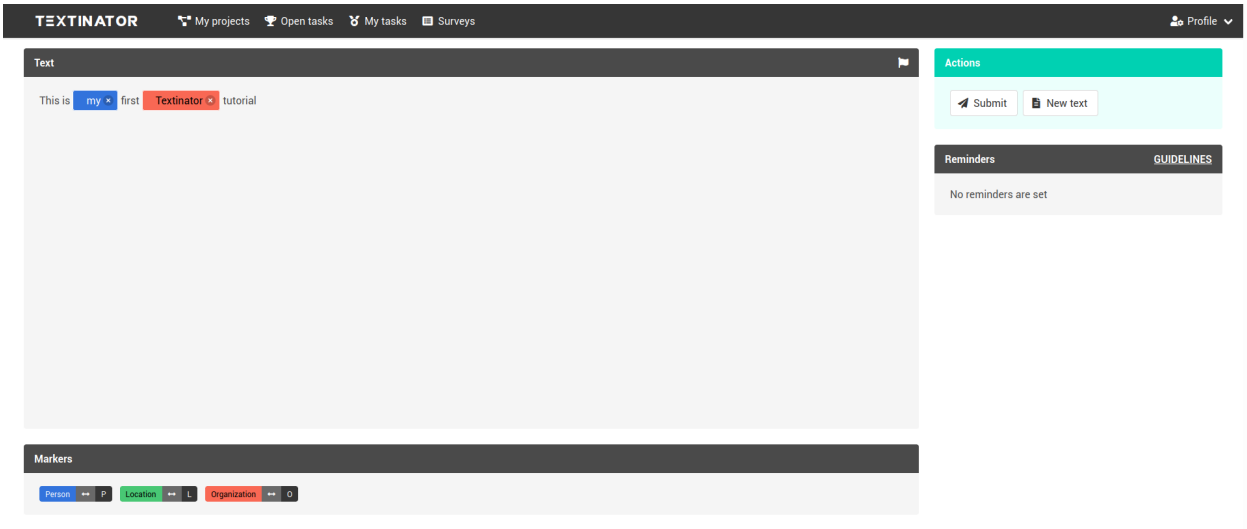
1.1.3 Multiple Choice Question Answering



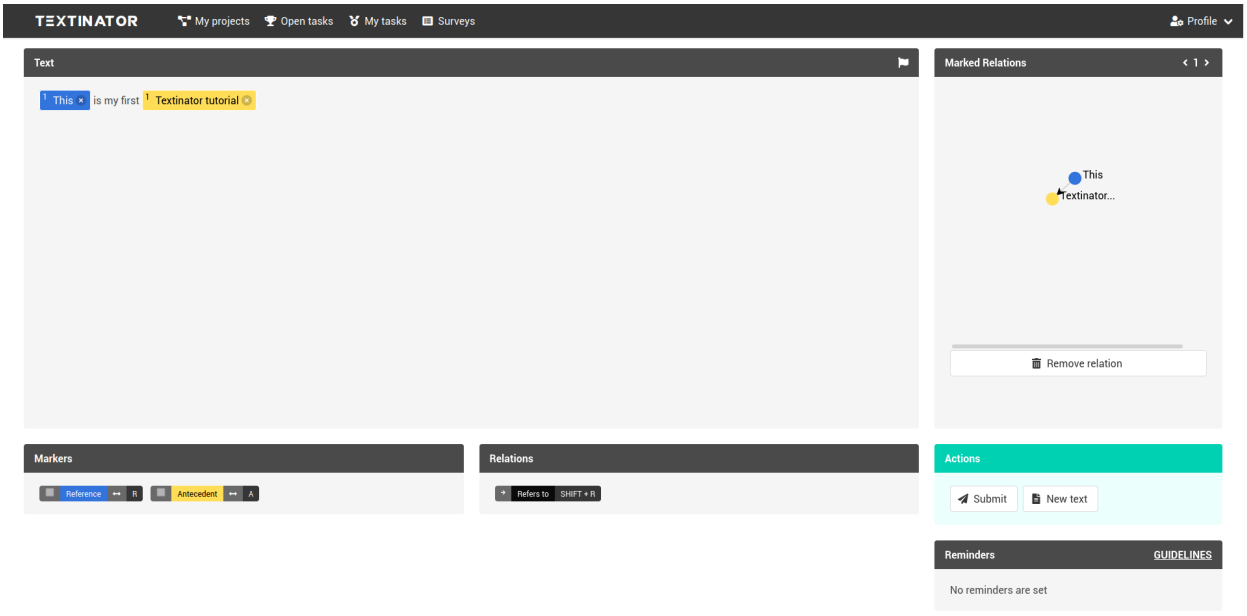
1.1.4 Multiple Choice Question Answering with Ranking



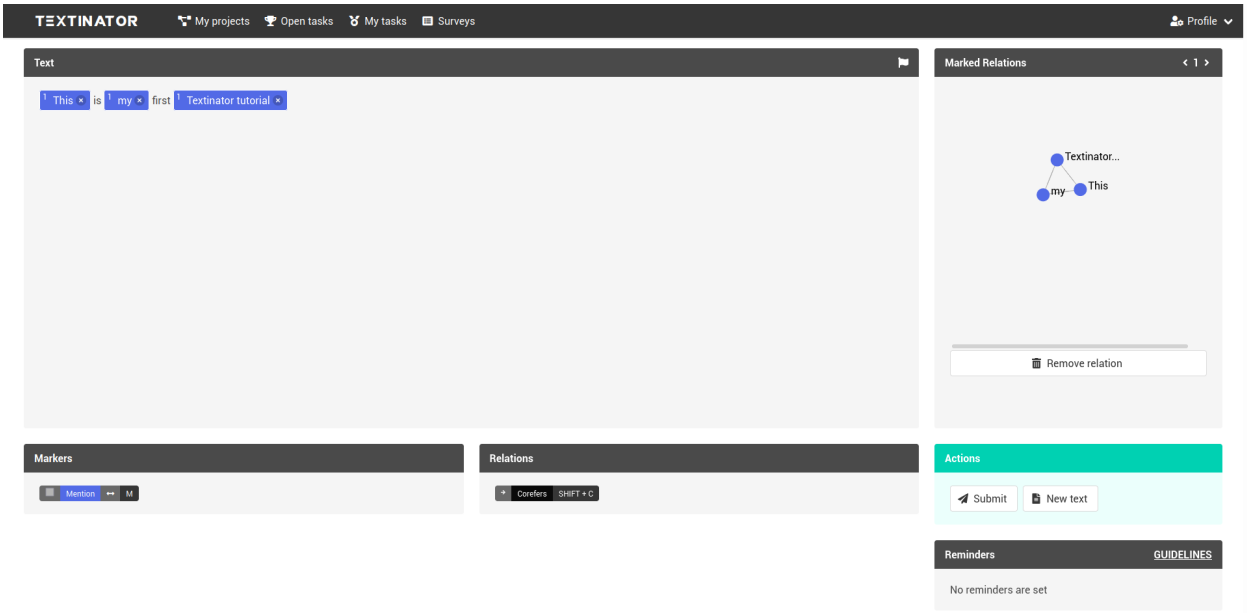
1.1.5 Named Entity Recognition



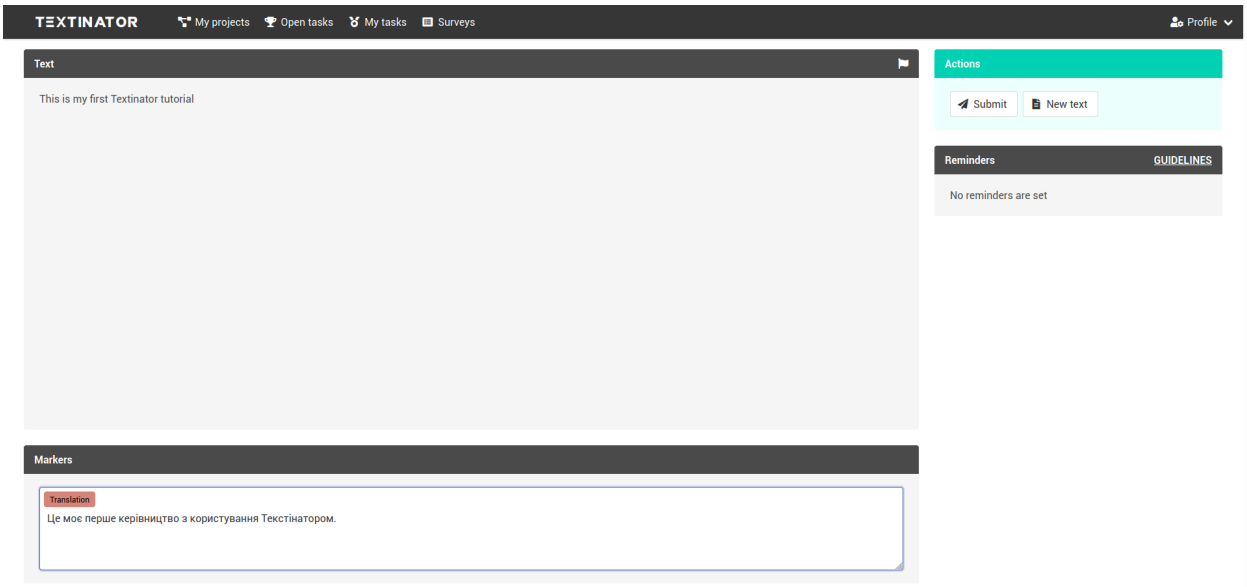
1.1.6 Pronoun Resolution



1.1.7 Co-reference Chain Resolution



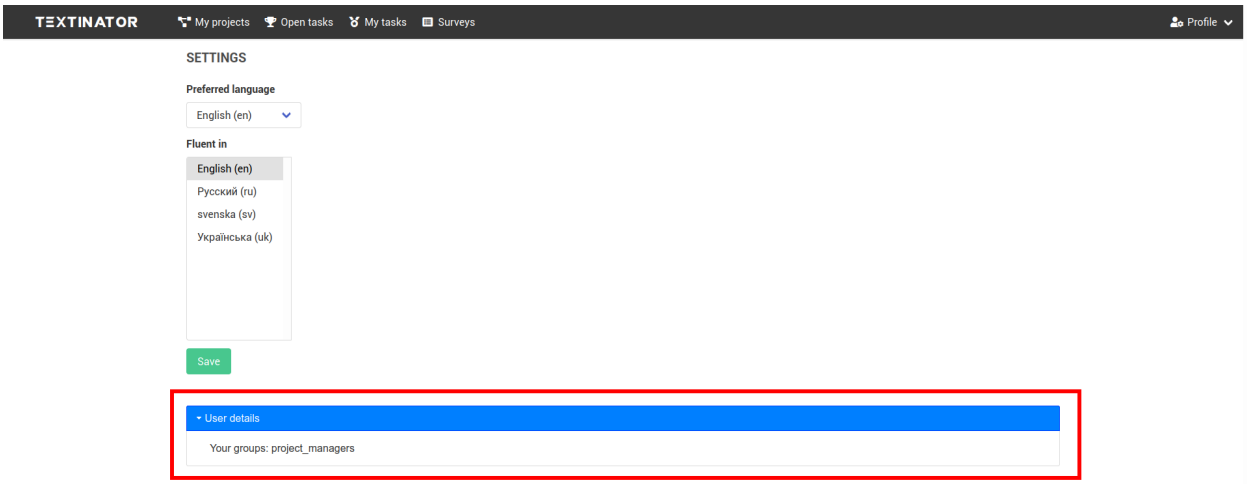
1.1.8 Machine Translation



1.2 [Part 2] Adding a new data source

Table of Contents

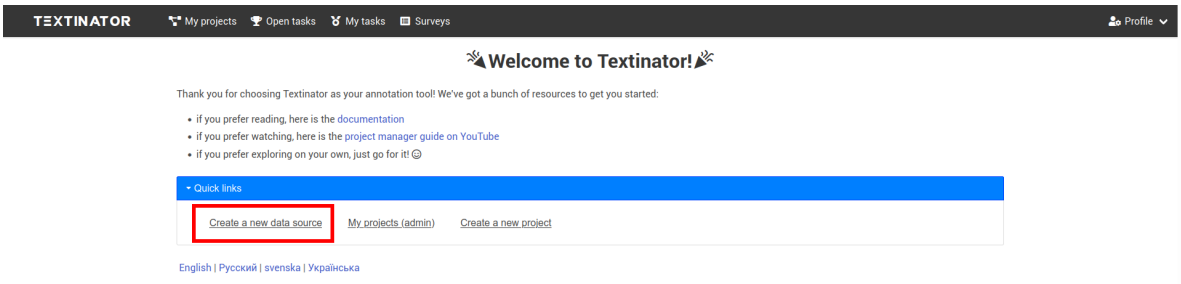
Data sources can only be added by staff members (check for the small cog near your user icon in the top right corner) that have been assigned to the *project_managers* user group. You can check if you have been assigned to that user group by clicking on *Profile* -> *Settings*, which should show the page, similar to the one on the screenshot below.



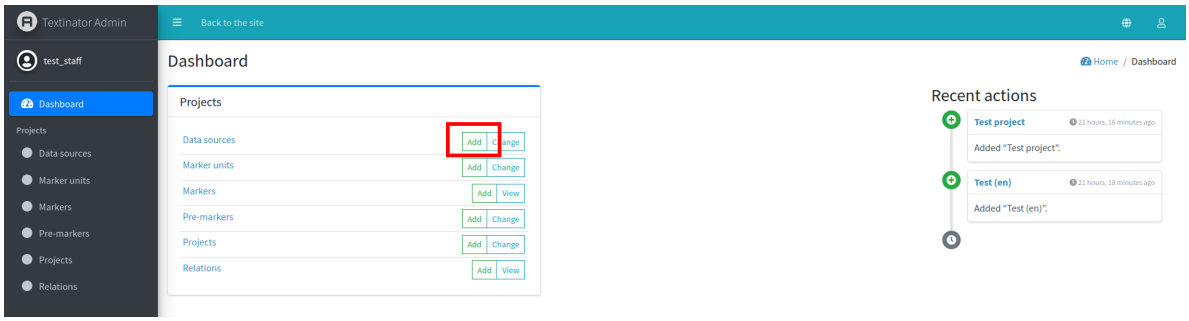
Check that the *User details* pane (marked by the right rectangle in the screenshot above) has *project_managers* in the list of your user groups. If it doesn't, please contact your system administrator to be added to that group.

After you have been added to the *project_managers* group, you can access the data source creation form in multiple ways:

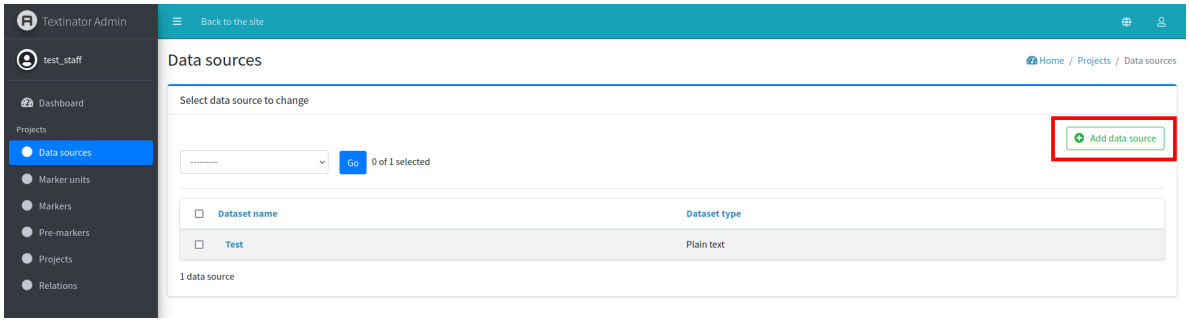
via *Quick links* pane on the *Welcome* page.



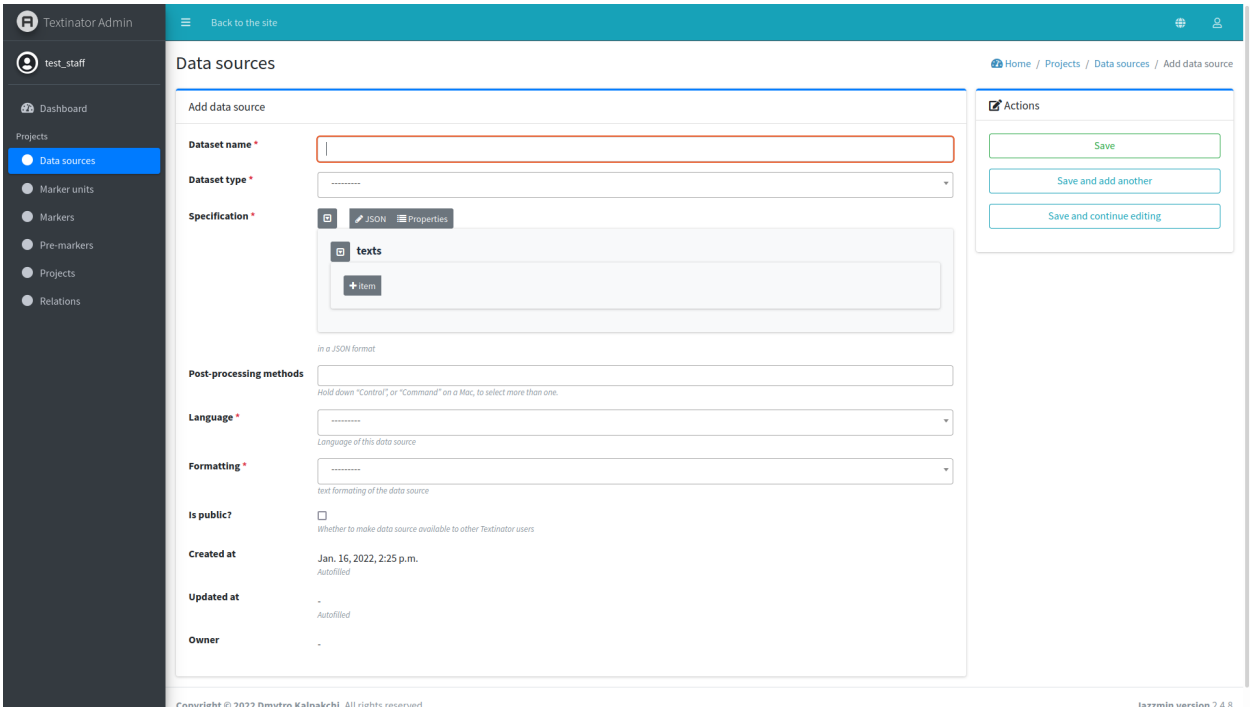
via *Admin panel dashboard*



via *Admin panel/Data sources*



All of these methods will lead to exactly the same form, shown below.



The following fields are **mandatory** for creating a data source:

- *name* - the name of your dataset (max. 50 characters). Although there are no strict requirement on uniqueness, make sure your name is unique enough, so that you can find your dataset when creating the project.
- *type* - currently Textinator supports 4 types of data sources:
  - plain text – input text directly in the admin interface (mostly for testing)

  - plain-text files – a bunch of files hosted on the same server as Textinator
  - JSON files – a bunch of JSON files hosted on the same server as Textinator
  - Texts API – a REST API that will be used for getting each datapoint (the endpoint should be specified)
- *specification* - a JSON-specification, dependent on type:
  - for *plain text* type you just enter a number of textual snippets to be used as data for annotation;
  - for *plain-text files* you need to specify files and/or folders containing your files (see below on where these files/folders should be located);
  - for *JSON files* you need to specify files and/or folders similar to *plain-text files*, but also a key in the JSON object that will contain the text;
  - for *Texts API* you need to specify only the endpoint to the server compatible with Texts API (see below).
- *language* - the language of the data
- *formatting* - formatting of the data, can be either plain text or formatted text (e.g., with tabs) or markdown.

Optional fields include:

- *post-processing methods* - any Python methods defined by your system administrator that can be used for cleaning the data (e.g., remove Wikipedia's infoboxes). Note that currently Textinator does NOT provide any such methods by default, so talk to your system administrator if you need any such methods.
- *is public?* - by default all data sources are private and can be accessed via UI only by the person who created the data source. If you want to make it accessible to all Textinator users, tick this option. Note, that all people with access to your server **will** be able to access the underlying data (unless you use Texts API).

1.2.1 Which data source type should I choose?

If you want to do quick and dirty annotation test, say, to check how well the annotators understand the instructions, you should use a *plain text* type and just copy-paste a couple of texts there. Recall that if your texts are pre-formatted (e.g., with tabs) or contain markdown, you should specify the formatting type accordingly.

In all other circumstances we **recommend** using *Texts API* for multiple reasons:

1. *Limiting access to your data*. If you use either *Plain-text files* or *JSON files*, they should be located on the very same server as Textinator. So at the very least your system administrator will have access to your data. A good way to avoid it is to setup your own server, compatible with Texts API (read below), so that you can have full control over who has access to the data.
2. *Decoupling*. Textinator is an annotation platform, not a data management platform.
3. *Flexibility*. Data comes in all possible shapes and forms and it would be a very hard task to support various data sources. For instance, some researchers might have data in MySQL or SQLite, others in MongoDB and others in ElasticSearch. Supporting all of these inputs, some of which may change their APIs in future, is a mammoth task. Instead, people can implement their own small REST APIs and just provide the endpoint to Textinator - much more flexible!
4. *Privacy considerations*. If you want to annotate e-mails or SMS, then authors of the data might request deletion of their data quoting laws such as GDPR in the European Union. Neither deleting only parts of the datasets nor anonymizing the data is possible via Textinator.
5. *Disk space limitation*. Textual datasets can get quite large (think Wikipedia), which will induce unnecessary overhead on the machine running Textinator - we want to avoid that. Furthermore, as time passes, even smaller datasets in large amounts may end up requiring unreasonable large amounts of space. At that point, one would need to set some kind of expiration policy as to when data should be auto-removed along with reminders to the data owners… So I will just reiterate Textinator is an annotation platform :)

The remaining two types of data sources (*plain-text files* and *JSON files*) are left for backward compatibility with pre-release versions of Textinator. They allow you to upload your data directly to the server via the secure tool of your choice (e.g., *scp* or *rsync*) to the data folder (*Textinator/data* by default). Then you can specify the paths to the files/folders relative to this data folder, given that they are either plain-text or JSON files.

1.2.2 What server is compatible with Texts API?

Note: Requires programming skills.

Texts API is pretty simple and requires your server to support 4 GET requests:

```
GET /get_datapoint?key=your-key HTTP/1.1
```

Response

```
{
    "text": "text-for-your-key"
}
```

```
GET /get_random_datapoint HTTP/1.1
```

Response

```
{
    "key": "key-for-the-random-datapoint",
    "text": "text-for-the-key-above"
}
```

```
GET /size HTTP/1.1
```

Response

```
{
    "size": "size-of-your-dataset"
}
```

```
GET /get_source_name?key=your-key HTTP/1.1
```

Response

```
{
    "name": "source-name-for-the-datapoint-under-your-key"
}
```

A simple example Flask server is provided in the example_texts_api folder in the GitHub repository.

1.2.3 What if I really want to upload data via UI?

> **Warning:** This feature is subject to change or removal in future.

Currently there is no *recommended* way of uploading your files into Textinator. However, if you really insist, there is a temporary workaround that has multiple limitations (introduced to discourage its usage):

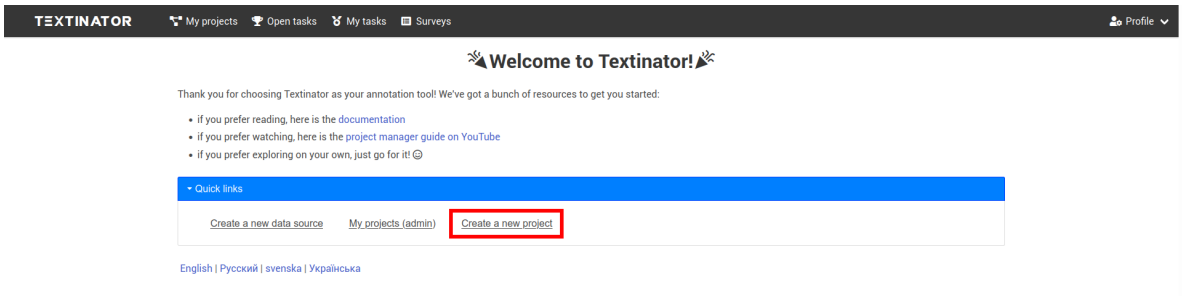
1. Your data will be accessible by **all** other staff members of Textinator. So this solution should only be used either if you are the only user of Textinator or there is an honor code in place.
2. The upload size is limited to 20MB per file.
3. You are still limited to either plain-text files or JSON files (that can contain plain text, preformatted text or markdown though).

In order to use this workaround, you need to ask your system administrator to add you to the *file_managers* user group. Then you will see "FileBrowser" in the menu of the admin UI and will be able to access Textinator's file browser. You will then need to create a folder with the same name as your username and upload your files in that folder. If you place your files in any other folder, they will **NOT** be seen by Textinator.

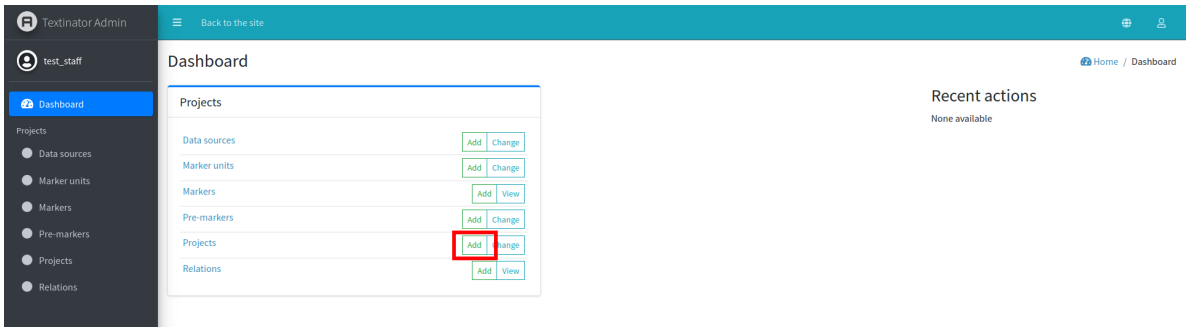
1.3 [Part 3] Creating your first project

After having created a data source, we can actually go ahead and create our first Textinator project. Similar to adding a new data source, there are 3 ways of accessing the project creation form.

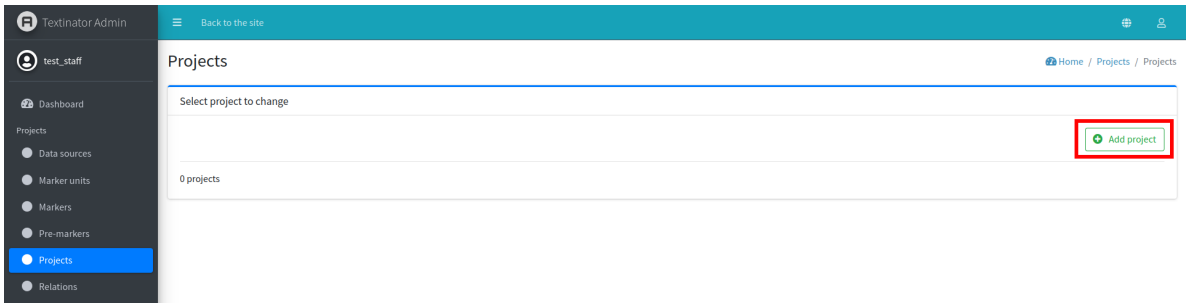
via *Quick links* pane on the *Welcome* page.



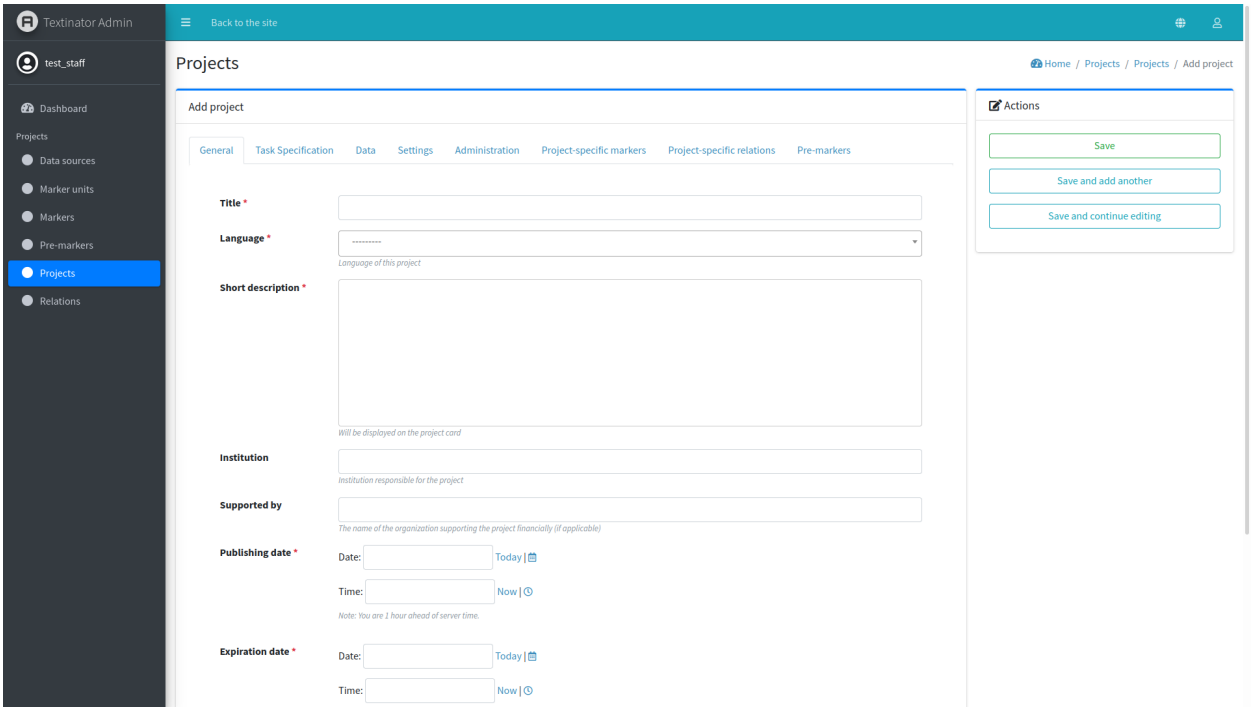
via *Admin panel dashboard*



via *Admin panel/Projects*



All of these methods will lead to exactly the same form, shown below.



The following fields are **mandatory** for creating a project:

Generic tab

- *title* - the title of your project (max. 50 characters);
- *language* - the language of your project, needs to match the language of the data sources (is used to filter out only annotators who have indicated to be fluent in that language);
- *short description* - a short and concise description of your project (to be shown in the Project card);
- *publishing date* - date of publishing (important for projects, open to public, which is true by default);
- *expiration date* - last date when the annotations can still be performed.

Task specification tab

- *type of the annotation task* - one of the *8 annotation tasks supported out of the box* or *Generic* for your custom annotation tasks.

Data tab

- *data sources* - the data to be annotated, needs to match the langauge of the project.

The following fields are *optional* and most of them are self-explanatory. We will highlight only those that relate to the mandatory fields.

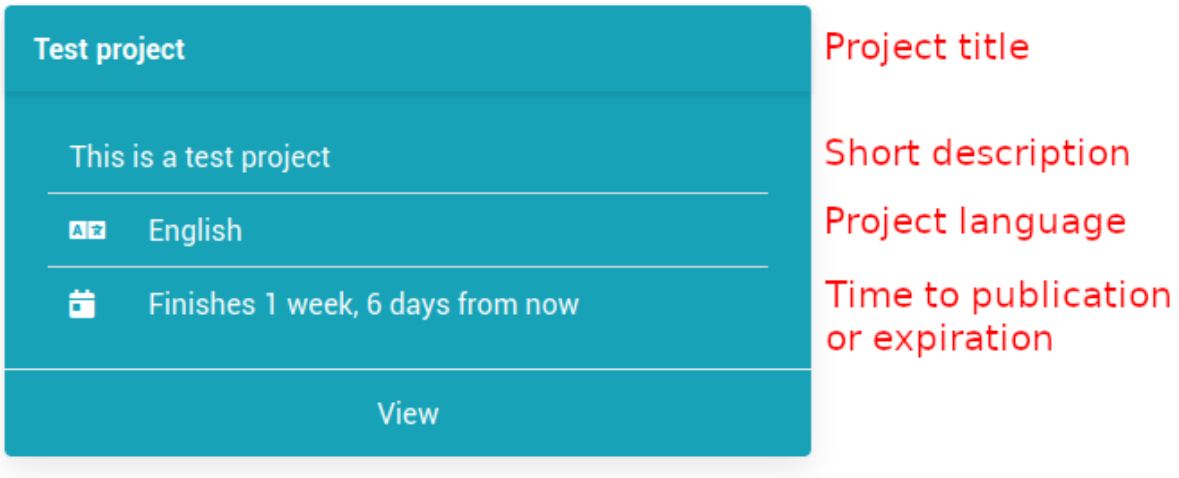
Task specification tab

- *guidelines* - the rich-text input for annotation guidelines, where you could give an elaborate description with examples (these guidelines will be accessible by annotators at all times as a modal window);
- *reminders* - the summary of essential points of the guidelines, visible at all times and cannot be hidden.

Settings tab

- *should the project be public?* - a switch of whether the project should be open to public (then publishing date plays a vital role). If the switch is off, the users will not see the project
- *should selecting the labels be allowed?* - whether annotated spans of text should be clickable (**essential** that it is turned on if you have any relations, since this is how relations are annotated in the current version of Textinator)

After you have created the project it should appear under *My projects* tab in Textinator and have a card that has the following anatomy.



If you have added a summary video, it will appear between the project title and short description.

1.4 [Part 4] Exploring a data explorer

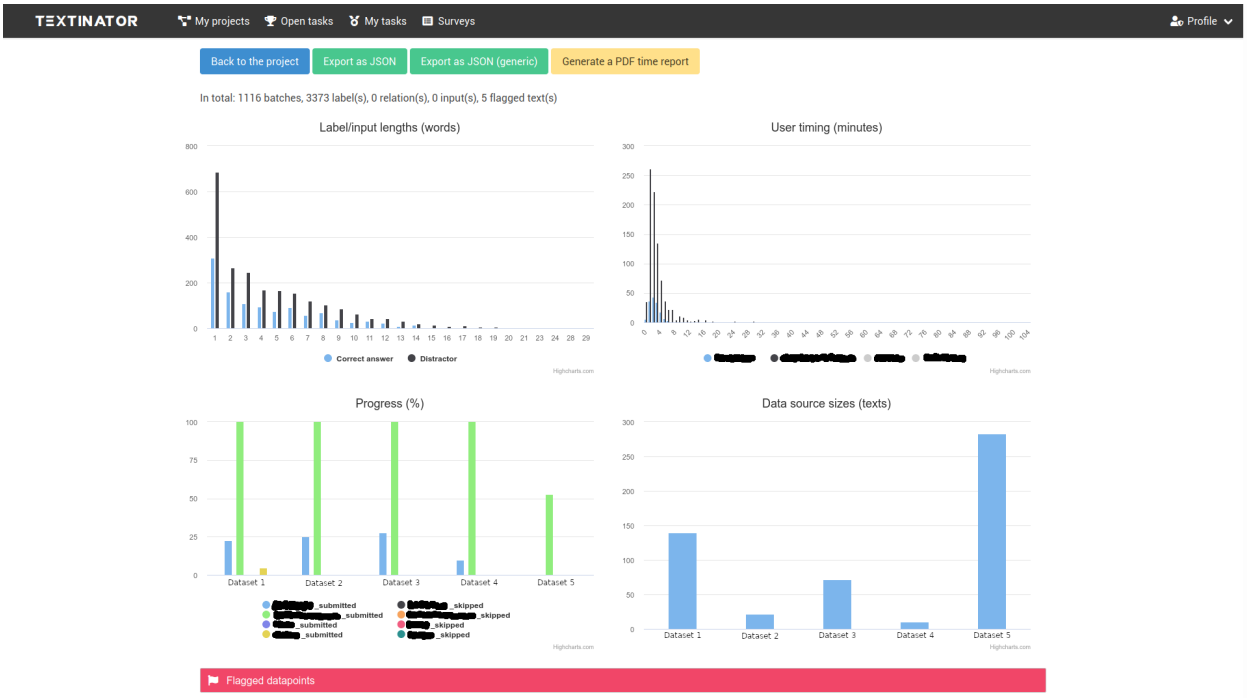
Table of Contents

The role of data explorer is to provide you with administration capabilities, as well as give a birds-eye view of the annotated data.

Note: The data overview functionality is expected to broaden in Textinator v1.1

1.4.1 Annotation statistics & progress tracking

Textinator provides a birds-eye view of the annotated data and the annotation process. Speficially, it shows the distribution of lengths for each marker that is present in the project (the top left graph in the screenshot below).



For administrative purposes, you can also track the timing per annotation for each annotator (top right graph), their annotation progress and how many texts they chose to skip (bottom left graph), as well the overall sizes of the data sources, as a reminder (bottom right graph). Note that the screenshot is taken from the real-world annotation project, so the names of the data sources and annotators are edited out for privacy reasons. On top of that if annotators flag any texts as problematic, you can also see their comments in the data explorer under the red pane called “Flagged texts” (see example in the screenshot below).

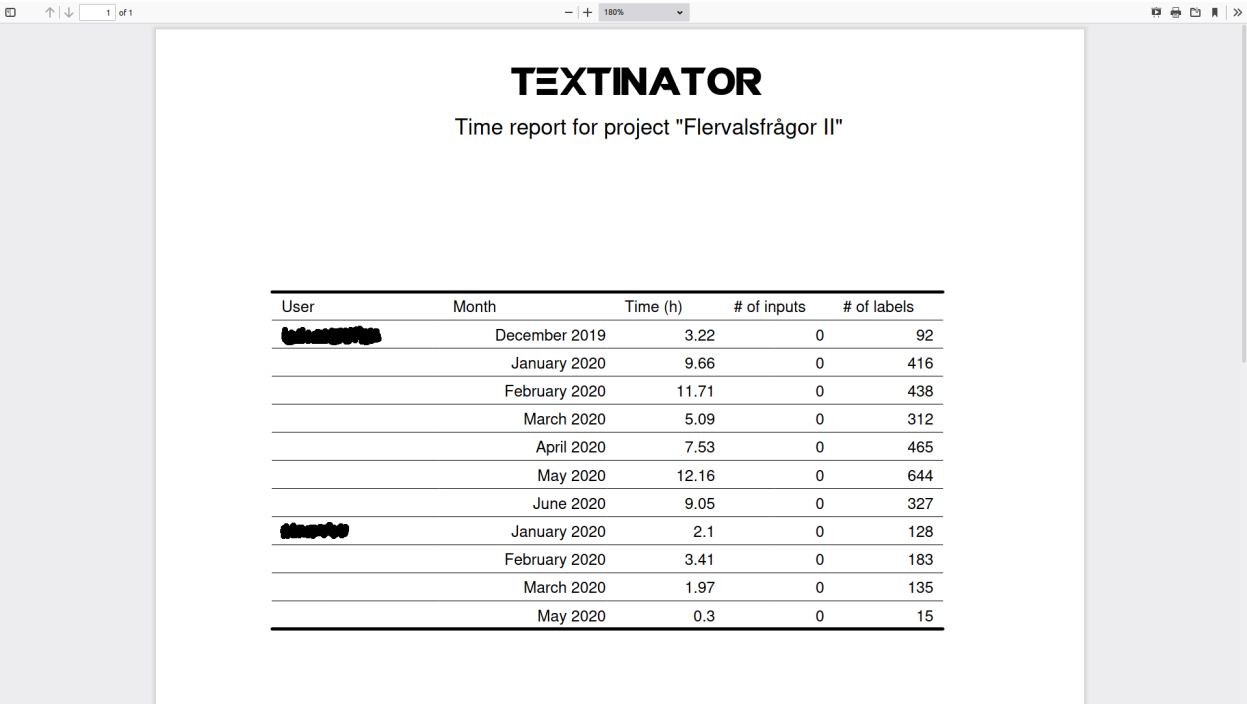
1.4.2 Exporting annotations

Textinator exports data in a custom concise JSON format specifically designed for each annotation task, supported out of the box. You will get data exported in this format if you click on the green "Export to JSON" button (see screenshots in the previous section).

If you have done any customization to an out-of-the-box task, we recommend using a generic export functionality, featuring a generic export format (thus less concise), but including all of your annotation. You can use generic export by clicking the green "Export to JSON (generic)" button.

1.4.3 PDF time report

Textinator is also capable of generating a per-month time report per annotator. Note that this report can **NOT** be a ground for payments, since Textinator provides only an *estimate* of the spent time. For instance, the time needed for breaks or potential research connected to the annotation process should also be counted, but is not accounted for in Textinator's report. The report should be mostly used as a sanity check and if the reported numbers and the real numbers differ by a substantial amount, it's just a flag to the project manager that this should be investigated further.



TEXTINATOR

Time report for project "Flervalsfrågor II"

User	Month	Time (h)	# of inputs	# of labels
[redacted]	December 2019	3.22	0	92
	January 2020	9.66	0	416
	February 2020	11.71	0	438
	March 2020	5.09	0	312
	April 2020	7.53	0	465
	May 2020	12.16	0	644
	June 2020	9.05	0	327
[redacted]	January 2020	2.1	0	128
	February 2020	3.41	0	183
	March 2020	1.97	0	135
	May 2020	0.3	0	15

1.5 [Part 5] Creating a custom annotation task

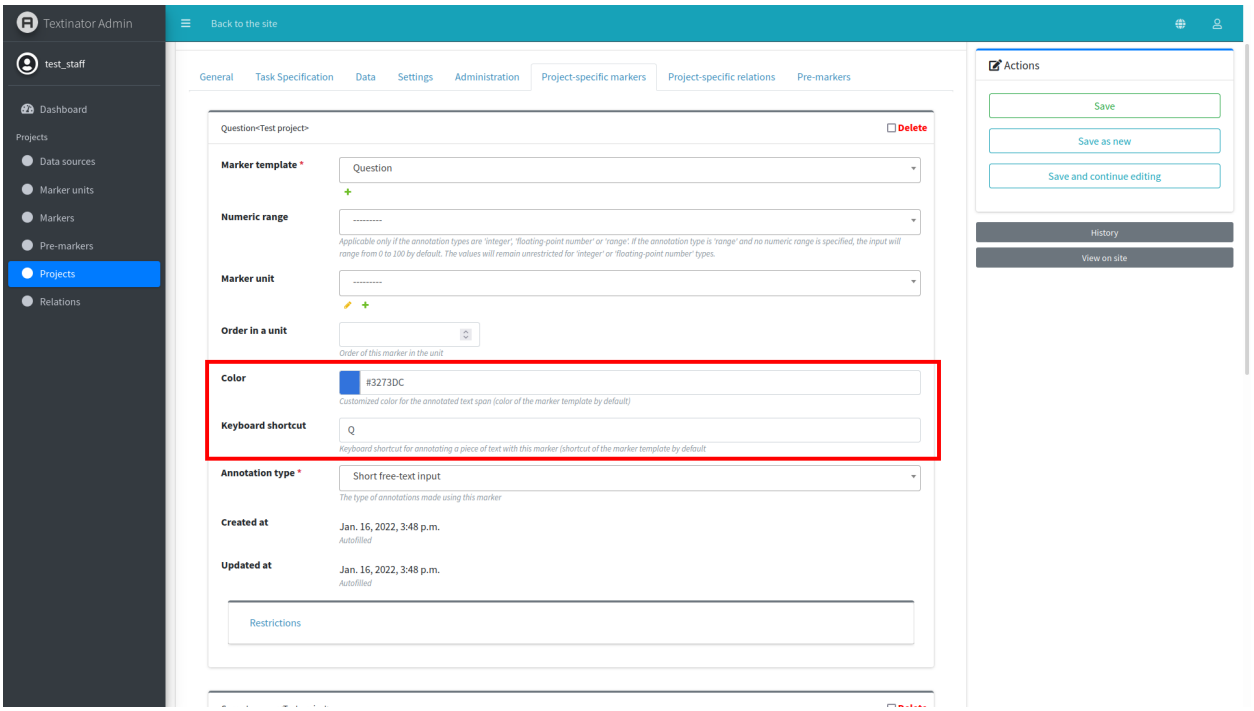
Table of Contents

1.5.1 Minor modifications to out-of-the-box task types

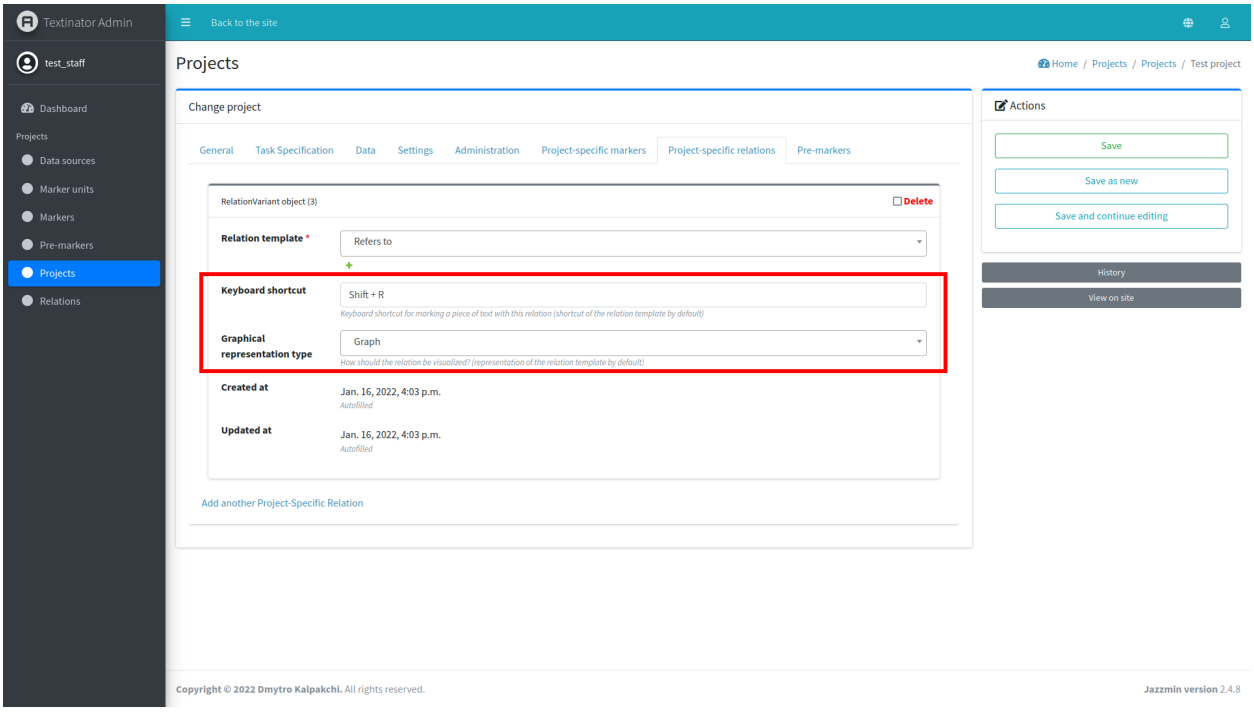
You can the following modifications to the out-of-the-box tasks without any ramifications to the data export:

- color of markers
- hotkeys for markers or relations
- changing visualization type for a relation (graph or list)
- adding custom restrictions on the number of markers (e.g., there should be at least 2 distractors per each submitted multiple choice question)

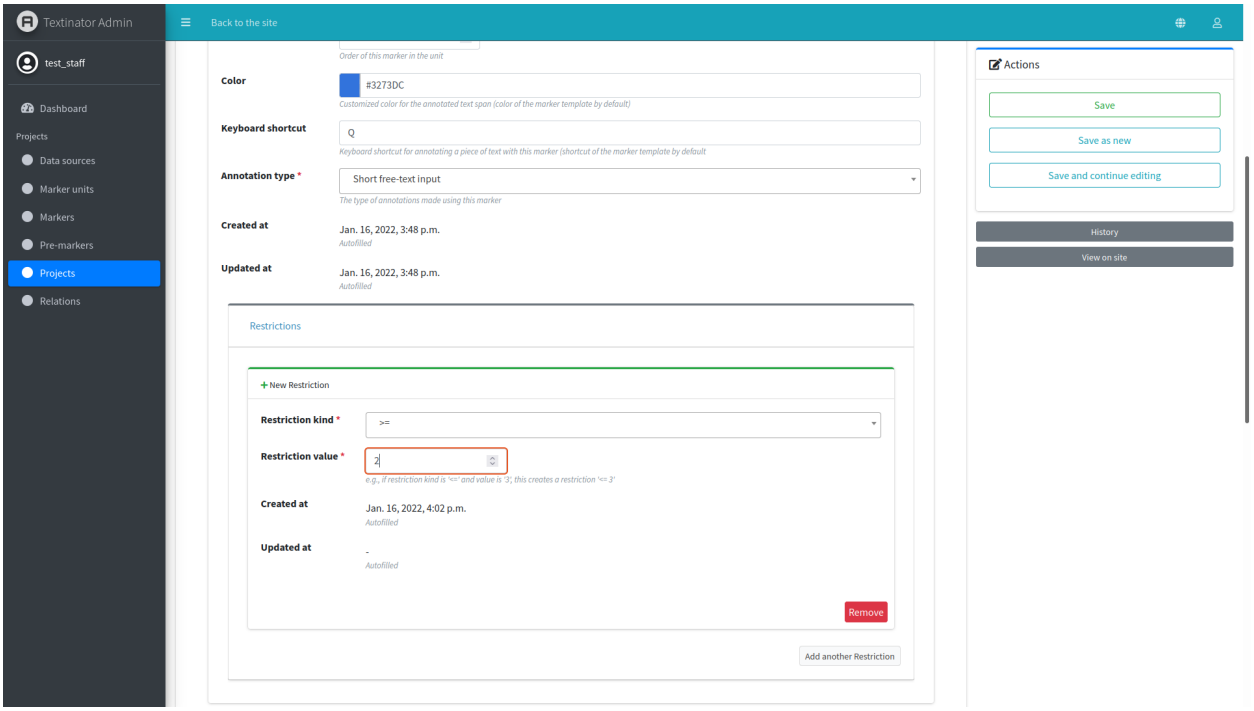
The changes of color and hotkeys for the markers can be done via the project's form *Project-specific markers* tab (relevant fields are marked by a red rectangle in the screenshot below).



The changes of hotkeys and visualization type for the relations can be done via the project's form *Project-specific relations* tab (relevant fields are marked by a red rectangle in the screenshot below).



Custom restrictions for the markers can be added via the *Restrictions* pane available at the bottom of each *Marker's* form. For instance, the restriction of having at least 2 markers of such kind can be added as shown in the screenshot below.



1.5.2 Major modifications to out-of-the-box task types

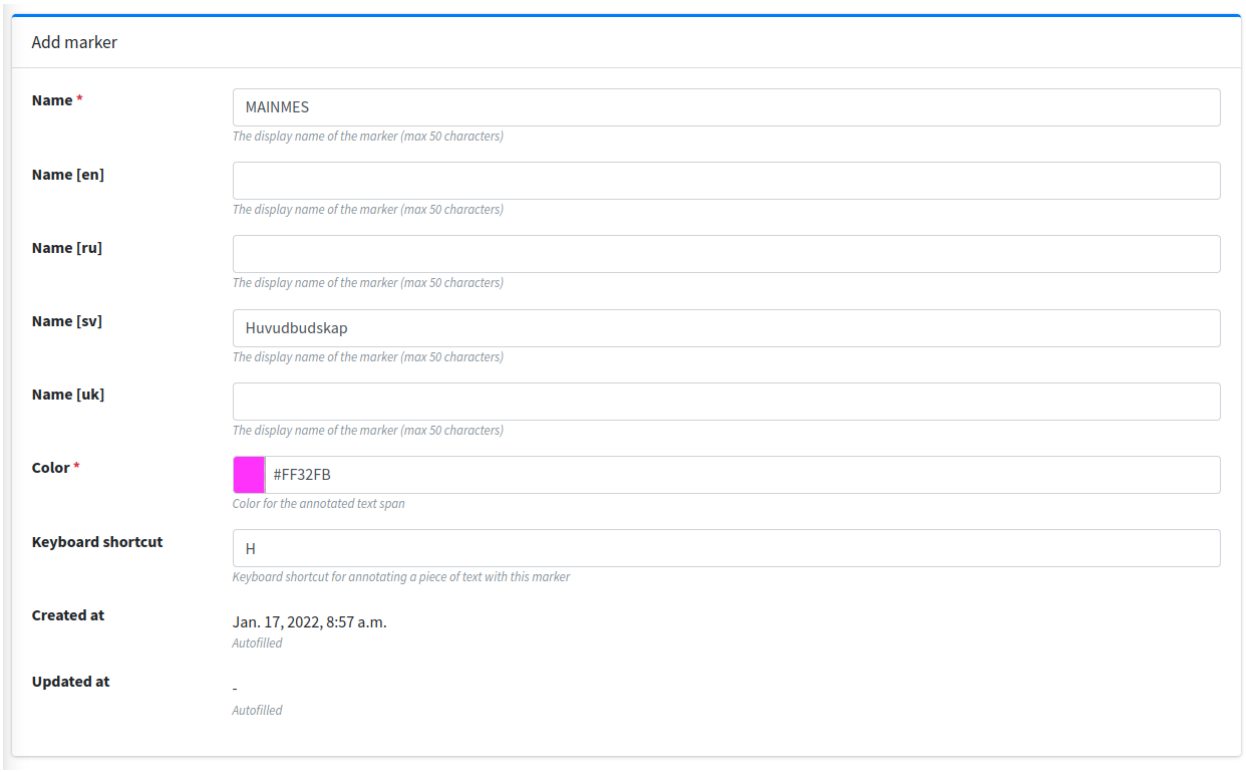
Adding custom markers or relations to the out-of-the-box tasks is also possible, but current export functionality is configured to work only with the default markers/relations to give as consice JSON file as possible. Having said that, you could try to add markers/relations to your task and see if the default export functionality still works. If it doesn't, you can use the generic export by clicking on the *Export JSON (generic)* button under *Data explorer*, which is guaranteed to contain all annotations, albeit in a somewhat longer format.

1.5.3 Defining custom markers

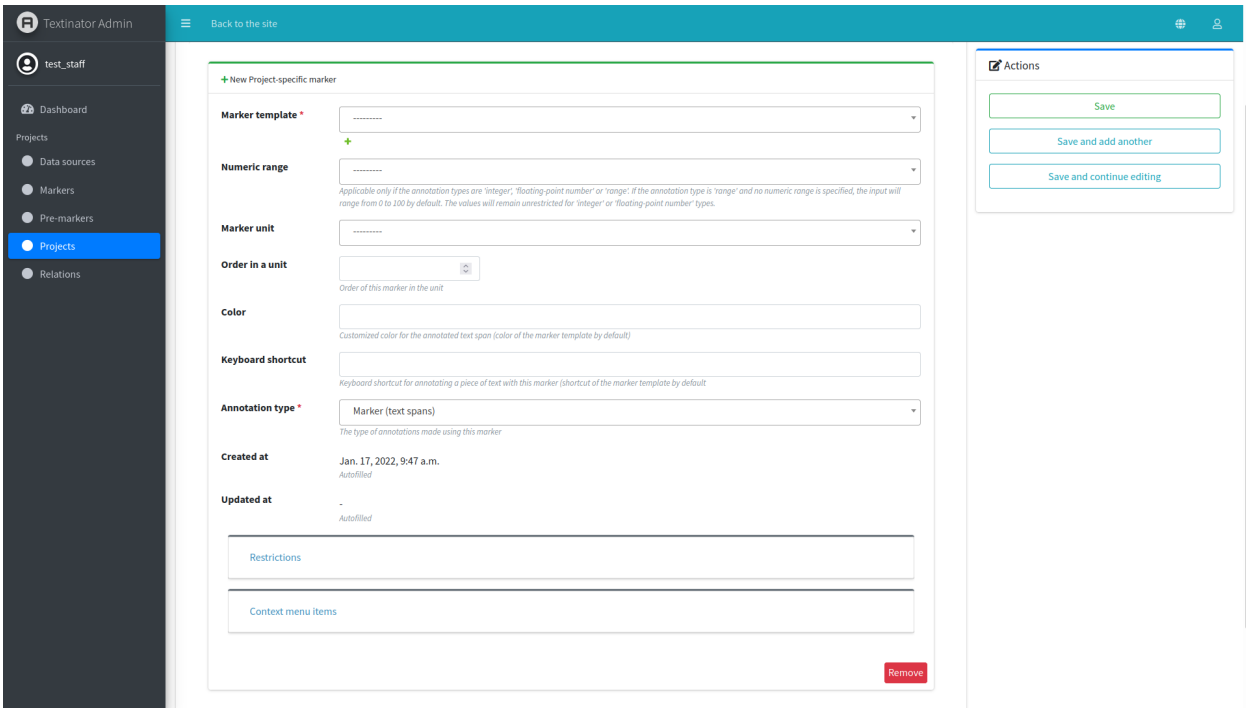
If you want to annotate a task currently unsupported by Textinator or make a major modification to an already existing task, you will have to define custom units of annotation. Textinator supports such customized definitions through *Markers* via *Admin panel/Markers*. You need just a couple of things to define a basic *Marker*:

- defining a marker name to be used when exporting data (mandatory);
- choose a color (mandatory);
- defining the translation of the marker name to the language (among supported by Textinator) that you are going to use for annotation (optional, but highly recommended);
- choose a shortcut for the marker *(optional)*.

Let's say we want the annotators to find and mark the main message of the text in Swedish, then filled-in *Marker* fields (corresponding to the properties listed above), would look like in the picture below.



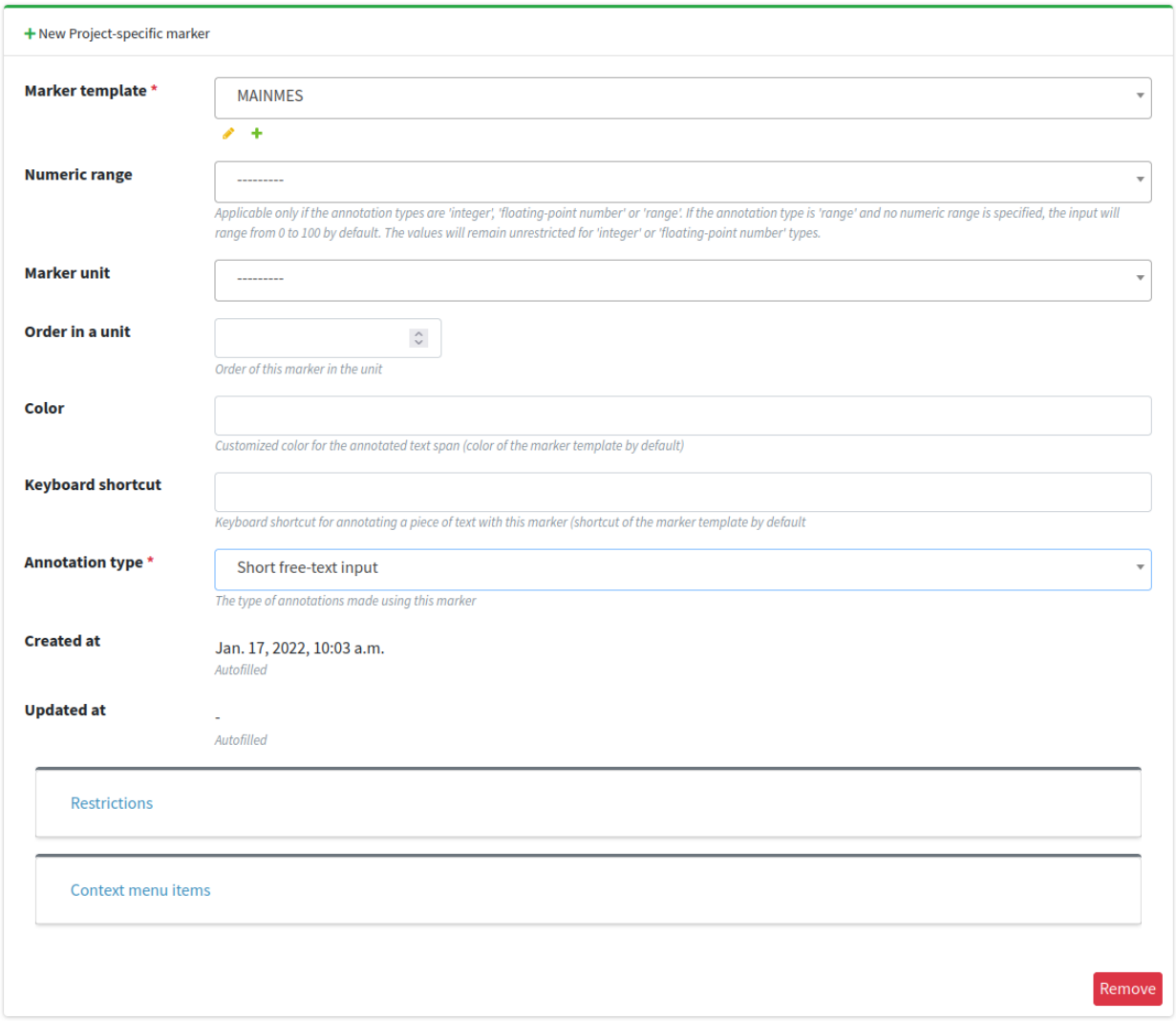
Now that we have defined a *Marker*, this definition will be accessible to all Textinator staff members. Now we need to add a marker following this definition to our project. In order to do that you should find the project of interest via *Admin panel/Projects*. Open the project and choose the tab called *Project-specific markers* and then click *Add another Project-Specific Marker*. You should get a form similar to the one shown below



Choose a *Marker* that you have defined before and define variant-specific properties:

1. You need to specify the type of annotations that should be made with this marker in the project. For instance, in some cases, you want to mark the correct answer in the text, in which case you should select *Marker (text spans)* as your annotation type. If you do not want the correct answers to be in the text, you might want to give annotators the freedom of providing them as a text input, in which case select *Short free-text input*. If you want to perform text classification, you will need to select *Marker (whole text)*. The other marker types are self-explanatory.
2. If you want your marker to be annotated as a part of the unit, you will need to specify a marker unit. For instance, when creating multiple choice questions, consisting of a question, a correct answer and 3 distractors, then all of them would be considered a unit. In which case you will need to create a unit first (by clicking on the green "+" button below the *Marker unit* field) and then choosing one and same unit for all 3 markers (question, correct answer and distractor).
3. If a marker belongs to a unit, you can also specify order of a marker in the unit by using *Order in a unit* field. For instance, if you want markers to appear in the order question - correct answer - distractor, then the *Order in a unit* field of the *Question* marker should have the value of 1, of the *Correct answer* marker - the value of 2 and of the *Distractor* marker - the value of 3.
4. If you require a specific number of annotations to be made by a marker prior to the submission, you could defined that using the *Restrictions* pane. For instance, if you need your annotators to enter at least two distractors, you need to add a *Restriction* of the kind >= and the value of 2. **Note** that you can add a restriction only **after** you have saved your marker variant for the first time.
5. If you wanted to define custom actions, available when right-clicking the marker, you could define them using *Context menu items* pane. **Note** that you can add a context menu item only **after** you have saved your marker variant for the first time. Also note that only actions previously defined by the system administrator can be used for context menu items.

In our example case, we want annotators to be able to enter main message as a free text and this is the only marker connected to it, so no units are required. We also do not need any restrictions or context menu items. Hence, the filled in form would look as below.

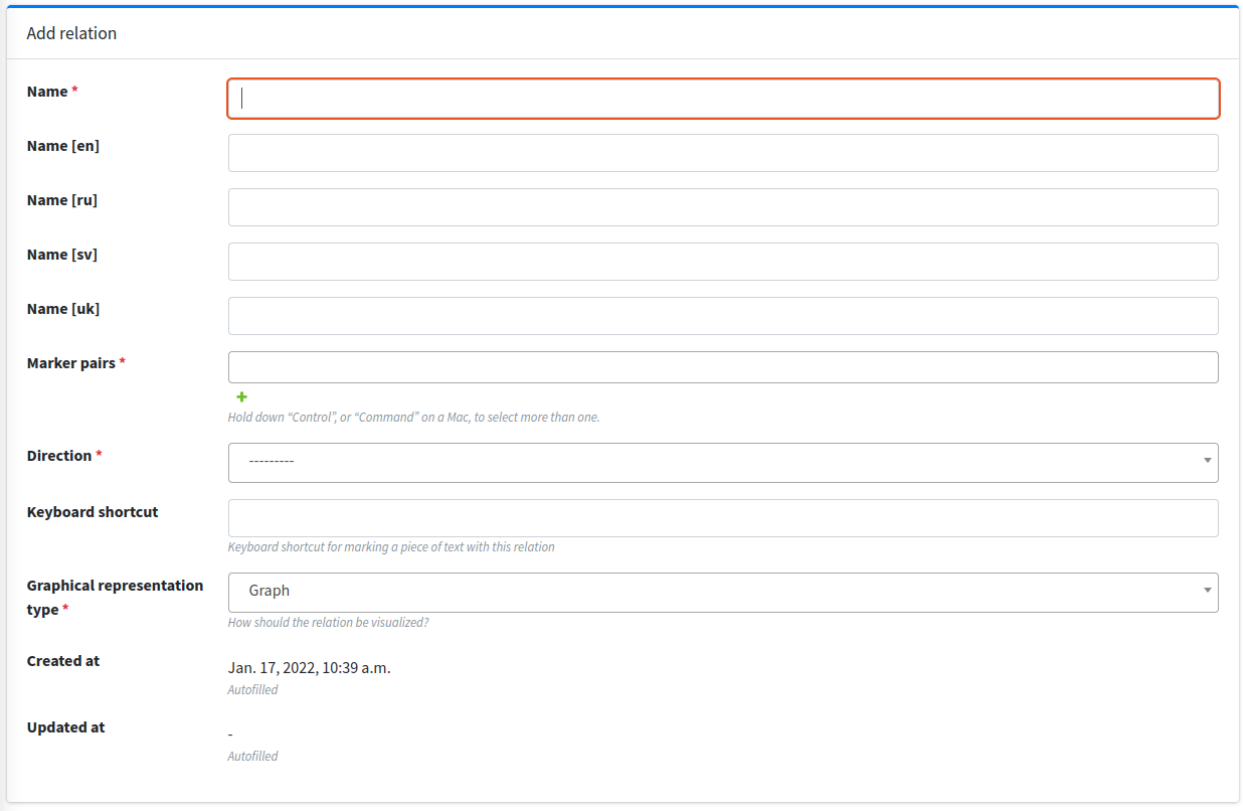


1.5.4 Defining custom relations

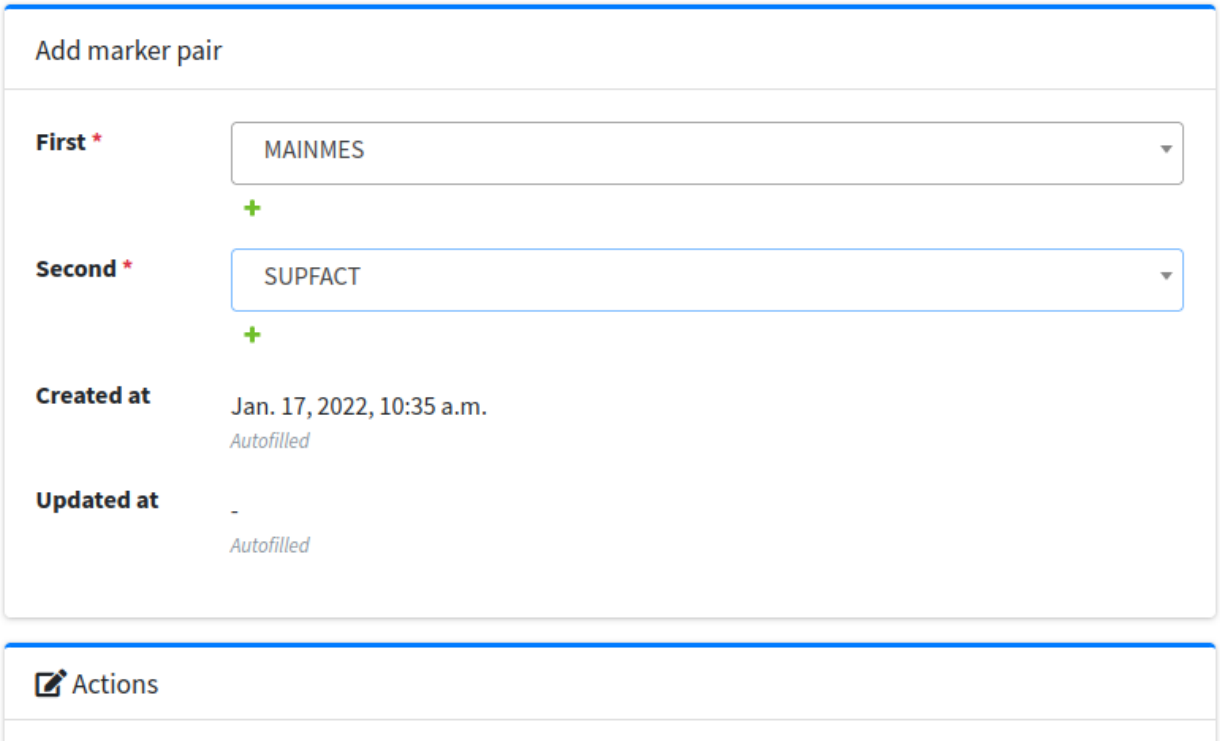
If you have more than one marker, then you might want to define custom relations between the markers. Textinator supports such customized definitions through *Relations*. You need just a couple of things to define a basic *Relation*:

- defining a relation name to be used when exporting data (mandatory);
- choose marker pairs, for which the relation is applicable (mandatory);
- choose the directionality of the relation (mandatory);
- choose graphical representation type, i.e., graph or list (mandatory);
- defining the translation of the relation name to the language (among supported by Textinator) that you are going to use for annotation (optional, but highly recommended);
- choose a shortcut for the relation *(optional)*.

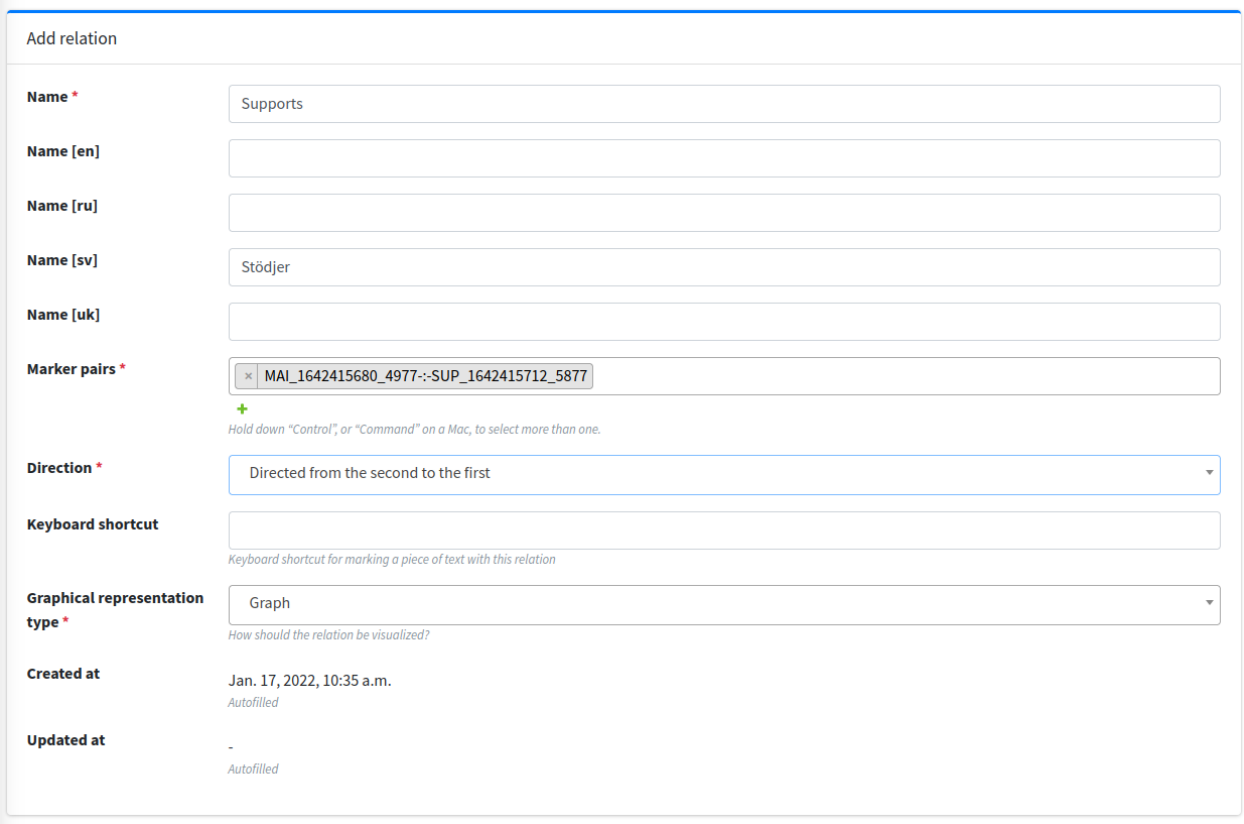
Let's say we want the annotators to specify a supporting fact for each main message they find. Then we need to define another marker called *Supporting fact* (using the same procedure as before). Then we can define a relation *Supports* between the *Supporting fact* and *Main message* via *Admin panel/Relations* using the form below.



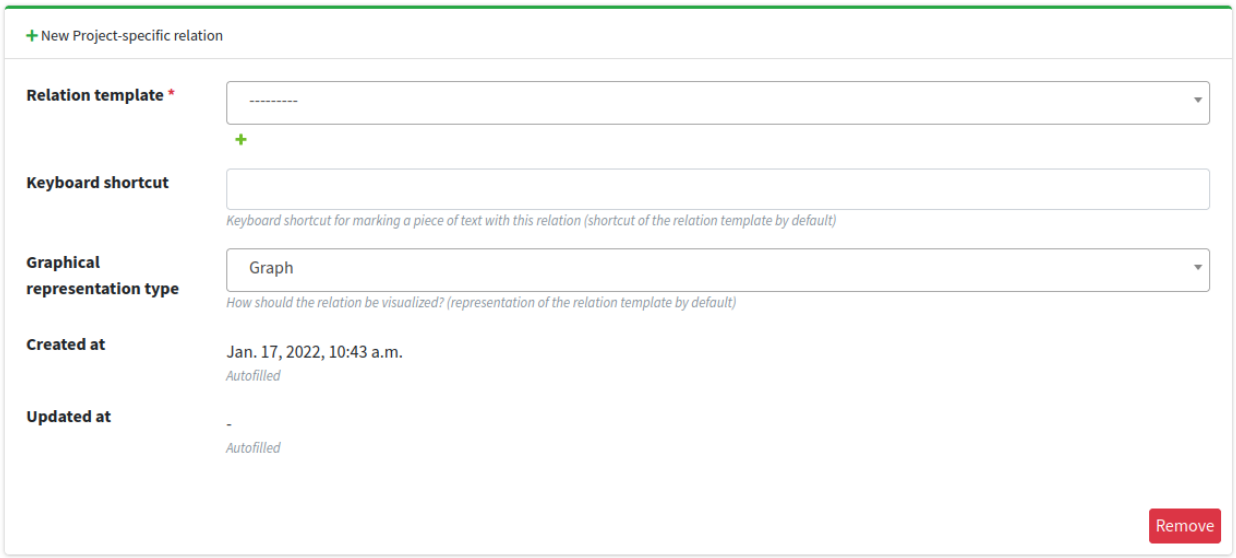
First you will need to define a marker pair of *Main message* and *Supporting fact*, which you can do by clicking on the green plus icon below the *Marker pairs* text field. This will bring the pop-up window, which would look as follows when filled in.



Afterwards, you should click on *Save* in the pop-up under *Actions* and the popup should close adding this pair to the *Marker pairs* field of the original relation. The completely filled-in form for the relation should look as shown below.



Now that we have defined a *Relation*, this definition will be accessible to all Textinator staff members. Now we need to add a relation following this definition to our project. In order to do that you should find the project of interest via *Admin panel/Projects*. Open the project and choose the tab called *Project-specific relations* and then click *Add another Project-Specific Relation*. You should get a form similar to the one shown below.

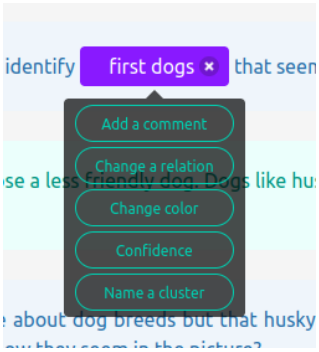


In this form you simply need to choose a newly created relation as a relation template - that's the only mandatory field. You could also customize the representation type or a hotkey, although you would typically want to do that if you are

re-using someone else's relation.

1.6 [Part 6] Associating actions with your markers

Remember that in Textinator the definition of a markable is called *Marker* and the specific instantiation of a *Marker* in a given text is called *Label*. Textinator provides a flexible way of associating a number of actions with each *Label* by simply right clicking on them and getting a context menu with those actions (as shown on the screenshot below, for instance).

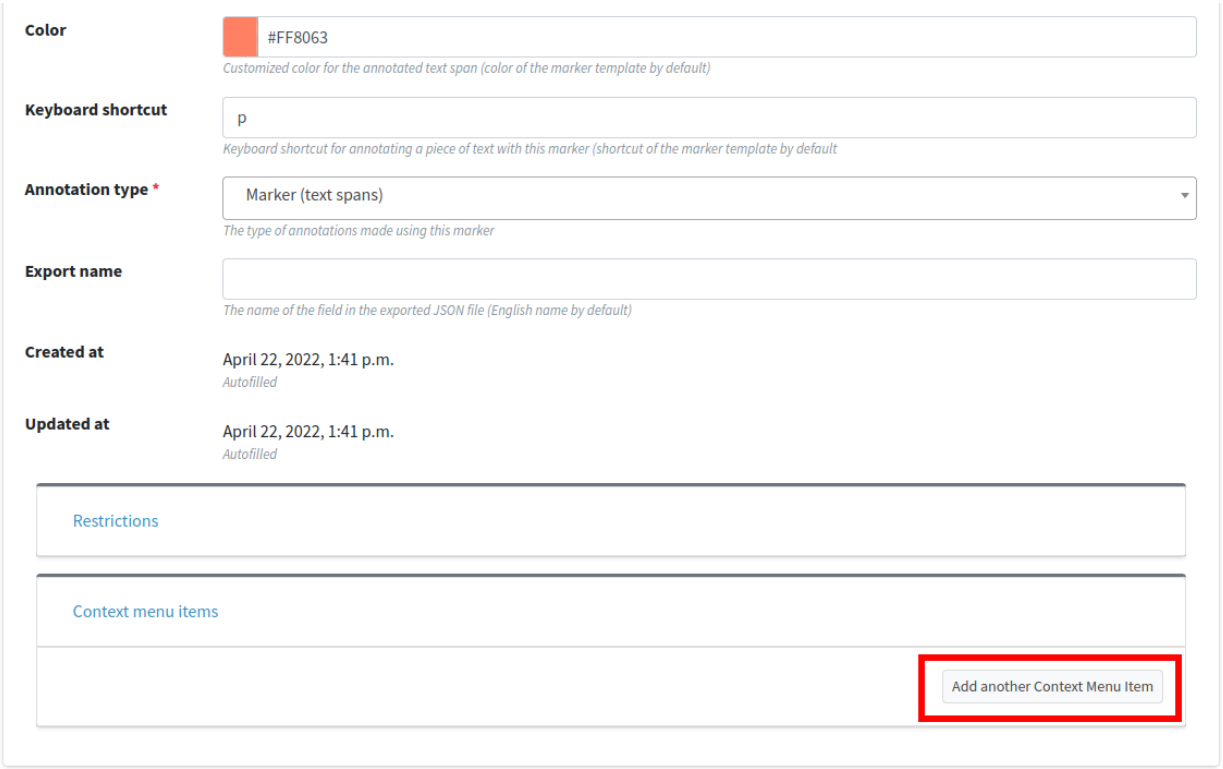


Each of the green buttons in the context menu is an action, associated with the given *Label* (as defined for the given *Marker*). Each action is an instantiation of one of the so called labeler plugins (read more about them in the developer documentation). Currently, there are four such plugins available out of the box:

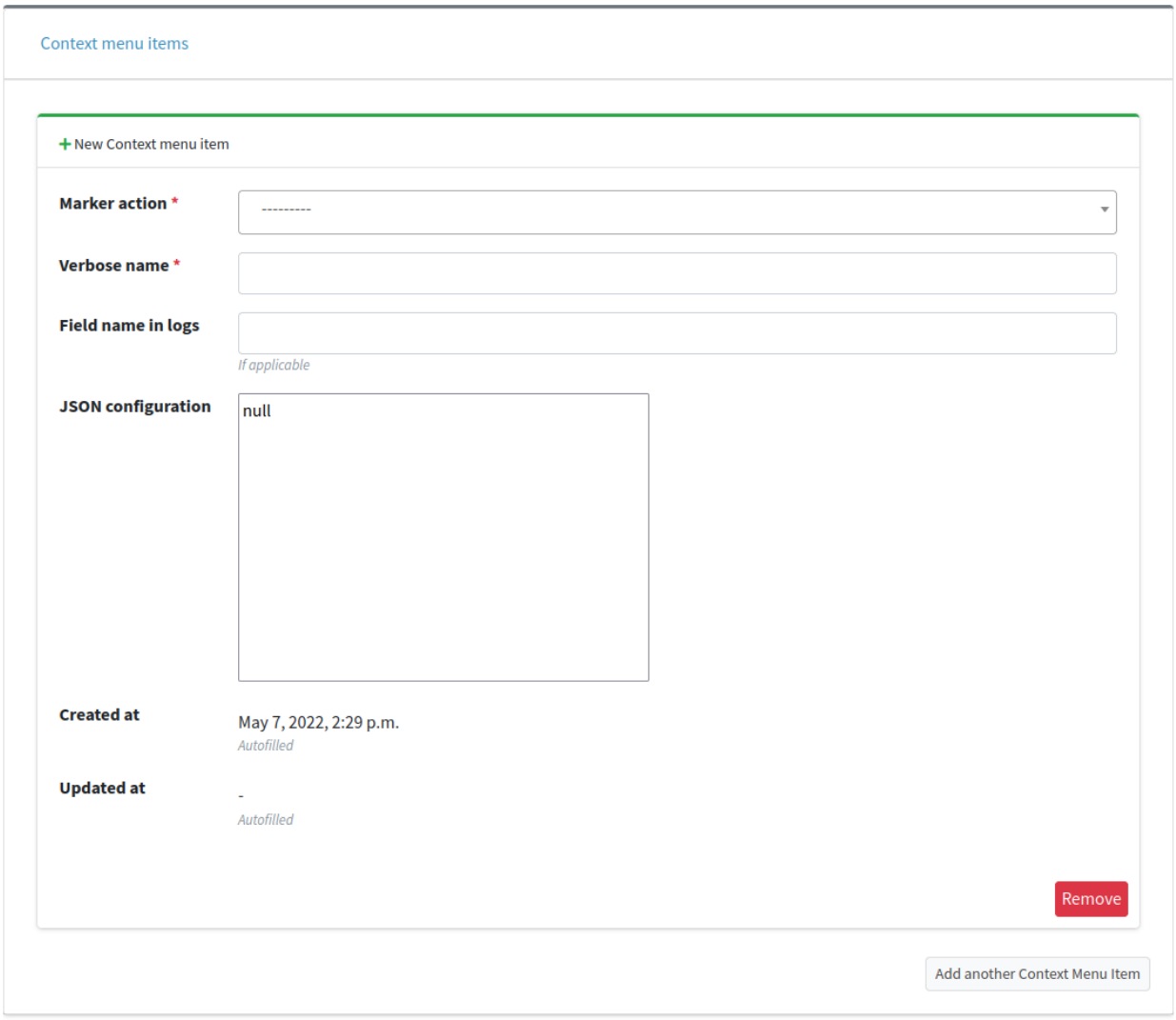
- A plugin adding a text field to a marker's context menu, potentially shared between markers
- A plugin allowing to change a relationship of a label
- A plugin allowing to change a color of a label, potentially shared between labels
- A plugin adding a slider to a marker's context menu

To add an action to any *Marker*, you need to navigate to the project page in the admin interface and open the *Project-specific markers* tab. There you need to find the marker you want to add your actions to and find the *Context menu items* subpane, as shown in the screenshot below.

Click on it to open the subpane and then click on *Add another Context menu item* button (as highlighted in red in the screenshot below).



This should result in the form for adding a new context menu item.



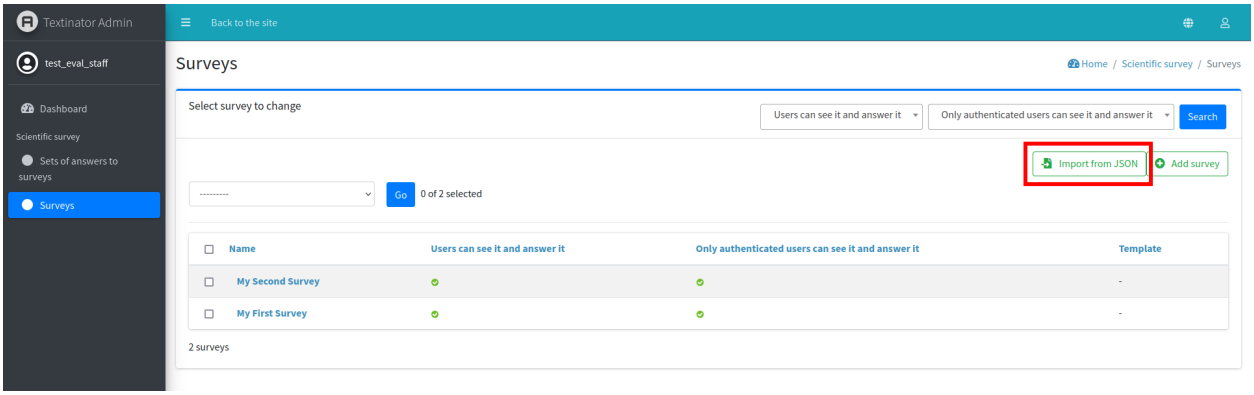
The form has the following fields to be filled in:

- *marker action* - one of the four aforementioned plugins;
- *verbose name* - the name of the context menu items, as will appear on one of the green context menu item buttons;
- *field name in logs* - the name of the fields in the `extra` dictionary when exported (by default the name of the marker action itself);
- *JSON configuration* - the configuration for the plugin (as defined in the documentation page for each plugin);

1.7 [Part 7] Setting up human evaluation

Textinator also provides rich capabilities for conducting human evaluation about the textual data in the form of surveys. In order to be able to work with surveys, you need to be a staff member and be added to the *evaluation_managers* group (you can check that similar to the how you checked the *project_managers* group earlier in the tutorial).

You could add a *Survey* in a similar fashion to either a *Dataset* or a *Project*, as we did before (via *Admin panel/Surveys/Add survey*). However, while the fields of the *Survey* creation form are self-explanatory, it can be tedious to add, say 50 survey items manually, which is why Textinator offers an import function! You can access an import function via *Admin panel/Surveys* as shown in the screenshot below.



An example of the required JSON format is given below (and more examples are available here).

```
{
  "name": "name-of-your-survey",
  "categories": [
    "first-category",
    "second-category"
  ],
  "items": [
    {
      "question": "Sentence: Does it work?<br>Paraphrase: Would it work?",
      "required": true,
      "category": 1,
      "extra": {
        "model": "A"
      },
      "order": -1,
      "answer_sets": [
        {
          "type": "radio",
          "name": "criterion-1",
          "choices": ["0", "1", "2"]
        },
        {
          "type": "radio",
          "name": "criterion-2",
          "choices": ["0", "1", "2", "3"]
        }
      ]
    }
```

(continues on next page)

(continued from previous page)

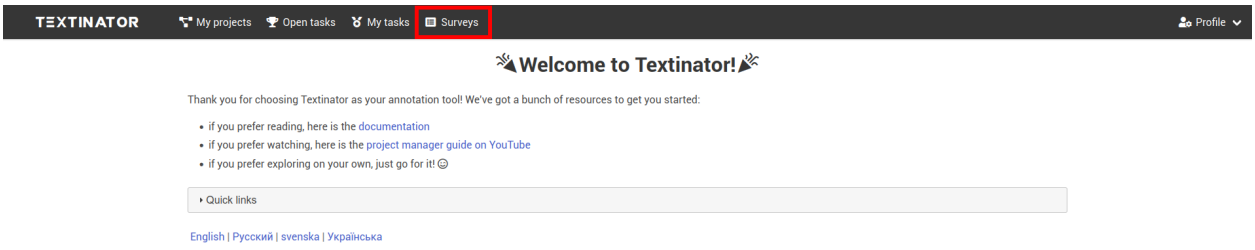
```
  ]
}
```

Each answer set should be one of the following types:

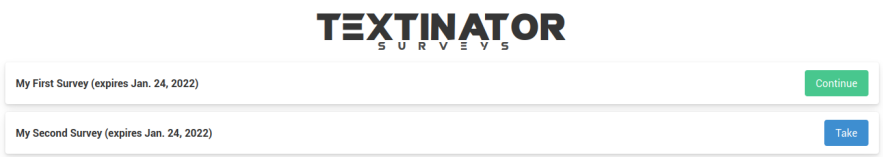
- "text" – provides a textarea for longer texts
- "short-text" – provides a regular HTML input for shorter texts
- "integer" – an HTML number input with only integers allowed
- "float" – an HTML number input with floats allowed
- "date" – a regular text input with date format validation
- "radio" – a radio button(s), with values specified in the "choices" argument
- "select" – a dropdown using an HTML <select> tag, with values specified in the "choices" argument
- "select-multiple" – a checkbox(es) with values specified in the "choices" argument

Note that the "choices" argument in each answer set affects the current answer set only if its type belongs to one of the last three types from the list above.

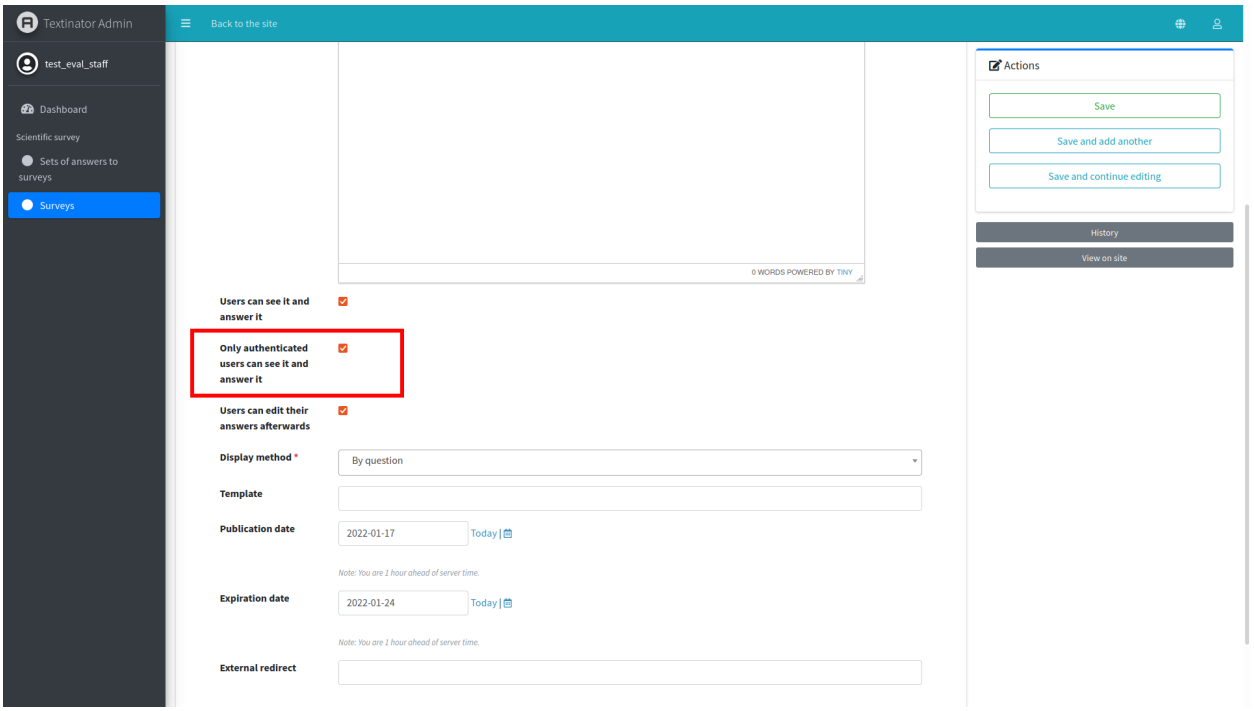
After you have successfully imported a survey, you can access it via *Surveys tab* (marked with a red rectangle in the screenshot below).



The tab should bring you to a separate *Textinator Surveys* page, which contains all currently available surveys.

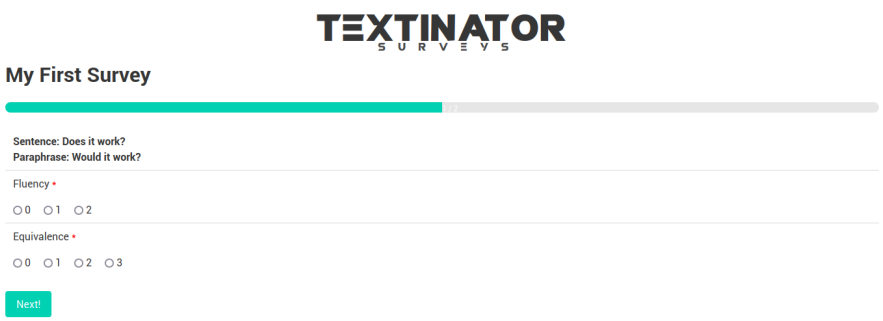


Note that the page has no obvious links to Textinator, since the human judges might be recruited via the crowdsourcing platform and we wanted to skip the whole registration-login workflow for Textinator. To enable that, there is a possibility to turn off the authentication requirement by simply ticking off the necessary checkbox in the settings for your *Survey* via *Admin panel/Surveys* (marked with a red rectangle in the screenshot below).



There is also a possibility to integrate the survey with crowdsourcing platforms requiring a redirect link after finishing the survey (e.g., Prolific). The redirect link could be specified under the *External redirect* field in the screenshot above.

After being done with configuration, the human judges can start taking your survey, one example of which is shown below. Have a great time running human evaluations!



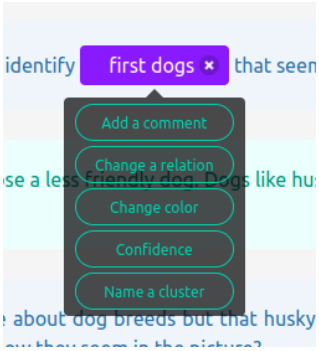
CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 Labeler Plugins API

Remember that in Textinator the definition of a markable is called *Marker* and the specific instantiation of a *Marker* in a given text is called *Label*. Textinator provides a flexible way of associating a number of actions with each *Label* by simply right clicking on them and getting a context menu with those actions (as shown on the screenshot below, for instance).



Each of the green buttons in the context menu is a specific instantiation of a Textinator's labeler plugin. There are four plugins available out of the box:

- A plugin adding a text field to a marker's context menu, potentially shared between markers
- A plugin allowing to change a relationship of a label
- A plugin allowing to change a color of a label, potentially shared between labels
- A plugin adding a slider to a marker's context menu

For a more detailed guide on how to use these, please have a look at the tutorial about associating custom actions with markers. However, if you know some JavaScript, you could add your own custom plugin as well, which is what this page aims at documenting.

Each labeler plugin is just a JavaScript file residing in the folder *static/scripts/labeler_plugins*. If you want to add your own plugin you just need to create a JS file in that directory (e.g. *my_first_plugin.js*). The created file should have the following structure.

```
/**
 * name: <your-plugin-name>
 * description: <your-plugin-description>
 * admin_filter: boolean
 * author: <author-name>
```

(continues on next page)

(continued from previous page)

```
 */

var plugin = function(cfg, labeler) {
  var config = {
    name: "<your-plugin-name>",
    verboseName: '<verbose-name>',  // shown in the context menu
    storeFor: "label",              // one of "label", "relation"
    dispatch: {},                   // the map that triggers events after certain other␣
↪events
    subscribe: [],                  // the events triggering plugin's re-rendering
    allowSingletons: false          // signifies whether markables with no shared␣
↪information should be allowed (only if storeFor: "relation")
  }

  // YOUR OWN PRIVATE FUNCTIONS

  return {
    name: config.name,
    verboseName: config.verboseName,
    storage: {},
    dispatch: config.dispatch,
    subscribe: config.subscribe,
    storeFor: config.storeFor,
    allowSingletons: config.allowSingletons,
    isAllowed: function(obj) {
      // YOUR CODE HERE
    },
    exec: function(label, menuItem) {
      // YOUR CODE HERE
    }
  }
};
```

As you can see the plugin file contains just one object called *plugin*, which is a function. Notice the comment block before the object definition, this is **mandatory** to have, since this helps Textinator to register the information about your plugin in the database. The plugin object itself implements a revealing module pattern, i.e. being a function that returns a JS object with a fixed number of properties, that can be considered public for this plugin. You can define any helper functions you want, replacing the comment `// YOUR OWN PRIVATE FUNCTIONS`, since those won't be visible outside of the plugin.

The `config` object contains the default values for the configuration of each plugin. The fields shown in the example file **MUST** be present, but you can include your own custom configuration fields to be used within `exec` function. The mandatory fields are:

- `name` defines the name of the field associated with the plugin in the exported data and in the database
- `verboseName` defines the name to be displayed in the context menu
- `storeFor` defines the scope for the plugin storage, i.e. whether values in the storage should be per label or per relation
- `subscribe` defines a list of JS events triggering when the plugin should be re-rendered
- `dispatch` defines a map of events triggering other events, note that for each `key: value`, `key` should not be registered in `subscribe`, but `value` typically should, since typically it is desirable to trigger plugin re-rendering. For instance, if the relation associated with the plugin has changed, Textinator triggers

`labeler_relationschange` event, which could be captured by your plugin and trigger another event associated with your plugin, e.g. `my_custom_event`, in which case you would need to specify `"dispatch": { "labeler_relationschange" : "my_custom_event" }`.

- `allowSingletons` will take effect only if `storeFor` equals `"relation"` and signifies then whether the plugin should get initialized for the markables that are not in anyr elations, a.k.a. singletons.

After defining the plugin, you will need to **restart** Textinator for your plugin to get registered. It will then be available for every *Marker* as a context menu item. Then for each *Label* of a *Marker* with a context menu items, all items will be initialized and renderd using the following procedure:

1. Check if a plugin is allowed for the given *Label* by calling `isAllowed(label)` for this plugin and the label in question.
2. **If allowed:**
    a) Run the plugin's `exec(label, menuItem)`, where `menuItem` is the context menu button for this plugin and `label`.
    b) For all events in the plugin's `subscribe`, register an event listener, triggering plugin's re-rendering (consisting of 1 and 2a).

CHAPTER

THREE

API

3.1 Models

class projects.models.**Batch**(**args*, ***kwargs*)

Each time an annotator submits any annotation(s), an annotation batch is created for this annotator and a unique UUID is assigned tot his batch.

All annotated *Markers* (instantiated as either *Inputs* or *Labels*) and *Relations* (instantiated as *LabelRelations*) are then binded to this batch.

Parameters

- **id** (*AutoField*) – Id
- **revision_of_id** (ForeignKey to *Batch*) – Revision of
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **uuid** (*UUIDField*) – Uuid
- **user_id** (ForeignKey to User) – User
- **is_flagged** (*BooleanField*) – Indicates whether the annotator has flagged the batch as having problems

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**CommonModel**(**args*, ***kwargs*)

Abstract model containing the fields for creation and update dates, as well as a stub for *to_json* method.

Parameters

- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled

save(**args*, ***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

class projects.models.**Context**(**args*, ***kwargs*)

An **instantiation** of a textual context that is currently annotated. This is stored specifically in Textinator to avoid the loss of annotations if something should happen to the original data sources.

We do specify which data source a context is from, so it could be deleted, should the need arise. However, it is not deleted automatically on deletion of the data source, again, to prevent the loss of annotations in case the data source deletion was accidental.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **datasource_id** (ForeignKey to *DataSource*) – Datasource
- **datapoint** (*CharField*) – As stored in the original dataset
- **content** (*TextField*) – Content

exception DoesNotExist

exception MultipleObjectsReturned

save(**args*, ***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

class projects.models.**DataAccessLog**(**args*, ***kwargs*)

Holds data access logs for each annotator per project. We keep track of:

- which datapoint and of which data source was accessed and when
- whether at least one annotation was submitted for that datapoint
- whether the datapoint was skipped without annotation (i.e., a new text was requested)
- whether the user flagged anything related to this datapoint (e.g., problems with text)

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **user_id** (ForeignKey to User) – User
- **project_id** (ForeignKey to *Project*) – Project
- **datasource_id** (ForeignKey to *DataSource*) – Datasource
- **datapoint** (*IntegerField*) – As ordered in the original dataset
- **flags** (*JSONField*) – Additional information provided by the annotator
- **is_submitted** (*BooleanField*) – Indicates whether the datapoint was successfully submitted by an annotator
- **is_skipped** (*BooleanField*) – Indicates whether the datapoint was skipped by an annotator

- **is_delayed** (*BooleanField*) – Indicates whether the datapoint for skipped and saved for later by an annotator

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**DataSource**(**args*, ***kwargs*)

Holds a **definition** of a datasource. Currently we support 4 *source_types*:

- plain text – input text directly in the admin interface (mostly for testing)
- plain text files – a bunch of files hosted on the same server as Textinator
- JSON files – a bunchf of JSON files hosted on the same server as Textinator
- Texts API – a REST API that will be used for getting each datapoint (the endpoint should be specified)

Texts API specification is available in the *example_texts_api* folder of the GitHub repository.

DataSource specifies 3 different formattings:

- plain text (without line breaks or tabs preserved)
- formatted text (with line breaks and tabs preserved)
- markdown

By default each DataSource is private, unless *is_public* switch is on.

owner of the DataSource is set automatically and is nullable. The reason behind allowing NULL values is that the data might be owned by the institution, not by the user and might also have projects connected to it. If people want their datasource deleted together with their user account, they need to request a manual deletion.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **name** (*CharField*) – Dataset name
- **source_type** (*CharField*) – Dataset type
- **spec** (*TextField*) – in a JSON format
- **language** (*CharField*) – Language of this data source
- **formatting** (*CharField*) – text formating of the data source
- **is_public** (*BooleanField*) – Whether to make data source available to other Textinator users
- **owner_id** (ForeignKey to User) – Owner
- **post_processing_methods** (*ManyToManyField*) – Post-processing methods

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**Input**(**args*, ***kwargs*)

Holds an **instantiation** of a *Marker* that does not require specifying the start-end boundaries of the text. This mostly concerns the cases when a user provides an input via HTML <*input*> tag.

Specifically this concerns *MarkerVariants* with the following annotation types:

- short (long) free-text input
- integer
- floating-point number
- range

group_order field specifies the order of the marker group that this *MarkerVariant* belongs to in the *MarkerUnit* (if such unit was defined) at submission time. To exemplify, let's say there is a definition of a *MarkerUnit* that consists of 3 to 5 marker groups, each of which has:

- *Question* marker (Q)
- *Correct answer* marker (C)

In the UI, the annotator will then see the following:

[(Q, C)+, (Q, C)+, (Q, C)+, (Q, C), (Q, C)]

The groups with a (+) are mandatory for submission (since a unit should hold at least 3 groups by a specification). *group_order* is meaningfull only if the annotator is allowed to rank the groups within a unit. If so, then *group_order* specifies the order of each (Q, C) group after ranking at submission time.

Parameters

- **id** (*AutoField*) – Id
- **revision_of_id** (ForeignKey to *Input*) – Revision of
- **group_order** (*PositiveIntegerField*) – At the submission time
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **content** (*TextField*) – Content
- **marker_id** (ForeignKey to *MarkerVariant*) – Marker
- **context_id** (ForeignKey to *Context*) – Context
- **batch_id** (ForeignKey to *Batch*) – Batch

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**Label**(**args*, ***kwargs*)

Holds an **instantiation** of a *Marker* that requires specifying the start-end boundaries of the text or is **NOT** provided via HTML <*input*> tag.

Specifically this concerns *MarkerVariants* with the following annotation types:

- marker (text spans)
- marker (whole text)

extra holds extra information associated with the annotation at submission time. This extra information is typically via marker actions (i.e., right-clicking a marker).

The meaning of *group_order* is exactly the same as for *Input*.

Parameters

- **id** (*AutoField*) – Id
- **revision_of_id** (ForeignKey to *Label*) – Revision of
- **group_order** (*PositiveIntegerField*) – At the submission time
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **start** (*PositiveIntegerField*) – Character-wise start position in the text
- **end** (*PositiveIntegerField*) – Character-wise end position in the text
- **marker_id** (ForeignKey to *MarkerVariant*) – Marker
- **extra** (*JSONField*) – in a JSON format
- **context_id** (ForeignKey to *Context*) – Context
- **undone** (*BooleanField*) – Indicates whether the annotator used 'Undo' button
- **batch_id** (ForeignKey to *Batch*) – Batch

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**LabelRelation**(**args*, ***kwargs*)

Holds an **instantiation** of a *Relation*.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **rule_id** (ForeignKey to *RelationVariant*) – Rule
- **first_label_id** (ForeignKey to *Label*) – First label
- **second_label_id** (ForeignKey to *Label*) – Second label
- **undone** (*BooleanField*) – Indicates whether the annotator used 'Undo' button
- **batch_id** (ForeignKey to *Batch*) – Batch
- **cluster** (*PositiveIntegerField*) – At the submission time
- **extra** (*JSONField*) – in a JSON format

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**Marker**(**args*, ***kwargs*)

Holds the **definition** for each unit of annotation in Textinator, called *Marker*. We create each *Marker* only when creating a new project and can re-use *Markers* between the projects (all *Markers* are available to all users).

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled

- **dt_updated** (*DateTimeField*) – Autofilled
- **name** (*CharField*) – The display name of the marker (max 50 characters)
- **name_en** (*TranslationCharField*) – The display name of the marker (max 50 characters)
- **name_nl** (*TranslationCharField*) – The display name of the marker (max 50 characters)
- **name_ru** (*TranslationCharField*) – The display name of the marker (max 50 characters)
- **name_es** (*TranslationCharField*) – The display name of the marker (max 50 characters)
- **name_sv** (*TranslationCharField*) – The display name of the marker (max 50 characters)
- **name_uk** (*TranslationCharField*) – The display name of the marker (max 50 characters)
- **code** (*CharField*) – Marker's nickname used internally
- **color** (*ColorField*) – Color for the annotated text span
- **shortcut** (*CharField*) – Keyboard shortcut for annotating a piece of text with this marker
- **suggestion_endpoint** (*URLField*) – Endpoint for the Suggestions API

exception DoesNotExist

exception MultipleObjectsReturned

clean_fields(*exclude=None*)

Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

get_deferred_fields()

Return a set containing names of deferred fields for this instance.

is_part_of_relation()

Check whether a given marker is part of definition for any *Relation*

refresh_from_db(*using=None*, *fields=None*)

Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save(**args*, ***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

class projects.models.**MarkerAction**(**args*, ***kwargs*)

Specifies an action that shows up after right-clicking the marker. Each action is implemented as a JavaScript plugin that should exist in *static/scripts/labeler_plugins* folder along with a specification of how to implement your own plugins if necessary.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled

- **name** (*CharField*) – Name
- **description** (*TextField*) – Description
- **file** (*CharField*) – a name of the JS plugin file in the */static/scripts/labeler_plugins* directory
- **admin_filter** (*CharField*) – Specifies the filter type in the data explorer interface (one of 'boolean', 'range'). If empty, then this action will be excluded from data explorer.

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**MarkerContextMenuItem**(**args*, ***kwargs*)

An M2M model binding *MarkerAction* and *MarkerVariant* and holding additional information.

- ***config* holds a JSON configuration, specified in the JS plugin file for this action.**
  By storing it here, we allow each config to be customized specifically for each *MarkerVariant*-*MarkerAction* binding.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **action_id** (ForeignKey to *MarkerAction*) – Marker action
- **marker_id** (ForeignKey to *MarkerVariant*) – Marker
- **verbose** (*CharField*) – Verbose name
- **verbose_admin** (*CharField*) – Verbose name in data explorer
- **field** (*CharField*) – If applicable
- **config** (*JSONField*) – Json configuration

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**MarkerPair**(**args*, ***kwargs*)

Holds a pair of markers and is used to define (constrain) the relations. For example, if the relation *Refers to* holds between *Antecedent* and *Reference*, then a marker pair of *Antecedent* and *Reference* will be created and assigned to that relation definition.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **first_id** (ForeignKey to *Marker*) – First
- **second_id** (ForeignKey to *Marker*) – Second

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.MarkerRestriction(*args, **kwargs)

Holds a **definition** of a count restriction that is placed on a *MarkerVariant*

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **variant_id** (ForeignKey to *MarkerVariant*) – Marker variant
- **kind** (*CharField*) – Restriction kind
- **value** (*PositiveIntegerField*) – e.g., if restriction kind is '<=' and value is '3', this creates a restriction '<= 3'
- **is_ignorable** (*BooleanField*) – whether the restriction can be ignored at the discretion of the annotator

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.MarkerUnit(*args, **kwargs)

Some annotation tasks might benefit from annotating groups of markers as one unit. This model stores the **definitions** of such units (shared across all users).

The unit configuration has two dimensions: - marker group, which is defined by a one-to-many relationship with MarkerVariant model - unit height, which provides minimum and maximum number of marker groups in this unit

minimum_required attribute defines a lower bound for a unit height, whereas *size* defines an upper bound.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **size** (*PositiveIntegerField*) – Default (and maximal) number of marker groups in the unit
- **minimum_required** (*PositiveIntegerField*) – Minimum required number of marker groups in the unit (can't be more than *size*)
- **is_rankable** (*BooleanField*) – Whether annotators should be allowed to rank marker groups in the unit

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.MarkerVariant(*args, **kwargs)

Holds a **project-specific definition** for a previously defined *Marker*. This model allows the project manager to customize a previously defined marker by:

- specifying different color or hotkey
- changing the annotation type of the marker (defined in settings.ANNOTATION_TYPES)

- assigning a marker to a unit

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **project_id** (ForeignKey to *Project*) – Project
- **marker_id** (ForeignKey to *Marker*) – Marker template
- **nrange_id** (ForeignKey to *Range*) – Applicable only if the annotation types are 'integer', 'floating-point number' or 'range'. If the annotation type is 'range' and no numeric range is specified, the input will range from 0 to 100 by default. The values will remain unrestricted for 'integer' or 'floating-point number' types.
- **unit_id** (ForeignKey to *MarkerUnit*) – Marker unit
- **order_in_unit** (*PositiveIntegerField*) – Order of this marker in the unit
- **are_suggestions_enabled** (*BooleanField*) – Indicates whether Suggestions API should be enabled for this marker (if endpoint is specified)
- **custom_suggestion_endpoint** (*URLField*) – Custom endpoint for the Suggestions API (by default the one from the marker template is used). Activates only if suggestions are enabled.
- **custom_color** (*ColorField*) – Customized color for the annotated text span (color of the marker template by default)
- **custom_shortcut** (*CharField*) – Keyboard shortcut for annotating a piece of text with this marker (shortcut of the marker template by default
- **anno_type** (*CharField*) – The type of annotations made using this marker
- **display_type** (*CharField*) – Only applicable if annotation type is *Marker (text spans)*
- **display_tab** (*CharField*) – A name of the tab to which this marker belongs (leave empty if you don't want to have any tabs)
- **export_name** (*CharField*) – The name of the field in the exported JSON file (English name by default)
- **choices** (*JSONField*) – Valid only if annotation type is *radio buttons* or *checkboxes*. Up to 2 levels of nesting allowed (more than 2 is impractical for the annotator)
- **actions** (*ManyToManyField*) – Actions associated with this marker

exception DoesNotExist

exception MultipleObjectsReturned

get_count_restrictions(*stringify=True*)

Get the restrictions (if any) on the number of markers per submitted instance

Args:
stringify (bool, optional): Whether to return the restrictions in a string format

Returns:
(str or list): Restrictions on the number of markers per submitted instance

max()

Returns:
int: The maximal number of markers of this kind per submitted instance

min()

Returns:
int: The minimal number of markers of this kind per submitted instance

save(**args*, ***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

class projects.models.**PostProcessingMethod**(**args*, ***kwargs*)

Holds function names to the post-processing methods that can be applied directly to textual data. Eligible methods are currently being pulled from *projects/helpers.py*, which is not a very elegant solution.

NOTE: this functionality is currently inactive and is a candidate for removal/overhaul in future releases.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **name** (*CharField*) – Verbose name
- **helper** (*CharField*) – Name as specified in *projects/helpers.py*

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**PreMarker**(**args*, ***kwargs*)

Static pre-markers to be automatically created before the annotation of a specific text begun.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **project_id** (ForeignKey to *Project*) – Project
- **marker_id** (ForeignKey to *MarkerVariant*) – Marker
- **tokens** (*TextField*) – Comma-separated tokens that should be highlighted with a marker

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**Project**(**args*, ***kwargs*)

Holds a **definition** of each Textinator project.

Parameters

- **id** (*AutoField*) – Id

- **dt_created** (*DateTimeField*) – Autofilled
- **title** (*CharField*) – Title
- **short_description** (*TextField*) – Will be displayed on the project card
- **institution** (*CharField*) – Institution responsible for the project
- **supported_by** (*CharField*) – The name of the organization supporting the project financially (if applicable)
- **guidelines** (*HTMLField*) – Guidelines for the annotation task
- **reminders** (*HTMLField*) – Reminders for essential parts of guidelines (keep them short and on point)
- **temporary_message** (*HTMLField*) – A temporary message for urgent communication with annotators (e.g., about maintenance work)
- **data_order** (*CharField*) – In what order should the data be presented?
- **disjoint_annotation** (*BooleanField*) – Should each annotator work with their own part of data?
- **show_datasource_identifiers** (*BooleanField*) – Should data source identifiers be shown?
- **task_type** (*CharField*) – Type of the annotation task
- **dt_publish** (*DateTimeField*) – Publishing date
- **dt_finish** (*DateTimeField*) – Expiration date
- **dt_updated** (*DateTimeField*) – Updated at
- **author_id** (ForeignKey to `User`) – Author
- **is_open** (*BooleanField*) – Should the project be public?
- **is_peer_reviewed** (*BooleanField*) – Should the annotations be peer reviewed?
- **allow_selecting_labels** (*BooleanField*) – Should selecting the labels be allowed?
- **disable_submitted_labels** (*BooleanField*) – Should submitted labels be disabled?
- **auto_text_switch** (*BooleanField*) – Automatic mode involves showing a new text on page refresh if at least one annotation was made on it (default). If this setting is turned off, the annotator only gets a new text when they choose to click on the 'Get new text' button.
- **max_markers_per_input** (*PositiveIntegerField*) – Maximal number of markers per input
- **has_intro_tour** (*BooleanField*) – WARNING: Intro tours are currently in beta
- **language** (*CharField*) – Language of this project
- **thumbnail** (*ImageField*) – A thumbnail of your project (ignored if not provided)
- **video_summary** (*FileBrowseField*) – Video introducing people to the annotation task at hand (if applicable)
- **video_remote** (*URLField*) – A URL for video summary to be embedded (e.g. from YouTube)
- **modal_configs** (*JSONField*) – JSON configuration for the modal windows in the project. Currently available keys for modals are: 'flagged'

- **editing_title_regex** (*TextField*) – The regular expression to be used for searching the annotated texts and using the first found result as a title of the batches to be edited
- **allow_editing** (*BooleanField*) – Should editing of own annotations be allowed?
- **editing_as_revision** (*BooleanField*) – By default editing happens directly in the annotated objects. If this setting is turned on, the original objects will remain intact and separate reivison objects will be created
- **allow_reviewing** (*BooleanField*) – Should peer reviewing be enabled?
- **collaborators** (*ManyToManyField*) – Collaborators
- **participants** (*ManyToManyField*) – Participants
- **markers** (*ManyToManyField*) – Project-specific markers
- **relations** (*ManyToManyField*) – Project-specific relations
- **datasources** (*ManyToManyField*) – All data sources must be of the same language as the project

exception DoesNotExist

exception MultipleObjectsReturned

data(*user*, *force_switch=False*)

Main method for getting data from the data sources and keeping track of who should annotate what.

The method proceeds as follows:

- If the annotator has previously requested a datapoint, but neither did any annotation, nor requested a new one, show the very same datapoint again. Otherwise, proceed.
- If the annotator did some annotation and the auto text switch is off, show the very same text again. Otherwise, proceed
- If sampling with replacement is turned off, exclude the previously annotated data.
- If disjoint annotation is turned on, then all previously annotated datapoints (by anyone) should be excluded, so that the sets of annotations for each annotator are disjoint.
- If disjoint annotation is off, then exclude only data previously annotated by the current user.
- Instantiate all datasources associated with this project
- Choose an unannotated datapoint uniformly at random across all datasources and return it.

Args:
user (User): Current user

Returns:
DatapointInfo: The instance holding the information about the datapoint to be annotated

free_markers(*intelligent_groups=False*)

Returns:
QuerySet: The set of marker variants that do NOT belong to marker unit (order by annotation type)

is_ordered(*parallel='*'*)

Check if the data order is static

Args:

parallel (str or bool, optional): indicates whether to check if the dataset order is parallel (True) or sequential (False)
or that the exact order is not important is not important ('*', by default)

Returns:
bool: Indicator of whether the data is to be presented in a specified (optionally, with sequential or parallel dataset order)

`is_sampled`(*replacement='*'*)

Check if the data order is randomly sampled

Args:

replacement (str or bool, optional): indicates whether to the check if sampling with replacement (True) or not (False)
or that the kind is not important ('*', by default)

Returns:
bool: Indicator of whether the data is to be sampled (optionally, with or without replacement)

`property marker_groups`

Returns:
QuerySet: The set of marker variants that belong to marker unit (order by annotation type)

`class` projects.models.**Range**(**args*, ***kwargs*)

Holds a definition for a numeric range, stores min, max and step (similar to Python's range). Currently this is used for specifying the possible numeric ranges for marker variants of types 'integer', 'floating-point value' and 'range'.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **min_value** (*FloatField*) – Minimal value
- **max_value** (*FloatField*) – Maximal value
- **step** (*FloatField*) – Step

`exception DoesNotExist`

`exception MultipleObjectsReturned`

`clean`()

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

`class` projects.models.**Relation**(**args*, ***kwargs*)

Holds a **definition** of a relation in Textinator. We create each *Relation* only when creating a new project and can re-use *Relations* between the projects (all *Relations* are available to all users).

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled

- **name** (*CharField*) – Name
- **name_en** (*TranslationCharField*) – Name [en]
- **name_nl** (*TranslationCharField*) – Name [nl]
- **name_ru** (*TranslationCharField*) – Name [ru]
- **name_es** (*TranslationCharField*) – Name [es]
- **name_sv** (*TranslationCharField*) – Name [sv]
- **name_uk** (*TranslationCharField*) – Name [uk]
- **direction** (*CharField*) – Direction
- **shortcut** (*CharField*) – Keyboard shortcut for marking a piece of text with this relation
- **representation** (*CharField*) – How should the relation be visualized?
- **pairs** (*ManyToManyField*) – Marker pairs

exception DoesNotExist

exception MultipleObjectsReturned

property between

Returns:
str: The string representation of the pairs of markers for which the relation can be annotated.

clean_fields(*exclude=None*)

Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

get_deferred_fields()

Return a set containing names of deferred fields for this instance.

refresh_from_db(*using=None*, *fields=None*)

Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

class projects.models.**RelationVariant**(**args*, ***kwargs*)

Holds a **project-specific definition** for a previously defined *Relation*. This model allows the project manager to customize a previously defined relation by:

- specifying different hotkey
- specifying a different visual representation (i.e., graph or list)

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **project_id** (ForeignKey to *Project*) – Project

- **relation_id** (ForeignKey to *Relation*) – Relation template
- **custom_shortcut** (*CharField*) – Keyboard shortcut for marking a piece of text with this relation (shortcut of the relation template by default)
- **custom_representation** (*CharField*) – How should the relation be visualized? (representation of the relation template by default)

exception DoesNotExist

exception MultipleObjectsReturned

property between

Returns:
str: The string representation of the pairs of markers for which the relation can be annotated.

save(**args*, ***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

class projects.models.**TaskTypeSpecification**(**args*, ***kwargs*)

Holds a specification for an annotation task type and is used when a project of the pre-defined annotation type is instantiated. The specification describes markers and relations that are to be used for this annotation task.

Default specifications are created during the first startup of the server and can be found in the *task_defaults.json*

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **task_type** (*CharField*) – Type of the annotation task
- **config** (*JSONField*) – Json configuration

exception DoesNotExist

exception MultipleObjectsReturned

class projects.models.**UserProfile**(**args*, ***kwargs*)

An M2M model binding *User* and *Project* and holding additional information

NOTE: the additional information is currently not in use and is a candidate for removal/overhaul in future releases.

Parameters

- **id** (*AutoField*) – Id
- **dt_created** (*DateTimeField*) – Autofilled
- **dt_updated** (*DateTimeField*) – Autofilled
- **user_id** (ForeignKey to User) – User
- **project_id** (ForeignKey to *Project*) – Project
- **allowed_reviewing** (*BooleanField*) – Whether the annotator is allowed to review for this project

exception DoesNotExist

exception MultipleObjectsReturned

PYTHON MODULE INDEX

INDEX

P

R

S

T

U