crawler_test web server API

Paulo V. Radtke – pvwradtke@gmail.com

Abstract

This document first discusses the design decisions behind this web crawler service, which takes a list of URLs and lists, for each page in the first two levels (parents and linked children), the URLs to the images used in those pages. We first talk about some choices made during its development. Then we discuss the implementation, detailing the program flow and the expected usage. Finally a list of shortcomings is discussed, as well a improvements that can be made in future versions.

Design Decisions

Some decisions were made in order to leverage previous experiences and speed up development, while meeting the constraint on using Docker. The first decision was to use Python, based on a previous experience writing a web crawler for text mining, using the BeautifulSoup library. This choice led to the next decision, to use the Flask framework to implement the web service, based on available documentation and to keep it simple (e.g. Flask doesn't even require a database).

To keep the Docker configuration straightforward, jobs were implemented using regular Python threads. While using multiprocessing could give better results and use multiple processor cores, we preferred to use a known approach to minimize the risk to implement the service in a short time. An alternative would have been to use a framework such as Redis. This would have given the ability to queue tasks, and purge old results for free after a certain timeout – which was manually implemented in the thread. However, this would increase complexity and not bring many benefits, so it was preferred to use only threads.

To share data between threads and the request functions, we used a dictionary that maintains the metadata for each job. Multiple jobs can run at the same time, and each job can use a different number of threads.

Obtaining the code and Usage

To clone he project, use the command below:

```
git clone git@github.com:pvwradtke/crawler_test.git
```

There is a README.md file in the repository with the usage details. Regarding the original request, it was not clear on how to pass along in the request the number of threads. Then it was decided that we should use as the input a fully formed JSON document with two values, **urls**, a list of URLs to crawl, and an optional **threads** integer value:

```
curl -X POST 'http://127.0.0.1:8080' -H "Content-Type: application/json" --data '{"urls": ["http://4chan.org/","https://golang.org/"], "threads": 4}'
```

This will start the crawling as a thread (the service returns the code 200 right away), using the provided URL list and 4 threads (if no value is provided, the default 1 is used). This outputs some info, including the job UUID:

```
{"job_id":"1ec8fff7-3a1f-41db-8ef7-7c4e6f452cc0","threads":"4","urls":"['https://golang.org/', 'http://4chan.org/']"}
```

It's possible to query the status for this job using the UUID above in the **status** page:

```
curl 'http://127.0.0.1:8080/status/1ec8fff7-3a1f-41db-8ef7-7c4e6f452cc0'
```

This will return a JSON with the number of URLs processed and the number of URLs waiting to be processed:

```
{"completed": 80, "inprogress": 32}
```

Once the **inprogress** value is equal to 0, the job has completed and we can retrieve the job's results using its UUID in the page **result:**

```
curl 'http://127.0.0.1:8080/result/1ec8fff7-3a1f-41db-8ef7-7c4e6f452cc0'
```

Tis returns a JSON document, where each node is an URL, with a list of the images used in the page (absolute links):


```
{
 "http://4chan.org/": [
  "http://i.4cdn.org/v/1611007947515s.jpg",
  "http://i.4cdn.org/vrpg/1610947985169s.jpg",
  ...
  "http://s.4cdn.org/image/fp/logo-transparent.png",
  "http://i.4cdn.org/sci/1610900463796s.jpg"
 ],
 "http://4chan.org/4channews.php": [
  "http://s.4cdn.org/image/news/Happybirthday_17th_th.jpg",
  "http://s.4cdn.org/image/fp/logo-transparent.png"
 ],
 "http://4chan.org/advertise": [
  "http://s.4cdn.org/image/fp/logo-transparent.png",
  ...
  "http://s.4cdn.org/image/advertise/top_ad_desktop.png",
  "http://s.4cdn.org/image/advertise/top_ad_mobile.png"
 ],
 "http://4chan.org/contact": [
  "http://s.4cdn.org/image/fp/logo-transparent.png"
 ],
```

```
...
```

Please note that if the job is still running a JSON document with the message asking to wait is returned.

Implementation

The Webservice

The **app/app.py** file in the repository defines the application. The first section defines the global variables, the **app** (our Flack application), **synclock**, a semaphore to synchronize threads and **jobs**, a dictionary used to store metadata for all jobs – we can both run and store results for multiple jobs.

There are some support functions, used in the code:

- **validate_url**: returns True or False if the URL is well formed (introduced because some links were to Javascript elements).
- **process_page**: the function that actually crawls an specific URL. It returns a tuple with a list of links and a list of images.

There are some functions to handle common HTTP server error codes, which are straightforward, but the remainder route functions are used in the normal application flow: launch job, check status and retrieve results. Let's look into them in the order they'd be called, to then understand the functions.

Launching a Job

Launching a job through the index page will call the **index** function. It first retrieves the JSON data from the POST request, getting the URL list and, optionally, the number of threads (defaults to 1) and the link levels we want to crawl (defaults to 2). If the link list is missing and 400 status is returned with a message.

Next thr function gets an UUID to identify the job and prepares the metadata. The **todo** list actually contains both the URL and the current depth level, to inform thee crawling threads later where the link is in the hierarchy. The job is done when the **todo** list is empty and no pages are in the **processing** list. A noteworthy field in the metadata is the results dictionary, which is empty for now. Next, the **index** function creates the thread that will run the job, using the **crawling_job** function, and starts it. This sill end the index function and return 200.

The **crawling_job** function runs as a thread, and coordinates the job, from getting the results up to the moment the job metadata times out. We use a *ThreadPoolExecutor* to create the required number of threads, calling the **crawling_thread** function. Affter running the crawling threads (results will be in the metadata stored in the **jobs** global variable), the thread sleeps for 600 seconds (10 minutes) and removes the metadata from the service (simulating what other queues, such as Redis, would do). Please note that this point is reached only after crawling_thread has completed (next paragraph).

The **crawling_thread** function may have several threads running it, and has a while loop that executes according to a boolean variable – it starts with True, but changes to false once all URLs have been crawled (which ends the crawling thread). Inside the loop, we first retrieve the next URL to crawl from the **todo** field in the metadata – index 0 is the URL and index 1 is the current depth. After parsing the

URL by calling **process_page**, the function stores the images and checks the next depth: if below the maximum value, it will addd the links from the page to the **todo** list for the current job ID (UUID), with the next depth level. Otherwise, no new links are added if the maximum depth was reached. When no new page to crawl is available on **todo** in the metadata, the function will wait until all pages currently under processing are finished – this is required because a page being processed may add new links. Once no new links are avaliable, the function ends.

Status and Retrieving Results

Requesting the job status will call the **jobstatus** function, which takes the job ID. If the ID exists in the **jobs** metadata, we return the number of processed pages (length of **finished**) and the under processing pages (the length of **todo** added to **processing**). It the pade doesn't exist, an error is returned.

When asking for the results, the **jobresults** function is called. It first checks that the job is valid (returns an error if it doesn't exist). This may happen either because the job ID is invalid or the results from a previous valid job ID havee expired – whoever invoked the crawling job is required to retrieve results within the next 10 minutes after the job ends. If the job is valid, we simple return the results.

The Docker Container

The docker container is very simple. It has only three standard files:

- **Dockerfile**: specifies theimage to use (Ubuntu), asks to install the requirements and setup the application.
- **requirements.txt**: the Python libraries to install with pip.
- **.dockerignore**: the files to ignore, also covering the .git directory.

There's no Docker compose file, as we used only one service.

Shortcomings and Improvements

A first general improvement is to better handle exceptions, to have a robust web service. As mentioned before, using threads only allows some processing speed gains related to I/O operations – reading the documents from their URLs. Whereas moving from 1 to 2 threads gives a significant performance boost (roughly half the time, e.g. from 41 to 23 seconds), getting up to 4 threads will not see the same gains (e.g. from 41 to 16 seconds). Thee expectation is that using multiprocessing and leveraging real processor cores can provide more effective gains that go beyond I/O gains.

Another improvement would be to use a framework to manage jobs, suh as Redis. This can manage in a transparent way the removal of results from the webservice. As of now, the thread function for each job sleeps for 10 minutes before removing the meta data from the job, including the results – we assume at this point that result doesn't need to be persisted, and that the caller is responsible to consume shortly and, if needed, store the information.

The crawler implemented the ability to specify a different depth in the inout JSON document. The request was to crawl 2 levels (parent and child), but specifying a different value using the **levels** value allows the crawler to go deeper. However this may not work everywhere, as the crawler doesn't filter out non-HTML content, and will try to download links even is they're pointing to binary files (e.g. .zip

files). An improvement in this case is to use the web server document information (assuming it's reliable) and only crawl HTML documents.

Finally, the crawler can currently perform on plain HTML documents, failing to handle pages that were created using Javascript code, such as in Angular and NodeJS pages. For this we'd need to use a module such as Selenium, that integrates to a system browser (e.g. Chromium on Linux) to render the page and get the HTML document AFTER the Javascript code has run.