In this session:

- You will learn to implement a simple web crawler to extract information from web pages, using the `Scrapy` framework
- We will use ElasticSearch for storing the information

1 Scrapy: a framework for web crawling

Scrapy is a python library for developing web crawlers and extracting information from web pages. This library helps to design and deploy crawlers that target specific web pages.

In this session we are going to explain the basics of implementing a web crawler, you have a more detailed example in the Scrapy documentation tutorial.

The basic code scaffolding for a web crawler is created automatically by the library, type in a terminal:

```
$ scrapy startproject caiscrapy
```

This will create a crawler project able to host different crawlers. As an example we are going to extract from the UPC Commons website the information in the pages that store all the TFGs presented by FIB students over the past years.

The url of this page is http://upcommons.upc.edu/handle/2099.1/18595/recent-submissions

In a terminal type the following:

```
$ cd caiscrapy/caiscrapy
$ scrapy genspider UPCCommonsTFG upcommons.upc.edu
```

This will generate the file `UPCCommonsTFG.py` inside the directory `spider` with a python class for the spider and its basic configuration. You will have to modify the variable `start_urls` so the starting point of the spider is:

`http://upcommons.upc.edu/handle/2099.1/18595/recent-submissions`

The class has only one method called `parse` that is the one that receives the pages generated by the crawler. This is the method that extracts information from the page and decides what links to follow. This method has only one parameter (`response`) that contains the response of the web server to the request of the spider. In its current state the spider does nothing, just downloads the first url and does not generate any output.

2 Parsing webpages

2.1 Extracting TFG information

In order to decide what and how to extract information from a website, first we have to analyze how the information is stored in its pages. Because `HTML` represents a webpage as a tree with different tags and information in their nodes we can use that structure to access what we want to extract.

Scrapy has two methods for extracting information from a webpage, one based on CSS and other in XPATH. We are going to use the first one because is simpler (but less powerful).

Fortunately for us the pages that we want to crawl are well structured and the `HTML` tags that have the information that we want are more or less marked. In order to *dissect* a webpage we can download it and open it with a text editor or we can use a browser like Chrome or Mozilla to inspect the page using `ctrl+shift+I`.

Looking closely to the structure of the page (and I mean closely) we can find that all the TFGs are inside a `<li>` tag of class `ds-artifact-item`. The method `css` of the object stored in the `response` parameter allows us to extract all the occurrences of this tag. We can iterate through all these elements to extract the information inside. The `css` method extracts the part of the `HTML` tree that begins with the `tag` passed as a parameter. We can use the attributes of the tag to be more selective. Each element that we extract can also be parsed using the same method.

Each element has a title, an url that links to the detailed information of the TFG, an author, a publisher, a date, the publication rights and an abstract.

This is the code that you will find in the function `parse` in the file `parse1.txt` within the files of this session:

```
for tfg in response.css('li.ds-artifact-item'):
    doc = {}
    doc['title'] = tfg.css('h4 a::text').extract_first()
    data = tfg.css('div.artifact-info')
    doc['url'] = response.urljoin(tfg.css('h4 a::attr(href)').extract_first())
    doc['author'] = data.css('span.author span::text').extract_first()
    doc['publisher'] = data.css('span.publisher::text').extract_first()
    doc['date'] = data.css('span.date::text').extract_first()
    doc['rights'] = data.css('span.rights::text').extract_first()
    doc['abstract'] = data.css('div.artifact-abstract::text').extract_first()

    yield doc
```

Basically we extract all the adequate `<li>` tags using the `css` method and for each one we use again `css` to parse the adequate tags for each field of information. The `extract_first` methods returns the first occurrence or `None` if there is none. For the url we add the domain of the web to complete the url if it is a relative one.

Everything is stored in a dictionary and returned to Scrapy using `yield`, this will make this function a generator, so the elements of the page are retrieved one by one when the crawler needs them.

Now you can try this web spider using:

```
$ scrapy crawl UPCCommonsTFG -o tfg.json
```

You will see in the standard output (mixed with the log of scrappy actions) the information from the first page of TFGs and it will also be stored in `tfg.json` in JSON format.

2.2 Going deeper

The information in the page with the list of TFGs is not complete. Each TFG has an individual page with more information like the full summary and a list of keywords. We have obtained the link to this page from each TFG and stored it in the `url` field.

Scrapy allows following links and adding the information from these links to the one that we already have. This can be done using the `Request` method. This method needs a `url`, the function that will process the webpage obtained from the url and also accepts the information that we have already collected.

The `parse2.txt` file has the updated code, we have substituted the `yield` of the `doc` dictionary by a `yield` of the value returned by the call to `Request`. This receives a new function for parsing the detailed page (`parse_detail`) and the already collected fields as the `meta` parameter. Notice that if we also return to Scrapy the information already in the `doc` dictionary we will have two different items for each TFG (the one from the list of TFGs and the one from the detailed page and that is not what we want).

The function `parse_detail` extracts the full summary and the keywords of a TFG. The designer of this page was kind enough to mark the tags as `expandable` (the full summary) and `descripcio` (the keywords). The summary in expandable is in different languages, but they are not indentified in the tags. In the solution that you have all is joined as one string. We could use a language detection algorithm to be able separate them, but this it is left as an exercise for you.

Substitute the first version of the code with the one in `parse2.txt` and run again the crawler to see the results.

2.3 Storing the items in ElasticSearch

Storing the data in a text file is ok, but it is more useful to store it in a database. Scrapy allows putting a pipeline in the middle of the scraping process and storing the data in a database (or anywhere you need to).

Substitute the automatically generated `pipelines.py` file with the one that you have with the session files. This file includes a class with methods that are called at the beginning and the end of the crawling process and each time an item is extracted. If you open the file you will see that a new index named `scrapy` is created in ElasticSearch and each item is stored as is. One additional possibility is to check if all the fields extracted are valid, dropping the item if not or changing the invalid values by a default, but that is beyond the scope of this session.

To activate the pipeline you will have to modify the `settings.py` file. Uncomment the line that has the `ITEM_PIPELINES` configuration and change it as:

```
ITEM_PIPELINES = {
    'caiscrapy.pipelines.CaiscrapyElasticPipeline': 300,
}
```

2.4 Going even deeper

A final step that is missing is to collect the information from more than just the first page of TFGs. For doing this we only have to extract the link in the page that points to the next page. The page designer was also kind enough to mark this link as a `<a>` tag of class `next-page-link`. The following code at the end of the `parse` function will do the trick:

```
next = response.css('a.next-page-link::attr(href)').extract_first()
if next is not None:
    next_page = response.urljoin(next)
    yield scrapy.Request(next_page, callback=self.parse)
```

Basically we search for the next page link and if it exist we follow it.

The `parse3.txt` file has the updated code.

Now the crawler will follow the next link of each page until there is no more pages.

2.5 Scraping all the way

Now you can run the crawler and get all the TFGs data. First start ElasticSearch and after that initiate the crawling process:

```
$ scrapy crawl UPCCommonsTFG
```

The process will take a couple of minutes (or more) and in the end you will have all the TFGs info stored in the database. Now we can query the index and search for information. You have a modified `SearchIndex.py` script among the session files. This script allows searching an index using LUCENE query syntax.

This syntax allows putting as a prefix of a word the field you want to use for the search, for example `author:jordi` will search for the word `jordi` only in the `author` field.

For example, you can try:

```
$ python SearchIndex.py --index scrapy --query keywords:bases AND keywords:dades
$ python SearchIndex.py --index scrapy --query keywords:machine AND keywords:learning
$ python SearchIndex.py --index scrapy --query description:game
$ python SearchIndex.py --index scrapy --query description:dades
$ python SearchIndex.py --index scrapy --query title:dades~2
$ python SearchIndex.py --index scrapy --query author:miquel~1
```

Invent your own queries and see the results.

Now you can play during the rest of the session with this crawler, for example, see if you can extract the director of the project from the detailed TFG page or any other information.

There are **no deliverables** for this session.