

Setting up the webapi

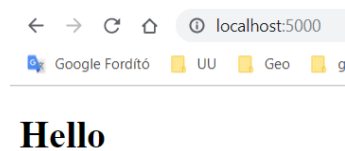
I used EME, which is a web framework built on top of Flask (and many more) to set up the webservice.

<https://github.com/oboforty/eme>

Setting up a website using eme is as simple as

```
$ python emetools.py forge:webapp --path="/uu/semester 3/applied cloud computing/labs/lab 3"
```

The built website is basically a Flask website with a more cleaner code architecture that is similar to other web frameworks. After setting up the whole thing with a stub for calling celery's result, here is how it looks like:



Eme's typical directory structure looks like this for a web app:

core/	-- business logic & data access code goes here
webapi/	-- website app files
controllers/	-- controller files, the main application-logic code: these are the ones that call core services usually
templates/	-- jinja2 html templates
public/	-- public assets, e.g. js, css
config.ini	-- configuration for server, routing & other eme related
website.py	-- main entry for website
entities.py	-- app-logic related entities, e.g. ApiResponse, a wrapper, that generates json outputs

Setting up Celery

In Openstack I started a new instance to act as a contextualization tool for the Celery service.

First of all, to set up python, celery, flask and eme manually:

```
sudo apt-get update
sudo apt-get install rabbitmq-server
sudo apt install python3-pip
pip3 install celery
pip3 install flask
pip3 install eme
sudo apt install python-celery-common
```

Py3 is already installed on Ubuntu18.

Then I wrote a testing script that would add two numbers. I created a RabbitMQ user acc/acc with vhost 'accvhost'.

```
sudo rabbitmqctl add_user acc acc
sudo rabbitmqctl add_vhost accvhost
sudo rabbitmqctl set_user_tags acc accvhost
sudo rabbitmqctl set_permissions -p accvhost acc ".*" ".*" ".*"
```

After writing a 'hello' test app in celery, I could confirm that it works using a worker and calling the testing script separately:

```
celery worker -A test --loglevel=info
python3 test.py
```

Pronoun count in Celery

I've written a second task, **count_pronouns** that will call a function that parses a file and counts the pronouns:

```
from celery import Celery

from core.celery import celeryconfig

from core.tweeter import count_tweets

app = Celery('tasks')
app.config_from_object(celeryconfig)

@app.task
def count_pronouns(files):
    pronoun_counts = {}

    # summarize for all files:
    for file in files:
        pronoun_counts.update(count_tweets(file))

    return pronoun_counts
```

And the actual pronoun count function:

```
import json
from collections import Counter, defaultdict

pronouns = ["han", "hon", "den", "det", "denna", "denne", "hen"]

def count_tweets(file):
    pronoun_counts = defaultdict(int)

    with open(file) as fh:
        for line in fh:
            line = line.strip()

            if not line:
                continue

            try:
                tweet = json.loads(line)

                if tweet['retweeted']:
                    # disregard retweets
                    continue

                # split tweet content into words & calculate word frequencies
                words = tweet["text"].lower().split()
                word_freq = Counter(words)

                for pronoun in pronouns:
                    pronoun_counts[pronoun] += word_freq.get(pronoun, 0)

                # uncomment for testing
                # break

            except Exception as e:
                # just in any case (EAFP):
                print('%s\t%s' % (str(e), 1))

                continue

    return dict(pronoun_counts)
```

Getting the website to work with Celery

After I wrote a python script that parses a tiny subset of the tweets, I added the code to the celery framework. I added the celery files (both worker tasks and the service that waits for the delay) to core. While the HomeController pronouns action (route: localhost:5000/pronouns) will call the celery service. This of course, assumes that a worker (or several workers) is running in the background.

Here's a list that explains the many files in core:

- **core/tweeter.py** contains the native python function that parses one json file and counts the Swedish pronouns.
- **core/celery/tasks.py** runs the above function as a celery task
- **core/celery/celeryconfig.py** is the celery config that is being used by **tasks.py**, as usual
- **core/celery/celery_service.py** is a wrapper for the EME core logic that calls the results of the word count service from celery. Also the list of files to parse is passed as an argument. This way I could forward a user-defined list of files for the celery service, FROM the web client (as HTTP GET params).


The WebApi only references and calls **celery_service.py**.

See attached files for the complete picture.

I initially ran the flask website without celery service to see if it works and the ports are all forwarded correctly:

```
self.server.addUrlRule({
    'GET /pronouns': 'home/count_pronouns',
})
...
def get_count_pronouns(self):
    pronouns = {}

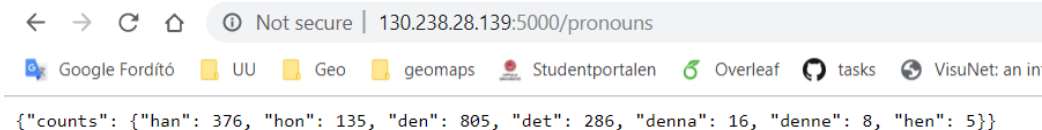
    return ApiResponse({
        "counts": pronouns
    })
```



After that I added the celery service with 1 file and restarted the celery worker:

```
def get_count_pronouns(self):
    pronouns = celery_service.count_pronouns([
        "/home/ubuntu/data/0ecd8e0-bc1a-4fb3-a015-9b8dc563a92f"
    ])

    return ApiResponse({
        "counts": pronouns
    })
```



Commands used (CWD: `/home/ubuntu/web`)

Celery:

```
celery worker -A run_celery --loglevel=info
```

Web server:

```
python3 run.py
```

Last steps

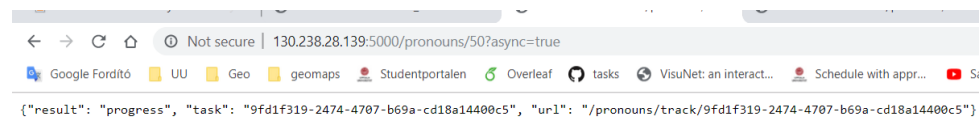
While these are nice results, the HTTP call is being blocked while the Celery worker finishes. This completely destroys the point of Celery, so I'm adding `async` calling and task ID checking.

I added an HTTP GET parameter to EME to provide the number of files to be parsed. `/pronouns/<nfiles>`

Finally, I added two more actions under `/pronouns/<nfiles>?async=true` and `/pronouns/track/<ID>` that will start the celery service, and only return the task's ID that can be tracked. These actions call functions that utilize `apply_async` instead of `delay` and can check the status of a task.

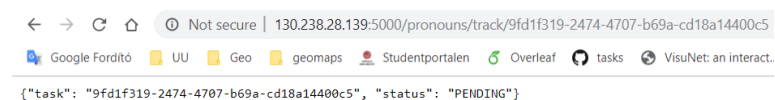
Final result:

`/pronouns/50?async=true:`



```
{\"result\": \"progress\", \"task\": \"9fd1f319-2474-4707-b69a-cd18a14400c5\", \"url\": \"/pronouns/track/9fd1f319-2474-4707-b69a-cd18a14400c5\"}
```

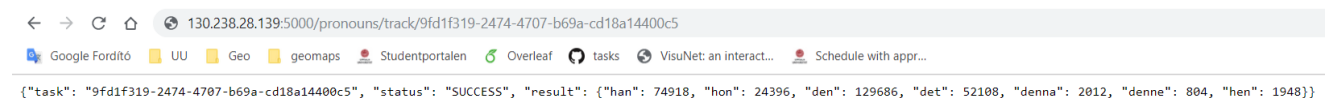
`/pronouns/track/9fd1f319-2474-4707-b69a-cd18a14400c5`



```
{\"task\": \"9fd1f319-2474-4707-b69a-cd18a14400c5\", \"status\": \"PENDING\"}
```

Reloading after a while:

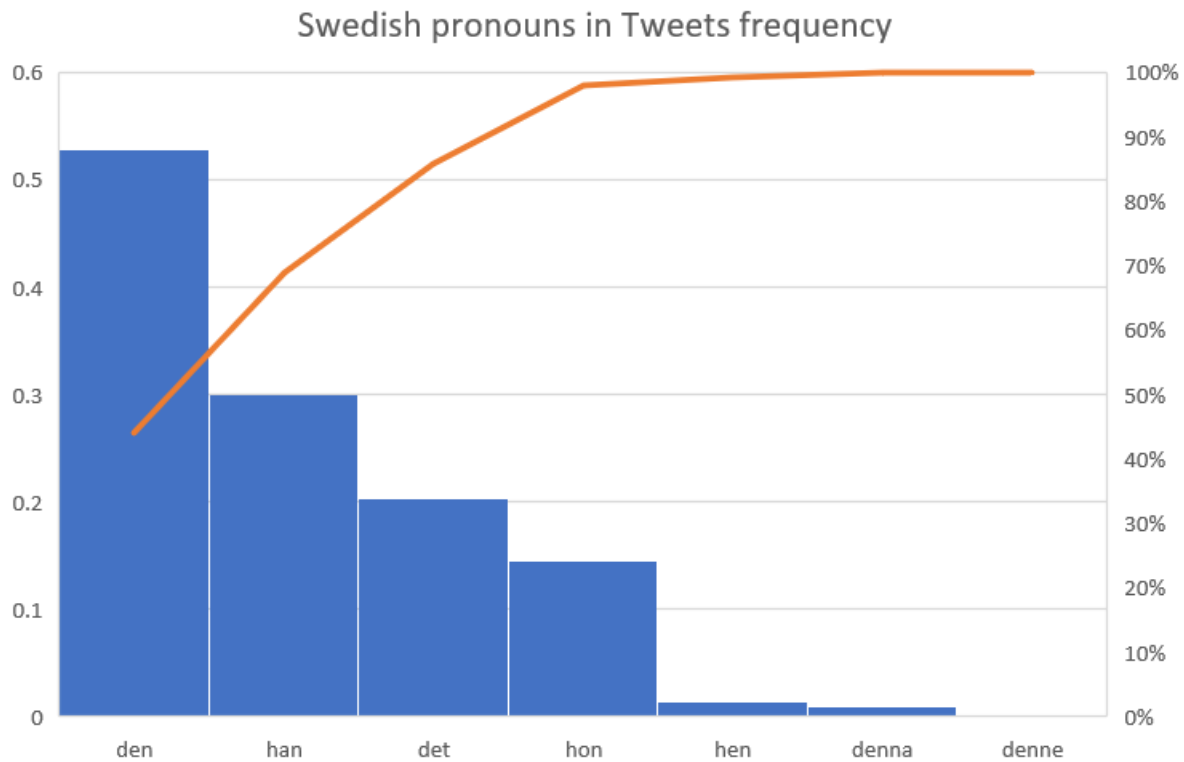
`/pronouns/track/9fd1f319-2474-4707-b69a-cd18a14400c5`



```
{\"task\": \"9fd1f319-2474-4707-b69a-cd18a14400c5\", \"status\": \"SUCCESS\", \"result\": {\"han\": 74918, \"hon\": 24396, \"den\": 129686, \"det\": 52108, \"denna\": 2012, \"denne\": 804, \"hen\": 1948}}
```

Visualization

I've added the result to excel and produced the following chart by dividing with the total number of tweets (that were processed by the files):



Dependencies

- Celery
- RabbitMQ

Pip:

- Celery
- Flask
- Eme

File dependencies:

- ~/data/* (tweet files, by celery)
- ~/cweb/webapi
- ~/cweb/core/celery/* (both by webapi and celery setup)

run.py and **run_celery.py** depend on these files, otherwise flask app is reflexive.