CRAWLER PROGRAMMING TUTORIAL

8/7/2015 - Version 1

Overview

Stargate

Most companies, publish job positions on their websites. At we want to offer this vacancies to our users. To accomplish this goal, we have set up a platform hosting a cluster of crawlers that navigates on companies websites extracting job positions, and writing this information on XML files.

The goal of this tutorial is to show how to program these job crawlers. Programmers should be familiar with Python programming.

Used Technologies

This is the detail of technologies used on crawlers.

- **python**: Main programming language to write the crawlers, it’s required to have a knowledge of python programming.
- **Scrapy**: Open source crawling framework, written on python, previous knowledge is not required but helps.
- **Selenium**: Test environment for web applications. We just use *Webdriver API*, to interact with page elements: buttons, text boxes, etc...
- **PhantomJs**: It is a headless browser, to automate interaction with web pages. Basically, Scrapy and Selenium does not interpret javascript code included in web pages, it PhantomJs job to execute javascript code to render web pages. This can be done transparently to programmer, that only has to decide when to use direct Response from scrapy and when to render response using Phantomjs.
- **Xpath**: language designed to browse HTML documents, providing access to DOM elements.
- **Regular Expressions**: There are cases when Xpath usage is not enough to extract data, mainly because required data is included in bigger texts. In that cases regular expressions can be used to extract requested portion of data.

beBee crawling platform structure

beBee crawling platform is a very simple framework to deploy crawlers, it provides all common functionality, leaving just navigation and data extracting tasks to be implemented on the crawlers. This way, crawlers become very simple and easy to program.

Folder structure

/output - The folder where the crawlers left XML output files.

/crawling - Configuration files and framework module files.

/crawling/spiders - Folder to deploy crawler files. Usually a crawler is implemented by a python file containing crawler code and some json files for text mappings.

My First Crawler, Amazon

We have developed an example crawler to extract Amazon's job vacancies. This example crawler is implemented in these files:

`/crawling/spiders/amazonGetItems.py` - The crawler

`/crawling/spiders/amazonCategoriesMap.json` - It is an array used to map job categories from amazon's style to beBee Category list. The array is writen in JSON format. This array maps each Job Category text in amazon with the corresponding numeric code in bebee for that job category.

Here is the list of beBee's job categories:
`https://www.bebee.com/api/v0/help/categories`

After running the crawler, a new file will appear:

`/crawling/spiders/amazonCategoriesMissing.json` - This is a file generated by the crawler, including the list of missing mappings on categories. This file can be used to enrich `amazonCategoriesMap.json`

`/crawling/spiders/amazonGetItems.ini` - Crawler's configuration file

How does the crawler work: amazonGetItems.py

```
class AmazonJobsSpider(scrapy.Spider):
```

File amazonGetItems.py defines the class AmazonJobsSpider, extending scrapy.Spider class. Scrapy defines more sophisticated classes, but scrapy.Spider provides all the basic functionality we need to write a crawler.

You can follow this link to learn more about available spider classes provided by Scrapy.

http://doc.scrapy.org/en/1.0/topics/spiders.html

```
        name = "amazonGetItems"
```

Attribute name has to be unique, can not be several crawlers using the same name. Scrapy will merge crawlers behaviour leading to unpredictable results.

```
      allowed_domains = ["amazon.jobs"]
```

allowed_domains defines the list of allowed domains for the crawler, any url with a different domain than allowed is ignored by the crawler.

More info:
http://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider.allowed_domains

```
      start_urls = (
            'http://www.amazon.jobs/results?searchStrings[]=all',
      )
```

This is the list of URLS to begin the crawling. Crawler will follow the links found on the initial URLS to scan the pages. Ir our Amazon Crawler example, this url will give as a paginated index linking to all vacancies webpages.

More information:
http://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider.start_urls

```
def set_crawler(self, crawler):
      # Change this class name
      super(AmazonJobsSpider, self).set_crawler(crawler)

      # Getting the BEBEE CONFIGURATION PARAMETERS from .ini
      # Second level configuration file takes precedence over settings.py
      config = ConfigParser.ConfigParser()
      if config.read('./crawling/spiders/' + self.name + '.ini'):
            for name, value in config.items('DEFAULT'):
                  crawler.settings.set(name, value)
      else:
            # NO .ini configuration file
            print "WARNING: no %s.ini config. using default values" % self.name

      # Getting the BEBEE CONFIGURATION PARAMETERS
      self.page_index    = crawler.settings.getint('BEBEE_SPIDER_FIRST_PAGE', 1)
      self.stop_index    = crawler.settings.getint('BEBEE_SPIDER_LAST_PAGE', 1)
      self.max_jobs      = crawler.settings.getint('BEBEE_SPIDER_MAX_ITEMS', 3)
      self.delay_crawl_page =
crawler.settings.getint('BEBEE_SPIDER_CRAWL_DELAY_PAGE', 5)
      self.delay_crawl_job     =
crawler.settings.getint('BEBEE_SPIDER_CRAWL_DELAY_ITEM', 1)
      self.max_execution_time =
crawler.settings.getint('BEBEE_SPIDER_MAX_EXECUTION_TIME', 1800)
      self.account_id    = crawler.settings.get('BEBEE_SPIDER_ACCOUNT_ID', '0')
      self.company_id    = crawler.settings.get('BEBEE_SPIDER_COMPANY_ID', '')

      # Logger start. This code need account_id
      self.beBeeLogger = BebeeLogger(account_id=self.account_id,
botName=self.name)
      self.beBeeLogger.init()
```

This method loads configuration values in the spider object.

Configuration parameters are:

- page_index: the first page to crawl in the pagination index.
- stop_index: the last page to crawl in the index. These two parameters can be used to break the crawling in several stages, you may not want to crawl the whole site in just un crawling execution.
- max_jobs: Maximum number of job vacancies to crawl, the crawler stops when it reaches this limit.
- delay_crawl_page: delay in seconds between index pages requests.
- delay_crawl_job: delay in seconds between job pages requests.
- max_execution_time: Maximum time to crawl, robot stops crawling when it reaches this limit.
- account_id: Constant number provided by beBee, identifies the crawler.
- company_id: Constant number provided by beBee, identifies the company publishing the job position.

```
def __init__(self):
      self.driver = webdriver.PhantomJS()
      self.driver.set_window_size(1024,768)

      # Mapper for geoname_id and country_code
      self.geoCache = MapperGeoCache()

      # List of unique categories
      self.uniqueCategoriesSet = set()

      # Load the dict for category mapper
      # Change this filename in each spider class
      with open('crawling/spiders/amazonCategoriesMap.json') as data_file:
            self.categories = json.load(data_file)

      # for counting elapsed time
      self.start_time = time()
```

This is spider’s class constructor. It creates a driver to manage Selenium API Webdriver. We’ll use this driver to navigate on index pagination.

We also use PhantomJs to execute Javascript code, PantomJs is a Headless browser.

To debug code, it’s useful to replace PantomJs with Firefox driver, so we can view page interaction. We need an X-Window environment to use Firefox driver.

After setting webdriver, we instantiate MapperGeoCache, that will provide methods to perform localization mappings.

Then we load dictionary file **`amazonCategoriesMap.json`**, this dictionary is used by categoryMapper method.

Finally, we record starting time, to control execution time.

```
def __del__(self):
      # Close selenium driver to avoid too much phantomJS running
      self.driver.close()

      # Saving the unique set of categories
      # Change this filename in each spider class
      fset = open('crawling/spiders/amazonCategoriesMissing.json', 'w')
      json.dump(list(self.uniqueCategoriesSet), fset)
      fset.close()

      # Log end
      self.beBeeLogger.end()
```

This code implements class destructor. In the destructor, we close selenium Webdriver, and save unmapped categories to **`amazonCategoriesMissing.json`** file.

```
def parse(self, response):
```

Parse method's task is to parse responses got from Scrapy from start urls.

```
self.driver.get(response.url)
```

We use Selenium webdriver to interact with webpage.

```
inputElement =
self.driver.find_element_by_xpath('//div[@data-ap="paginator"]//input')
inputElement.send_keys(Keys.CONTROL + "a");
inputElement.send_keys(str(page_index))
inputElement.send_keys(Keys.ENTER)
```

We locate the selector of input element that controls pagination. We set page number (page_index) and press ENTER.

Look that input selector is located using xpath. In this url you can find more information about this:

http://selenium-python.readthedocs.org/en/latest/locating-elements.html#locating-by-xpath

```
WebDriverWait(self.driver, 10).until(EC.presence_of_element_located((By.XPATH, 
'//table//tr[10]/td[2]/a')))
      clickElems  = self.driver.find_elements_by_xpath('//table//tr/td[1]')
      linkElems   = self.driver.find_elements_by_xpath('//table//tr/td[2]/a')
      categoryElems     = self.driver.find_elements_by_xpath('//table//tr/td[3]')
      locationElems     = self.driver.find_elements_by_xpath('//table//tr/td[4]')
```

We wait for the response, setting a timeout of 10 seconds. We save the information we are interested in: links to follow, categories, location names. Later we iterate linkElems to save information into arrays for further processing.

```
if (page_index == stop_index) or (max_jobs <= totalJobs) or (max_execution_time 
<= (time() - self.start_time)):
            break
```

Checking stop conditions in index loop.

Once we have got all job urls, we do a loop to extract vacancies information (items).

```
      # Data loop (foreach link)
      for i,link in enumerate(links):
            error_location = False
            error_category = False

            # Object to create XML
            item = BebeeItem()
            item['title'] = titles[i]
            item['offer_id'] = 
(re.search("^http://www.amazon.jobs/jobs/([0-9]*)/.*", link)).group(1)
            item['lang_code'] = 'en-US'
            item['url'] = link
            item['date'] = dt.strftime('%Y%m%d')
            item['account_id'] = str(self.account_id)
            item['company_id'] = str(self.company_id)
            item['location_name'] = locations[i]
            item['category_name'] = categories[i]
```

The loop extract several fields for each item, we can find detailed information about item fields in document **`beBee - Formato XML.pdf`**

Job vacancy (item) fields:

- title
- offer_id

- lang_code
- url
- date: formated YYYYMMDD
- description
- account_id: Constant number provided by beBee.
- company_id: Constant number provided by beBee, identifies publishing company.

```
# GEONAME MANAGEMENT
try:
item['geoname_id'] = self.geoCache.getGeonameId(locations[i])
item['country_code'] = self.geoCache.getCountryCode(locations[i])
except:
      error_message = "%s location not found in GeoName" % str(locations[i])
      print error_message
      error_location = True
      self.beBeeLogger.failure(item['offer_id'], error_message)
```

Mapping vacancy location. We use two methods:

- *getGeonameId*
- *getCountryCode*

Both methods get just an input text that should contain city name and country name (or code), it supports any format. Keep in mind that these calls must be inside try/except blocks, to prevent crawler stop in case of geolocation failure.

```
# CATEGORY MANAGEMENT
category_id = self.categoryMapper(item['category_name'])
if category_id:
      item['category_id'] = category_id
else:
      error_message = "category not found: %s" % str(item['category_name'])
      print error_message
      self.uniqueCategoriesSet.add(item['category_name'])
      error_category = True
      self.beBeeLogger.failure(item['offer_id'], error_message)
```

Category mapping using *categoryMapper* method.

```
# Crawl job description
request = scrapy.Request(item['url'], callback=self.parse_description)
request.meta['item'] = item
yield request
```

This code section extracts job position description. This information can not be extracted in jobs index, so we have to crawl job page to extract description (and any other information not included in the index).

To do this, we create a *Request* for url *item['url']*, and we process the response with *parse_description* method. Look that *Request* already include previously extracted data.

```
# Get job description
def parse_description(self, response):
      item = response.meta['item']
      description = ""
      descriptionDivList = response.xpath("//h2[contains(.//text(), 'Job Description')]/..").re(r'(?s)Job Description(.*)')
      for desc in descriptionDivList:
            description += unicode.strip(remove_tags(desc))
      item['description'] = description
      return item
```

This method scraps full job description, look that description is extracted using regular expression, because description text is surround by other data.

How to run the crawler

To test your script, we recommend you to crawl only a few pages. In a console, go to the root directory (the one that contains the crawling folder) and run the command

```
scrapy crawl amazonGetItems
```

This should generate a file in the output directory with the jobs crawled.

Things you should take into account

It's very important to analyse carefully website structure before start crawler coding. Identify where are located the data we want to scrap, and how to reach them. There will be times when all required information can be found in the index, and times when we have to crawl two or three webpages to extract all required information for an Item (job position). Sometimes, we will need to click buttons to navigate, and sometimes we would need to fill an input textbox to search vacancies.

Any job website has it's own structure and we need to design crawler taking this structure into account, to achieve the optimal and simplest way to extract information. It's convenient to know the different crawler types provided by scrapy to choose the one that fit better for our job website. In most cases, all methods required to process informations will already implemented by scrapy.

We strongly recommend to read scrapy documentation:
http://doc.scrapy.org/en/1.0/topics/spiders.html

Also, it’s very interesting to read Scrapy tutorial published in Scrapy website:
http://doc.scrapy.org/en/1.0/topics/spiders.html

Good Luck !