Reference guide

Comic search engine

1. Objective

The goal is to build a search engine for comics. The data is extracted from bdgest and more specifically the bedetheque subdomain. The web application has a nice look and feel, a wide range of search criterias and relevant visualizations.

2. Structure

The project consists of three main components: a MongoDB database, a Scrapy data extractor and a Flask web application. It is available at this GitHub repository. In order to clone the project, run the following command on your machine (you need to have git installed):

```
git clone https://github.com/nicolasvo95/OUAP-4314.git
```

The MongoDB database is NoSQL. The hierarchy is database, then collection, document and fields within a document. In the project, the database is called **bdgest**, the collections are **authors, series** and **comics**. The fields for each collection can be found here. Since the project's main language is Python, pymongo is used to query the database. The data extraction is done with the Scrapy framework. The web application is developed with Flask and the visualizations with HighchartsJS. The appearance and responsivity are obtained with MaterializeCSS.

3. Deployment

Concerning the deployment of the project, it is done via Docker Compose. Therefore, Docker and Docker Compose are required. See how to install them here. It allows for the deployment of all three containers at once, thus, meanwhile the scraped data is fed to the database, the web application can already be used.

If you already wish to deploy the project, simply run the following:

```
docker-compose up
```

If you wish to deploy in detached mode (i.e. in background):

```
docker-compose up -d
```

Check the logs with:

```
docker-compose logs
```

Check the status of the containers with:

```
docker-compose ps
```

Gracefully shut the containers down:

```
docker-compose down
```

N.B.: you may need to pull the Docker image for the mongo database. In that case, run the following:

```
docker pull mongo
```

4. Scrapy

4.1. Settings

The settings.py script controls the way the scrap is done. In fact, websites have defense mechanisms such as temporary IP blacklisting in order to prevent excessive scraping. The file allows to set a delay between each new fetching of a page and many other limitations. For example, the definition of specific user agents and customized middleware behaviour. In the case of the project, it was found acceptable to simply put a limit between each new request as well as a modified download delay:

```
AUTOTHROTTLE_START_DELAY = 3
DOWNLOAD_DELAY = 2
```

4.2. Items

The items.py script defines the structure of the three items author, series and comic. Since bdgest already provides a distinct ID for authors, series and comics, they are also retrieved and used as ID for our database. In MongoDB, the **_id** field defines a document ID, which by default is an automatically generated ObjectID. However it is not as convenient to read and manipulate.

4.3. Pipelines

The pipelines.py script defines how the items are inserted into the Mongo database. Indeed, each item has a specific destination collection. Upon receiving, the name of the item is simply checked:

```
if isinstance(item, AuthorsItem):
```

When it is checked that the item is not empty, it is inserted:

```
self.collection.insert_one(dict(item))
```

4.4. Spider

The authors.py script defines the actual spider. First, the Scrapy items are imported from the items.py file:

```
from ..items import AuthorsItem, SeriesItem, ComicsItem
```

Also the already existing databases are dropped for a fresh start:

```
client = MongoClient("mongo")
client.drop_database("bdgest")
```

The spider is named **authors**, its list of allowed domains consists simply of **bedetheque.com**. The list of start URLs consists of the pages referencing the authors by alphabetical order with the addition of a page for names starting by a special character. A tidy list comprehension covers all the pages:

```
start_urls =
['https://www.bedetheque.com/liste_auteurs_BD_{0}.html'.format(x) for x
in string.ascii_lowercase]
start_urls.append('https://www.bedetheque.com/liste_auteurs_BD_0.html')
```

The first **parse** function simply retrieves the URL for each author page from the pages listed in **start_urls**. Then a **yield** calls a Scrapy Request with the author page URL as parameter and the **parse_authors** function as callback.

parse_authors retrieves many informations about each author: first name, last name, nickname, birth date, death date, image, country and personal webpage. An additional full name is created by joining first name, last name and nickname. On the author page, the link to each of his/her series can be found. It is the parameter of the next **yield** with **parse_series** as callback function.

parse_series retrieves informations about both series and comics. Indeed, the full list of series can be retrieved by ending the URL with __10000.html:

```
var_url = re.sub('.html$', '__10000.html', var_url) #show all comics of
a series
```

It also happens that for each series page, all the comics are displayed and with many

informations, which in a first time meant we did not need to go further and actually fetch each comic's individual page.

When a comic is done being scraped, its item is sent to the pipeline. When all of a series' comics have been scraped, its item is sent to the pipeline. Finally when an author's series have all been scraped, its item is sent to the pipeline.

The idea behind the use of **yield** is to have the **parse** functions behave like **generators**. Therefore, when a series URL needs to be parsed, it can be done and when it is finished being parsed, we go back to the **parse_author** function but at its previous state and keep feeding the rest of series pages to be parsed.

5. Flask

Inside the app folder:

- views.py controls the routes, how the data is processed and which templates are rendered
- forms.py defines the structure of each form, i.e. the fields, their labels, their type

Inside the templates folder:

- macros.html contains the jinja macros which render the forms, tables of results and information cards
- dictionaries.html contains some variables so that they do not need to be redefined at many places
- base.html is the base template for all pages of the web application
- index.html defines the homepage and its three forms
- author.html, series.html and comic.html define the result pages for each author, series and comic respectively
- author_id.html defines the page of an author in particular
- series_id.html defines the page of a series in particular

The static folder contains the CSS and JS libraries responsible for the appearance and mobile responsivity of the web app.

5.1. run.py

It is the file which launches the web application. The port can be specified as well as the use of multithreading.

```
app.run(debug=True, port=1000, host='0.0.0.0', threaded=True)
```

5.2. forms.py

The project uses the WTForms library known for efficiently formatting forms in Flask web applications. In this project, there is a form for each author, series and comic. Each form has many fields and labels which are neatly defined in the forms.py script.

5.3. views.py

The file first imports the previously defined forms:

```
from .forms import AuthorForm, SeriesForm, ComicForm
```

The Mongo client is first instantiated and the database set to **bdgest**:

```
client = MongoClient("mongo")
db = client['bdgest']
```

A few functions were written in order to keep the code as *DRY* as possible:

- **series_by_id** and **author_by_id** simply return the name of the series or author knowing its ID
- **table_comic** and **table_series** are functions capable of returning lists of comics and series in a processed format. Not every field in the database are wanted for display on the website, therefore only the desired fields are kept:

```
document_updated.update({key: (document[key] if document.get(key) else
"") for key in ("scenario", "illustration", "editor", "legal_deposit")})
```

And special keys in the document starting with **redirect** help creating the redirections to other routes on the website, i.e. author and series specific page. It will be explained in depth in the macros.html template.

- **kardesh** is a function which resolves the task of searching through multiple collections (fun fact: kardesh means brother in Turkish). For example, in order to search for a series with a name “a” whose author has a name “b”, a first preliminary query of the authors collection for a name starting with “b” returns the list of author IDs matching the criteria. The initial query is updated to answer the condition that its author IDs need to belong to this list of author IDs.

The routes are:

- **home** (template index.html): homepage with the three forms
- **author** (template author.html): result page of the author form. The birth date and death date are formatted to day-month-year.
- **series** (template series.html): it is a bit more efficient than the **author** route with more dictionaries to automatically fetch the results of the input fields and to properly build

the Mongo query string. It essentially returns a list of series which answer the search.

- **comic** (template comic.html): similar to **series**
- **author_id** (template author_id.html): takes an **author_id** as argument. It outputs information about an author, his/her list of series and comics. Mongo aggregations are executed in order to get the data which feeds the visualizations in JS:

```
    pipeline = [
        {"$match": {"author_id": author_id}},
        {"$group": {"_id":"$genre", "count":{"$sum":1}}}
    ]
    output["pie"] = db.command('aggregate', 'series', pipeline=pipeline,
explain=False)["cursor"]["firstBatch"]
```

The aggregation with pymongo requires a defined pipeline which then has to be executed for a specific collection in the database. The output is in the **cursor** and **firstBatch** keys of the result dictionary. However, it was observed that this structure of the result dictionary is not the same for all versions of MongoDB, therefore, adjustments may need to be made.

- **series_id** (template series_id.html): simply displays information about the series and the list of comics belonging to the series in question.

5.4. Templates

The **macros.html** template contains three functions:

- **render_form**: renders the form based on lists of labels and field names which are WTForm-formatted
- **render_table**: renders the tables of authors, series and comics. It treats regular links (a href) and redirections differently: if the variable starts with “redirect”, it links to a route, else it is a regular link to the outside world.
- **render_card**: renders a card with the information contained in all the fields of an author or a series document

The **dictionaries.html** template contains some defined lists which are imported and used by other templates. The lists define the titles for the tables of authors, series and comics to be displayed. Defining them here allows us to define them once for every single template which needs them. Therefore, the code is kept *DRY*.

The **base.html** template is used by all of the coming templates. It defines the head, i.e. the imports of CSS and JS libraries, the compatibility with mobile devices and also the navbar.

The **index.html** template defines the homepage. The three forms are loaded in most part with the **render_table** macro.

author.html, series.html and **comic.html** display the results of the search forms.

author_id.html and **series_id.html** display the individual pages of an author and of a series.