[API Access 1](#_Toc141865310)

[Mastodon API 1](#_Toc141865311)

[Toot-Object 1](#_Toc141865312)

[Load-Function 1](#_Toc141865313)

[Triggers 1](#_Toc141865314)

[Trigger-Parent 2](#_Toc141865315)

[Media-Trigger 2](#_Toc141865316)

[Image-Trigger 2](#_Toc141865317)

[GIF-Trigger 2](#_Toc141865318)

[Video-Trigger 2](#_Toc141865319)

[Audio-Trigger 3](#_Toc141865320)

[Language-Trigger 3](#_Toc141865321)

[Poll-Trigger 3](#_Toc141865322)

[Mentions-Trigger 3](#_Toc141865323)

[Time-Trigger 3](#_Toc141865324)

[Before-Trigger 4](#_Toc141865325)

[After-Trigger 4](#_Toc141865326)

[Phrase-Trigger 4](#_Toc141865327)

[Composite Triggers 4](#_Toc141865328)

[And-Trigger 4](#_Toc141865329)

[Or-Trigger 5](#_Toc141865330)

[Not-Trigger 5](#_Toc141865331)

[Filter-Toots 5](#_Toc141865332)

# API Access

An API access refers to the ability to interact with and make use of an **Application Programming Interface** (API). An API is a set of rules and protocols that allows different software applications to communicate and interact with each other. It defines the methods and data formats that applications can use to request and exchange information.

APIs enable developers to access the functionality of a specific software, service, or platform, such as **retrieving data**, performing operations, or integrating with other systems. For example, social media platforms like Facebook, Twitter or [**Mastodon**](https://mastodon.social/home)provide APIs that allow developers to access user data, post updates, or retrieve posts.

API access is typically provided through a combination of an API key or token and specific endpoints or URLs. The API key acts as a unique identifier that grants permission to access the API, while endpoints are URLs that specify the different actions or operations available through the API.

APIs are crucial for enabling the integration of various services and systems, allowing developers to build more powerful and complex applications by leveraging the functionality and data provided by other services.

## Mastodon API

## Mastodon is a social networking platform that is part of the decentralized social media movement.

It was created by Eugen Rochko and launched in 2016 as an alternative to traditional, centralized social media platforms like Twitter. Mastodon is open-source software, which means its source code is freely available for anyone to use, modify, and distribute.

Unlike traditional social media platforms, Mastodon operates on a federated model. Instead of having a single central server that hosts all user accounts and content, Mastodon instances are independently operated servers, known as "instances", which are connected to form a larger network. Each instance is its own community with its own rules and moderation policies, and users can choose which instance they want to join based on their interests or preferences.

Users on Mastodon can post messages, called "toots," which can include text, images, videos, and links. They can follow other users from their own instance or other instances, and interactions are possible across instances within the Mastodon network.

To get access to the Mastodon API you should follow the steps below:

1. Create a Mastodon account at the 'Mastodon.social’ server.
2. Generate an access token, that will allow your code to interact with the Mastodon API on your behalf.  
   (Log-in into your account 🡪 Go to your account settings 🡪 Look for “Development” or “API” in settings 🡪 Create a new application by providing a name, website and brief description)
3. Use the API Library [Mastodon.py](https://mastodonpy.readthedocs.io/en/stable/) to make calls from yur application to the API.
4. Authenticate you application using “Client ID”, “Client Secret” and “Access token”, as well as the server (“api\_base\_url”) your profile is hosted at.

## Toot-Object

## A toot in Mastodon is like a tweet in Twitter. In this problem set you are going to filter those toots, to which you got access to using the API.

## We want to store any information about an object that we can then pass around in the rest of our program. Your task, in this problem, is to write a class, toot, starting with a constructor that takes (content, account, user\_id, hashtags, bookmark, no\_replies, url, toot\_id, count\_replies, pubdate, mentions, media, language, poll) as arguments and stores them appropriately. Remember how to use the init method to create your constructor.

## Load-Function

The Mastodon.py documentation includes a function, “timeline\_hashtag” that takes a hashtag, and returns a JSON dictionary with various information about all toots which have that particular hashtag. This information should be stored in the object “toot” described above.

It is you turn now to write a function called “load” which takes a hashtag, loads all toots that have the hashtag into objects into a list and returns this list.   
When writing this function, make sure to save the toot context as a string and not as html script. To do so you can use the ”get\_text\_content()” function. We already implemented this function in the file “content\_Processor\_ps7.py”, so you don’t have to worry about that.

# Triggers

Triggers are rules that determine whether or not a social network post on Mastodon meets the triggers criteria. They can be based on the content of the posts, such as specific phrases or media types, or on the time when the posts were published. Triggers can be used individually or combined using composite triggers to create more sophisticated filtering rules. It is your task to create these filters which are using the posts (e.g. Toots) you are loading in the Load-Function. After creating all trigger classes you then need to write a function that goes through a sample of triggers and posts and checks whether or not a specific post meets all given trigger criteria.

## Trigger-Parent

This is an abstract class representing the parent class for all triggers. It contains the **evaluate** method, which is used to evaluate whether an alert should be generated for a given social network post on Mastodon. All trigger classes inherit from this class. Because -as you know- if an inheriting subclass does not overwrite the methods of the super class, it uses the super class method instead of an own. Therefore you need to overwrite the **evaluate** method in the subclasses as otherwise the **evaluate** method from the super class Trigger will get called and return a **NotImplementedError**.

## Media-Trigger

* The **MediaTrigger** is a trigger that fires when a toot contains any media attachments such as images, videos, gifs, or audio files.
* To implement this trigger, you need to access the **media** attribute of the **toot** object, which represents the media attachments.
* Check if the **media** attribute is not empty. If it's not empty, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

### Image-Trigger

* The **Image-Trigger** is a trigger that fires when a social network post contains one or more image attachments.
* To implement this trigger, you need to access the **media** attribute of the **post** object, which represents the media attachments.
* Check if the **media** attribute is not empty and contains at least one image attachment. If it does, return **True**, indicating that the trigger should fire. Otherwise, return **False**.
* Hint: For this task it is very helpful to check the mastodon documentation about the returning dictionaries, especially [media dicts](https://mastodonpy.readthedocs.io/en/stable/02_return_values.html#media-dicts).

### GIF-Trigger

* The **GIF-Trigger** is a trigger that fires when a social network post contains one or more GIF (Graphics Interchange Format) attachments.
* To implement this trigger, you need to access the **media** attribute of the **post** object, which represents the media attachments.
* Check if the **media** attribute is not empty and contains at least one GIF attachment. If it does, return **True**, indicating that the trigger should fire. Otherwise, return **False**.
* Hint: For this task it is very helpful to check the mastodon documentation about the returning dictionaries, especially [media dicts](https://mastodonpy.readthedocs.io/en/stable/02_return_values.html#media-dicts).

### Video-Trigger

* The **Video-Trigger** is a trigger that fires when a social network post contains one or more video attachments.
* To implement this trigger, you need to access the **media** attribute of the **post** object, which represents the media attachments.
* Check if the **media** attribute is not empty and contains at least one video attachment. If it does, return **True**, indicating that the trigger should fire. Otherwise, return **False**.
* Hint: For this task it is very helpful to check the mastodon documentation about the returning dictionaries, especially [media dicts](https://mastodonpy.readthedocs.io/en/stable/02_return_values.html#media-dicts).

### Audio-Trigger

* The **Audio-Trigger** is a trigger that fires when a social network post contains one or more audio attachments.
* To implement this trigger, you need to access the **media** attribute of the **post** object, which represents the media attachments.
* Check if the **media** attribute is not empty and contains at least one audio attachment. If it does, return **True**, indicating that the trigger should fire. Otherwise, return **False**.
* Hint: For this task it is very helpful to check the mastodon documentation about the returning dictionaries, especially [media dicts](https://mastodonpy.readthedocs.io/en/stable/02_return_values.html#media-dicts).

## Language-Trigger

* The **LanguageTrigger** is a trigger that fires when a toot is written in a specific language.
* For this task it is important to check the specific notation Mastodon uses. In the documentation there is more information about what language codes they use and a (very short) explanation on how it looks like.
* To implement this trigger, you need to pass the desired language as an argument when creating an instance of **LanguageTrigger** to save the passed in argument as an attribute of the **LanguageTrigger.**
* In the **evaluate** method, access the **language** attribute of a **toot** object to check if it matches the specified language, e.g., the attribute of the trigger.
* If the **language** matches the specified language, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

## Poll-Trigger

* The **PollTrigger** is a trigger that fires when a toot contains a poll, quite similar to the media trigger.
* To implement this trigger, you need to access the **poll** attribute of the **toot** object, which represents the poll details.
* Check if the **poll** attribute is not empty. If it's not empty, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

## Mentions-Trigger

* The **MentionsTrigger** is a trigger that fires when a toot mentions other users, again quite similar to the poll and the media trigger.
* To implement this trigger, you need to access the **mentions** attribute of the **toot** object, which represents the mentioned users.
* Check if the **mentions** attribute is not empty. If it's not empty, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

## Time-Trigger

* The **TimeTrigger** is a trigger that fires based on the publication time of a toot.
* To implement this trigger, you need to pass the trigger time as a string in the format of "YYYY-MM-DD HH:MM:SS" (in EST timezone) when creating an instance of **TimeTrigger**.
* Convert the trigger time from a string to a **datetime** object and store it as an attribute (**ptime**) of the **TimeTrigger**.
* Note that **TimeTrigger** should be an abstract class, and you won't directly instantiate it.
* Therefore no **evaluate** method needs to be constructed.

### Before-Trigger

* The **BeforeTrigger** is a trigger that fires when a post was published strictly before the trigger time.
* Implement **BeforeTrigger** as a subclass of **TimeTrigger**.
* Recall that being a subclass means it inherits all methods of the super class. Therefore the **\_\_init\_\_** function of **TimeTrigger** will also be inherited and you do not need to construct a new one.
* In the **evaluate** method, access the **pubdate** attribute of the **post** object to get the publication time of the post.
* Convert the publication time to the EST timezone and compare it with the trigger time (**ptime**). If the post was published before the trigger time, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

### After-Trigger

* The **AfterTrigger** is a trigger that fires when a post was published strictly after the trigger time.
* Implement **AfterTrigger** as a subclass of **TimeTrigger**.
* As this trigger is quite the same to the **BeforeTrigger**, recall what you took into consideration then.
* In the **evaluate** method, access the **pubdate** attribute of the **post** object to get the publication time of the post.
* Convert the publication time to the EST timezone and compare it with the trigger time (**ptime**). If the post was published after the trigger time, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

## Phrase-Trigger

* The **PhraseTrigger** is a trigger that fires when a toot contains a specific phrase in its content.
* To implement this trigger, you need to pass the desired phrase as an argument when creating an instance of **PhraseTrigger**.
* In the **evaluate** method, access the **content** attribute of the **toot** object to get the text content of the toot.
* Convert both the **content** and the specified **phrase** to lowercase (to make the comparison case-insensitive).
* After that replace all string punctuation of the content with an empty string, e.g., delete all punctuations.
* Check if the **phrase** is present in the **content**. If it is, return **True**, indicating that the trigger should fire. Otherwise, return **False**.

## Composite Triggers

### And-Trigger

* The **AndTrigger** is a trigger that fires on a toot only if both inputted triggers fire on that toot.
* To implement this trigger, you need to pass two triggers as arguments when creating an instance of **AndTrigger**.
* In the **evaluate** method, evaluate both triggers for the toot and return **True** only if both triggers return **True**.

### Or-Trigger

* The **OrTrigger** is a trigger that fires on a toot if either one (or both) of the inputted triggers fires on that toot.
* To implement this trigger, you need to pass two triggers as arguments when creating an instance of **OrTrigger**.
* In the **evaluate** method, evaluate both triggers for the toot and return **True** if at least one trigger returns **True**.

### Not-Trigger

* The **NotTrigger** is a trigger that inverts the output of another trigger.
* To implement this trigger, you need to pass another trigger as an argument when creating an instance of **NotTrigger**.
* In the **evaluate** method, evaluate the passed in trigger for the toot and return the inverse of the result.

# Filter-Toots

* The **filter\_toots** function takes a list of posts (toots) and a list of triggers as input.
* It returns a list of only the toots for which all trigger in the trigger list fire.
* Iterate over each toot in the toots list and then iterate over each trigger in the trigger list.
* Evaluate each trigger for the toot using the triggers **evaluate** method. If all triggers return **True** for the toot, add the toot to the **triggered\_toots** list.
* Finally, return the **triggered\_toots** list containing only the toots that satisfy all the specified triggers.