主要代码：

from \_\_future\_\_ import unicode\_literals

from builtins import str

import datetime, json, logging, os, sys

from django.db import connection

from scrapy import signals

from scrapy.spiders import Spider

from scrapy.spiders import CrawlSpider

from pydispatch import dispatcher

from scrapy.exceptions import CloseSpider, UsageError

import django

django.setup()

from django.core.exceptions import ObjectDoesNotExist

from django.db.models import Q

from dynamic\_scraper.models import Log, LogMarker

class NoParsingFilter(logging.Filter):

def filter(self, record=True):

return False

npf = NoParsingFilter()

logging.getLogger('twisted').addFilter(npf)

class DjangoBaseSpider(CrawlSpider):

name = None

action\_successful = False

mandatory\_vars = ['ref\_object', 'scraper', 'scrape\_url',]

allowed\_domains = []

start\_urls = []

conf = {

"DO\_ACTION": False,

"RUN\_TYPE": 'SHELL',

"SPLASH\_ARGS": {},

"IMAGES\_STORE\_FORMAT": 'FLAT',

"CUSTOM\_PROCESSORS": [],

"LOG\_ENABLED": True,

"LOG\_LEVEL": 'ERROR',

"LOG\_LIMIT": 250,

"CONSOLE\_LOG\_LEVEL": 'DEBUG',

}

bcolors = {

"HEADER": '\033[95m',

"OK": '\033[92m',

"INFO": '\033[33m',

"ERROR": '\033[91m',

"ENDC": '\033[0m',

"BOLD": '\033[1m',

"UNDERLINE": '\033[4m',

}

mp\_request\_kwargs = {}

fp\_request\_kwargs = {}

dp\_request\_kwargs = {}

def \_\_init\_\_(self, \*args, \*\*kwargs):

msg = "Django settings used: {s}".format(s=os.environ.get("DJANGO\_SETTINGS\_MODULE"))

self.dds\_logger = logging.getLogger('dds')

self.dds\_logger.info(msg)

super(DjangoBaseSpider, self).\_\_init\_\_(None, \*\*kwargs)

self.\_check\_mandatory\_vars()

def \_set\_ref\_object(self, ref\_object\_class, \*\*kwargs):

self.dds\_logger = logging.getLogger('dds')

if not 'id' in kwargs:

msg = "{cs}You have to provide the ID of the reference {type} object.{ce}".format(

type=ref\_object\_class.\_\_name\_\_, cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"])

self.dds\_logger.error(msg)

self.output\_usage\_help()

raise UsageError()

try:

self.ref\_object = ref\_object\_class.objects.get(pk=kwargs['id'])

except ObjectDoesNotExist:

msg = "{cs}{type} object with ID {id} not found.{ce}".format(

id=kwargs['id'], type=ref\_object\_class.\_\_name\_\_,

cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"])

self.dds\_logger.error(msg)

self.output\_usage\_help()

raise UsageError()

def \_set\_config(self, log\_msg, \*\*kwargs):

#run\_type|rt

if 'rt' in kwargs:

kwargs['run\_type'] = kwargs['rt']

if 'run\_type' in kwargs:

self.conf['RUN\_TYPE'] = kwargs['run\_type']

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "run\_type " + self.conf['RUN\_TYPE']

#do\_action

if 'do\_action' in kwargs:

if kwargs['do\_action'] == 'yes':

self.conf['DO\_ACTION'] = True

else:

self.conf['DO\_ACTION'] = False

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "do\_action " + str(self.conf['DO\_ACTION'])

else:

self.log("Attribute do\_action not set, not saving to Django DB.", logging.INFO)

if self.settings['DSCRAPER\_SPLASH\_ARGS']:

self.conf['SPLASH\_ARGS'] = self.settings['DSCRAPER\_SPLASH\_ARGS']

if 'wait' not in self.conf['SPLASH\_ARGS']:

self.conf['SPLASH\_ARGS']['wait'] = 0.5

if self.settings['DSCRAPER\_IMAGES\_STORE\_FORMAT']:

self.conf['IMAGES\_STORE\_FORMAT'] = self.settings['DSCRAPER\_IMAGES\_STORE\_FORMAT']

if self.conf["IMAGES\_STORE\_FORMAT"] == 'FLAT':

msg = "Use simplified FLAT images store format (save the original or one thumbnail image)"

self.dds\_logger.info(msg)

if self.settings['DSCRAPER\_IMAGES\_THUMBS'] and len(self.settings['DSCRAPER\_IMAGES\_THUMBS']) > 0:

msg = "IMAGES\_THUMBS setting found, saving images as thumbnail images "

msg += "with size {size} (first entry)".format(

size=next(iter(self.settings['DSCRAPER\_IMAGES\_THUMBS'].keys())))

else:

msg = "IMAGES\_THUMBS setting not found, saving images with original size"

self.dds\_logger.info(msg)

elif self.conf["IMAGES\_STORE\_FORMAT"] == 'ALL':

msg = "Use ALL images store format (Scrapy behaviour, save both original and thumbnail images)"

self.dds\_logger.info(msg)

else:

msg = "Use THUMBS images store format (save only the thumbnail images)"

self.dds\_logger.info(msg)

if self.settings['DSCRAPER\_CUSTOM\_PROCESSORS']:

self.conf['CUSTOM\_PROCESSORS'] = self.settings['DSCRAPER\_CUSTOM\_PROCESSORS']

if self.settings['DSCRAPER\_LOG\_ENABLED']:

self.conf['LOG\_ENABLED'] = self.settings['DSCRAPER\_LOG\_ENABLED']

if self.settings['DSCRAPER\_LOG\_LEVEL']:

self.conf['LOG\_LEVEL'] = self.settings['DSCRAPER\_LOG\_LEVEL']

if self.settings['LOG\_LEVEL']:

self.conf['CONSOLE\_LOG\_LEVEL'] = self.settings['LOG\_LEVEL']

if self.settings['DSCRAPER\_LOG\_LIMIT']:

self.conf['LOG\_LIMIT'] = self.settings['DSCRAPER\_LOG\_LIMIT']

if log\_msg == "":

log\_msg = "{}"

self.log("Runtime config: " + log\_msg, logging.INFO)

if self.conf['CONSOLE\_LOG\_LEVEL'] == 'DEBUG':

logging.getLogger('twisted').removeFilter(npf)

if self.conf['CONSOLE\_LOG\_LEVEL'] != 'DEBUG':

logging.getLogger('scrapy.middleware').addFilter(npf)

self.log("Configuration set: " + str(self.conf), logging.DEBUG)

dispatcher.connect(self.spider\_closed, signal=signals.spider\_closed)

def \_check\_mandatory\_vars(self):

if self.conf['RUN\_TYPE'] == 'TASK':

if not getattr(self, 'scheduler\_runtime', None):

msg = "You have to provide a scheduler\_runtime when running with run\_type TASK."

self.dds\_logger.error(msg)

raise CloseSpider(msg)

msg = "SchedulerRuntime (" + str(self.scheduler\_runtime) + ") found."

self.log(msg, logging.INFO)

for var in self.mandatory\_vars:

attr = getattr(self, var, None)

if not attr:

msg = "Missing attribute {a}.".format(a=var)

self.dds\_logger.error(msg)

raise CloseSpider(msg)

if self.scraper.status == 'P' or self.scraper.status == 'I':

msg = 'Scraper status set to {s}!'.format(s=self.scraper.get\_status\_display())

self.log(msg, logging.WARNING)

raise CloseSpider(msg)

def \_set\_request\_kwargs(self):

try:

self.scraper.get\_main\_page\_rpt()

except ObjectDoesNotExist:

msg = "Scraper must have exactly one main page request page type!"

self.dds\_logger.error(msg)

raise CloseSpider()

no\_fp\_rpt = True

for rpt in self.scraper.requestpagetype\_set.all():

if rpt.page\_type == 'MP':

pt\_dict = self.mp\_request\_kwargs

elif rpt.page\_type == 'FP':

pt\_dict = self.fp\_request\_kwargs

no\_fp\_rpt = False

else:

self.dp\_request\_kwargs[rpt.page\_type] = {}

pt\_dict = self.dp\_request\_kwargs[rpt.page\_type]

if rpt.headers != '':

try:

headers = json.loads(rpt.headers)

except ValueError:

msg = "Incorrect HTTP header attribute ({a}): not a valid JSON dict!".format(a=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

if not isinstance(headers, dict):

msg = "Incorrect HTTP header attribute ({a}): not a valid JSON dict!".format(a=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

pt\_dict['headers'] = headers

if rpt.body != '':

pt\_dict['body'] = rpt.body

if rpt.cookies != '':

try:

cookies = json.loads(rpt.cookies)

except ValueError:

msg = "Incorrect cookies attribute ({a}): not a valid JSON dict!".format(a=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

if not isinstance(cookies, dict):

msg = "Incorrect cookies attribute ({a}): not a valid JSON dict!".format(a=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

pt\_dict['cookies'] = cookies

if rpt.meta != '':

try:

meta = json.loads(rpt.meta)

except ValueError:

msg = "Incorrect meta attribute ({a}): not a valid JSON dict!".format(a=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

if not isinstance(meta, dict):

msg = "Incorrect meta attribute ({a}): not a valid JSON dict!".format(a=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

pt\_dict['meta'] = meta

if no\_fp\_rpt:

self.fp\_request\_kwargs = self.mp\_request\_kwargs.copy()

def \_set\_meta\_splash\_args(self):

for rpt in self.scraper.requestpagetype\_set.all():

if rpt.page\_type == 'MP':

pt\_dict = self.mp\_request\_kwargs

elif rpt.page\_type == 'FP':

pt\_dict = self.fp\_request\_kwargs

else:

pt\_dict = self.dp\_request\_kwargs[rpt.page\_type]

if rpt.content\_type == 'H' and rpt.render\_javascript:

if 'meta' not in pt\_dict:

pt\_dict['meta'] = {}

pt\_dict['meta']['splash'] = {

'endpoint': 'render.html',

'args': self.conf['SPLASH\_ARGS'].copy()

}

def struct\_log(self, msg):

level = self.conf['CONSOLE\_LOG\_LEVEL']

if level == 'INFO' or level == 'DEBUG':

self.log(msg, logging.INFO)

else:

self.log(msg, logging.WARNING)

def spider\_closed(self):

if self.conf['RUN\_TYPE'] == 'TASK' and self.conf['DO\_ACTION']:

time\_delta, factor, num\_crawls = self.scheduler.calc\_next\_action\_time(\

self.action\_successful,\

self.scheduler\_runtime.next\_action\_factor,\

self.scheduler\_runtime.num\_zero\_actions)

self.scheduler\_runtime.next\_action\_time = datetime.datetime.now() + time\_delta

self.scheduler\_runtime.next\_action\_factor = factor

self.scheduler\_runtime.num\_zero\_actions = num\_crawls

self.scheduler\_runtime.save()

msg = "Scheduler runtime updated (Next action time: "

msg += "{nat}, ".format(nat=str(self.scheduler\_runtime.next\_action\_time.strftime("%Y-%m-%d %H:%m")))

msg += "Next action factor: {naf}, ".format(naf=str(self.scheduler\_runtime.next\_action\_factor))

msg += "Zero actions: {za})".format(za=str(self.scheduler\_runtime.num\_zero\_actions))

self.log(msg, logging.INFO)

self.log("Closing Django DB connection.", logging.INFO)

connection.close()

def log(self, message, level=logging.DEBUG):

if self.conf['RUN\_TYPE'] == 'TASK' and self.conf['DO\_ACTION']:

if self.conf['LOG\_ENABLED'] and level >= getattr(logging, self.conf['LOG\_LEVEL']):

l = Log()

l.message = message

l.ref\_object = self.ref\_object.\_\_class\_\_.\_\_name\_\_ + " (" + str(self.ref\_object.pk) + ")"

l.type = 'None'

l.level = int(level)

l.spider\_name = self.name

l.scraper = self.scraper

# Look for corresponding log markers

lms = LogMarker.objects.filter(

Q(ref\_object=l.ref\_object) | Q(ref\_object=''),

Q(spider\_name=l.spider\_name) | Q(spider\_name=''),

Q(scraper=l.scraper) | Q(scraper\_\_isnull=True),

)

for lm in lms:

if lm.message\_contains in l.message:

if lm.custom\_type:

l.type = lm.custom\_type

else:

l.type = lm.get\_mark\_with\_type\_display()

l.save()

#Delete old logs

if Log.objects.count() > self.conf['LOG\_LIMIT']:

items = Log.objects.all()[self.conf['LOG\_LIMIT']:]

for item in items:

item.delete()

self.dds\_logger.log(level, message)

import sys  
  
from flask import Flask, jsonify, request  
  
sys.path.append('../')  
  
from Manager.ProxyManager import ProxyManager  
  
app = Flask(\_\_name\_\_)  
  
  
api\_list = {  
 'get': u'get an usable proxy',  
 'refresh': u'refresh proxy pool',  
 'get\_all': u'get all proxy from proxy pool',  
 'delete?proxy=127.0.0.1:8080': u'delete an unable proxy',  
}  
  
  
@app.route('/')  
def index():  
 return jsonify(api\_list)  
  
  
@app.route('/get/')  
def get():  
 proxy = ProxyManager().get()  
 return proxy  
  
  
@app.route('/refresh/')  
def refresh():  
 *# TODO refresh会有守护程序定时执行，由api直接调用性能较差，暂不使用  
 # ProxyManager().refresh()* pass  
 return 'success'  
  
  
@app.route('/get\_all/')  
def getAll():  
 proxies = ProxyManager().getAll()  
 return jsonify(list(proxies))  
  
  
@app.route('/delete/', methods=['GET'])  
def delete():  
 proxy = request.args.get('proxy')  
 ProxyManager().delete(proxy)  
 return 'success'  
  
  
@app.route('/get\_status/')  
def get\_status():  
 status = ProxyManager().get\_status()  
 return jsonify(status)  
  
  
def run():  
 app.run(host='0.0.0.0', port=5000)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 run()

from pymongo import MongoClient  
import random  
import json  
  
  
class MongodbClient(object):  
  
 def \_\_init\_\_(self, name, host, port):  
 self.name = name  
 self.client = MongoClient(host, port)  
 self.db = self.client.proxy  
  
  
 def changeTable(self, name):  
 self.name = name  
  
  
 def get(self):  
 proxy = self.getAll()  
 return random.choice(proxy) if proxy else None  
  
  
 def put(self, value):  
 if self.db[self.name].find\_one({'proxy': value}):  
 return None  
 else:  
 self.db[self.name].insert({'proxy': value})  
  
  
 def pop(self):  
 value = self.get()  
 if value:  
 self.delete(value)  
 return value  
  
  
 def delete(self, value):  
 self.db[self.name].remove({'proxy': value})  
  
  
 def getAll(self):  
 return [p['proxy'] for p in self.db[self.name].find()]  
  
  
 def clean(self):  
 self.client.drop\_database('proxy')  
  
  
 def delete\_all(self):  
 self.db[self.name].remove()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 db = MongodbClient('first', 'localhost', 27017)  
 db.put('127.0.0.1:1')  
 db2 = MongodbClient('second', 'localhost', 27017)  
 db2.put('127.0.0.1:2')  
 db.clean()

import sys  
import time  
import logging  
from threading import Thread  
from apscheduler.schedulers.blocking import BlockingScheduler  
  
sys.path.append('../')  
  
from Util.utilFunction import validUsefulProxy  
from Manager.ProxyManager import ProxyManager  
from Util.LogHandler import LogHandler  
  
\_\_author\_\_ = 'JHao'  
  
logging.basicConfig()  
  
  
class ProxyRefreshSchedule(ProxyManager):  
 *"""  
 代理定时刷新  
 """* def \_\_init\_\_(self):  
 ProxyManager.\_\_init\_\_(self)  
 self.log = LogHandler('refresh\_schedule')  
  
 def validProxy(self):  
 *"""  
 验证raw\_proxy\_queue中的代理, 将可用的代理放入useful\_proxy\_queue  
 :return:  
 """* self.db.changeTable(self.raw\_proxy\_queue)  
 raw\_proxy = self.db.pop()  
 self.log.info('%s start validProxy\_a' % time.ctime())  
 exist\_proxy = self.db.getAll()  
 while raw\_proxy:  
 if validUsefulProxy(raw\_proxy) and (raw\_proxy not in exist\_proxy):  
 self.db.changeTable(self.useful\_proxy\_queue)  
 self.db.put(raw\_proxy)  
 self.log.info('validProxy\_a: %s validation pass' % raw\_proxy)  
 else:  
 self.log.debug('validProxy\_a: %s validation fail' % raw\_proxy)  
 self.db.changeTable(self.raw\_proxy\_queue)  
 raw\_proxy = self.db.pop()  
 self.log.info('%s validProxy\_a complete' % time.ctime())  
  
  
def refreshPool():  
 pp = ProxyRefreshSchedule()  
 pp.validProxy()  
  
  
def main(process\_num=30):  
 p = ProxyRefreshSchedule()  
  
 *# 获取新代理* p.refresh()  
  
 *# 检验新代理* pl = []  
 for num in range(process\_num):  
 proc = Thread(target=refreshPool, args=())  
 pl.append(proc)  
  
 for num in range(process\_num):  
 pl[num].start()  
  
 for num in range(process\_num):  
 pl[num].join()  
  
  
def run():  
 *# main()* sched = BlockingScheduler()  
 sched.add\_job(main, 'interval', minutes=5)  
 sched.start()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 run()

import sys  
  
sys.path.append('../')  
  
from Util.utilFunction import validUsefulProxy  
from Manager.ProxyManager import ProxyManager  
from Util.LogHandler import LogHandler  
  
  
class ProxyValidSchedule(ProxyManager):  
 def \_\_init\_\_(self):  
 ProxyManager.\_\_init\_\_(self)  
 self.log = LogHandler('valid\_schedule')  
  
 def \_\_validProxy(self):  
 *"""  
 验证代理  
 :return:  
 """* while True:  
 self.db.changeTable(self.useful\_proxy\_queue)  
 for each\_proxy in self.db.getAll():  
 if isinstance(each\_proxy, bytes):  
 each\_proxy = each\_proxy.decode('utf-8')  
  
 if validUsefulProxy(each\_proxy):  
 *# 成功计数器加1* self.db.inckey(each\_proxy, 1)  
 self.log.debug('validProxy\_b: {} validation pass'.format(each\_proxy))  
 else:  
 *# 失败计数器减一* self.db.inckey(each\_proxy, -1)  
 *# self.db.delete(each\_proxy)* self.log.info('validProxy\_b: {} validation fail'.format(each\_proxy))  
 value = self.db.getvalue(each\_proxy)  
 if value and int(value) < -5:  
 *# 计数器小于-5删除该代理* self.db.delete(each\_proxy)  
 self.log.info('validProxy\_a running normal')  
  
 def main(self):  
 self.\_\_validProxy()  
  
  
def run():  
 p = ProxyValidSchedule()  
 p.main()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 p = ProxyValidSchedule()  
 p.main()

import os  
from Util.utilClass import ConfigParse  
from Util.utilClass import LazyProperty  
  
  
class GetConfig(object):  
 *"""  
 to get config from config.ini  
 """* def \_\_init\_\_(self):  
 self.pwd = os.path.split(os.path.realpath(\_\_file\_\_))[0]  
 self.config\_path = os.path.join(os.path.split(self.pwd)[0], 'Config.ini')  
 self.config\_file = ConfigParse()  
 self.config\_file.read(self.config\_path)  
  
 @LazyProperty  
 def db\_type(self):  
 return self.config\_file.get('DB', 'type')  
  
 @LazyProperty  
 def db\_name(self):  
 return self.config\_file.get('DB', 'name')  
  
 @LazyProperty  
 def db\_host(self):  
 return self.config\_file.get('DB', 'host')  
  
 @LazyProperty  
 def db\_port(self):  
 return int(self.config\_file.get('DB', 'port'))  
  
 @LazyProperty  
 def proxy\_getter\_functions(self):  
 return self.config\_file.options('ProxyGetter')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 gg = GetConfig()  
 print(gg.db\_type)  
 print(gg.db\_name)  
 print(gg.db\_host)  
 print(gg.db\_port)  
 print(gg.proxy\_getter\_functions)

import os  
  
import logging  
  
from logging.handlers import TimedRotatingFileHandler  
  
*# 日志级别*CRITICAL = 50  
FATAL = CRITICAL  
ERROR = 40  
WARNING = 30  
WARN = WARNING  
INFO = 20  
DEBUG = 10  
NOTSET = 0  
  
CURRENT\_PATH = os.path.dirname(os.path.abspath(\_\_file\_\_))  
ROOT\_PATH = os.path.join(CURRENT\_PATH, os.pardir)  
LOG\_PATH = os.path.join(ROOT\_PATH, 'log')  
  
  
class LogHandler(logging.Logger):  
 *"""  
 LogHandler  
 """* def \_\_init\_\_(self, name, level=DEBUG):  
 self.name = name  
 self.level = level  
 logging.Logger.\_\_init\_\_(self, self.name, level=level)  
 self.\_\_setFileHandler\_\_()  
 self.\_\_setStreamHandler\_\_()  
  
 def \_\_setFileHandler\_\_(self, level=None):  
 *"""  
 set file handler  
 :param level:  
 :return:  
 """* file\_name = os.path.join(LOG\_PATH, '{name}.log'.format(name=self.name))  
 *# 设置日志回滚, 保存在log目录, 一天保存一个文件, 保留15天* file\_handler = TimedRotatingFileHandler(filename=file\_name, when='D', interval=1, backupCount=15)  
 file\_handler.suffix = '%Y%m%d.log'  
 if not level:  
 file\_handler.setLevel(self.level)  
 else:  
 file\_handler.setLevel(level)  
 formatter = logging.Formatter('%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s')  
  
 file\_handler.setFormatter(formatter)  
 self.file\_handler = file\_handler  
 self.addHandler(file\_handler)  
  
 def \_\_setStreamHandler\_\_(self, level=None):  
 *"""  
 set stream handler  
 :param level:  
 :return:  
 """* stream\_handler = logging.StreamHandler()  
 formatter = logging.Formatter('%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s')  
 stream\_handler.setFormatter(formatter)  
 if not level:  
 stream\_handler.setLevel(self.level)  
 else:  
 stream\_handler.setLevel(level)  
 self.addHandler(stream\_handler)  
  
 def resetName(self, name):  
 *"""  
 reset name  
 :param name:  
 :return:  
 """* self.name = name  
 self.removeHandler(self.file\_handler)  
 self.\_\_setFileHandler\_\_()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 log = LogHandler('test')  
 log.info('this is a test msg')

class LazyProperty(object):  
 *"""  
 LazyProperty  
 explain: http://www.spiderpy.cn/blog/5/  
 """* def \_\_init\_\_(self, func):  
 self.func = func  
  
 def \_\_get\_\_(self, instance, owner):  
 if instance is None:  
 return self  
 else:  
 value = self.func(instance)  
 setattr(instance, self.func.\_\_name\_\_, value)  
 return value  
  
  
try:  
 from configparser import ConfigParser *# py3*except:  
 from ConfigParser import ConfigParser *# py2*class ConfigParse(ConfigParser):  
 *"""  
 rewrite ConfigParser, for support upper option  
 """* def \_\_init\_\_(self):  
 ConfigParser.\_\_init\_\_(self)  
  
 def optionxform(self, optionstr):  
 return optionstr  
  
  
class Singleton(type):  
 *"""  
 Singleton Metaclass  
 """* \_inst = {}  
  
 def \_\_call\_\_(cls, \*args, \*\*kwargs):  
 if cls not in cls.\_inst:  
 cls.\_inst[cls] = super(Singleton, cls).\_\_call\_\_(\*args)  
 return cls.\_inst[cls]

import requests  
from lxml import etree  
  
from Util.LogHandler import LogHandler  
from Util.WebRequest import WebRequest  
  
logger = LogHandler(\_\_name\_\_)  
  
  
*# noinspection PyPep8Naming*def robustCrawl(func):  
 def decorate(\*args, \*\*kwargs):  
 try:  
 return func(\*args, \*\*kwargs)  
 except Exception as e:  
 logger.info(u"sorry, 抓取出错。错误原因:")  
 logger.info(e)  
  
 return decorate  
  
  
*# noinspection PyPep8Naming*def verifyProxyFormat(proxy):  
 *"""  
 检查代理格式  
 :param proxy:  
 :return:  
 """* import re  
 verify\_regex = r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}:\d{1,5}"  
 return True if re.findall(verify\_regex, proxy) else False  
  
  
*# noinspection PyPep8Naming*def getHtmlTree(url, \*\*kwargs):  
 *"""  
 获取html树  
 :param url:  
 :param kwargs:  
 :return:  
 """* header = {'Connection': 'keep-alive',  
 'Cache-Control': 'max-age=0',  
 'Upgrade-Insecure-Requests': '1',  
 'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_3) AppleWebKit/537.36 (KHTML, like Gecko)',  
 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8',  
 'Accept-Encoding': 'gzip, deflate, sdch',  
 'Accept-Language': 'zh-CN,zh;q=0.8',  
 }  
 *# TODO 取代理服务器用代理服务器访问* wr = WebRequest()  
 html = wr.get(url=url, header=header).content  
 return etree.HTML(html)  
  
  
*# noinspection PyPep8Naming*def validUsefulProxy(proxy):  
 *"""  
 检验代理是否可用  
 :param proxy:  
 :return:  
 """* proxies = {"https": "https://{proxy}".format(proxy=proxy)}  
 try:  
 *# 超过40秒的代理就不要了* r = requests.get('https://www.baidu.com', proxies=proxies, timeout=40, verify=False)  
 if r.status\_code == 200:  
 logger.debug('%s is ok' % proxy)  
 return True  
 except Exception as e:  
 logger.info(e)  
 return False

import requests  
import random  
import time  
  
  
class WebRequest(object):  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 pass  
  
 @property  
 def user\_agent(self):  
 *"""  
 return an User-Agent at random  
 :return:  
 """* ua\_list = [  
 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.1599.101',  
 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.122',  
 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71',  
 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95',  
 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.71',  
 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; QQDownload 732; .NET4.0C; .NET4.0E)',  
 'Mozilla/5.0 (Windows NT 5.1; U; en; rv:1.8.1) Gecko/20061208 Firefox/2.0.0 Opera 9.50',  
 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:34.0) Gecko/20100101 Firefox/34.0',  
 ]  
 return random.choice(ua\_list)  
  
 @property  
 def header(self):  
 *"""  
 basic header  
 :return:  
 """* return {'User-Agent': self.user\_agent,  
 'Accept': '\*/\*',  
 'Connection': 'keep-alive',  
 'Accept-Language': 'zh-CN,zh;q=0.8'}  
  
 def get(self, url, header=None, retry\_time=5, timeout=30,  
 retry\_flag=list(), retry\_interval=5, \*args, \*\*kwargs):  
 *"""  
 get method  
 :param url: target url  
 :param header: headers  
 :param retry\_time: retry time when network error  
 :param timeout: network timeout  
 :param retry\_flag: if retry\_flag in content. do retry  
 :param retry\_interval: retry interval(second)  
 :param args:  
 :param kwargs:  
 :return:  
 """* headers = self.header  
 if header and isinstance(header, dict):  
 headers.update(header)  
 while True:  
 try:  
 html = requests.get(url, headers=headers, timeout=timeout)  
 if filter(lambda key: key in html.content, retry\_flag):  
 raise Exception  
 else:  
 return html  
 except Exception as e:  
 print(e)  
 retry\_time -= 1  
 if retry\_time <= 0:  
 return  
 time.sleep(retry\_interval)

from \_\_future\_\_ import unicode\_literals

from builtins import str

from builtins import map

from builtins import range

import ast, datetime, importlib, json, logging, scrapy

from jsonpath\_rw import jsonpath, parse

from jsonpath\_rw.lexer import JsonPathLexerError

from scrapy.selector import Selector

from scrapy.http import Request, FormRequest

from scrapy.loader import ItemLoader

from scrapy.loader.processors import Join, TakeFirst

from scrapy.exceptions import CloseSpider

from django.db.models.signals import post\_save

from django.utils.encoding import smart\_text

from dynamic\_scraper.spiders.django\_base\_spider import DjangoBaseSpider

from dynamic\_scraper.models import ScraperElem

from dynamic\_scraper.utils.loader import JsonItemLoader

from dynamic\_scraper.utils.scheduler import Scheduler

from dynamic\_scraper.utils import processors

class DjangoSpider(DjangoBaseSpider):

tmp\_non\_db\_results = {}

non\_db\_results = {}

current\_output\_num\_mp\_response\_bodies = 0

current\_output\_num\_dp\_response\_bodies = 0

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.mandatory\_vars.append('scraped\_obj\_class')

self.mandatory\_vars.append('scraped\_obj\_item\_class')

super(DjangoSpider, self).\_\_init\_\_(self, \*args, \*\*kwargs)

@classmethod

def from\_crawler(cls, crawler, \*args, \*\*kwargs):

spider = cls(\*args, \*\*kwargs)

spider.\_set\_crawler(crawler)

spider.\_set\_config(\*\*kwargs)

spider.\_set\_request\_kwargs()

for cp\_path in spider.conf['CUSTOM\_PROCESSORS']:

try:

custom\_processors = importlib.import\_module(cp\_path)

except ImportError:

msg = "Custom processors from {path} could not be imported, processors won't be applied".format(

path=cp\_path,

)

spider.log(msg, logging.WARNING)

post\_save.connect(spider.\_post\_save\_tasks, sender=spider.scraped\_obj\_class)

spider.\_set\_start\_urls(spider.scrape\_url)

spider.scheduler = Scheduler(spider.scraper.scraped\_obj\_class.scraper\_scheduler\_conf)

spider.from\_page = 'MP'

spider.loader = None

spider.dummy\_loader = None

spider.items\_read\_count = 0

spider.items\_save\_count = 0

msg = 'Spider for {roc} "{ro}" ({pk}) initialized.'.format(

roc=spider.ref\_object.\_\_class\_\_.\_\_name\_\_,

ro=str(spider.ref\_object),

pk=str(spider.ref\_object.pk),

)

spider.log(msg, logging.INFO)

return spider

def output\_usage\_help(self):

out = (

'',

'DDS Usage',

'=========',

' scrapy crawl [scrapy\_options] SPIDERNAME -a id=REF\_OBJECT\_ID [dds\_options]',

'',

'Options',

'-------',

'-a do\_action=(yes|no) Save output to DB, default: no (Test Mode or File Output)',

'-L LOG\_LEVEL (scrapy option) Setting the log level for both Scrapy and DDS',

'-a run\_type|rt=(TASK|SHELL) Simulate task based scraper run, default: SHELL',

'-a max\_items\_read|mir=[Int] Limit number of items to read',

'-a max\_items\_save|mis=[Int] Limit number of items to save',

'-a max\_pages\_read|mpr=[Int] Limit number of pages to read (static pagination)',

'-a start\_page|sp=[PAGE] Start at page PAGE, e.g. 5, F (static pagination)',

'-a end\_page|ep=[PAGE] End scraping at page PAGE, e.g. 10, M (static pagination)',

'-a num\_pages\_follow|npf=[Int] Number of pages to follow (dynamic pagination)',

'-a output\_num\_mp\_response\_bodies|omp=[Int] Output response body content of MP for debugging',

'-a output\_num\_dp\_response\_bodies|odb=[Int] Output response body content of DP for debugging',

'',

)

for out\_str in out:

self.dds\_logger.info(out\_str)

def \_set\_request\_kwargs(self):

super(DjangoSpider, self).\_set\_request\_kwargs()

for rpt in self.scraper.requestpagetype\_set.all():

if rpt.form\_data != '':

try:

form\_data = json.loads(rpt.form\_data)

except ValueError:

msg = "Incorrect form\_data attribute ({pt}): not a valid JSON dict!".format(pt=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

if not isinstance(form\_data, dict):

msg = "Incorrect form\_data attribute ({pt}): not a valid JSON dict!".format(pt=rpt.page\_type)

self.dds\_logger.error(msg)

raise CloseSpider()

def \_set\_config(self, \*\*kwargs):

log\_msg = ""

#max\_items\_read|mir

if 'mir' in kwargs:

kwargs['max\_items\_read'] = kwargs['mir']

if 'max\_items\_read' in kwargs:

try:

self.conf['MAX\_ITEMS\_READ'] = int(kwargs['max\_items\_read'])

except ValueError:

msg = "You have to provide an integer value as max\_items\_read parameter!"

self.dds\_logger.error(msg)

raise CloseSpider()

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "max\_items\_read " + str(self.conf['MAX\_ITEMS\_READ'])

else:

self.conf['MAX\_ITEMS\_READ'] = self.scraper.max\_items\_read

#max\_items\_save|mis

if 'mis' in kwargs:

kwargs['max\_items\_save'] = kwargs['mis']

if 'max\_items\_save' in kwargs:

try:

self.conf['MAX\_ITEMS\_SAVE'] = int(kwargs['max\_items\_save'])

except ValueError:

msg = "You have to provide an integer value as max\_items\_save parameter!"

self.dds\_logger.error(msg)

raise CloseSpider()

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "max\_items\_save " + str(self.conf['MAX\_ITEMS\_SAVE'])

else:

self.conf['MAX\_ITEMS\_SAVE'] = self.scraper.max\_items\_save

#max\_pages\_read|mpr

if 'mpr' in kwargs:

kwargs['max\_pages\_read'] = kwargs['mpr']

if 'max\_pages\_read' in kwargs:

try:

self.conf['MAX\_PAGES\_READ'] = int(kwargs['max\_pages\_read'])

except ValueError:

msg = "You have to provide an integer value as max\_pages\_read parameter!"

self.dds\_logger.error(msg)

raise CloseSpider()

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "max\_pages\_read " + str(self.conf['MAX\_PAGES\_READ'])

else:

self.conf['MAX\_PAGES\_READ'] = None

#start\_page|sp

if 'sp' in kwargs:

kwargs['start\_page'] = kwargs['sp']

if 'start\_page' in kwargs:

self.conf['START\_PAGE'] = kwargs['start\_page']

else:

self.conf['START\_PAGE'] = None

#end\_page|ep

if 'ep' in kwargs:

kwargs['end\_page'] = kwargs['ep']

if 'end\_page' in kwargs:

self.conf['END\_PAGE'] = kwargs['end\_page']

else:

self.conf['END\_PAGE'] = None

#num\_pages\_follow|npf

if 'npf' in kwargs:

kwargs['num\_pages\_follow'] = kwargs['npf']

if 'num\_pages\_follow' in kwargs:

try:

self.conf['NUM\_PAGES\_FOLLOW'] = int(kwargs['num\_pages\_follow'])

except ValueError:

msg = "You have to provide an integer value as num\_pages\_follow parameter!"

self.dds\_logger.error(msg)

raise CloseSpider()

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "num\_pages\_follow " + str(self.conf['NUM\_PAGES\_FOLLOW'])

else:

self.conf['NUM\_PAGES\_FOLLOW'] = self.scraper.num\_pages\_follow

#output\_num\_mp\_response\_bodies|omp

if 'omp' in kwargs:

kwargs['output\_num\_mp\_response\_bodies'] = kwargs['omp']

if 'output\_num\_mp\_response\_bodies' in kwargs:

try:

self.conf['OUTPUT\_NUM\_MP\_RESPONSE\_BODIES'] = int(kwargs['output\_num\_mp\_response\_bodies'])

except ValueError:

msg = "You have to provide an integer value as output\_num\_mp\_response\_bodies parameter!"

self.dds\_logger.error(msg)

raise CloseSpider()

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "output\_num\_mp\_response\_bodies " + str(self.conf['OUTPUT\_NUM\_MP\_RESPONSE\_BODIES'])

else:

self.conf['OUTPUT\_NUM\_MP\_RESPONSE\_BODIES'] = 0

#output\_num\_dp\_response\_bodies|odp

if 'odp' in kwargs:

kwargs['output\_num\_dp\_response\_bodies'] = kwargs['odp']

if 'output\_num\_dp\_response\_bodies' in kwargs:

try:

self.conf['OUTPUT\_NUM\_DP\_RESPONSE\_BODIES'] = int(kwargs['output\_num\_dp\_response\_bodies'])

except ValueError:

msg = "You have to provide an integer value as output\_num\_dp\_response\_bodies parameter!"

self.dds\_logger.error(msg)

raise CloseSpider()

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "output\_num\_dp\_response\_bodies " + str(self.conf['OUTPUT\_NUM\_DP\_RESPONSE\_BODIES'])

else:

self.conf['OUTPUT\_NUM\_DP\_RESPONSE\_BODIES'] = 0

super(DjangoSpider, self).\_set\_config(log\_msg, \*\*kwargs)

def limit\_page\_nums(self, pages):

if self.conf['START\_PAGE']:

index = 0

exists = False

for page in pages:

if str(page) == self.conf['START\_PAGE']:

pages = pages[index:]

exists = True

break

index += 1

if not exists:

msg = "The provided start page doesn't exist in the range of page values!"

self.dds\_logger.error(msg)

raise CloseSpider()

if self.conf['END\_PAGE']:

index = 0

exists = False

for page in pages:

if str(page) == self.conf['END\_PAGE']:

pages = pages[:index+1]

exists = True

break

index += 1

if not exists:

msg = "The provided end page doesn't exist in the range of page values!"

self.dds\_logger.error(msg)

raise CloseSpider()

return pages

def \_set\_start\_urls(self, scrape\_url):

self.start\_urls = []

if self.scraper.pagination\_type in ['R', 'F',]:

if not self.scraper.pagination\_page\_replace:

msg = 'Please provide a pagination\_page\_replace context corresponding to pagination\_type!'

self.dds\_logger.error(msg)

raise CloseSpider()

if self.scraper.pagination\_type == 'R':

try:

pages = self.scraper.pagination\_page\_replace

pages = pages.split(',')

if len(pages) > 3:

raise Exception

pages = list(range(\*list(map(int, pages))))

except Exception:

msg = 'Pagination\_page\_replace for pagination\_type "RANGE\_FUNCT" ' +\

'has to be provided as python range function arguments ' +\

'[start], stop[, step] (e.g. "1, 50, 10", no brackets)!'

self.dds\_logger.error(msg)

raise CloseSpider()

pages = self.limit\_page\_nums(pages)

if self.scraper.pagination\_type == 'F':

try:

pages = self.scraper.pagination\_page\_replace

pages = pages.strip(', ')

pages = ast.literal\_eval("[" + pages + ",]")

except:

msg = 'Wrong pagination\_page\_replace format for pagination\_type "FREE\_LIST", ' +\

"Syntax: 'Replace string 1', 'Another replace string 2', 'A number 3', ..."

self.dds\_logger.error(msg)

raise CloseSpider()

pages = self.limit\_page\_nums(pages)

if self.scraper.pagination\_type in ['R', 'F',]:

append\_str = self.scraper.pagination\_append\_str

if scrape\_url[-1:] == '/' and append\_str[0:1] == '/':

append\_str = append\_str[1:]

self.pages = pages

if self.conf['MAX\_PAGES\_READ']:

self.pages = self.pages[0:self.conf['MAX\_PAGES\_READ']]

for page in self.pages:

url = scrape\_url + append\_str.format(page=page)

self.start\_urls.append(url)

if not self.scraper.pagination\_on\_start and not self.conf['START\_PAGE']:

self.start\_urls.insert(0, scrape\_url)

self.pages.insert(0, "")

if self.scraper.pagination\_type in ['N', 'O',]:

self.start\_urls.append(scrape\_url)

self.pages = ["",]

num = len(self.start\_urls)

if (num == 1):

url\_str = 'URL'

else:

url\_str = 'URLs'

self.log("Scraper set to run on {num} start {url\_str}.".format(

num=num, url\_str=url\_str), logging.INFO)

def \_prepare\_mp\_req\_data(self, kwargs\_orig, form\_data\_orig, page, follow\_page=''):

kwargs = kwargs\_orig.copy()

if 'meta' not in kwargs:

kwargs['meta'] = {}

form\_data = None

if form\_data\_orig:

form\_data = json.loads(form\_data\_orig).copy()

if 'headers' in kwargs:

kwargs['headers'] = json.loads(json.dumps(kwargs['headers']).replace('{page}', str(page)))

kwargs['headers'] = json.loads(json.dumps(kwargs['headers']).replace('{follow\_page}', str(follow\_page)))

if 'body' in kwargs:

kwargs['body'] = kwargs['body'].replace('{page}', str(page))

kwargs['body'] = kwargs['body'].replace('{follow\_page}', str(follow\_page))

if 'cookies' in kwargs:

kwargs['cookies'] = json.loads(json.dumps(kwargs['cookies']).replace('{page}', str(page)))

kwargs['cookies'] = json.loads(json.dumps(kwargs['cookies']).replace('{follow\_page}', str(follow\_page)))

if form\_data:

form\_data = json.loads(json.dumps(form\_data).replace('{page}', str(page)))

form\_data = json.loads(json.dumps(form\_data).replace('{follow\_page}', str(follow\_page)))

return kwargs, form\_data

def \_log\_page\_info(self, page\_num, follow\_page\_num, url, rpt, form\_data, kwargs):

self.dds\_logger.info('')

self.dds\_logger.info(self.bcolors['BOLD'] + '======================================================================================' + self.bcolors['ENDC'])

self.struct\_log("{es}{es2}Scraping data from page {p}({fp}).{ec}{ec}".format(

p=page\_num, fp=follow\_page\_num, es=self.bcolors['BOLD'], es2=self.bcolors['HEADER'], ec=self.bcolors['ENDC']))

self.struct\_log("URL : {url}".format(url=url))

self.\_log\_request\_info(rpt, form\_data, kwargs)

self.dds\_logger.info(self.bcolors['BOLD'] + '======================================================================================' + self.bcolors['ENDC'])

def start\_requests(self):

index = 0

rpt = self.scraper.get\_main\_page\_rpt()

follow\_page\_num = 0

for url in self.start\_urls:

self.\_set\_meta\_splash\_args()

page\_num = index + 1

kwargs, form\_data = self.\_prepare\_mp\_req\_data(self.mp\_request\_kwargs, self.scraper.get\_main\_page\_rpt().form\_data, self.pages[index])

kwargs['meta']['page\_num'] = page\_num

kwargs['meta']['follow\_page\_num'] = follow\_page\_num

kwargs['meta']['rpt'] = rpt

self.\_log\_page\_info(page\_num, follow\_page\_num, url, rpt, form\_data, kwargs)

index += 1

if rpt.request\_type == 'R':

yield Request(url, callback=self.parse, method=rpt.method, dont\_filter=rpt.dont\_filter, \*\*kwargs)

else:

yield FormRequest(url, callback=self.parse, method=rpt.method, formdata=form\_data, dont\_filter=rpt.dont\_filter, \*\*kwargs)

def \_check\_for\_double\_item(self, item):

idf\_elems = self.scraper.get\_id\_field\_elems()

num\_item\_idfs = 0

for idf\_elem in idf\_elems:

idf\_name = idf\_elem.scraped\_obj\_attr.name

if idf\_name in item:

num\_item\_idfs += 1

cnt\_double = 0

if len(idf\_elems) > 0 and num\_item\_idfs == len(idf\_elems):

qs = self.scraped\_obj\_class.objects

for idf\_elem in idf\_elems:

idf\_name = idf\_elem.scraped\_obj\_attr.name

qs = qs.filter(\*\*{idf\_name:item[idf\_name]})

cnt\_double = qs.count()

# Mark item as DOUBLE item

if cnt\_double > 0:

item.\_is\_double = True

return item, True

else:

item.\_is\_double = False

return item, False

def \_get\_processors(self, scraper\_elem):

procs\_str = scraper\_elem.processors

attr\_type = scraper\_elem.scraped\_obj\_attr.attr\_type

if scraper\_elem.use\_default\_procs:

procs = [TakeFirst(), processors.string\_strip,]

else:

procs = []

if not procs\_str:

return procs

procs\_tmp = list(procs\_str.split(','))

for p in procs\_tmp:

p = p.strip()

added = False

if hasattr(processors, p):

procs.append(getattr(processors, p))

added = True

for cp\_path in self.conf['CUSTOM\_PROCESSORS']:

try:

custom\_processors = importlib.import\_module(cp\_path)

if hasattr(custom\_processors, p):

procs.append(getattr(custom\_processors, p))

added = True

except ImportError:

pass

if not added:

self.log("Processor '{p}' is not defined!".format(p=p), logging.ERROR)

procs = tuple(procs)

return procs

def \_set\_loader\_context(self, context\_str):

try:

context\_str = context\_str.strip(', ')

context = ast.literal\_eval("{" + context\_str + "}")

context['spider'] = self

self.loader.context = context

self.dummy\_loader.context = context

except SyntaxError:

self.log("Wrong context definition format: " + context\_str, logging.ERROR)

def \_scrape\_item\_attr(self, scraper\_elem, response, from\_page, item\_num):

if(from\_page == scraper\_elem.request\_page\_type or

(from\_page == 'FP' and scraper\_elem.request\_page\_type == 'MP')):

procs = self.\_get\_processors(scraper\_elem)

self.\_set\_loader\_context(scraper\_elem.proc\_ctxt)

if not scraper\_elem.scraped\_obj\_attr.save\_to\_db:

name = 'tmp\_field'

loader = self.dummy\_loader

else:

name = scraper\_elem.scraped\_obj\_attr.name

loader = self.loader

static\_ctxt = loader.context.get('static', '')

self.log("Applying the following processors: {p\_list}".format(

p\_list=str([p.\_\_name\_\_ if hasattr(p, '\_\_name\_\_') else type(p).\_\_name\_\_ for p in procs])),

logging.DEBUG)

if processors.static in procs and static\_ctxt:

loader.add\_value(name, static\_ctxt)

elif(scraper\_elem.reg\_exp):

loader.add\_xpath(name, scraper\_elem.x\_path, \*procs, re=scraper\_elem.reg\_exp)

else:

loader.add\_xpath(name, scraper\_elem.x\_path, \*procs)

if not scraper\_elem.scraped\_obj\_attr.save\_to\_db:

item = loader.load\_item()

if name in item:

self.tmp\_non\_db\_results[item\_num][scraper\_elem.scraped\_obj\_attr.name] = item[name]

rpt = self.scraper.requestpagetype\_set.filter(page\_type=from\_page)[0]

rpt\_str = rpt.get\_content\_type\_display()

if rpt.render\_javascript:

rpt\_str += '-JS'

rpt\_str += '|' + rpt.method

page\_str = str(response.request.meta['page\_num'])

page\_str += '(' + str(response.request.meta['follow\_page\_num']) + ')-'

msg = '{page\_type: <4} {rpt\_str: <13} {cs}{name: <20}{ce} {page}{num} '.format(page=page\_str, num=str(item\_num), name=name, rpt\_str=rpt\_str, page\_type=from\_page, cs=self.bcolors["BOLD"], ce=self.bcolors["ENDC"])

c\_values = loader.get\_collected\_values(name)

if len(c\_values) > 0:

val\_str = c\_values[0]

if self.conf['CONSOLE\_LOG\_LEVEL'] != 'DEBUG':

val\_str = (val\_str[:400] + '..') if len(val\_str) > 400 else val\_str

msg += smart\_text(val\_str)

else:

msg += 'None'

self.log(msg, logging.INFO)

def \_set\_loader(self, response, from\_page, xs, item):

self.from\_page = from\_page

rpt = self.scraper.get\_rpt(from\_page)

if not (self.from\_page == 'MP' or self.from\_page == 'FP'):

item = response.request.meta['item']

if rpt.content\_type == 'J':

json\_resp = json.loads(response.body\_as\_unicode())

self.loader = JsonItemLoader(item=item, selector=json\_resp)

else:

self.loader = ItemLoader(item=item, response=response)

else:

if rpt.content\_type == 'J':

self.loader = JsonItemLoader(item=item, selector=xs)

else:

self.loader = ItemLoader(item=item, selector=xs)

self.loader.default\_output\_processor = TakeFirst()

self.loader.log = self.log

def \_set\_dummy\_loader(self, response, from\_page, xs, item):

self.from\_page = from\_page

rpt = self.scraper.get\_rpt(from\_page)

if not (self.from\_page == 'MP' or self.from\_page == 'FP'):

item = response.request.meta['item']

if rpt.content\_type == 'J':

json\_resp = json.loads(response.body\_as\_unicode())

self.dummy\_loader = JsonItemLoader(item=DummyItem(), selector=json\_resp)

else:

self.dummy\_loader = ItemLoader(item=DummyItem(), response=response)

else:

if rpt.content\_type == 'J':

self.dummy\_loader = JsonItemLoader(item=DummyItem(), selector=xs)

else:

self.dummy\_loader = ItemLoader(item=DummyItem(), selector=xs)

self.dummy\_loader.default\_output\_processor = TakeFirst()

self.dummy\_loader.log = self.log

def parse\_item(self, response, xs=None, from\_page=None, item\_num=None):

#logging.info(str(response.request.meta))

#logging.info(response.body\_as\_unicode())

if not from\_page:

from\_page = response.request.meta['from\_page']

item\_num = response.request.meta['item\_num']

self.\_set\_loader(response, from\_page, xs, self.scraped\_obj\_item\_class())

self.\_set\_dummy\_loader(response, from\_page, xs, self.scraped\_obj\_item\_class())

if from\_page == 'MP' or from\_page == 'FP':

self.items\_read\_count += 1

else:

if self.current\_output\_num\_dp\_response\_bodies < self.conf['OUTPUT\_NUM\_DP\_RESPONSE\_BODIES']:

self.current\_output\_num\_dp\_response\_bodies += 1

self.log("Response body ({url})\n\n\*\*\*\*\* RP\_DP\_{num}\_START \*\*\*\*\*\n{resp\_body}\n\*\*\*\*\* RP\_DP\_{num}\_END \*\*\*\*\*\n\n".format(

url=response.url,

resp\_body=response.body,

num=self.current\_output\_num\_dp\_response\_bodies), logging.INFO)

elems = self.scraper.get\_scrape\_elems()

for elem in elems:

if not elem.scraped\_obj\_attr.save\_to\_db:

self.\_set\_dummy\_loader(response, from\_page, xs, self.scraped\_obj\_item\_class())

self.\_scrape\_item\_attr(elem, response, from\_page, item\_num)

# Dealing with Django Char- and TextFields defining blank field as null

item = self.loader.load\_item()

for key, value in list(item.items()):

if value == None and \

self.scraped\_obj\_class().\_meta.get\_field(key).blank and \

not self.scraped\_obj\_class().\_meta.get\_field(key).null:

item[key] = ''

if not (from\_page == 'MP' or from\_page == 'FP'):

item, is\_double = self.\_check\_for\_double\_item(item)

if response.request.meta['last']:

self.non\_db\_results[id(item)] = self.tmp\_non\_db\_results[item\_num].copy()

return item

else:

return item

def \_replace\_placeholders(self, text\_str, item, item\_num, only\_mp):

applied = []

if type(text\_str).\_\_name\_\_ != 'str' and type(text\_str).\_\_name\_\_ != 'unicode':

return text\_str, applied

standard\_elems = self.scraper.get\_standard\_elems()

for scraper\_elem in standard\_elems:

if not only\_mp or scraper\_elem.request\_page\_type == 'MP':

name = scraper\_elem.scraped\_obj\_attr.name

placeholder = '{' + name + '}'

if not scraper\_elem.scraped\_obj\_attr.save\_to\_db:

if name in self.tmp\_non\_db\_results[item\_num] and \

self.tmp\_non\_db\_results[item\_num][name] != None and \

placeholder in text\_str:

text\_str = text\_str.replace(placeholder, self.tmp\_non\_db\_results[item\_num][name])

applied.append(placeholder)

else:

if name in item and \

item[name] != None and \

placeholder in text\_str:

text\_str = text\_str.replace(placeholder, item[name])

applied.append(placeholder)

return text\_str, applied

def \_do\_req\_info\_replacements(self, item, item\_num, page, json\_dict, info\_str):

json\_dict = json.loads(json.dumps(json\_dict).replace('{page}', str(page)))

for key, value in list(json\_dict.items()):

new\_value, applied = self.\_replace\_placeholders(value, item, item\_num, True)

json\_dict[key] = new\_value

if len(applied) > 0:

msg = "Request info placeholder(s) applied (item {id}): {a}".format(

a=str(applied), id=item.\_dds\_id\_str)

self.log(msg, logging.DEBUG)

self.log(info\_str + " [" + str(key) + "] before: " + str(value), logging.DEBUG)

self.log(info\_str + " [" + str(key) + "] after : " + str(new\_value), logging.DEBUG)

return json\_dict

def parse(self, response):

xs = Selector(response)

base\_objects = []

base\_elem = self.scraper.get\_base\_elem()

rpt = response.request.meta['rpt']

page\_num = response.request.meta['page\_num']

page = self.pages[page\_num - 1]

follow\_page\_num = response.request.meta['follow\_page\_num']

if rpt.page\_type == 'MP':

if self.current\_output\_num\_mp\_response\_bodies < self.conf['OUTPUT\_NUM\_MP\_RESPONSE\_BODIES']:

self.current\_output\_num\_mp\_response\_bodies += 1

self.log("Response body ({url})\n\n\*\*\*\*\* RP\_MP\_{num}\_START \*\*\*\*\*\n{resp\_body}\n\*\*\*\*\* RP\_MP\_{num}\_END \*\*\*\*\*\n\n".format(

url=response.url,

resp\_body=response.body,

num=self.current\_output\_num\_mp\_response\_bodies), logging.INFO)

if rpt.content\_type == 'J':

json\_resp = None

try:

json\_resp = json.loads(response.body\_as\_unicode())

except ValueError:

msg = "JSON response for MP could not be parsed!"

self.log(msg, logging.ERROR)

if json\_resp:

try:

jsonpath\_expr = parse(base\_elem.x\_path)

except JsonPathLexerError:

msg = "JsonPath for base elem could not be processed!"

self.dds\_logger.error(msg)

raise CloseSpider()

base\_objects = [match.value for match in jsonpath\_expr.find(json\_resp)]

if len(base\_objects) > 0:

base\_objects = base\_objects[0]

else:

base\_objects = response.xpath(base\_elem.x\_path)

if(len(base\_objects) == 0):

self.log("{cs}No base objects found.{ce}".format(

cs=self.bcolors["INFO"], ce=self.bcolors["ENDC"]), logging.ERROR)

if(self.conf['MAX\_ITEMS\_READ']):

items\_left = min(len(base\_objects), self.conf['MAX\_ITEMS\_READ'] - self.items\_read\_count)

base\_objects = base\_objects[0:items\_left]

for obj in base\_objects:

item\_num = self.items\_read\_count + 1

self.tmp\_non\_db\_results[item\_num] = {}

page\_str = str(page\_num) + '(' + str(follow\_page\_num) + ')'

self.dds\_logger.info("")

self.dds\_logger.info(self.bcolors['BOLD'] + '--------------------------------------------------------------------------------------' + self.bcolors['ENDC'])

self.struct\_log("{cs}Starting to crawl item {i} from page {p}.{ce}".format(

i=str(item\_num), p=page\_str, cs=self.bcolors["HEADER"], ce=self.bcolors["ENDC"]))

self.dds\_logger.info(self.bcolors['BOLD'] + '--------------------------------------------------------------------------------------' + self.bcolors['ENDC'])

item = self.parse\_item(response, obj, rpt.page\_type, item\_num)

item.\_dds\_item\_page = page\_num

item.\_dds\_item\_follow\_page = follow\_page\_num

item.\_dds\_item\_id = item\_num

item.\_dds\_id\_str = str(item.\_dds\_item\_page) + '(' + str(item.\_dds\_item\_follow\_page) + ')-' + str(item.\_dds\_item\_id)

if item:

only\_main\_page\_idfs = True

idf\_elems = self.scraper.get\_id\_field\_elems()

for idf\_elem in idf\_elems:

if idf\_elem.request\_page\_type != 'MP':

only\_main\_page\_idfs = False

is\_double = False

if only\_main\_page\_idfs:

item, is\_double = self.\_check\_for\_double\_item(item)

# Don't go on reading detail pages when...

# No detail page URLs defined or

# DOUBLE item with only main page IDFs and no standard update elements to be scraped from detail pages or

# generally no attributes scraped from detail pages

cnt\_sue\_detail = self.scraper.get\_standard\_update\_elems\_from\_detail\_pages().count()

cnt\_detail\_scrape = self.scraper.get\_from\_detail\_pages\_scrape\_elems().count()

if self.scraper.get\_detail\_page\_url\_elems().count() == 0 or \

(is\_double and cnt\_sue\_detail == 0) or cnt\_detail\_scrape == 0:

self.non\_db\_results[id(item)] = self.tmp\_non\_db\_results[item\_num].copy()

yield item

else:

#self.run\_detail\_page\_request()

url\_elems = self.scraper.get\_detail\_page\_url\_elems()

for url\_elem in url\_elems:

if not url\_elem.scraped\_obj\_attr.save\_to\_db:

url\_before = self.tmp\_non\_db\_results[item\_num][url\_elem.scraped\_obj\_attr.name]

url, applied = self.\_replace\_placeholders(url\_before, item, item\_num, True)

self.tmp\_non\_db\_results[item\_num][url\_elem.scraped\_obj\_attr.name] = url

else:

url\_before = item[url\_elem.scraped\_obj\_attr.name]

url, applied = self.\_replace\_placeholders(url\_before, item, item\_num, True)

item[url\_elem.scraped\_obj\_attr.name] = url

if len(applied) > 0:

msg = "Detail page URL placeholder(s) applied (item {id}): {a}".format(

a=str(applied), id=item.\_dds\_id\_str)

self.log(msg, logging.DEBUG)

self.log("URL before: " + url\_before, logging.DEBUG)

self.log("URL after : " + url, logging.DEBUG)

dp\_rpt = self.scraper.get\_rpt\_for\_scraped\_obj\_attr(url\_elem.scraped\_obj\_attr)

kwargs = self.dp\_request\_kwargs[dp\_rpt.page\_type].copy()

if 'meta' not in kwargs:

kwargs['meta'] = {}

kwargs['meta']['page\_num'] = page\_num

kwargs['meta']['follow\_page\_num'] = follow\_page\_num

kwargs['meta']['item'] = item

kwargs['meta']['from\_page'] = dp\_rpt.page\_type

kwargs['meta']['item\_num'] = item\_num

kwargs['meta']['rpt'] = dp\_rpt

if 'headers' in kwargs:

kwargs['headers'] = self.\_do\_req\_info\_replacements(item, item\_num, page, kwargs['headers'], "HEADERS")

if 'body' in kwargs:

body\_before = kwargs['body']

kwargs['body'] = kwargs['body'].replace('{page}', str(page))

kwargs['body'], applied = self.\_replace\_placeholders(kwargs['body'], item, item\_num, True)

if len(applied) > 0:

msg = "Request info placeholder(s) applied (item {id}): {a}".format(

a=str(applied), id=item.\_dds\_id\_str)

self.log(msg, logging.DEBUG)

self.log("BODY before: " + body\_before, logging.DEBUG)

self.log("BODY after : " + kwargs['body'], logging.DEBUG)

if 'cookies' in kwargs:

kwargs['cookies'] = self.\_do\_req\_info\_replacements(item, item\_num, page, kwargs['cookies'], "COOKIES")

form\_data = None

if dp\_rpt.request\_type == 'F' and dp\_rpt.form\_data:

form\_data = json.loads(dp\_rpt.form\_data).copy()

form\_data = self.\_do\_req\_info\_replacements(item, item\_num, page, form\_data, "FORM DATA")

if url\_elem == url\_elems[len(url\_elems)-1]:

kwargs['meta']['last'] = True

else:

kwargs['meta']['last'] = False

self.\_set\_meta\_splash\_args()

#logging.info(str(kwargs))

self.log(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>", logging.INFO)

msg = "{cs}Calling {dp} URL for item {id}...{ce}".format(

dp=dp\_rpt.page\_type, id=item.\_dds\_id\_str,

cs=self.bcolors["HEADER"], ce=self.bcolors["ENDC"])

self.log(msg, logging.INFO)

msg = "URL : {url}".format(url=url)

self.log(msg, logging.INFO)

self.\_log\_request\_info(dp\_rpt, form\_data, kwargs)

self.log(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>", logging.INFO)

if dp\_rpt.request\_type == 'R':

yield response.follow(url, callback=self.parse\_item, method=dp\_rpt.method, dont\_filter=dp\_rpt.dont\_filter, \*\*kwargs)

else:

yield FormRequest(url, callback=self.parse\_item, method=dp\_rpt.method, formdata=form\_data, dont\_filter=dp\_rpt.dont\_filter, \*\*kwargs)

for key, value in list(item.items()):

#Fixing some extremely weird Python 2 encoding failure, 2017-06-29

if type(value).\_\_name\_\_ == 'str':

try:

value = value.decode('utf-8')

except AttributeError:

pass

if value and (type(value).\_\_name\_\_ in ['str', 'unicode']) and '{page}' in value:

msg = "Applying page placeholder on {k}...".format(k=key)

self.log(msg, logging.DEBUG)

self.log("Value before: " + value, logging.DEBUG)

value = value.replace('{page}', str(page))

item[key] = value

self.log("Value after: " + value, logging.DEBUG)

else:

self.log("Item could not be read!", logging.ERROR)

mir\_reached = False

if self.conf['MAX\_ITEMS\_READ'] and (self.conf['MAX\_ITEMS\_READ'] - self.items\_read\_count <= 0):

mir\_reached = True

if self.scraper.follow\_pages\_url\_xpath and not mir\_reached:

if not self.conf['NUM\_PAGES\_FOLLOW'] or follow\_page\_num < self.conf['NUM\_PAGES\_FOLLOW']:

url = response.xpath(self.scraper.follow\_pages\_url\_xpath).extract\_first()

if url is not None:

self.\_set\_meta\_splash\_args()

follow\_page = ''

if self.scraper.follow\_pages\_page\_xpath:

follow\_page = response.xpath(self.scraper.follow\_pages\_page\_xpath).extract\_first()

form\_data\_orig = None

if self.scraper.get\_follow\_page\_rpts().count() > 0:

f\_rpt = self.scraper.get\_follow\_page\_rpts()[0]

form\_data\_orig = self.scraper.get\_follow\_page\_rpts()[0].form\_data

else:

f\_rpt = self.scraper.get\_main\_page\_rpt()

form\_data\_orig = self.scraper.get\_main\_page\_rpt().form\_data

kwargs, form\_data = self.\_prepare\_mp\_req\_data(self.fp\_request\_kwargs, form\_data\_orig, page, follow\_page)

follow\_page\_num += 1

kwargs['meta']['page\_num'] = page\_num

kwargs['meta']['follow\_page\_num'] = follow\_page\_num

kwargs['meta']['rpt'] = f\_rpt

self.\_log\_page\_info(page\_num, follow\_page\_num, url, f\_rpt, form\_data, kwargs)

if f\_rpt.request\_type == 'R':

yield response.follow(url, callback=self.parse, method=f\_rpt.method, dont\_filter=f\_rpt.dont\_filter, \*\*kwargs)

else:

url = response.urljoin(url)

yield FormRequest(url, callback=self.parse, method=f\_rpt.method, formdata=form\_data, dont\_filter=f\_rpt.dont\_filter, \*\*kwargs)

def \_log\_request\_info(self, rpt, form\_data, kwargs):

level = logging.DEBUG

extra\_info = False

if 'headers' in kwargs:

self.log("HEADERS : " + str(kwargs['headers']), level)

extra\_info = True

if 'body' in kwargs:

self.log("BODY : " + str(kwargs['body']), level)

extra\_info = True

if 'cookies' in kwargs:

self.log("COOKIES : " + str(kwargs['cookies']), level)

extra\_info = True

if rpt.request\_type == 'F' and form\_data:

self.log("FORM DATA : " + str(form\_data), level)

extra\_info = True

if not extra\_info:

self.log("No additional request information sent.", level)

def \_post\_save\_tasks(self, sender, instance, created, \*\*kwargs):

if instance and created:

self.scraper.last\_scraper\_save = datetime.datetime.now()

self.scraper.save()

class DummyItem(scrapy.Item):

tmp\_field = scrapy.Field()

from \_\_future\_\_ import unicode\_literals

from builtins import str

import datetime, json, logging, os

from jsonpath\_rw import jsonpath, parse

from jsonpath\_rw.lexer import JsonPathLexerError

from scrapy import signals

from scrapy.exceptions import CloseSpider

from scrapy.http import Request

from pydispatch import dispatcher

from dynamic\_scraper.spiders.django\_base\_spider import DjangoBaseSpider

from dynamic\_scraper.models import ScraperElem

from dynamic\_scraper.utils.scheduler import Scheduler

class DjangoChecker(DjangoBaseSpider):

name = "django\_checker"

mandatory\_vars = ['ref\_object', 'scraper',]

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(DjangoChecker, self).\_\_init\_\_(self, \*args, \*\*kwargs)

@classmethod

def from\_crawler(cls, crawler, \*args, \*\*kwargs):

spider = cls(\*args, \*\*kwargs)

spider.\_set\_crawler(crawler)

spider.\_set\_config(\*\*kwargs)

spider.\_check\_checker\_config()

spider.\_set\_request\_kwargs()

spider.\_set\_meta\_splash\_args()

spider.scheduler = Scheduler(spider.scraper.scraped\_obj\_class.checker\_scheduler\_conf)

dispatcher.connect(spider.response\_received, signal=signals.response\_received)

msg = "Checker for " + spider.ref\_object.\_\_class\_\_.\_\_name\_\_ + " \"" + str(spider.ref\_object) + "\" (" + str(spider.ref\_object.pk) + ") initialized."

spider.log(msg, logging.INFO)

return spider

def output\_usage\_help(self):

out = (

'',

'DDS Usage',

'=========',

' scrapy crawl [scrapy\_options] CHECKERNAME -a id=REF\_OBJECT\_ID [dds\_options]',

'',

'Options',

'-------',

'-a do\_action=(yes|no) Delete on checker success, default: no (Test Mode)',

'-L LOG\_LEVEL (scrapy option) Setting the log level for both Scrapy and DDS',

'-a run\_type=(TASK|SHELL) Simulate task based checker run, default: SHELL',

'-a output\_response\_body=(yes|no) Output response body content for debugging',

'',

)

for out\_str in out:

self.dds\_logger.info(out\_str)

def \_set\_config(self, \*\*kwargs):

log\_msg = ""

#output\_response\_body

if 'output\_response\_body' in kwargs and kwargs['output\_response\_body'] == 'yes':

self.conf['OUTPUT\_RESPONSE\_BODY'] = True

if len(log\_msg) > 0:

log\_msg += ", "

log\_msg += "output\_response\_body " + str(self.conf['OUTPUT\_RESPONSE\_BODY'])

else:

self.conf['OUTPUT\_RESPONSE\_BODY'] = False

super(DjangoChecker, self).\_set\_config(log\_msg, \*\*kwargs)

def \_check\_checker\_config(self):

if self.scraper.checker\_set.count() == 0:

msg = '{cs}No checkers defined for scraper.{ce}'.format(

cs=self.bcolors["INFO"], ce=self.bcolors["ENDC"])

self.dds\_logger.warning(msg)

self.output\_usage\_help()

raise CloseSpider(msg)

def \_del\_ref\_object(self):

if self.action\_successful:

self.log("Item already deleted, skipping.", logging.INFO)

return

from scrapy.utils.project import get\_project\_settings

settings = get\_project\_settings()

try:

img\_elem = self.scraper.get\_image\_elem()

if hasattr(self.ref\_object, img\_elem.scraped\_obj\_attr.name):

img\_name = getattr(self.ref\_object, img\_elem.scraped\_obj\_attr.name)

thumb\_paths = []

if settings.get('IMAGES\_THUMBS') and len(settings.get('IMAGES\_THUMBS')) > 0:

for key in settings.get('IMAGES\_THUMBS').keys():

thumb\_paths.append(('thumbnail, {k}'.format(k=key), os.path.join(settings.get('IMAGES\_STORE'), 'thumbs', key, img\_name),))

del\_paths = []

if self.conf['IMAGES\_STORE\_FORMAT'] == 'FLAT':

del\_paths.append(('original, flat path', os.path.join(settings.get('IMAGES\_STORE'), img\_name),))

if self.conf['IMAGES\_STORE\_FORMAT'] == 'ALL':

del\_paths.append(('original, full/ path', os.path.join(settings.get('IMAGES\_STORE'), 'full' , img\_name),))

del\_paths += thumb\_paths

if self.conf['IMAGES\_STORE\_FORMAT'] == 'THUMBS':

del\_paths += thumb\_paths

for path in del\_paths:

if os.access(path[1], os.F\_OK):

try:

os.unlink(path[1])

self.log("Associated image ({n}, {p}) deleted.".format(n=img\_name, p=path[0]), logging.INFO)

except Exception:

self.log("Associated image ({n}, {p}) could not be deleted!".format(n=img\_name, p=path[0]), logging.ERROR)

else:

self.log("Associated image ({n}, {p}) could not be found!".format(n=img\_name, p=path[0]), logging.WARNING)

except ScraperElem.DoesNotExist:

pass

self.ref\_object.delete()

self.scraper.last\_checker\_delete = datetime.datetime.now()

self.scraper.save()

self.action\_successful = True

self.log("{cs}Item deleted.{ce}".format(

cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"]), logging.INFO)

def start\_requests(self):

for checker in self.scraper.checker\_set.all():

url = getattr(self.ref\_object, checker.scraped\_obj\_attr.name)

rpt = self.scraper.get\_rpt\_for\_scraped\_obj\_attr(checker.scraped\_obj\_attr)

kwargs = self.dp\_request\_kwargs[rpt.page\_type].copy()

if 'meta' not in kwargs:

kwargs['meta'] = {}

kwargs['meta']['checker'] = checker

kwargs['meta']['rpt'] = rpt

self.\_set\_meta\_splash\_args()

if url:

if rpt.request\_type == 'R':

yield Request(url, callback=self.parse, method=rpt.method, dont\_filter=True, \*\*kwargs)

else:

yield FormRequest(url, callback=self.parse, method=rpt.method, formdata=self.dp\_form\_data[rpt.page\_type], dont\_filter=True, \*\*kwargs)

def response\_received(self, \*\*kwargs):

checker = kwargs['response'].request.meta['checker']

rpt = kwargs['response'].request.meta['rpt']

# 404 test

if kwargs['response'].status == 404:

if self.scheduler\_runtime.num\_zero\_actions == 0:

self.log("{cs}Checker test returned second 404 ({c}). Delete reason.{ce}".format(

c=str(checker), cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"]), logging.INFO)

if self.conf['DO\_ACTION']:

self.\_del\_ref\_object()

else:

self.log("{cs}Checker test returned first 404 ({c}).{ce}".format(

str(checker), cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"]), logging.INFO)

self.action\_successful = True

def parse(self, response):

# x\_path test

checker = response.request.meta['checker']

rpt = response.request.meta['rpt']

if self.conf['OUTPUT\_RESPONSE\_BODY']:

self.log("Response body ({url})\n\n\*\*\*\*\* RP\_START \*\*\*\*\*\n{resp\_body}\n\*\*\*\*\* RP\_END \*\*\*\*\*\n\n".format(

url=response.url,

resp\_body=response.body.decode('utf-8')), logging.INFO)

if checker.checker\_type == '4':

self.log("{cs}No 404 result ({c} checker type).{ce}".format(

c=str(checker), cs=self.bcolors["OK"], ce=self.bcolors["ENDC"]), logging.INFO)

if self.conf['DO\_ACTION']:

self.dds\_logger.info("{cs}Item kept.{ce}".format(

cs=self.bcolors["OK"], ce=self.bcolors["ENDC"]))

return

if rpt.content\_type == 'J':

json\_resp = json.loads(response.body\_as\_unicode())

try:

jsonpath\_expr = parse(checker.checker\_x\_path)

except JsonPathLexerError:

msg = "Invalid checker JSONPath ({c})!".format(c=str(checker))

self.dds\_logger.error(msg)

raise CloseSpider()

test\_select = [match.value for match in jsonpath\_expr.find(json\_resp)]

#self.log(unicode(test\_select), logging.INFO)

else:

try:

test\_select = response.xpath(checker.checker\_x\_path).extract()

except ValueError:

self.log("Invalid checker XPath ({c})!".format(c=str(checker)), logging.ERROR)

return

if len(test\_select) > 0 and checker.checker\_x\_path\_result == '':

self.log("{cs}Elements for XPath found on page (no result string defined) ({c}). Delete reason.{ce}".format(

c=str(checker), cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"]), logging.INFO)

if self.conf['DO\_ACTION']:

self.\_del\_ref\_object()

return

elif len(test\_select) > 0 and test\_select[0] == checker.checker\_x\_path\_result:

self.log("{cs}XPath result string '{s}' found on page ({c}). Delete reason.{ce}".format(

s=checker.checker\_x\_path\_result, c=str(checker), cs=self.bcolors["ERROR"], ce=self.bcolors["ENDC"]), logging.INFO)

if self.conf['DO\_ACTION']:

self.\_del\_ref\_object()

return

else:

self.log("{cs}XPath result string not found ({c}).{ce}".format(

c=str(checker), cs=self.bcolors["OK"], ce=self.bcolors["ENDC"]), logging.INFO)

if self.conf['DO\_ACTION']:

self.dds\_logger.info("{cs}Item kept.{ce}".format(

cs=self.bcolors["OK"], ce=self.bcolors["ENDC"]))

Return

from \_\_future\_\_ import unicode\_literals

from builtins import next

from builtins import str

from builtins import object

import hashlib, logging, ntpath

from dynamic\_scraper.models import ScraperElem

from django.utils.encoding import smart\_text

from scrapy.pipelines.images import ImagesPipeline

from scrapy.utils.python import to\_bytes

from scrapy.exceptions import DropItem

from scrapy.http import Request

from scrapy.utils.project import get\_project\_settings

settings = get\_project\_settings()

class DjangoImagesPipeline(ImagesPipeline):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(DjangoImagesPipeline, self).\_\_init\_\_(\*args, \*\*kwargs)

self.thumbs = settings.get('IMAGES\_THUMBS', {})

def get\_media\_requests(self, item, info):

try:

img\_elem = info.spider.scraper.get\_image\_elem()

if img\_elem.scraped\_obj\_attr.name in item and item[img\_elem.scraped\_obj\_attr.name]:

if not hasattr(self, 'conf'):

self.conf = info.spider.conf

return Request(item[img\_elem.scraped\_obj\_attr.name])

except (ScraperElem.DoesNotExist, TypeError):

pass

def file\_path(self, request, response=None, info=None):

url = request.url

image\_guid = hashlib.sha1(to\_bytes(url)).hexdigest()

if self.conf["IMAGES\_STORE\_FORMAT"] == 'FLAT':

return '{ig}.jpg'.format(ig=image\_guid)

elif self.conf["IMAGES\_STORE\_FORMAT"] == 'THUMBS':

return 'thumbs/{p}/{ig}.jpg'.format(p=next(iter(list(settings.get('IMAGES\_THUMBS').keys()))), ig=image\_guid)

else:

return 'full/{ig}.jpg'.format(ig=image\_guid)

def thumb\_path(self, request, thumb\_id, response=None, info=None):

url = request.url

image\_guid = hashlib.sha1(to\_bytes(url)).hexdigest()

if self.conf["IMAGES\_STORE\_FORMAT"] == 'FLAT':

return '{ig}.jpg'.format(ig=image\_guid)

else:

return 'thumbs/{p}/{ig}.jpg'.format(p=thumb\_id, ig=image\_guid)

def item\_completed(self, results, item, info):

try:

img\_elem = info.spider.scraper.get\_image\_elem()

except ScraperElem.DoesNotExist:

return item

results\_list = [x for ok, x in results if ok]

if len(results\_list) > 0:

item[img\_elem.scraped\_obj\_attr.name] = ntpath.basename(results\_list[0]['path'])

else:

item[img\_elem.scraped\_obj\_attr.name] = None

return item

class NoParsingFilter(logging.Filter):

def filter(self, record=True):

return False

class ValidationPipeline(object):

def process\_item(self, item, spider):

if spider.conf['CONSOLE\_LOG\_LEVEL'] != 'DEBUG':

logging.getLogger('scrapy.core.scraper').addFilter(NoParsingFilter())

#Process processor placeholders

for key, value in list(item.items()):

standard\_elems = spider.scraper.get\_standard\_elems()

for scraper\_elem in standard\_elems:

name = scraper\_elem.scraped\_obj\_attr.name

placeholder = '{' + name + '}'

value = smart\_text(value)

if not scraper\_elem.scraped\_obj\_attr.save\_to\_db:

if value and name in spider.non\_db\_results[id(item)] and \

spider.non\_db\_results[id(item)][name] != None and \

placeholder in value:

msg = "Applying placeholder {p} on {k}...".format(p=placeholder, k=key)

spider.log(msg, logging.DEBUG)

spider.log("Value before: " + value, logging.DEBUG)

value = value.replace(placeholder, str(spider.non\_db\_results[id(item)][name]))

spider.log("Value after: " + value, logging.DEBUG)

item[key] = value

else:

if value and name in item and \

item[name] != None and \

placeholder in value:

msg = "Applying placeholder {p} on {k}...".format(p=placeholder, k=key)

spider.log(msg, logging.DEBUG)

spider.log("Value before: " + value, logging.DEBUG)

value = value.replace(placeholder, str(item[name]))

spider.log("Value after: " + value, logging.DEBUG)

item[key] = value

idf\_elems = spider.scraper.get\_id\_field\_elems()

is\_double = item.\_is\_double

exist\_objects = spider.scraped\_obj\_class.objects

for idf\_elem in idf\_elems:

idf\_name = idf\_elem.scraped\_obj\_attr.name

if idf\_name in item and is\_double:

exist\_objects = exist\_objects.filter(\*\*{idf\_name:item[idf\_name]})

if is\_double:

mandatory\_elems = spider.scraper.get\_standard\_update\_elems()

else:

mandatory\_elems = spider.scraper.get\_mandatory\_scrape\_elems()

for elem in mandatory\_elems:

if elem.scraped\_obj\_attr.save\_to\_db and (\

not elem.scraped\_obj\_attr.name in item or\

(elem.scraped\_obj\_attr.name in item and not item[elem.scraped\_obj\_attr.name])):

msg = "{cs}Item {id} dropped, mandatory elem {elem} missing!{ce}".format(

id=item.\_dds\_id\_str, elem=elem.scraped\_obj\_attr.name, cs=spider.bcolors['ERROR'], ce=spider.bcolors['ENDC'])

spider.log(msg, logging.ERROR)

raise DropItem()

if spider.conf['MAX\_ITEMS\_SAVE'] and spider.items\_save\_count >= spider.conf['MAX\_ITEMS\_SAVE']:

spider.log("{cs}Max items save reached ({num}), item {id} not saved or further processed.{ce}".format(

num=str(spider.conf['MAX\_ITEMS\_SAVE']), id=item.\_dds\_id\_str, cs=spider.bcolors["INFO"], ce=spider.bcolors["ENDC"]), logging.INFO)

raise DropItem()

if not spider.conf['DO\_ACTION']:

spider.log("{cs}Item {id} not saved to Django DB (Test Mode or File Output).{ce}".format(

id=item.\_dds\_id\_str, cs=spider.bcolors["INFO"], ce=spider.bcolors["ENDC"]), logging.WARNING)

else:

if is\_double:

standard\_update\_elems = spider.scraper.get\_standard\_update\_elems()

updated\_attribute\_list = ''

if len(standard\_update\_elems) > 0 and len(exist\_objects) == 1:

exist\_object = exist\_objects[0]

dummy\_object = spider.scraped\_obj\_class()

for elem in standard\_update\_elems:

attr\_name = elem.scraped\_obj\_attr.name

if attr\_name in item and hasattr(exist\_object, attr\_name):

setattr(dummy\_object, attr\_name, item[attr\_name])

if str(getattr(dummy\_object, attr\_name)) != str(getattr(exist\_object, attr\_name)):

setattr(exist\_object, attr\_name, item[attr\_name])

if len(updated\_attribute\_list) > 0:

updated\_attribute\_list += ', '

updated\_attribute\_list += attr\_name

if len(updated\_attribute\_list) > 0:

exist\_object.save()

spider.action\_successful = True

msg = "{cs}Item {id} already in DB, attributes updated: {attr\_str}{ce}".format(

id=item.\_dds\_id\_str, attr\_str=updated\_attribute\_list, cs=spider.bcolors["OK"], ce=spider.bcolors["ENDC"])

spider.struct\_log(msg)

raise DropItem()

else:

msg = "{cs}Double item {id}, not saved.{ce}".format(

id=item.\_dds\_id\_str, cs=spider.bcolors["INFO"], ce=spider.bcolors["ENDC"])

spider.dds\_logger.warning(msg)

raise DropItem()

spider.items\_save\_count += 1

return item

from \_\_future\_\_ import unicode\_literals

from django.utils.encoding import python\_2\_unicode\_compatible

from builtins import range

from builtins import str

from builtins import object

import datetime

from django.db import models

from django.db.models import Q

@python\_2\_unicode\_compatible

class ScrapedObjClass(models.Model):

name = models.CharField(max\_length=200)

scraper\_scheduler\_conf = models.TextField(default='\

"MIN\_TIME": 15,\n\

"MAX\_TIME": 10080,\n\

"INITIAL\_NEXT\_ACTION\_FACTOR": 10,\n\

"ZERO\_ACTIONS\_FACTOR\_CHANGE": 20,\n\

"FACTOR\_CHANGE\_FACTOR": 1.3,\n')

checker\_scheduler\_conf = models.TextField(default='\

"MIN\_TIME": 1440,\n\

"MAX\_TIME": 10080,\n\

"INITIAL\_NEXT\_ACTION\_FACTOR": 1,\n\

"ZERO\_ACTIONS\_FACTOR\_CHANGE": 5,\n\

"FACTOR\_CHANGE\_FACTOR": 1.3,\n')

comments = models.TextField(blank=True)

def \_\_str\_\_(self):

return self.name

class Meta(object):

verbose\_name = "Scraped object class"

verbose\_name\_plural = "Scraped object classes"

ordering = ['name',]

@python\_2\_unicode\_compatible

class ScrapedObjAttr(models.Model):

ATTR\_TYPE\_CHOICES = (

('S', 'STANDARD'),

('T', 'STANDARD (UPDATE)'),

('B', 'BASE'),

('U', 'DETAIL\_PAGE\_URL'),

('I', 'IMAGE'),

)

name = models.CharField(max\_length=200)

order = models.IntegerField(default=100)

obj\_class = models.ForeignKey(ScrapedObjClass)

attr\_type = models.CharField(max\_length=1, choices=ATTR\_TYPE\_CHOICES)

id\_field = models.BooleanField(default=False)

save\_to\_db = models.BooleanField(default=True)

def \_\_str\_\_(self):

return self.name + " (" + str(self.obj\_class) + ")"

class Meta(object):

ordering = ['order',]

@python\_2\_unicode\_compatible

class Scraper(models.Model):

STATUS\_CHOICES = (

('A', 'ACTIVE'),

('M', 'MANUAL'),

('P', 'PAUSED'),

('I', 'INACTIVE'),

)

WORK\_STATUS\_CHOICES = (

('R2', 'REVISION NEEDED (MAJOR)'),

('R1', 'REVISION NEEDED (MINOR)'),

('UR', 'UNRESOLVED'),

('BR', 'BROKEN'),

('W', 'WORKING'),

('RC', 'RELEASE CANDIDATE'),

('BE', 'BETA'),

('A', 'ALPHA'),

('D', 'DRAFT'),

('S', 'SUSPENDED'),

('U', 'UNKNOWN'),

('N', 'NOT SET'),

)

CONTENT\_TYPE\_CHOICES = (

('H', 'HTML'),

('X', 'XML'),

('J', 'JSON'),

)

REQUEST\_TYPE\_CHOICES = (

('R', 'Request'),

('F', 'FormRequest'),

)

METHOD\_CHOICES = (

('GET', 'GET'),

('POST', 'POST'),

)

PAGINATION\_TYPE = (

('N', 'NONE'),

('R', 'RANGE\_FUNCT (+FOLLOW)'),

('F', 'FREE\_LIST (+FOLLOW)'),

('O', 'FOLLOW'),

)

name = models.CharField(max\_length=200)

scraped\_obj\_class = models.ForeignKey(ScrapedObjClass)

help\_text = "Runtime status of the scraper, used by scheduling mechanism."

status = models.CharField(max\_length=1, choices=STATUS\_CHOICES, default='P', help\_text=help\_text)

help\_text = "Internal work/progress status of the scraper."

work\_status = models.CharField(max\_length=2, choices=WORK\_STATUS\_CHOICES, default='N', help\_text=help\_text)

help\_text = "Optional owner when working on scrapers with various people"

owner = models.CharField(max\_length=12, blank=True, help\_text=help\_text)

max\_items\_read = models.IntegerField(blank=True, null=True, help\_text="Max number of items to be read (empty: unlimited).")

max\_items\_save = models.IntegerField(blank=True, null=True, help\_text="Max number of items to be saved (empty: unlimited).")

pagination\_type = models.CharField(max\_length=1, choices=PAGINATION\_TYPE, default='N')

pagination\_on\_start = models.BooleanField(default=False)

pagination\_append\_str = models.CharField(max\_length=200, blank=True, help\_text="Syntax: /somepartofurl/{page}/moreurlstuff.html")

pagination\_page\_replace = models.TextField(blank=True,

help\_text="RANGE\_FUNCT: uses Python range funct., syntax: [start], stop[, step], FREE\_LIST: 'Replace text 1', 'Some other text 2', 'Maybe a number 3', ...")

help\_text = "Optional, follow links from a single non-paginated or all statically paginated (RANGE\_FUNCT, FREE\_LIST) main pages"

follow\_pages\_url\_xpath = models.TextField(blank=True, help\_text=help\_text)

help\_text = "Optional additional XPath for the page number, can be used in {follow\_page} placeholder."

follow\_pages\_page\_xpath = models.TextField(blank=True, help\_text=help\_text)

help\_text = "Optionally limit number of pages to follow (default: follow until XPath fails)"

num\_pages\_follow = models.IntegerField(blank=True, null=True, help\_text=help\_text)

last\_scraper\_save\_alert\_period = models.CharField(max\_length=5, blank=True,

help\_text="Optional, used for scraper monitoring with 'check\_last\_scraper\_saves' management cmd, \

syntax: [HOURS]h or [DAYS]d or [WEEKS]w (e.g. '6h', '5d', '2w')")

next\_last\_scraper\_save\_alert = models.DateTimeField(default=datetime.datetime.now,

help\_text="Next time the last scraper save will be alerted, normally set on management cmd run.",)

last\_checker\_delete\_alert\_period = models.CharField(max\_length=5, blank=True,

help\_text="Optional, used for scraper monitoring with 'check\_last\_checker\_deletes' management cmd, \

syntax: [HOURS]h or [DAYS]d or [WEEKS]w (e.g. '6h', '5d', '2w')")

next\_last\_checker\_delete\_alert = models.DateTimeField(default=datetime.datetime.now,

help\_text="Next time the last checker delete will be alerted, normally set on management cmd run.",)

comments = models.TextField(blank=True)

last\_scraper\_save = models.DateTimeField(null=True, blank=True)

last\_checker\_delete = models.DateTimeField(null=True, blank=True)

def get\_alert\_period\_timedelta(self, attribute\_str):

if getattr(self, attribute\_str) and len(getattr(self, attribute\_str)) >= 2:

period\_str = getattr(self, attribute\_str)[-1]

num\_str = getattr(self, attribute\_str)[:-1]

if period\_str in ('h', 'd', 'w',):

try:

num\_int = int(num\_str)

if period\_str == 'h':

return datetime.timedelta(0, 0, 0, 0, 0, num\_int)

if period\_str == 'd':

return datetime.timedelta(num\_int)

if period\_str == 'w':

return datetime.timedelta(0, 0, 0, 0, 0, 0, num\_int)

except ValueError:

return None

else:

return None

else:

return None

def get\_last\_scraper\_save\_alert\_period\_timedelta(self):

return self.get\_alert\_period\_timedelta('last\_scraper\_save\_alert\_period')

def get\_last\_checker\_delete\_alert\_period\_timedelta(self):

return self.get\_alert\_period\_timedelta('last\_checker\_delete\_alert\_period')

def get\_main\_page\_rpt(self):

return self.requestpagetype\_set.get(page\_type='MP')

def get\_follow\_page\_rpts(self):

return self.requestpagetype\_set.filter(page\_type='FP')

def get\_detail\_page\_rpts(self):

return s.requestpagetype\_set.filter(~Q(page\_type='MP'))

def get\_rpt(self, page\_type):

return self.requestpagetype\_set.get(page\_type=page\_type)

def get\_rpt\_for\_scraped\_obj\_attr(self, soa):

return self.requestpagetype\_set.get(scraped\_obj\_attr=soa)

def get\_base\_elems(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='B')

def get\_base\_elem(self):

return self.scraperelem\_set.get(scraped\_obj\_attr\_\_attr\_type='B')

def get\_detail\_page\_url\_elems(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='U')

def get\_detail\_page\_url\_id\_elems(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='U', scraped\_obj\_attr\_\_id\_field=True)

def get\_standard\_elems(self):

q1 = Q(scraped\_obj\_attr\_\_attr\_type='S')

q2 = Q(scraped\_obj\_attr\_\_attr\_type='T')

return self.scraperelem\_set.filter(q1 | q2)

def get\_id\_field\_elems(self):

q1 = Q(scraped\_obj\_attr\_\_id\_field=True)

return self.scraperelem\_set.filter(q1)

def get\_standard\_fixed\_elems(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='S')

def get\_standard\_update\_elems(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='T')

def get\_standard\_update\_elems\_from\_detail\_pages(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='T').filter(~Q(request\_page\_type='MP'))

def get\_image\_elems(self):

return self.scraperelem\_set.filter(scraped\_obj\_attr\_\_attr\_type='I')

def get\_image\_elem(self):

return self.scraperelem\_set.get(scraped\_obj\_attr\_\_attr\_type='I')

def get\_scrape\_elems(self):

q1 = Q(scraped\_obj\_attr\_\_attr\_type='S')

q2 = Q(scraped\_obj\_attr\_\_attr\_type='T')

q3 = Q(scraped\_obj\_attr\_\_attr\_type='U')

q4 = Q(scraped\_obj\_attr\_\_attr\_type='I')

return self.scraperelem\_set.filter(q1 | q2 | q3 | q4)

def get\_mandatory\_scrape\_elems(self):

q1 = Q(scraped\_obj\_attr\_\_attr\_type='S')

q2 = Q(scraped\_obj\_attr\_\_attr\_type='T')

q3 = Q(scraped\_obj\_attr\_\_attr\_type='U')

q4 = Q(scraped\_obj\_attr\_\_attr\_type='I')

return self.scraperelem\_set.filter(q1 | q2 | q3 | q4).filter(mandatory=True)

def get\_from\_detail\_pages\_scrape\_elems(self):

return self.scraperelem\_set.filter(~Q(request\_page\_type='MP'))

def \_\_str\_\_(self):

return self.name + " (" + self.scraped\_obj\_class.name + ")"

class Meta(object):

ordering = ['name', 'scraped\_obj\_class',]

@python\_2\_unicode\_compatible

class RequestPageType(models.Model):

TYPE\_CHOICES = tuple([("MP", "Main Page"), ("FP", "Follow Page"),] + [("DP{n}".format(n=str(n)), "Detail Page {n}".format(n=str(n))) for n in list(range(1, 26))])

CONTENT\_TYPE\_CHOICES = (

('H', 'HTML'),

('X', 'XML'),

('J', 'JSON'),

)

REQUEST\_TYPE\_CHOICES = (

('R', 'Request'),

('F', 'FormRequest'),

)

METHOD\_CHOICES = (

('GET', 'GET'),

('POST', 'POST'),

)

help\_text = "One main page RPT, an optional follow page RPT (if follow pagination is used) and detail page RPTs for all DETAIL\_PAGE\_URLs"

page\_type = models.CharField(max\_length=3, choices=TYPE\_CHOICES, help\_text=help\_text)

scraped\_obj\_attr = models.ForeignKey(ScrapedObjAttr, blank=True, null=True, help\_text="Empty for main page, attribute of type DETAIL\_PAGE\_URL scraped from main page for detail pages.")

scraper = models.ForeignKey(Scraper)

content\_type = models.CharField(max\_length=1, choices=CONTENT\_TYPE\_CHOICES, default='H', help\_text="Data type format for scraped pages of page type (for JSON use JSONPath instead of XPath)")

render\_javascript = models.BooleanField(default=False, help\_text="Render Javascript on pages (ScrapyJS/Splash deployment needed, careful: resource intense)")

request\_type = models.CharField(max\_length=1, choices=REQUEST\_TYPE\_CHOICES, default='R', help\_text="Normal (typically GET) request (default) or form request (typically POST), using Scrapys corresponding request classes (not used for checker).")

method = models.CharField(max\_length=10, choices=METHOD\_CHOICES, default='GET', help\_text="HTTP request via GET or POST.")

headers = models.TextField(blank=True, help\_text='Optional HTTP headers sent with each request, provided as a JSON dict (e.g. {"Referer":"http://referer\_url"}, use double quotes!)), can use {main page attribute}, {page} and {follow\_page} placeholders.')

body = models.TextField(blank=True, help\_text="Optional HTTP message body provided as a unicode string, can use {main page attribute}, {page} and {follow\_page} placeholders.")

cookies = models.TextField(blank=True, help\_text="Optional cookies as JSON dict (use double quotes!), can use {main page attribute}, {page} and {follow\_page} placeholders.")

meta = models.TextField(blank=True, help\_text="Optional Scrapy meta attributes as JSON dict (use double quotes!), see Scrapy docs for reference.")

form\_data = models.TextField(blank=True, help\_text="Optional HTML form data as JSON dict (use double quotes!), only used with FormRequest request type, can use {main page attribute}, {page} and {follow\_page} placeholders.")

dont\_filter = models.BooleanField(default=False, help\_text="Do not filter duplicate requests, useful for some scenarios with requests falsely marked as being duplicate (e.g. uniform URL + pagination by HTTP header).")

comments = models.TextField(blank=True)

def \_\_str\_\_(self):

ret\_str = self.get\_page\_type\_display()

if self.scraped\_obj\_attr:

ret\_str += ' (' + str(self.scraped\_obj\_attr) + ')'

return ret\_str

@python\_2\_unicode\_compatible

class Checker(models.Model):

CHECKER\_TYPE = (

('4', '404'),

('X', '404\_OR\_X\_PATH'),

)

scraped\_obj\_attr = models.ForeignKey(ScrapedObjAttr, help\_text="Attribute of type DETAIL\_PAGE\_URL, several checkers for same DETAIL\_PAGE\_URL attribute possible.")

scraper = models.ForeignKey(Scraper)

checker\_type = models.CharField(max\_length=1, choices=CHECKER\_TYPE, default='4')

checker\_x\_path = models.TextField(blank=True)

checker\_x\_path\_result = models.TextField(blank=True)

checker\_ref\_url = models.URLField(max\_length=500, blank=True)

comments = models.TextField(blank=True)

def \_\_str\_\_(self):

return str(self.scraped\_obj\_attr) + ' > ' + self.get\_checker\_type\_display()

@python\_2\_unicode\_compatible

class ScraperElem(models.Model):

REQUEST\_PAGE\_TYPE\_CHOICES = tuple([("MP", "Main Page")] + [("DP{n}".format(n=str(n)), "Detail Page {n}".format(n=str(n))) for n in list(range(1, 26))])

help\_text = "The different attributes to be scraped, exactly one attribute of type BASE necessary."

scraped\_obj\_attr = models.ForeignKey(ScrapedObjAttr, help\_text=help\_text)

scraper = models.ForeignKey(Scraper)

x\_path = models.TextField(blank=True, help\_text='XPath or JSONPath expression, leave blank on "static" processor use.')

reg\_exp = models.TextField(blank=True, help\_text="Optional filtering by regular expression (e.g. 'Scrape only (.\*) the text in between').")

help\_text = "Corresponding Request Page Types created for this scraper."

request\_page\_type = models.CharField(max\_length=3, choices=REQUEST\_PAGE\_TYPE\_CHOICES, default="MP", help\_text=help\_text)

help\_text = "Use the default processors (Scrapy TakeFirst, DDS string\_strip) for convenience."

use\_default\_procs = models.BooleanField(default=True, help\_text=help\_text)

help\_text = 'Optional comma-separated list of processors used (e.g. "pre\_url, post\_string").'

processors = models.TextField(blank=True, help\_text=help\_text)

help\_text = "Comma-separated aditional context (depending on processor) (e.g. 'pre\_url': 'http://append\_before.org/', 'post\_string': '?append\_after=True')."

proc\_ctxt = models.TextField(blank=True, help\_text=help\_text)

help\_text = "Drop item if attribute could not be scraped."

mandatory = models.BooleanField(default=True, help\_text=help\_text)

def \_\_str\_\_(self):

return '{s} > {soa} Attribute ({rpt})'.format(

s=str(self.scraper),

soa=self.scraped\_obj\_attr.name,

rpt=self.get\_request\_page\_type\_display())

class Meta(object):

ordering = ['scraped\_obj\_attr\_\_order',]

@python\_2\_unicode\_compatible

class SchedulerRuntime(models.Model):

TYPE = (

('S', 'SCRAPER'),

('C', 'CHECKER'),

)

runtime\_type = models.CharField(max\_length=1, choices=TYPE, default='P')

next\_action\_time = models.DateTimeField(default=datetime.datetime.now)

next\_action\_factor = models.FloatField(blank=True, null=True)

num\_zero\_actions = models.IntegerField(default=0)

def \_\_str\_\_(self):

return str(self.id)

class Meta(object):

ordering = ['next\_action\_time',]

class LogMarker(models.Model):

TYPE\_CHOICES = (

('PE', 'Planned Error'),

('DD', 'Dirty Data'),

('IM', 'Important'),

('IG', 'Ignore'),

('MI', 'Miscellaneous'),

('CU', 'Custom'),

)

message\_contains = models.CharField(max\_length=255)

help\_text = "Use the string format from the log messages"

ref\_object = models.CharField(max\_length=200, blank=True)

help\_text = 'Choose "Custom" and enter your own type in the next field for a custom type'

mark\_with\_type = models.CharField(max\_length=2, choices=TYPE\_CHOICES, help\_text=help\_text)

custom\_type = models.CharField(max\_length=25, blank=True)

spider\_name = models.CharField(max\_length=200, blank=True)

scraper = models.ForeignKey(Scraper, blank=True, null=True)

class Log(models.Model):

LEVEL\_CHOICES = (

(50, 'CRITICAL'),

(40, 'ERROR'),

(30, 'WARNING'),

(20, 'INFO'),

(10, 'DEBUG'),

)

message = models.CharField(max\_length=255)

ref\_object = models.CharField(max\_length=200)

type = models.CharField(max\_length=25, blank=True)

level = models.IntegerField(choices=LEVEL\_CHOICES)

spider\_name = models.CharField(max\_length=200)

scraper = models.ForeignKey(Scraper, blank=True, null=True)

date = models.DateTimeField(default=datetime.datetime.now)

@staticmethod

def numeric\_level(level):

numeric\_level = 0

for choice in Log.LEVEL\_CHOICES:

if choice[1] == level:

numeric\_level = choice[0]

return numeric\_level

class Meta(object):

ordering = ['-date']