django-dynamic-scraper Documentation

Release 0.11-beta

Holger Drewes

May 13, 2016

Contents

Django Dynamic Scraper (DDS) is an app for Django build on top of the scraping framework Scrapy. While preserving many of the features of Scrapy it lets you dynamically create and manage spiders via the Django admin interface.

Note: Lot's of new features added recently :

- `Django 1.9`/`Scrapy 1.1` support
- Beta `Python 3` support
- `Javascript` rendering
- Scraping `JSON` content
- More flexible ID and detail page URL(s) concept
- Several checkers for a single scraper
- Custom `HTTP Header/Body`, `Cookies`, `GET/POST` requests
- `Scrapy Meta` attributes
- Scraper/Checker `Monitoring`

See *Release Notes* for further details!

CHAPTER 1

Features

- Create and manage scrapers for your Django models in the Django admin interface
- Many features of Scrapy like regular expressions, processors, pipelines (see Scrapy Docs)
- Image/screenshot scraping
- Dynamic scheduling depending on crawling success via Django Celery
- Checkers to check if items once scraped are still existing

CHAPTER 2

User Manual

2.1 Introduction

With Django Dynamic Scraper (DDS) you can define your Scrapy scrapers dynamically via the Django admin interface and save your scraped items in the database you defined for your Django project. Since it simplifies things DDS is not usable for all kinds of scrapers, but it is well suited for the relatively common case of regularly scraping a website with a list of updated items (e.g. news, events, etc.) and than dig into the detail page to scrape some more infos for each item.

Here are some examples for some use cases of DDS: Build a scraper for ...

- Local music events for different event locations in your city
- New organic recipes for asian food
- The latest articles from blogs covering fashion and style in Berlin
- ...Up to your imagination! :-)

Django Dynamic Scraper tries to keep its data structure in the database as separated as possible from the models in your app, so it comes with its own Django model classes for defining scrapers, runtime information related to your scraper runs and classes for defining the attributes of the models you want to scrape. So apart from a few foreign key relations your Django models stay relatively independent and you don't have to adjust your model code every time DDS's model structure changes.

The DDS repository on GitHub contains an example project in the `example_project` folder, showing how to create a scraper for open news content on the web (starting with Wikinews from Wikipedia). The source code from this example is used in the *Getting started* guide.

2.2 Installation

2.2.1 Requirements

The **basic requirements** for Django Dynamic Scraper are:

- Python 2.7+ or Python 3.4+
- Django 1.8/1.9 (newer versions untested)
- Scrapy 1.1 (older versions like `0.24` not supported any more!)
- scrapy-djangoitem 1.1

- Python JSONPath RW 1.4+
- Python-Future (preparing the code base to run with Python 2/3) 0.15+

If you want to use the **scheduling mechanism** of DDS you also have to install `django-celery`:

- django-celery 3.1.17 (newer versions untested)

For **scraping images** you will need the Pillow Library:

- Pillow Libray (PIL fork) 2.5+

Since `v.0.4.1` `DDS` has basic `ScrapyJS/Splash` support for rendering/processing `Javascript` before scraping the page. For this to work you have to install and configure (see: *Setting up ScrapyJS/Splash (Optional)*) `ScrapyJS`:

- *scrapy-splash <https://github.com/scrapy-plugins/scrapy-splash>* 0.6

2.2.2 Release Compatibility Table

Have a look at the following table for an overview which `Django`, `Scrapy`, `Python` and `django-celery` versions are supported by which `DDS` version. Due to dev resource constraints backwards compatibility for older `Django` or `Scrapy` releases for new `DDS` releases normally can not be granted.

DDS Version	Django	Scrapy	Python	django-celery/Celery/Kombu
0.11	1.8/1.9	1.1	2.7+/3.4+	3.1.17/3.1.20/3.0.33
0.4-0.9	1.7/1.8	0.22/0.24	2.7	3.1.16 (newer untested)
0.3	1.4-1.6	0.16/0.18	2.7	3.0+ (3.1+ untested)
0.2	1.4	0.14	2.7	(3.0 untested)

Note: Please get in touch (GitHub) if you have any additions to this table. A library version is counted as supported if the DDS testsuite is running through (see: *Running the test suite*).

2.2.3 Installation with Pip

Django Dynamic Scraper can be found on the PyPI Package Index (see package description). For the installation with Pip, first install the requirements above. Then install DDS with:

```
pip install django-dynamic-scraper
```

2.2.4 Manual Installation

For manually installing Django Dynamic Scraper download the DDS source code from GitHub or clone the project with git into a folder of your choice:

```
git clone https://github.com/holgerd77/django-dynamic-scraper.git .
```

Then you have to met the requirements above. You can do this by manually installing the libraries you need with `pip` or `easy_install`, which may be a better choice if you e.g. don't want to risk your Django installation to be touched during the installation process. However if you are sure that there is no danger ahead or if you are running DDS in a new `virtualenv` environment, you can install all the requirements above together with:

```
pip install -r requirements.txt
```

Then either add the `dynamic_scraper` folder to your `PYTHONPATH` or your project manually or install DDS with:

```
python setup.py install
```

Note, that the requirements are NOT included in the `setup.py` script since this caused some problems when testing the installation and the requirements installation process with `pip` turned out to be more stable.

Now, to use DDS in your Django project add `'dynamic_scraper'` to your `INSTALLED_APPS` in your project settings.

2.2.5 Setting up Scrapy

Scrapy Configuration

For getting Scrapy to work the recommended way to start a new Scrapy project normally is to create a directory and template file structure with the `scrapy startproject myscrapyproject` command on the shell first. However, there is (initially) not so much code to be written left and the directory structure created by the `startproject` command cannot really be used when connecting Scrapy to the Django Dynamic Scraper library. So the easiest way to start a new scrapy project is to just manually add the `scrapy.cfg` project configuration file as well as the Scrapy `settings.py` file and adjust these files to your needs. It is recommended to just create the Scrapy project in the same Django app you used to create the models you want to scrape and then place the modules needed for scrapy in a sub package called `scraper` or something similar. After finishing this chapter you should end up with a directory structure similar to the following (again illustrated using the open news example):

```
example_project/
  scrapy.cfg
  open_news/
    models.py # Your models.py file
    scraper/
      settings.py
      spiders.py
      (checkers.py)
      pipelines.py
      (tasks.py)
```

Your `scrapy.cfg` file should look similar to the following, just having adjusted the reference to the settings file and the project name:

```
[settings]
default = open_news.scraper.settings

#Scrapy till 0.16
[deploy]
#url = http://localhost:6800/
project = open_news

#Scrapy with separate scrapyd (0.18+)
[deploy:scrapyd1]
url = http://localhost:6800/
project = open_news
```

And this is your `settings.py` file:

```
import os

PROJECT_ROOT = os.path.abspath(os.path.dirname(__file__))
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "example_project.settings") #Changed in DDS v.0.3

BOT_NAME = 'open_news'
```

```
SPIDER_MODULES = ['dynamic_scraper.spiders', 'open_news.scraper',]
USER_AGENT = '%s/%s' % (BOT_NAME, '1.0')

#Scrapy 0.20+
ITEM_PIPELINES = {
    'dynamic_scraper.pipelines.ValidationPipeline': 400,
    'open_news.scraper.pipelines.DjangoWriterPipeline': 800,
}

#Scrapy up to 0.18
ITEM_PIPELINES = [
    'dynamic_scraper.pipelines.ValidationPipeline',
    'open_news.scraper.pipelines.DjangoWriterPipeline',
]
```

The `SPIDER_MODULES` setting is referencing the basic spiders of DDS and our `scraper` package where Scrapy will find the (yet to be written) spider module. For the `ITEM_PIPELINES` setting we have to add (at least) two pipelines. The first one is the mandatory pipeline from DDS, doing stuff like checking for the mandatory attributes we have defined in our scraper in the DB or preventing double entries already existing in the DB (identified by the url attribute of your scraped items) to be saved a second time.

Setting up ScrapyJS/Splash (Optional)

More and more webpages only show their full information load after various `Ajax` calls and/or `Javascript` function processing. For being able to scrape those websites `DDS` supports `ScrapyJS/Spash` starting with `v.0.4.1` for basic JS rendering/processing.

For this to work you have to install `Splash` (the Javascript rendering service) installed - probably via `Docker`- (see installation instructions), and then `ScrapyJS` with:

```
pip install scrapyjs
```

Afterwards follow the configuration instructions on the ScrapyJS GitHub page.

For customization of `Splash` args `DSCRAPER_SPLASH_ARGS` setting can be used (see: *Settings*).

ScrapyJS can later be used via activating it for certain scrapers in the corresponding `Django Admin` form.

Note: Resources needed for completely rendering a website on your scraping machine are vastly larger then for just requesting/working on the plain HTML text without further processing, so make use of `ScrapyJS/Splash` capability on when needed!

2.3 Getting started

2.3.1 Creating your Django models

Create your model classes

When you want to build a Django app using Django Dynamic Scraper to fill up your models with data you have to provide *two model classes*. The *first class* stores your scraped data, in our news example this is a class called `Article` storing articles scraped from different news websites. The *second class* is a reference class for this first model class, defining where the scraped items belong to. Often this class will represent a website, but it could also represent a category, a topic or something similar. In our news example we call the class `NewsWebsite`. Below is the code for this two model classes:

```
from django.db import models
from dynamic_scraper.models import Scraper, SchedulerRuntime
from scrapy_djangoitem import DjangoItem


class NewsWebsite(models.Model):
    name = models.CharField(max_length=200)
    url = models.URLField()
    scraper = models.ForeignKey(Scraper, blank=True, null=True, on_delete=models.SET_NULL)
    scraper_runtime = models.ForeignKey(SchedulerRuntime, blank=True, null=True, on_delete=models.SE

    def __unicode__(self):
        return self.name


class Article(models.Model):
    title = models.CharField(max_length=200)
    news_website = models.ForeignKey(NewsWebsite)
    description = models.TextField(blank=True)
    url = models.URLField()
    checker_runtime = models.ForeignKey(SchedulerRuntime, blank=True, null=True, on_delete=models.SE

    def __unicode__(self):
        return self.title


class ArticleItem(DjangoItem):
    django_model = Article
```

As you can see, there are some foreign key fields defined in the models referencing DDS models. The `NewsWebsite` class has a reference to the *Scraper* DDS model, which contains the main scraper with information about how to scrape the attributes of the article objects. The `scraper_runtime` field is a reference to the *SchedulerRuntime* class from the DDS models. An object of this class stores scheduling information, in this case information about when to run a news website scraper for the next time. The `NewsWebsite` class also has to provide the url to be used during the scraping process. You can either use (if existing) the representative url field of the model class, which is pointing to the nicely-layouted overview news page also visited by the user. In this case we are choosing this way with taking the `url` attribute of the model class as the scrape url. However, it often makes sense to provide a dedicated `scrape_url` (you can name the attribute freely) field for cases, when the representative url differs from the scrape url (e.g. if list content is loaded via ajax, or if you want to use another format of the content - e.g. the rss feed - for scraping).

The `Article` class to store scraped news articles also has a reference to the *SchedulerRuntime* DDS model class called `checker_runtime`. In this case the scheduling object holds information about the next existance check (using the `url` field from `Article`) to evaluate if the news article still exists or if it can be deleted (see *Defining item checkers*).

Last but not least: Django Dynamic Scraper uses the DjangoItem class from Scrapy for being able to directly store the scraped data into the Django DB. You can store the item class (here: `ArticleItem`) telling Scrapy which model class to use for storing the data directly underneath the associated model class.

Note: For having a loose coupling between your runtime objects and your domain model objects you should declare the foreign keys to the DDS objects with the `blank=True, null=True, on_delete=models.SET_NULL` field options. This will prevent a cascading delete of your reference object as well as the associated scraped objects when a DDS object is deleted accidentally.

Deletion of objects

If you delete model objects via the Django admin interface, the runtime objects are not deleted as well. If you want this to happen, you can use Django's pre_delete signals in your `models.py` to delete e.g. the `checker_runtime` when deleting an article:

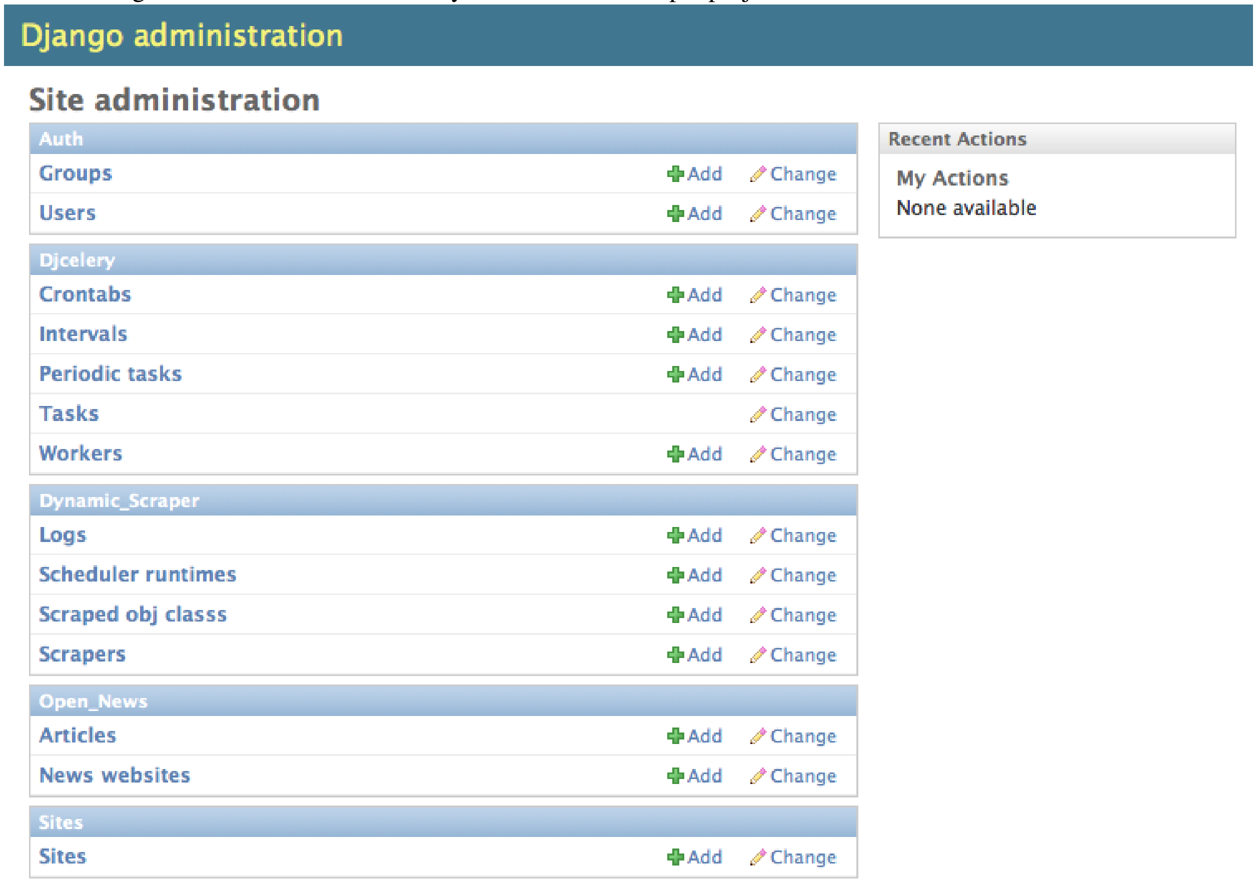
```
@receiver(pre_delete)
def pre_delete_handler(sender, instance, using, **kwargs):
    ....

    if isinstance(instance, Article):
        if instance.checker_runtime:
            instance.checker_runtime.delete()

pre_delete.connect(pre_delete_handler)
```

2.3.2 Defining the object to be scraped

If you have done everything right up till now and even synced your DB :-) your Django admin should look similar to the following screenshot below, at least if you follow the example project:



Django administration

Site administration

Auth		
Groups	Add	Change
Users	Add	Change
Djcelery		
Crontabs	Add	Change
Intervals	Add	Change
Periodic tasks	Add	Change
Tasks		Change
Workers	Add	Change
Dynamic_Scraper		
Logs	Add	Change
Scheduler runtimes	Add	Change
Scraped obj classs	Add	Change
Scrapers	Add	Change
Open_News		
Articles	Add	Change
News websites	Add	Change
Sites		
Sites	Add	Change

Recent Actions

My Actions

None available

Before being able to create scrapers in Django Dynamic Scraper you have to define which parts of the Django model class you defined above should be filled by your scraper. This is done via creating a new *ScrapedObjClass* in your Django admin interface and then adding several *ScrapedObjAttr* datasets to it, which is done inline in the form for the *ScrapedObjClass*. All attributes for the object class which are marked as to be saved to the database have to be named like the attributes in your model class to be scraped. In our open news example we want the title, the description, and the url of an Article to be scraped, so we add these attributes with the corresponding names to the scraped obj class.

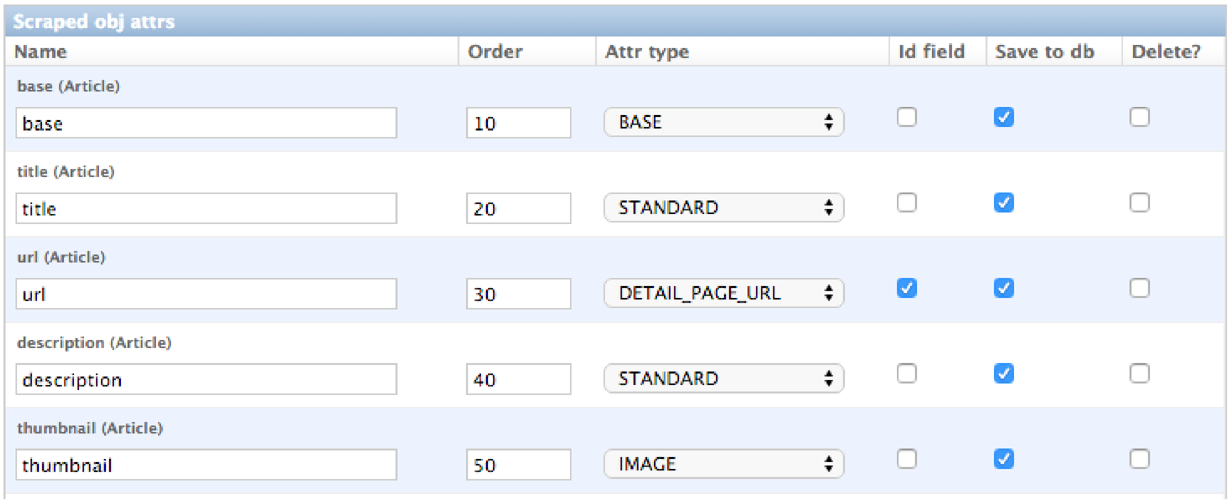
The reason why we are redefining these attributes here, is that we can later define x_path elements for each of theses attributes dynamically in the scrapers we want to create. When Django Dynamic Scraper is scraping items, the **general workflow of the scraping process** is as follows:

- The DDS scraper is scraping base elements from the overview page of items beeing scraped, with each base element encapsulating an item summary, e.g. in our open news example an article summary containing the title of the article, a screenshot and a short description. The encapsuling html tag often is a `div`, but could also be a `td` tag or something else.
- If provided the DDS scraper is then scraping the url from this item summary block leading to a detail page of the item providing more information to scrape
- All the real item attributes (like a title, a description, a date or an image) are then scraped either from within the item summary block on the overview page or from a detail page of the item. This can be defined later when creating the scraper itself.

To define which of the scraped obj attributes are just simple standard attributes to be scraped, which one is the base attribute (this is a bit of an artificial construct) and which one eventually is a url to be followed later, we have to choose an attribute type for each attribute defined. There is a choice between the following types (taken from `dynamic_scraper.models.ScrapedObjAttr`):

```
ATTR_TYPE_CHOICES = (
    ('S', 'STANDARD'),
    ('T', 'STANDARD (UPDATE)'),
    ('B', 'BASE'),
    ('U', 'DETAIL_PAGE_URL'),
    ('I', 'IMAGE'),
)
```

`STANDARD`, `BASE` and `DETAIL_PAGE_URL` should be clear by now, `STANDARD (UPDATE)` behaves like `STANDARD`, but these attributes are updated with the new values if the item is already in the DB. `IMAGE` represents attributes which will hold images or screenshots. So for our open news example we define a base attribute called 'base' with type `BASE`, two standard elements 'title' and 'description' with type `STANDARD` and a url field called 'url' with type `DETAIL_PAGE_URL`. Your definition form for your scraped obj class should look similar to the screenshot below:


Scraped obj attrs

Name	Order	Attr type	Id field	Save to db	Delete?
base (Article): base	10	BASE	☐	✓	☐
title (Article): title	20	STANDARD	☐	✓	☐
url (Article): url	30	DETAIL_PAGE_URL	✓	✓	☐
description (Article): description	40	STANDARD	☐	✓	☐
thumbnail (Article): thumbnail	50	IMAGE	☐	✓	☐

To prevent double entries in the DB you also have to set one or more object attributes of type `STANDARD` or `DETAIL_PAGE_URL` as `ID Fields`. If you provide a `DETAIL_PAGE_URL` for your object scraping, it is often a good idea to use this also as an `ID Field`, since the different URLs for different objects should be unique by definition in most cases. Using a single `DETAIL_PAGE_URL` ID field is also prerequisite if you want to use the checker functionality (see: *Defining item checkers*) of DDS for dynamically detecting and deleting items not existing any more.

Also note that these `ID Fields` just provide unique identification of an object for within the scraping process. In your model class defined in the chapter above you can use other ID fields or simply use a classic numerical auto-incremented ID provided by your database.

Note: If you define an attribute as `STANDARD (UPDATE)` attribute and your scraper reads the value for this attribute from the detail page of the item, your scraping process requires **much more page requests**, because the scraper has to look at all the detail pages even for items already in the DB to compare the values. If you don't use the update functionality, use the simple `STANDARD` attribute instead!

Note: The `order` attribute for the different object attributes is just for convenience and determines the order of the attributes when used for defining `XPaths` in your scrapers. Use 10-based or 100-based steps for easier resorting (e.g. '100', '200', '300', ...).

2.3.3 Defining your scrapers

General structure of a scraper

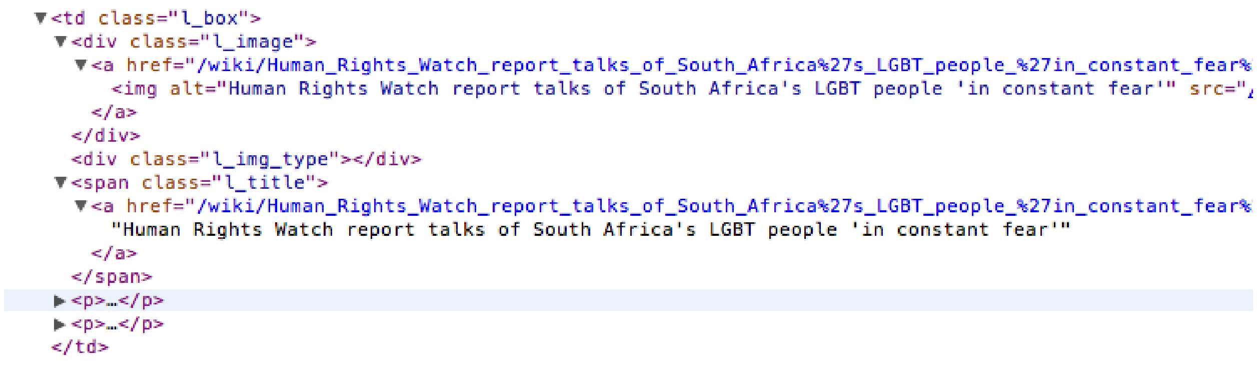
Scrapers for Django Dynamic Scraper are also defined in the Django admin interface. You first have to give the scraper a name and select the associated *ScrapedObjClass*. In our open news example we call the scraper 'Wikinews Scraper' and select the *ScrapedObjClass* named 'Article' defined above.

The main part of defining a scraper in DDS is to create several scraper elements, each connected to a *ScrapedObjAttr* from the selected *ScrapedObjClass*. Each scraper element define how to extract the data for the specific *ScrapedObjAttr* by following the main concepts of Scrapy for scraping data from websites. In the fields named 'x_path' and 'reg_exp' an XPath and (optionally) a regular expression is defined to extract the data from the page, following Scrapy's concept of XPathSelectors. The 'request_page_type' select box tells the scraper, if the data for the object attibute for the scraper element should be extracted from the overview page or a detail page of the specific item. For every chosen page type here you have to define a corresponding request page type in the admin form above. The fields 'processors' and 'processors_ctxt' are used to define output processors for your scraped data like they are defined in Scrapy's Item Loader section. You can use these processors e.g. to add a string to your scraped data or to bring a scraped date in a common format. More on this later. Finally, the 'mandatory' check box is indicating whether the data scraped by the scraper element is a necessary field. If you define a scraper element as necessary and no data could be scraped for this element the item will be dropped. Note, that you always have to keep attributes mandatory, if the corresponding attributes of your domain model class is a mandatory field, otherwise the scraped item can't be saved in the DB.

For the moment, keep the `status` to `MANUAL` to run the spider via the command line during this tutorial. Later you will change it to `ACTIVE`.

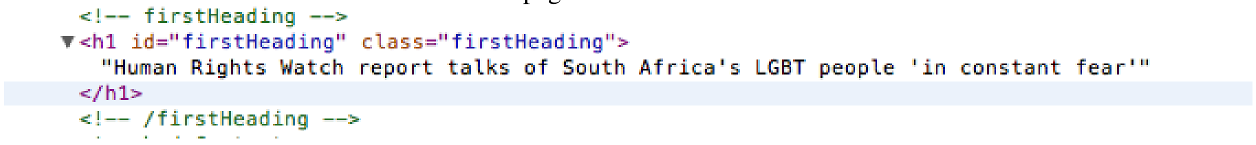
Creating the scraper of our open news example

Let's use the information above in the context of our Wikinews example. Below you see a screenshot of an html code extract from the Wikinews overview page like it is displayed by the developer tools in Google's Chrome browser:



```
▼<td class="l_box">
  ▼<div class="l_image">
    ▼<a href="/wiki/Human_Rights_Watch_report_talks_of_South_Africa%27s_LGBT_people_%27in_constant_fear%
       <img alt="Human Rights Watch report talks of South Africa's LGBT people 'in constant fear'" src="/
     </a>
   </div>
   <div class="l_img_type"></div>
  ▼<span class="l_title">
    ▼<a href="/wiki/Human_Rights_Watch_report_talks_of_South_Africa%27s_LGBT_people_%27in_constant_fear%
       "Human Rights Watch report talks of South Africa's LGBT people 'in constant fear'"
     </a>
   </span>
  ▶<p>…</p>
  ▶<p>…</p>
 </td>
```

The next screenshot is from a news article detail page:



```
<!-- firstHeading -->
▼<h1 id="firstHeading" class="firstHeading">
   "Human Rights Watch report talks of South Africa's LGBT people 'in constant fear'"
 </h1>
 <!-- /firstHeading -->
```

We will use these code snippets in our examples.

Note: If you don't want to manually create the necessary DB objects for the example project, you can also run `python manage.py loaddata open_news/open_news_dds_[DDS_VERSION].json` from within the `example_project` directory in your favorite shell to have all the objects necessary for the example created automatically. Use the file closest to the current DDS version. If you run into problems start installing the fitting DDS version for the fixture, then update the DDS version and apply the latest Django migrations.

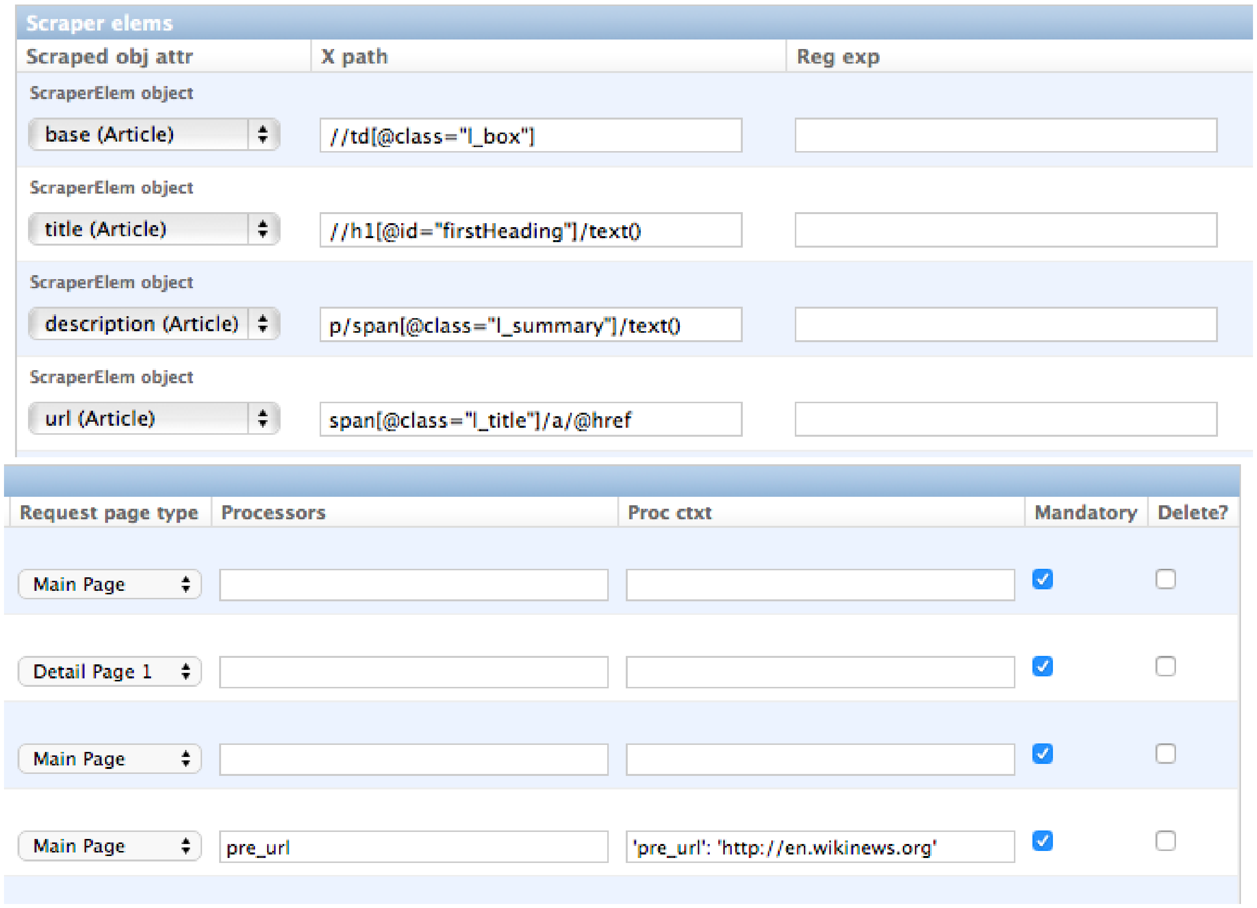
Note: The WikiNews site changes its code from time to time. I will try to update the example code and text in the docs, but I won't keep pace with the screenshots so they can differ slightly compared to the real world example.

1. First we have to define a base scraper element to get the enclosing DOM elements for news item summaries. On the Wikinews overview page all news summaries are enclosed by `<td>` tags with a class called 'l_box', so `//td[@class="l_box"]` should do the trick. We leave the rest of the field for the scraper element on default.

2. It is not necessary but just for the purpose of this example let's scrape the title of a news article from the article detail page. On an article detail page the headline of the article is enclosed by a `<h1>` tag with an id named 'firstHeading'. So `//h1[@id="firstHeading"]/span/text()` should give us the headline. Since we want to scrape from the detail page, we have to activate the 'from_detail_page' check box.

3. All the standard elements we want to scrape from the overview page are defined relative to the base element. Therefore keep in mind to leave the trailing double slashes of XPath definitions. We scrape the short description of a news item from within a `<span>` tag with a class named 'l_summary'. So the XPath is `p/span[@class="l_summary"]/text()`.

4. And finally the url can be scraped via the XPath `span[@class="l_title"]/a/@href`. Since we only scrape the path of our url with this XPath and not the domain, we have to use a processor for the first time to complete the url. For this purpose there is a predefined processor called 'pre_url'. You can find more predefined processors in the `dynamic_scraper.utils.processors` module. 'pre_url' is simply doing what we want, namely adding a base url string to the scraped string. To use a processor, just write the function name in the processor field. Processors can be given some extra information via the processors_ctxt field. In our case we need the spefic base url our scraped string should be appended to. Processor context information is provided in a dictionary like form: `'processor_name': 'context'`, in our case: `'pre_url': 'http://en.wikinews.org'`. Together with our scraped string this will create the complete url.


Scraper elems

Scraped obj attr	X path	Reg exp
ScraperElem object		
base (Article)	//td[@class="l_box"]	
ScraperElem object		
title (Article)	//h1[@id="firstHeading"]/text()	
ScraperElem object		
description (Article)	p/span[@class="l_summary"]/text()	
ScraperElem object		
url (Article)	span[@class="l_title"]/a/@href	

Request page type	Processors	Proc ctxt	Mandatory	Delete?
Main Page			✓	☐
Detail Page 1			✓	☐
Main Page			✓	☐
Main Page	pre_url	'pre_url': 'http://en.wikinews.org'	✓	☐

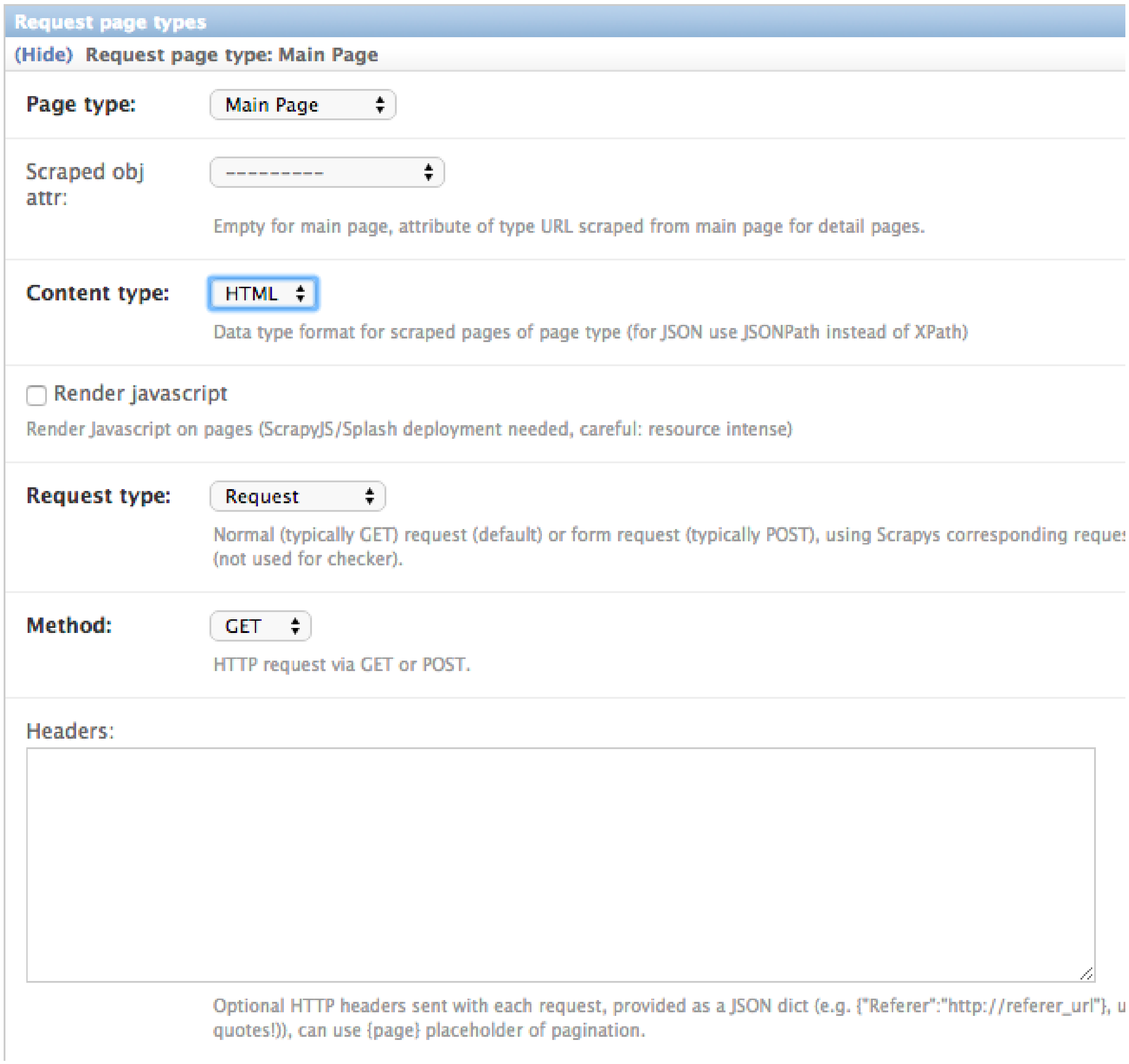
This completes the xpath definitions for our scraper. The form you have filled out should look similar to the screenshot above (which is broken down to two rows due to space issues).

Note: You can also **scrape** attributes of your object **from outside the base element** by using the `..` notation in your XPath expressions to get to the parent nodes!

Note: Starting with `DDS v.0.8.11` you can build your **detail page URLs** with placeholders for **main page attributes** in the form of `{ATTRIBUTE_NAME}`, see *Attribute Placeholders* for further reference.

Adding corresponding request page types

For all page types you used for your `ScraperElemes` you have to define corresponding `RequestPageType` objects in the `Scraper` admin form. There has to be exactly one main page and 0-25 detail page type objects.


Request page types

(Hide) Request page type: Main Page

Page type: Main Page

Scraped obj attr: ---------

Empty for main page, attribute of type URL scraped from main page for detail pages.

Content type: HTML

Data type format for scraped pages of page type (for JSON use JSONPath instead of XPath)

☐ Render javascript

Render Javascript on pages (ScrapyJS/Splash deployment needed, careful: resource intense)

Request type: Request

Normal (typically GET) request (default) or form request (typically POST), using Scrapys corresponding reque
(not used for checker).

Method: GET

HTTP request via GET or POST.

Headers:

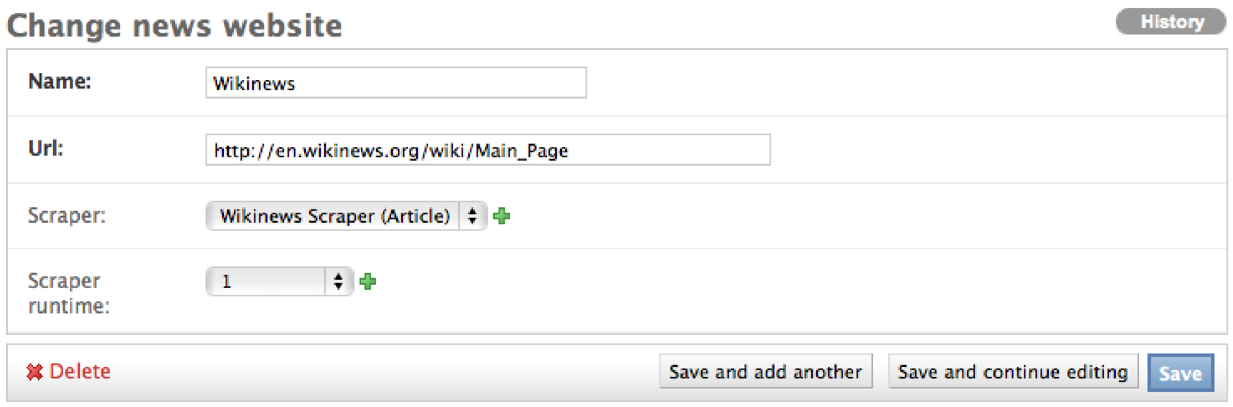
Optional HTTP headers sent with each request, provided as a JSON dict (e.g. {"Referer":"http://referer_url"}, u
quotes!)), can use {page} placeholder of pagination.

Within the `RequestPageType` object you can define request settings like the content type (`HTML`, `XML`,...), the request method (`GET` or `POST`) and others for the specific page type. With this it is e.g. possible to scrape HTML content from all the main pages and `JSON` content from the followed detail pages. For more information on this have a look at the *Advanced Request Options* section.

Create the domain entity reference object (NewsWebsite) for our open news example

Now - finally - we are just one step away of having all objects created in our Django admin. The last dataset we have to add is the reference object of our domain, meaning a `NewsWebsite` object for the Wikinews Website.

To do this open the NewsWebsite form in the Django admin, give the object a meaningful name ('Wikinews'), assign the scraper and create an empty *SchedulerRuntime* object with `SCRAPER` as your `runtime_type`.



Name:	Wikinews
Url:	http://en.wikinews.org/wiki/Main_Page
Scraper:	Wikinews Scraper (Article)
Scraper runtime:	1

2.3.4 Connecting Scrapy with your Django objects

For Scrapy to work with your Django objects we finally set up two static classes, the one being a spider class, inheriting from *DjangoSpider*, the other being a finalising pipeline for saving our scraped objects.

Adding the spider class

The main work left to be done in our spider class - which is inheriting from the *DjangoSpider* class of Django Dynamic Scraper - is to instantiate the spider by connecting the domain model classes to it in the `__init__` function:

```
from dynamic_scraper.spiders.django_spider import DjangoSpider
from open_news.models import NewsWebsite, Article, ArticleItem


class ArticleSpider(DjangoSpider):

    name = 'article_spider'

    def __init__(self, *args, **kwargs):
        self._set_ref_object(NewsWebsite, **kwargs)
        self.scraper = self.ref_object.scraper
        self.scrape_url = self.ref_object.url
        self.scheduler_runtime = self.ref_object.scraper_runtime
        self.scraped_obj_class = Article
        self.scraped_obj_item_class = ArticleItem
        super(ArticleSpider, self).__init__(self, *args, **kwargs)
```

Adding the pipeline class

Since you maybe want to add some extra attributes to your scraped items, DDS is not saving the scraped items for you but you have to do it manually in your own item pipeline:

```
import logging
from django.db.utils import IntegrityError
from scrapy.exceptions import DropItem
from dynamic_scraper.models import SchedulerRuntime

class DjangoWriterPipeline(object):
    def process_item(self, item, spider):
      if spider.conf['DO_ACTION']: #Necessary since DDS v.0.9+
            try:
                item['news_website'] = spider.ref_object

                checker_rt = SchedulerRuntime(runtime_type='C')
                checker_rt.save()
                item['checker_runtime'] = checker_rt

                item.save()
                spider.action_successful = True
                spider.log("Item saved.", logging.INFO)

            except IntegrityError as e:
                spider.log(str(e), logging.ERROR)
                spider.log(str(item._errors), logging.ERROR)
                raise DropItem("Missing attribute.")
      else:
          if not item.is_valid():
              spider.log(str(item._errors), logging.ERROR)
              raise DropItem("Missing attribute.")

      return item
```

The things you always have to do here is adding the reference object to the scraped item class and - if you are using checker functionality - create the runtime object for the checker. You also have to set the `action_successful` attribute of the spider, which is used internally by DDS when the spider is closed.

2.3.5 Running/Testing your scraper

You can run/test spiders created with Django Dynamic Scraper from the command line similar to how you would run your normal Scrapy spiders, but with some additional arguments given. The syntax of the DDS spider run command is as following:

```
scrapy crawl [--output=FILE --output-format=FORMAT] SPIDERNAME -a id=REF_OBJECT_ID
                        [-a do_action=(yes|no) -a run_type=(TASK|SHELL)
                        -a max_items_read={Int} -a max_items_save={Int}
                        -a max_pages_read={Int}
                        -a output_num_mp_response_bodies={Int} -a output_num_dp_response_bodies={Int
```

- With `-a id=REF_OBJECT_ID` you provide the ID of the reference object items should be scraped for, in our example case that would be the Wikinews `NewsWebsite` object, probably with ID 1 if you haven't added other objects before. This argument is mandatory.
- By default, items scraped from the command line are not saved in the DB. If you want this to happen, you have to provide `-a do_action=yes`.
- With `-a run_type=(TASK|SHELL)` you can simulate task based scraper runs invoked from the command line. This can be useful for testing, just leave this argument for now.
- With `-a max_items_read={Int}` and `-a max_items_save={Int}` you can override the scraper settings for these params.
- With `-a max_pages_read={Int}` you can limit the number of pages read when using pagination
- With `-a output_num_mp_response_bodies={Int}` and `-a output_num_dp_response_bodies={Int}` you can log the complete response body content of the {Int} first main/detail page responses to the screen for

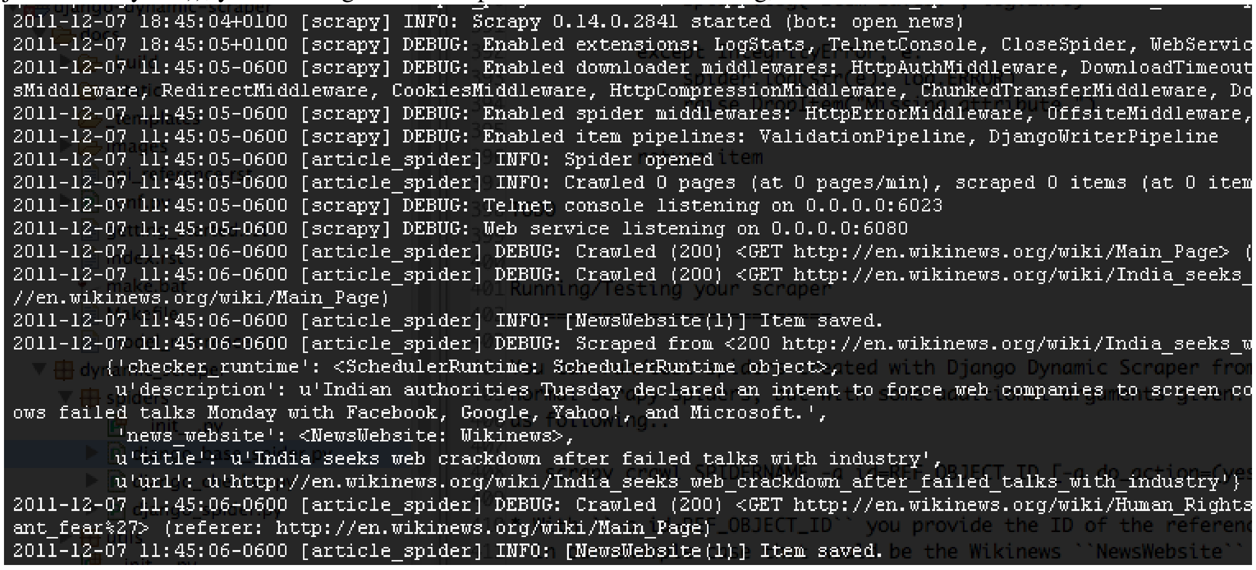
debugging (beginnings/endings are marked with a unique string in the form `RP_MP_{num}_START` for using full-text search for orientation)

- If you don't want your output saved to the Django DB but to a custom file you can use Scrapy's build-in output options `--output=FILE` and `--output-format=FORMAT` to scrape items into a file. Use this without setting the `-a do_action=yes` parameter!

So, to invoke our Wikinews scraper, we have the following command:

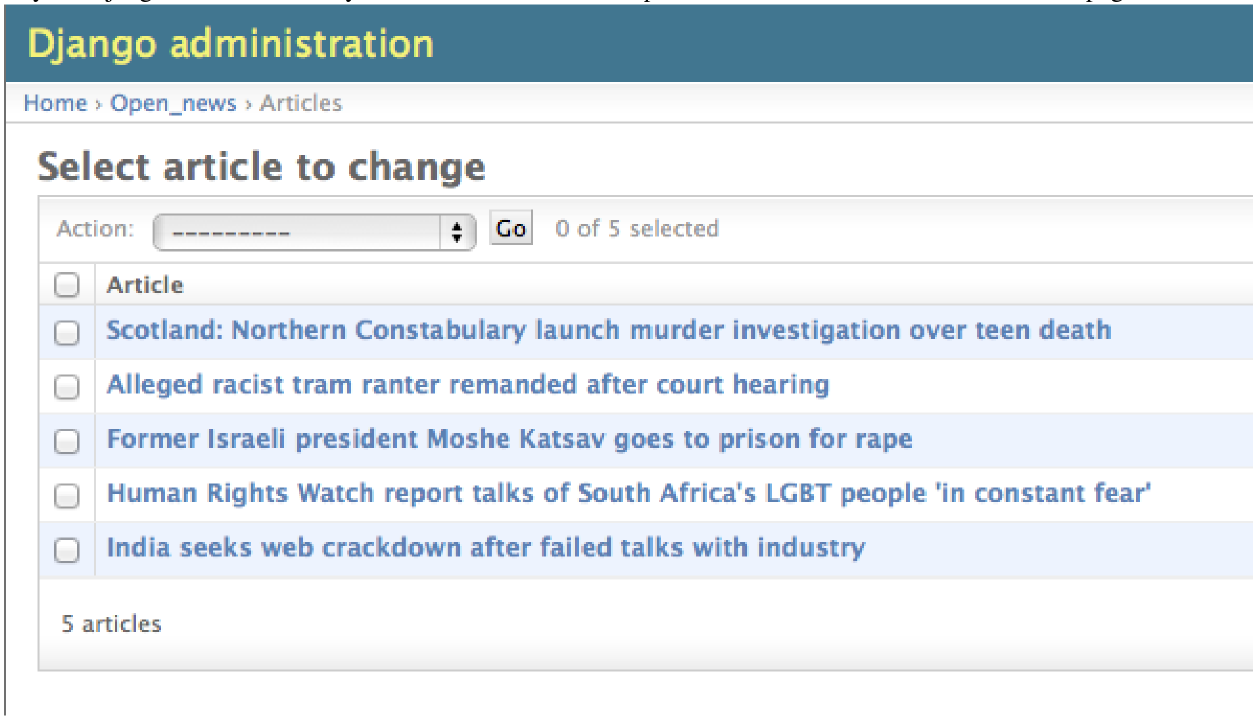
```
scrapy crawl article_spider -a id=1 -a do_action=yes
```

If you have done everything correctly (which would be a bit unlikely for the first run after so many single steps, but just in theory... :-)), you should get some output similar to the following, of course with other headlines:



```
2011-12-07 18:45:04+0100 [scrapy] INFO: Scrapy 0.14.0.2841 started (bot: open_news)
2011-12-07 18:45:05+0100 [scrapy] DEBUG: Enabled extensions: LogStats, TelnetConsole, CloseSpider, WebServic
2011-12-07 11:45:05-0600 [scrapy] DEBUG: Enabled downloader middlewares: HttpAuthMiddleware, DownloadTimeout
sMiddleware, RedirectMiddleware, CookiesMiddleware, HttpCompressionMiddleware, ChunkedTransferMiddleware, Do
2011-12-07 11:45:05-0600 [scrapy] DEBUG: Enabled spider middlewares: HttpErrorMiddleware, OffsiteMiddleware,
2011-12-07 11:45:05-0600 [scrapy] DEBUG: Enabled item pipelines: ValidationPipeline, DjangoWriterPipeline
2011-12-07 11:45:05-0600 [article_spider] INFO: Spider opened
2011-12-07 11:45:05-0600 [article_spider] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 item
2011-12-07 11:45:05-0600 [scrapy] DEBUG: Telnet console listening on 0.0.0.0:6023
2011-12-07 11:45:05-0600 [scrapy] DEBUG: Web service listening on 0.0.0.0:6080
2011-12-07 11:45:06-0600 [article_spider] DEBUG: Crawled (200) <GET http://en.wikinews.org/wiki/Main_Page> (
2011-12-07 11:45:06-0600 [article_spider] DEBUG: Crawled (200) <GET http://en.wikinews.org/wiki/India_seeks_
//en.wikinews.org/wiki/Main_Page)
2011-12-07 11:45:06-0600 [article_spider] INFO: [NewsWebsite(1)] Item saved.
2011-12-07 11:45:06-0600 [article_spider] DEBUG: Scraped from <200 http://en.wikinews.org/wiki/India_seeks_w
        {'checker_runtime': <SchedulerRuntime: SchedulerRuntime object>,
         u'description': u'Indian authorities Tuesday declared an intent to force web companies to screen co
ows failed talks Monday with Facebook, Google, Yahoo!, and Microsoft.',
         'news_website': <NewsWebsite: Wikinews>,
         u'title': u'India seeks web crackdown after failed talks with industry',
         u'url': u'http://en.wikinews.org/wiki/India_seeks_web_crackdown_after_failed_talks_with_industry'}
2011-12-07 11:45:06-0600 [article_spider] DEBUG: Crawled (200) <GET http://en.wikinews.org/wiki/Human_Rights
ant_fear%27> (referer: http://en.wikinews.org/wiki/Main_Page)
2011-12-07 11:45:06-0600 [article_spider] INFO: [NewsWebsite(1)] Item saved.
```

In your Django admin interface you should now see the scraped articles listed on the article overview page:



Django administration

Home › Open_news › Articles

Select article to change

Action: --------- Go 0 of 5 selected

	Article
☐	Scotland: Northern Constabulary launch murder investigation over teen death
☐	Alleged racist tram ranter remanded after court hearing
☐	Former Israeli president Moshe Katsav goes to prison for rape
☐	Human Rights Watch report talks of South Africa's LGBT people 'in constant fear'
☐	India seeks web crackdown after failed talks with industry

5 articles

Phew.

Your first scraper with Django Dynamic Scraper is working. Not so bad! If you do a second run and there haven't been any new bugs added to the DDS source code in the meantime, no extra article objects should be added to the DB. If you try again later when some news articles changed on the Wikinews overview page, the new articles should be added to the DB.

2.4 Advanced topics

2.4.1 Defining item checkers

Django Dynamic Scraper comes with a built-in mechanism to check, if items once scraped are still existing or if they could be deleted from the database. The entity providing this mechanism in DDS is called a `checker`. A `checker` is like a scraper also using the scraping logic from Scrapy. But instead of building together a new scraped item, it just checks the detail page referenced by a `DETAIL_PAGE_URL` of a scraped item. Depending on the `checker_type` and the result of the detail page check, the scraped item is kept or will be deleted from the DB.

Creating a checker class

To get a checker up and running you first have to create a checker class for each of your scraped object domain models. In our open news example, this would be a class called `ArticleChecker` in a module called `checkers` in our `scraper` directory:

```
from dynamic_scraper.spiders.django_checker import DjangoChecker
from open_news.models import Article


class ArticleChecker(DjangoChecker):

    name = 'article_checker'

    def __init__(self, *args, **kwargs):
        self._set_ref_object(Article, **kwargs)
        self.scraper = self.ref_object.news_website.scraper
        #self.scrape_url = self.ref_object.url (Not used any more in DDS v.0.8.3+)
        self.scheduler_runtime = self.ref_object.checker_runtime
        super(ArticleChecker, self).__init__(self, *args, **kwargs)
```

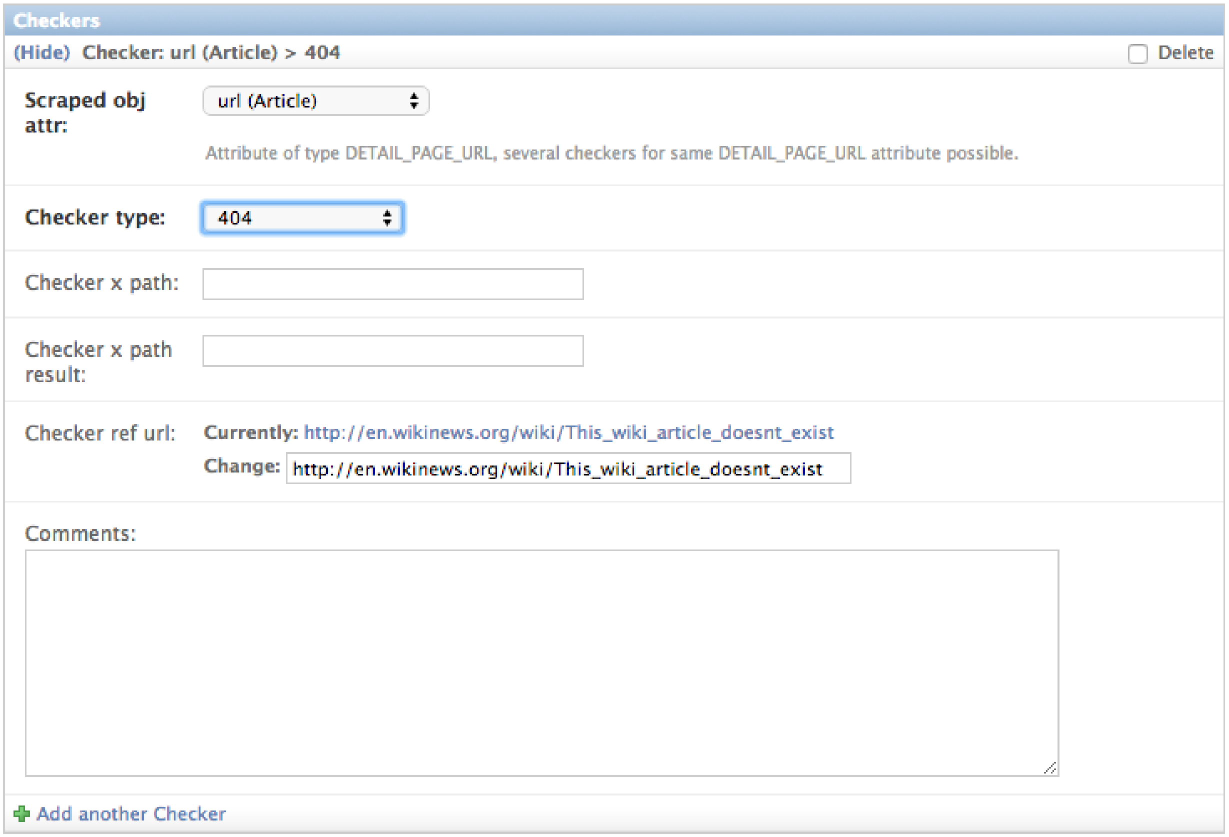
The checker class inherits from the *DjangoChecker* class from DDS and mainly gives the checker the information what to check and what parameters to use for checking. Be careful that the reference object is now the scraped object itself, since the checker is scraping from the item page url of this object. Furthermore the checker needs its configuration data from the scraper of the reference object. The scheduler runtime is used to schedule the next check. So if you want to use checkers for your scraped object, you have to provide a foreign key to a *SchedulerRuntime* object in your model class. The scheduler runtime object also has to be saved manually in your pipeline class (see: *Adding the pipeline class*).

Checker Configuration

You can create one or more checkers per scraper in the `Django admin`. A checker is connected to a `DETAIL_PAGE_URL` attribute and has a certain type, defining the checker behaviour. If you define more than one checker for a scraper an item is deleted when one of the checkers succeed.

There are momentarily the following checker types to choose from:

`404`	Item is deleted after check has returned 404 HTTP status code 2x in a row
`404_OR_X_PATH`	Same as 404 + check for an x_path value in the result


Checkers

(Hide) Checker: url (Article) > 404 — Delete

Scraped obj attr: url (Article)

Attribute of type DETAIL_PAGE_URL, several checkers for same DETAIL_PAGE_URL attribute possible.

Checker type: 404

Checker x path:

Checker x path result:

Checker ref url: Currently: http://en.wikinews.org/wiki/This_wiki_article_doesnt_exist

Change: http://en.wikinews.org/wiki/This_wiki_article_doesnt_exist

Comments:

Add another Checker

For selecting a checker type and providing the parameters for an x_path checker you have to look for an example item page url from the website to be scraped which references an item not existing any more. If the urls to your scraped items are build using an item ID you can e.g. try to lower this ID or increase it to a very large number. Be creative! In our Wikinews example it is a bit different, since the news article url there is build using the title of the article. So for the checker we take a random article url to a not existing article: "http://en.wikinews.org/wiki/Random_article_text".

If your url found is responding with a 404 when invoked, you can simply choose `404` as your checker type. For a `404_OR_X_PATH` checker you have to provide an XPath for your chosen url which will extract a string from that url uniquely indicating, that the content originally expected is not there any more. For our Wikinews example and the url we choose above there is a text and a url provided suggesting to create the currently not existing wiki page, so we can use the XPath `//a[@href="http://en.wikinews.org/wiki/This_wiki_article_doesnt_exist"]/text()` and the result string "create this page" to uniquely identifying a scraped item not existing any more. It is also possible to leave out the result string. Then the checker already succeeds when the given xpath is finding elements on the page.

Note: Attention! Make sure that the XPath/result string combination you choose is NOT succeeding on normal item pages, otherwise the checker will delete all your items!

Note: To make sure your items aren't deleted accidentally on a 404 response, 404 checks are only deleted on the second try while XPath checks are deleted at once. So to save crawling resources always try to realize your checking with XPath checks, otherwise the crawler need double the amount of checks!

Running your checkers

You can test your DDS checkers the same way you would run your scrapers:

```
scrapy crawl CHECKERNAME -a id=REF_OBJECT_ID [-a do_action=(yes|no) -a run_type=(TASK|SHELL)]
```

As a reference object ID you now have to provide the ID of a scraped item to be checked. With `do_action=yes` an item is really deleted, otherwise the checker is only tested without actually manipulating the DB.

If you want to test a check on an item scraped in the open news example project, change the url of the item in the DB to the checker reference url, look for the item ID and then run:

```
scrapy crawl article_checker -a id=ITEM_ID -a do_action=yes
```

If everything works well, your item should have been deleted.

Run checker tests

Django Dynamic Scraper comes with a build-in scraper called `checker_test` which can be used to test your checkers against the defined reference url. You can run this checker on the command line with the following command:

```
scrapy crawl checker_test -a id=SCRAPER_ID
```

This scraper is useful both to look, if you have chosen a valid `checker_x_path_ref_url` and corresponding `checker_x_path` and `checker_x_path_result` values as well as to see over time if your reference urls stay valid.

For running all checker tests at once there exists a simple Django management command called `run_checker_tests`, which executes the `checker_test` scraper for all of your defined scrapers and outputs Scrapy log messages on `WARNING` level and above:

```
python manage.py run_checker_tests [--only-active --report-only-errors --send-admin-mail]
```

The option `only-active` will limit execution to active scrapers, `--report-only-errors` will more generously pass the test on some not so severe cases (e.g. a checker ref url returning `404` for a `404_OR_X_PATH` checker type). Executing the command with the `--send-admin-mail` flag will send an email to Django admins if checker configurations are not working which can be useful if you want to run this command as a cronjob.

2.4.2 Scheduling scrapers/checkers

Introduction

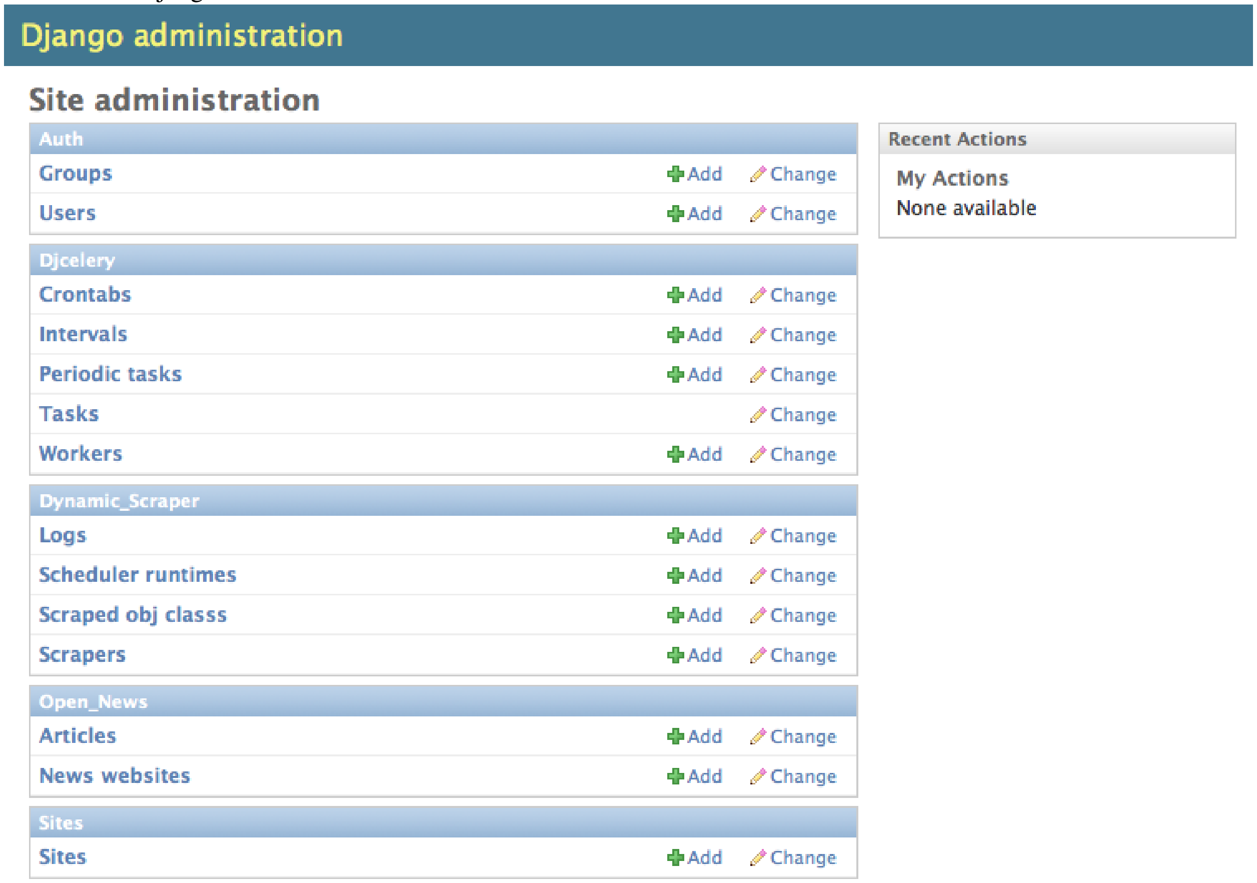
Django Dynamic Scraper comes with a build-in mechanism to schedule the runs of your scrapers as well as your checkers. After each run DDS dynamically calculates the next execution time depending on the success of the run. For a scraper that means, that the time between two scraper runs is shortened when new items could be scraped from a page and is prolonged if not. For a checker, it means that a next check is prolonged if the check was not successful, meaning that the item was not deleted. If it was deleted - well: than it was deleted! No further action! :-) The parameters for this calculation (e.g. a min/max time period between two actions) are defined for each *ScrapedObjClass* in the DB.

DDS is using django-celery to actually run your scrapers. Celery is a distributed task queue system for Python, which means that you can run a celery daemon which takes task orders from somewhere and then executes the corresponding tasks in a sequential way so that no task is lost, even if the system is under heavy load. In our use case Celery is "just" working as a comfortable cron job replacement, which can be controlled via the Django admin interface. The scheduler of DDS is using the scheduler runtime objects we defined for our example scraper and checker in the sections before. The scheduler runtime objects contain some dynamic information for the calculation of the next execution time of the scraper as well as the next execution time itself. For django-celery a task for each *ScrapedObjClass* has to be defined, which can than be started and stopped in the Django admin interface. Each task is executed periodically in a configurable time frame (e.g. ever hour). The task is then running all the scrapers associated with its *ScrapedObjClass*, which next execution time lies in the past. After each run, the next next execution time is calculated by the scraper and saved into its scheduler runtime object. The next time this time lies in the past, the scraper is run again.

Note: The number of spiders/checkers run at each task run is limited by the `DSCRAPER_MAX_SPIDER_RUNS_PER_TASK` and `DSCRAPER_MAX_CHECKER_RUNS_PER_TASK` settings which can be adopted in your custom settings file (see: *Settings*).

Installing/configuring django-celery for DDS

This paragraph is covering only the specific installation issues with django-celery in regard of installing it for the use with DDS, so you should be familiar with the basic functionality of Celery and take general installation infos from the django-celery website. If you have successfully installed and configured django-celery, you should see the `Djcelery` tables in the Django admin interface:



Django administration

Site administration

Auth		
Groups	Add	Change
Users	Add	Change

Djcelery		
Crontabs	Add	Change
Intervals	Add	Change
Periodic tasks	Add	Change
Tasks		Change
Workers	Add	Change

Dynamic_Scraper		
Logs	Add	Change
Scheduler runtimes	Add	Change
Scraped obj classs	Add	Change
Scrapers	Add	Change

Open_News		
Articles	Add	Change
News websites	Add	Change

Sites		
Sites	Add	Change

Recent Actions

My Actions

None available

For `django-celery` to work, Celery also needs a message broker for the actual message transport. For our relatively simple use case, kombu is the easiest and recommended choice. Kombu is automatically installed as a dependency when you install `django-celery` and you can add it to your `INSTALLED_APPS`:

```
    INSTALLED_APPS = (
    ...
    'kombu.transport.django',
    'djcelery',
)
```

Then we can configure django-celery in combination with kombu in our `settings.py` file. A starter configuration could look similar to this:

```
# django-celery settings
import djcelery
djcelery.setup_loader()
BROKER_HOST = "localhost"
BROKER_PORT = 5672
BROKER_BACKEND = "django"
BROKER_USER = "guest"
BROKER_PASSWORD = "guest"
BROKER_VHOST = "/"
CELERYBEAT_SCHEDULER = 'djcelery.schedulers.DatabaseScheduler'
```

Defining your tasks

For defining tasks for your scrapers and checkers which can be selected for periodical runs in the Django admin interface, you have to define two short methods in a Python module in which your tasks are declared and make sure, that your tasks are found by django-celery. The easiest way to do this is by placing your methods in a module called `tasks.py` in the main directory of your app. The tasks should then be found automatically. The two methods in our open news example look like this:

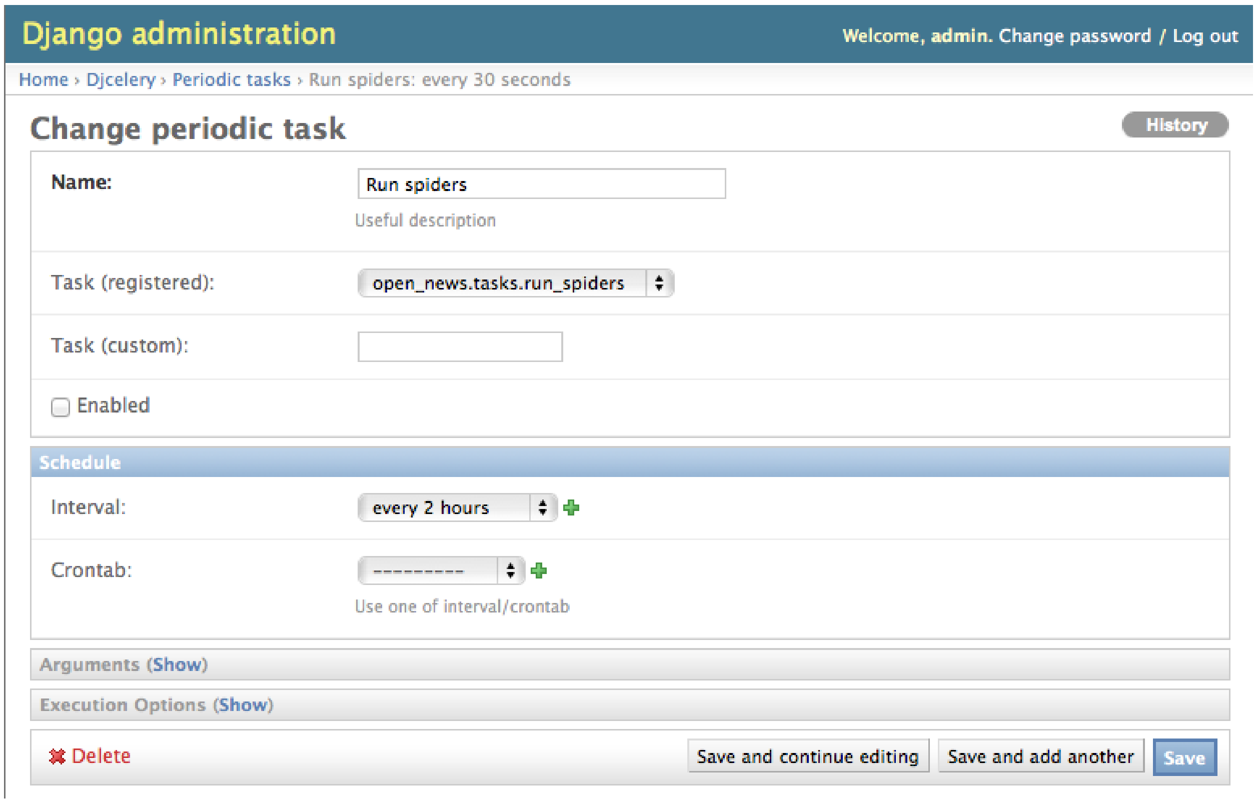
```
from celery.task import task
from django.db.models import Q
from dynamic_scraper.utils.task_utils import TaskUtils
from open_news.models import NewsWebsite, Article

@task()
def run_spiders():
    t = TaskUtils()
    #Optional: Django field lookup keyword arguments to specify which reference objects (NewsWebsite
    #to use for spider runs, e.g.:
    kwargs = {
        'scrape_me': True, #imaginary, model NewsWebsite hat no attribute 'scrape_me' in example
    }
    #Optional as well: For more complex lookups you can pass Q objects vi args argument
    args = (Q(name='Wikinews'),)
    t.run_spiders(NewsWebsite, 'scraper', 'scraper_runtime', 'article_spider', *args, **kwargs)

@task()
def run_checkers():
    t = TaskUtils()
    #Optional: Django field lookup keyword arguments to specify which reference objects (Article)
    #to use for checker runs, e.g.:
    kwargs = {
        'check_me': True, #imaginary, model Article hat no attribute 'check_me' in example
    }
    #Optional as well: For more complex lookups you can pass Q objects vi args argument
    args = (Q(id__gt=100),)
    t.run_checkers(Article, 'news_website__scraper', 'checker_runtime', 'article_checker', *args, **
```

The two methods are decorated with the Celery task decorator to tell Celery that these methods should be regarded as tasks. In each task, a method from the `TaskUtils` module from DDS is called to run the spiders/checkers ready for the next execution.

Now you can create a peridoc task both for your scraper and your checker in the Django admin interface:


Django administration

Welcome, admin. Change password / Log out

Home › Djcelery › Periodic tasks › Run spiders: every 30 seconds

Change periodic task

Name:	Run spiders Useful description
Task (registered):	open_news.tasks.run_spiders
Task (custom):	
Enabled	☐

Schedule

Interval:	every 2 hours
Crontab:	--------- Use one of interval/crontab

Arguments (Show)

Execution Options (Show)

In the peridoc task form you should be able to select your tasks defined above. Create an interval how often these tasks are performed. In our open news example, 2 hours should be a good value. Please keep in mind, that these are not the values how often a scraper/checker is actually run. If you define a two hour timeframe here, it just means, that ever two hours, the task method executed is checking for scrapers/checkers with a next execution time (defined by the associated `scheduler_runtime`) lying in the past and run these scrapers. The actual time period between two runs is determined by the next execution time itself which is calculated dynamically and depending on the scheduling configuration you'll learn more about below. For the scrapers to run, remember also that you have to set the scraper active in the associated `scraper` object.

Run your tasks

To actually run the task (respectively set our scheduling system to work as a whole) we have to run two different daemon processes. The first one is the `celeryd` daemon from django-celery which is responsible for collecting and executing tasks. We have to run `celeryd` with the -B option to also run the celerybeat task scheduler which executes periodical tasks defined in Celery. Start the daemon with:

```
python manage.py celeryd -l info -B --settings=example_project.settings
```

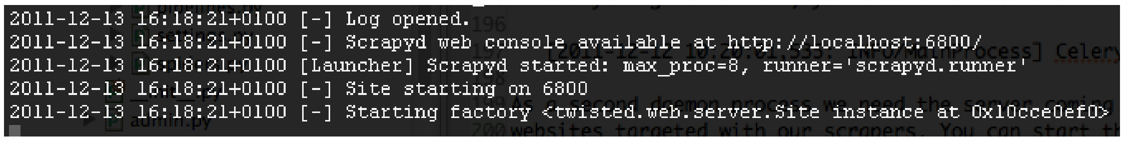
If everything works well, you should now see the following line in your command line output:

```
[2011-12-12 10:20:01,535: INFO/MainProcess] Celerybeat: Starting...
```

As a second daemon process we need the server from the separate `scrapyd` project to actually crawl the different websites targeted with our scrapers. Make sure you have deployed your Scrapy project (see: *Scrapy Configuration*) and run the server with:

```
scrapyd
```

You should get an output similar to the following:



```
2011-12-13 16:18:21+0100 [-] Log opened.
2011-12-13 16:18:21+0100 [-] Scrapyd web console available at http://localhost:6800/
2011-12-13 16:18:21+0100 [Launcher] Scrapyd started: max_proc=8, runner='scrapyd.runner'
2011-12-13 16:18:21+0100 [-] Site starting on 6800
2011-12-13 16:18:21+0100 [-] Starting factory <twisted.web.server.Site instance at 0x10cce0ef0>
```

For testing your scheduling system, you can temporarily set your time interval of your periodic task to a lower interval, e.g. 1 minute. Now you should see a new task coming in and being executed every minute:

```
Got task from broker: open_news.tasks.run_spiders[5a3fed53-c26a-4f8f-b946-8c4a2c7c5c83]
Task open_news.tasks.run_spiders[5a3fed53-c26a-4f8f-b946-8c4a2c7c5c83] succeeded in 0.052549123764s:
```

The executed task should then run the scrapers/checkers which you should see in the output of the Scrapy server:

```
Process started: project='default' spider='article_spider' job='41f27199259e11e192041093e90a480a' pic
Process finished: project='default' spider='article_spider' job='41f27199259e11e192041093e90a480a' p
```

Note: Note that you can vary the log level for debugging as well as other run parameters when you start the servers, see the man/help pages of the celery and the Scrapy daemons.

Note: Please see this configuration described here just as a hint to get started. If you want to use this in production you have to provide extra measures to make sure that your servers run constantly and that they are secure. See the specific server documentation for more information.

Scheduling configuration

Now coming to the little bit of magic added to all this stuff with dynamic scheduling. The basis for the dynamic scheduling in DDS is layed both for your scrapers and your checkers with the scheduling configuration parameters in your scraped object class definitions in the Django admin interface. The default configuration for a scraper looks like this:

```
"MIN_TIME": 15,
"MAX_TIME": 10080,
"INITIAL_NEXT_ACTION_FACTOR": 10,
"ZERO_ACTIONS_FACTOR_CHANGE": 20,
"FACTOR_CHANGE_FACTOR": 1.3,
```

Scheduling now works as follows: the inital time period between two scraper runs is calculated by taking the product of the `MIN_TIME` and the `INITIAL_NEXT_ACTION_FACTOR`, with minutes as the basic time unit for `MIN_TIME` and `MAX_TIME`:

```
initial time period := 15 Minutes (MIN_TIME) * 10 (INITIAL_NEXT_ACTION_FACTOR) = 150 Minutes = 2 1/2
```

Now, every time a scraper run was successful, the new next action factor is calculated by dividing the actual next action factor by the `FACTOR_CHANGE_FACTOR`. So a successful scraper run would lead to the following new time period:

```
new next action factor (NAF) := 10 (INITIAL_NEXT_ACTION_FACTOR) / 1.3 (FACTOR_CHANGE_FACTOR) = 7.69
time period after successful run := 15 Minutes * 7.69 (NAF) = 115 Minutes
```

So if it turns out that your scraper always find new items the time period between two runs gets smaller and smaller until the defined `MIN_TIME` is reached which is taken as a minimum time period between two scraper runs. If your scraper was not successful (meaning, that no new items were found) these unsucessful actions (scraper runs) are counted as `ZERO_ACTIONS`. If a number of unsuccessful actions greater than `ZERO_ACTIONS_FACTOR_CHANGE` is counted, a new next action factor is calculated, this time by taking the product of the actual action factor and the FACTOR_CHANGE_FACTOR (calculation restarting from initial values for the example):

```
new next action factor (NAF) := 10 (INITIAL_NEXT_ACTION_FACTOR) * 1.3 (FACTOR_CHANGE_FACTOR) = 13
time period after 21 unsuccessful runs := 15 Minutes * 13 (NAF) = 195 Minutes
```

So the time period between two scraper runs becomes larger. If there is never a new item found for your scraper this will go on until the calculated time period reaches the MAX_TIME defined.

In the real world application of this mechanism normally neither the MIN_TIME nor the MAX_TIME should be reached. The normal situation is that your scraper often finds nothing new on the page to be scraped and than after x executed runs finds new items provided on the website to be scraped. If this x is generally lower than your defined ZERO_ACTIONS_FACTOR_CHANGE number, the time period is becoming shorter over time. But since this means more scraper runs in the same time chances are high that with these narrower scheduled runs less zero actions occur and leads at some point to an again increased next action factor. So some kind of (relatively) stable next action factor should be reached over time, representing in the best case a good compromise between the needs of actuality of your scrapers and not to much resources wasted on running your scraper on websites not updated in between two runs.

Note: Since this is a relatively complex mechanism also depending on a large part on the update process of your scraped website, it will probably take some time to get a bit a feeling for how the scheduling is developing and to what action factors it tends to, so don't try to find the perfect solution in the first run. Instead, start with a (maybe rather too conservatively calculated) start configuration and adjust your parameters over time. You can observe the development of your action factors in the scheduler runtime objects.

Note: Please be aware that scraping is a resource consuming task, for your server but as well for the server of the websites you are scraping. Try to find a balanced solution, not just setting your MIN_TIME to 1 minute or similar.

Note: If you don't need dynamic scheduling, you can also just set the MIN_TIME and the MAX_TIME to the same values and just ignore the rest.

Scheduling of your checkers works very similar to the scraper scheduling, the inital configuration is as follows:

```
"MIN_TIME": 1440,
"MAX_TIME": 10080,
"INITIAL_NEXT_ACTION_FACTOR": 1,
"ZERO_ACTIONS_FACTOR_CHANGE": 5,
"FACTOR_CHANGE_FACTOR": 1.3,
```

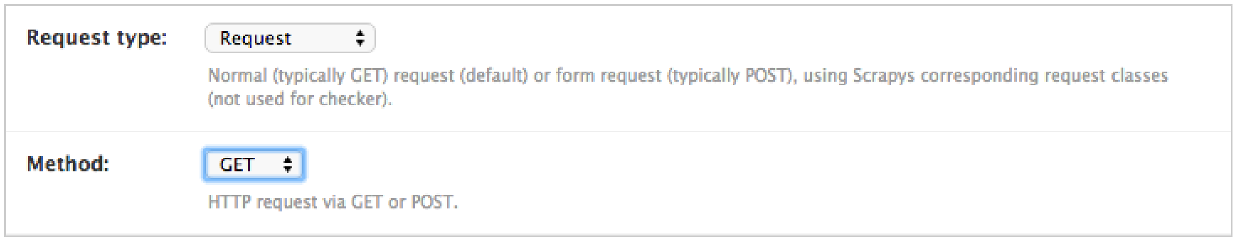
Since the checker scheduling is terminated with the success of a checker run (meaning the item and the associated scheduler runtime is deleted), there is only the prolonging time period part of the scheduler actually working. So scraped items are checked in a (relatively, defined by your configuration) short time period at first. If the item turns out to be persistently existing, the checks are prolonged till MAX_TIME is reached.

2.4.3 Advanced Request Options

Since DDS v.0.7+ you have more options to fine-tune your scraping requests by e.g. providing additional values for cookies or HTTP headers. These values are internally passed to Scrapy's Request object. You can find the extended request options in the Request options tab in the Scraper form of your Django project admin. For the different page types like the (paginated) main pages and the detail pages following scraped urls you can define different request options.

Note: Parameters for the different options are passed as `JSON` dicts. Make sure to use `double quotes` for attribute values and to leave the `comma` for the last attribute key-value pair.

Request Type and Method

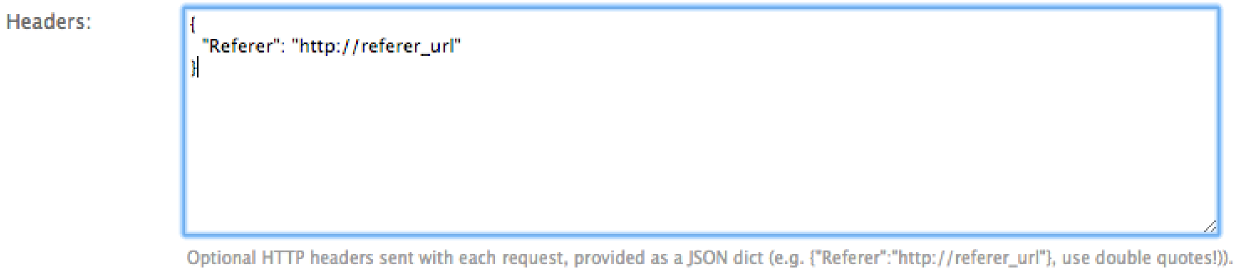


Request type: Request
Normal (typically GET) request (default) or form request (typically POST), using Scrapys corresponding request classes (not used for checker).

Method: GET
HTTP request via GET or POST.

The request type - corresponding to Scrapy's Request classes - and the type of the request being sent as `GET` or `POST`. Normally you will choose `GET` together with a classic `Request` and `POST` with a `FormRequest` but for special cases you are free too choose here.

HTTP Headers



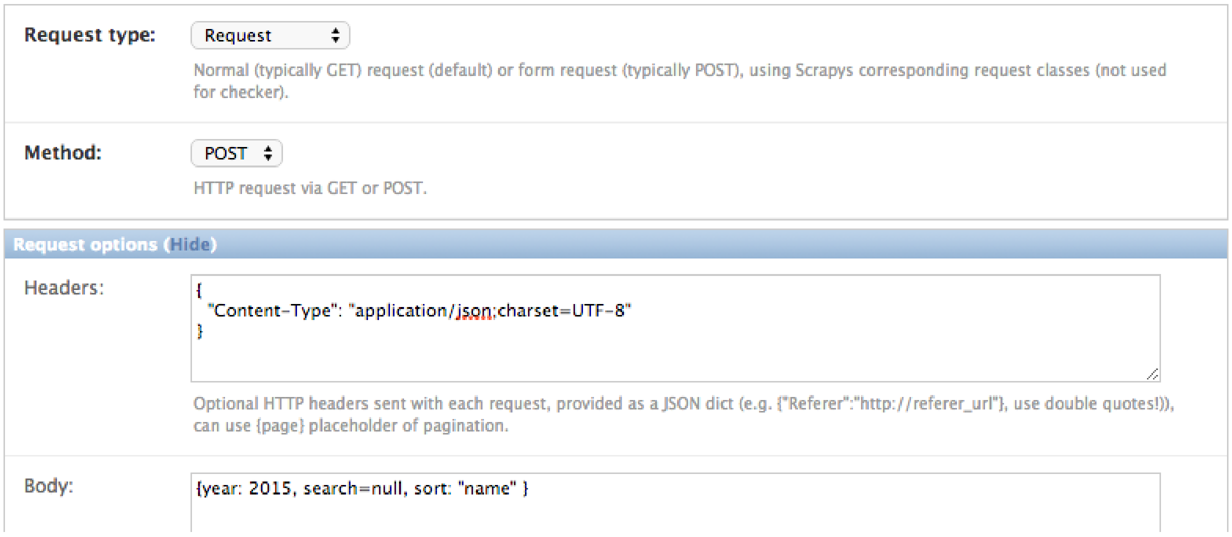
Headers:

```
{
  "Referer": "http://referer_url"
}
```

Optional HTTP headers sent with each request, provided as a JSON dict (e.g. {"Referer":"http://referer_url"}, use double quotes!)).

For setting/changing specific `HTTP header` fields like the referer URL use the `headers` text field in the request options.

HTTP Body



Request type: Request
Normal (typically GET) request (default) or form request (typically POST), using Scrapys corresponding request classes (not used for checker).

Method: POST
HTTP request via GET or POST.

Request options (Hide)

Headers:

```
{
  "Content-Type": "application/json;charset=UTF-8"
}
```

Optional HTTP headers sent with each request, provided as a JSON dict (e.g. {"Referer":"http://referer_url"}, use double quotes!)), can use {page} placeholder of pagination.

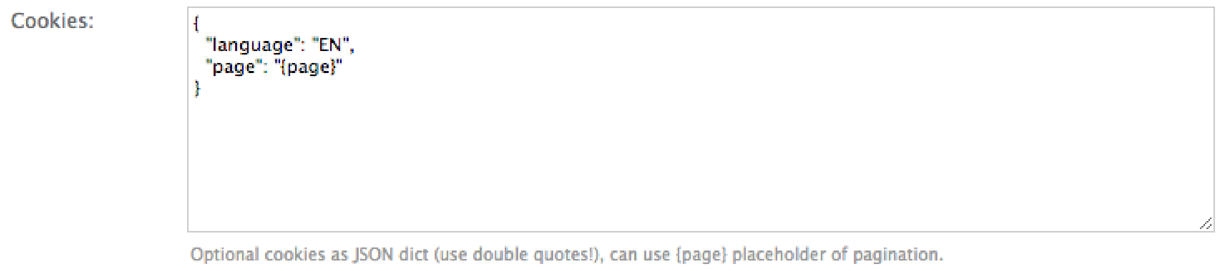
Body:

```
{year: 2015, search=null, sort: "name" }
```

Setting/changing the `HTTP body`. This can be useful for some special-case scenarios, for example if you want to send a `POST` request with content type for the request altered and sending `POST` parameters as a `JSON` dict.

Note: Don't be fooled, especially by the example provided: data for the body attribute is NOT provided as `JSON` but as a `string`. While e.g. the `Headers` field always has to be in `JSON` format, the `Body` text is just randomly `JSON` in this example, but it could also be `This is my body text.`.

Request Cookies



```
{
  "language": "EN",
  "page": "{page}"
}
```

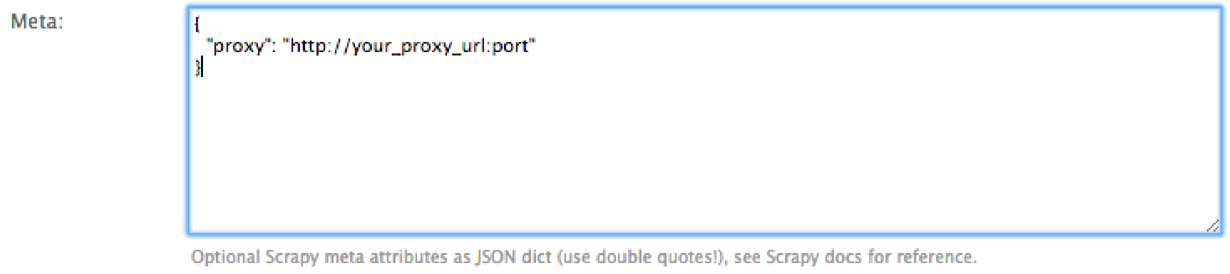
Optional cookies as JSON dict (use double quotes!), can use {page} placeholder of pagination.

Sometime the output of a website you want to scrape might depend on the values of some cookies sent to the server. For this occasion you can use the `Cookies` form in the request options tab, e.g. for setting the language of a website to `english`.

You can also use the `{page}` placeholder. This placeholder is replaced for consecutive pages according to your pagination parameters (see: *Pagination*).

Note: If you want to pass a `session ID` for a site as a `cookie`, you can open the desired website in your browser and copy-paste the session ID from the development console for immediately following scraper runs.

Scrapy Meta Options

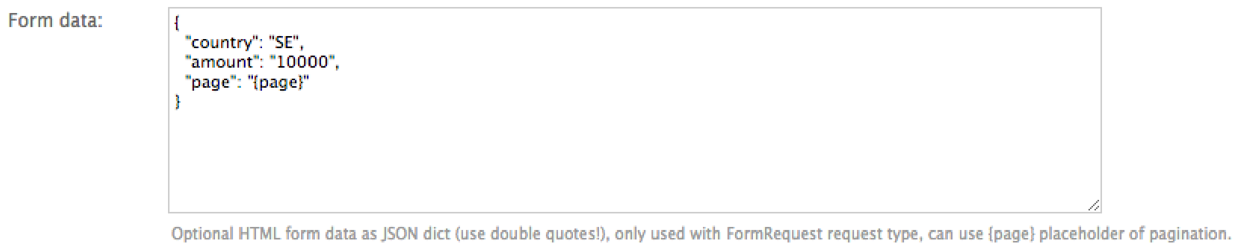


```
{
  "proxy": "http://your_proxy_url:port"
}
```

Optional Scrapy meta attributes as JSON dict (use double quotes!), see Scrapy docs for reference.

Changing Scrapy meta attributes, see Scrapy docs for reference.

Form Data


Form data:

```
{
  "country": "SE",
  "amount": "10000",
  "page": "{page}"
}
```

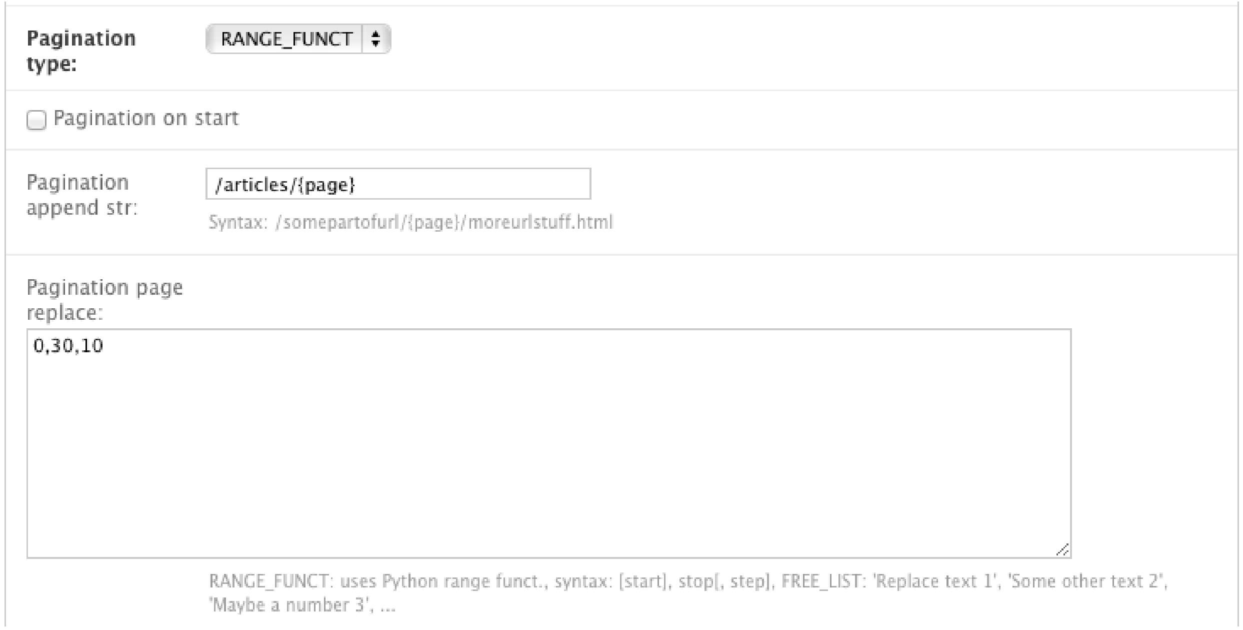
Optional HTML form data as JSON dict (use double quotes!), only used with FormRequest request type, can use {page} placeholder of pagination.

If you want to scrape data provided on a website via a web form, data is often returned via `POST` request after sending various `POST request parameters` for narrowing the results. For this scenario use the `FormRequest` request type and `POST` as method in the scraper admin and provide the adequate form data as a JSON dictionary in the request options.

You can also use the `{page}` placeholder. This placeholder is replaced for consecutive pages according to your pagination parameters (see: *Pagination*).

2.4.4 Pagination

Django Dynamic Scraper supports pagination for scraping your objects from several overview pages or archives. The following screenshot shows the pagination parameters which can be defined in the Django admin for each scraper:


Pagination type: RANGE_FUNCT

Pagination on start

Pagination append str: /articles/{page}

Syntax: /somepartofurl/{page}/moreurlstuff.html

Pagination page replace:

0,30,10

RANGE_FUNCT: uses Python range funct., syntax: [start], stop[, step], FREE_LIST: 'Replace text 1', 'Some other text 2', 'Maybe a number 3', ...

For using pagination you have to switch the `pagination_type` in your scraper definition from `NONE` to your desired type. The main concept of pagination is, that you define a `pagination_append_str` with a placeholder `{page}`, which is replaced through a list generated by selecting the `pagination_type` and giving a corresponding `pagination_page_replace` context. There are the following pagination types to choose from:

Pagination type: RANGE_FUNCT

This pagination type uses the python range function. As a replace context the same arguments like in the range function are used: `range([start], stop[, step])`. The integer list created by this function will be used as an input to replace the "{page}" template tag in the append string to form the different urls.

So the parameters in our example above in the screenshot will lead - together with "http://www.urltoscrape.org" as the base scrape url of your scraper runtime - to the following urls to be scraped:

1. http://www.urltoscrape.org/articles/0
2. http://www.urltoscrape.org/articles/10
3. http://www.urltoscrape.org/articles/20
4. http://www.urltoscrape.org/articles/30

Pagination type: FREE_LIST

If the urls from an archive are formed differently you can use this pagination type and just provide a list with different fitting replacements, the syntax is as follow: `'Replace text 1', 'Some other text 2', 'Maybe a number 3', ...`.

So if you define a list as follows: `'a-d', 'e-h', 'i-n', 'o-z'`, you get the following urls:

1. http://www.urltoscrape.org/articles/a-d
2. http://www.urltoscrape.org/articles/e-h
3. http://www.urltoscrape.org/articles/i-n
4. http://www.urltoscrape.org/articles/o-z

2.4.5 Scraping JSON content

Beside creating `HTML` or `XML` scrapers where you can use classic `XPath` notation, `DDS` supports also scraping pages encoded in `JSON` (`v.0.5.0` and above), e.g. for crawling web APIs or ajax call result pages.

For scraping `JSON`, `JSONPath` is used, an `XPath`-like expression language for digging into `JSON`. For reference see expressions as defined here:

- GitHub - python-jsonpath-rw Library
- JSONPath - XPath for JSON

Note: Using `JSONPath` in `DDS` works for standard `JSON` page results, but is not as heavily tested as using `XPath` for data extraction. If you are working with more complex `JSONPath` queries and run into problems, please report them on GitHub!

2.4.6 Scraping images/screenshots

Django Dynamic Scraper is providing a custom image pipeline build on Scrapy's item pipeline for downloading images to scrape and download images associated to your items scraped and and save a reference to each image together with the scraped item in the DB.

Configuration

For using image scraping in DDS you have to provide some additional parameters in your Scrapy *settings.py* file:

```
import os.path

PROJECT_ROOT = os.path.abspath(os.path.dirname(__file__))

ITEM_PIPELINES = [
    'dynamic_scraper.pipelines.DjangoImagesPipeline',
    'dynamic_scraper.pipelines.ValidationPipeline',
    'open_news.scraper.pipelines.DjangoWriterPipeline',
]

IMAGES_STORE = os.path.join(PROJECT_ROOT, '../thumbnails')

IMAGES_THUMBS = {
    'small': (170, 170),
}
```

In your settings file you have to add the `DjangoImagesPipeline` from DDS to your `ITEM_PIPELINES` and define a folder to store images scraped. Don't forget to create this folder in your file system and give it adequate permissions. You can also use the thumbnail creation capabilities already build in Scrapy by defining the thumbnail size via the `IMAGES_THUMBS` parameter.

Choosing store format for images

Different from Scrapy behaviour DDS is by default storing only one image in a flat store format directly under the `IMAGES_STORE` directory (Scrapy is creating a `full/` subdirectory for the original image). If you use the `IMAGES_THUMBS` setting, the scaled down thumbnail image will replace the image with the original size. Due to this simplification you can only use one entry in your `IMAGES_THUMBS` dictionary and the name of the key there doesn't matter.

Starting with `DDS v.0.3.9` you can change this behaviour with the `DSCRAPER_IMAGES_STORE_FORMAT` setting:

```
DSCRAPER_IMAGES_STORE_FORMAT = 'FLAT'   # The original image or - if available - one thumbnail image
DSCRAPER_IMAGES_STORE_FORMAT = 'ALL'    # Both the original image and all given thumbnail sizes
DSCRAPER_IMAGES_STORE_FORMAT = 'THUMBS' # Only the thumbnails
```

`FLAT` is the default setting with the behaviour described above. The `ALL` setting restores the Scrapy behaviour, the original images are stored in a `full/` directory under `IMAGES_STORE`, thumbnail files - if available - in separate sub directories for different thumbnail sizes (e.g. `thumbs/small/`).

Setting `DSCRAPER_IMAGES_STORE_FORMAT` to `THUMBS`, keeps only the thumbnail files, this setting makes only sense with setting the `IMAGES_THUMBS` setting as well. With `ALL` or `THUMBS` you can also use different sizes for thumbnail creation.

Note: Differing from the Scrapy output, an image is stored in the DB just by name, omitting path information like `full/`

Note: For image scraping to work you need the Pillow Library (PIL fork).

Updating domain model class/scraped obj class definition

When Scrapy is downloading images it creates a new unique random file name for each image saved in your image folder defined above. To keep a reference to the image associated with a scraped item DDS will save this filename in a field you have to define in your model class. In our open news example, we use 'thumbnail' as a field name:

```
class Article(models.Model):
    title = models.CharField(max_length=200)
    news_website = models.ForeignKey(NewsWebsite)
    description = models.TextField(blank=True)
    thumbnail = models.CharField(max_length=200)
    checker_runtime = models.ForeignKey(SchedulerRuntime)

    def __unicode__(self):
        return self.title
```

Note, that since there is just the filename of the image saved, you should declare this field as a simple CharField and not using UrlField or ImageField.

Now you have to update your *ScrapedObjClass* definition in the Django admin interface. Add a new attribute with the same name like in your model class and choose *IMAGE* as the attribute type. *IMAGE* is a special type to let your scraper know, that the image pipeline of DDS should be used when scraping this attribute.

Extending/Testing the scraper

At last we have to add a new scraper elem to our scraper, again in the Django admin interface, which scrapes and builds together the url of the image for the image pipeline to download later. Let's have a look at the Wikinews website of our open news example. On the news article overview page there is also an image presented with each article summary, which we want to scrape. `div[@class="l_image"]/a/img/@src` should provide us with the url of that image. Since the image urls we scrape with our XPath are starting with a double slash '//' and not with 'http://', we also have to use a pre_url processor with `'pre_url': 'http:'` as the processor context to complete the url.

That's it! If you now run your scraper, you should see lines like the following in the output (if you are in debug mode) and you should end up with the images saved in your defined images folder and the names of these images stored in the image field of your domain model in the DB:

```
DEBUG: Image (downloaded): Downloaded image from <GET http://upload.wikimedia.org/wikipedia/commons/t
...
u'thumbnail': '1bb3308a4c70b912ba6cf9d67344bb53476d70a2.jpg',
```

So now you have all these images, but how to rid of them if you don't need them any more? If you use a checker to delete scraped items not existing any more, your images will be automatically deleted as well. However, if you manually delete scraped items in your database, you have to delete the associated file yourself.

2.4.7 Where to go from here

So now that you have got your scraper up and running and maybe even integrated some of the advanced stuff like pagination or scraping images, does that mean that life will become boring because there is nothing to be done left? Definitely not! Here are some ideas about what to do next:

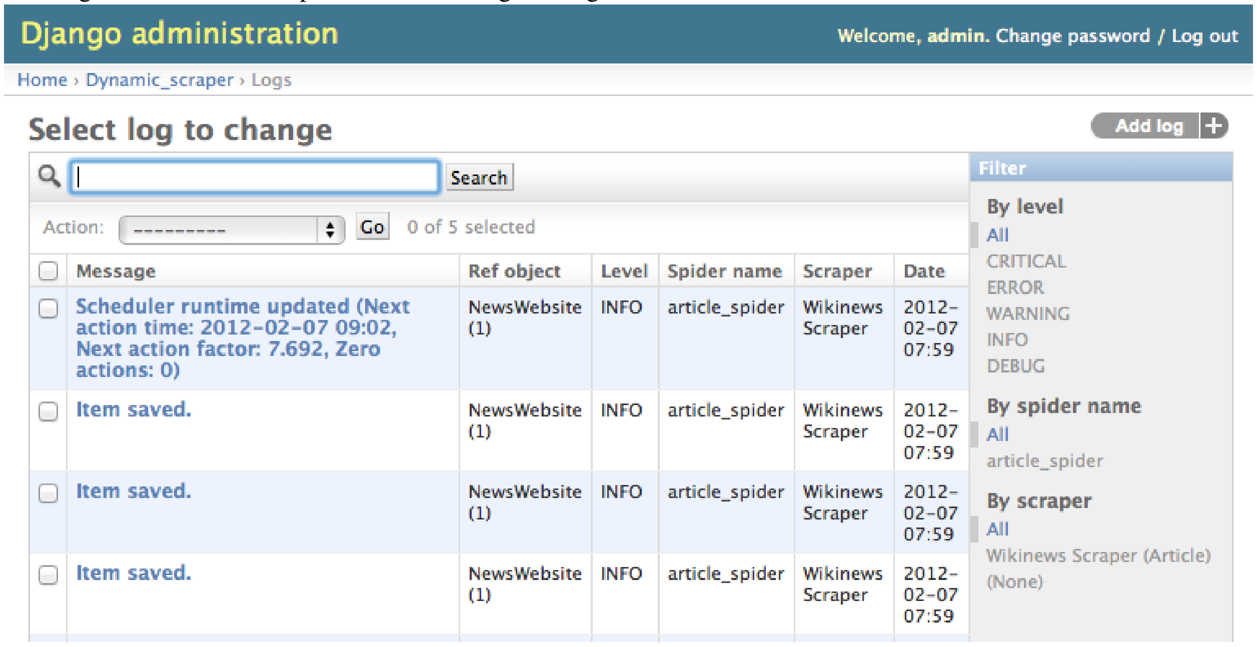
- Contribute to Django Dynamic Scraper through the experiences you made while using it (see *How to contribute*)
- Make your scraped data searchable with Django Haystack
- Provide an API to your scraped data so that others can use it with Django Tastypie
- Or... just do something no one has ever done before! :-)

2.5 Basic services

2.5.1 Logging / Log Markers

Introduction

Django Dynamic Scraper provides its own logging mechanism in addition to the build-in logging from Scrapy. While the Scrapy logging is mainly for debugging your scrapers during creation time, the DDS logging aims to get an overview how your scheduled scraper runs are doing over time, if scrapers and checkers defined with DDS are still working and how often scraper or cheker runs go wrong.



Message	Ref object	Level	Spider name	Scraper	Date
Scheduler runtime updated (Next action time: 2012-02-07 09:02, Next action factor: 7.692, Zero actions: 0)	NewsWebsite (1)	INFO	article_spider	Wikinews Scraper	2012-02-07 07:59
Item saved.	NewsWebsite (1)	INFO	article_spider	Wikinews Scraper	2012-02-07 07:59
Item saved.	NewsWebsite (1)	INFO	article_spider	Wikinews Scraper	2012-02-07 07:59
Item saved.	NewsWebsite (1)	INFO	article_spider	Wikinews Scraper	2012-02-07 07:59

In the screenshot above you see an overview of the log table in the Django admin in which new log messages are saved. In addition context information like the name of the spider run or the associated reference object or scraper is provided. By using the filtering options it is possible to track down the messages targeted to the actual needs, e.g. you can filter all the errors occurred while running your checkers.

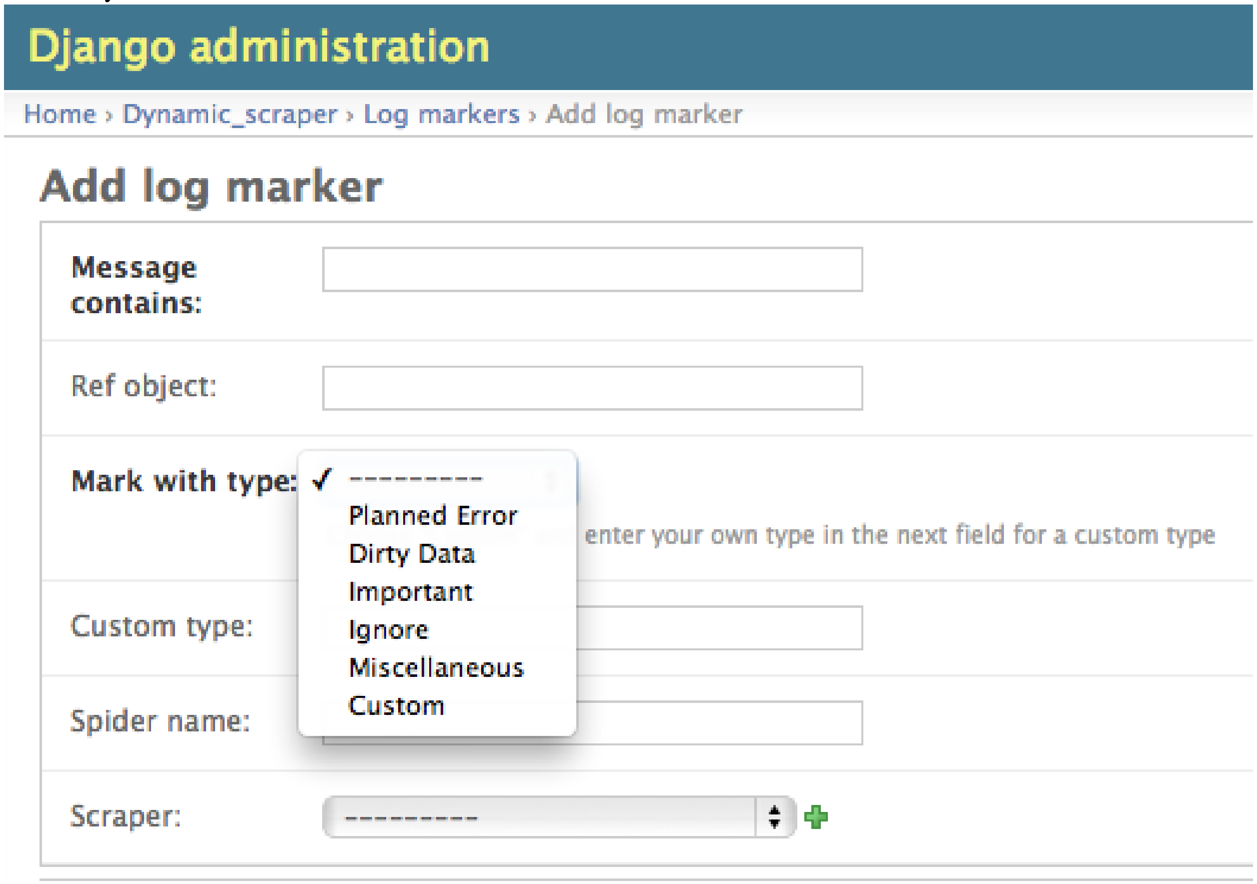
Logging: When and Where

When DDS scrapers are run from the command line both the logging messages from Scrapy as well as the DDS logging messages are provided. In the Django model log table, only the DDS messages are kept.

DDS only saves the DDS log messages in the DB when running with `run_type=TASK` and `do_action=yes`. This is configuration used when running scrapers or checkers via the scheduler. When you run your scraper via the command line you have to provide these options manually to have your DDS log messages saved in the DB (see *Running/Testing your scraper*) in addition to be displayed on the screen.

Log Markers: Meaning to your logs

Going through log entries and finding out what's wrong with your scrapers can be relatively tricky. One reason for that is that not all log entries are equally meaningful. Sometimes scraping errors could just be planned when creating the scraper, e.g. when using pagination for pages from 1 to 100, knowing that there are no items on some pages in between, leading to "No base objects" log entries. Or the data being scraped is a bit dirty, occasionally missing a mandatory field.


Django administration

Home › Dynamic_scraper › Log markers › Add log marker

Add log marker

Message contains:	
Ref object:	
Mark with type:	✓ ---------
Custom type:	
Spider name:	
Scraper:	---------

To get more meaning from your logs log markers come into play. Log markers are rules to mark all new log entries with a special type while the log marker exists. For the pagination above you can e.g. create a log marker, which marks all log entries as "Planned Error" type which contain the message "No base objects" and are coming from the corresponding scraper. With creating rules for the most common types of errors like these it becomes easier to concentrate on the potentially more severe errors by filtering down to the "None" type entries in your logs.

Note: Attention! Keep in mind that log markers can only be hints to a certain source of an error. When looking at the pagination example above it can also be the case that a "No base objects" error occur on a page where there should be some items and the scraper really not working any more. So be cautious! Log markers can only give a better orientation with your log entries and don't necessarily are telling the truth in all situations.

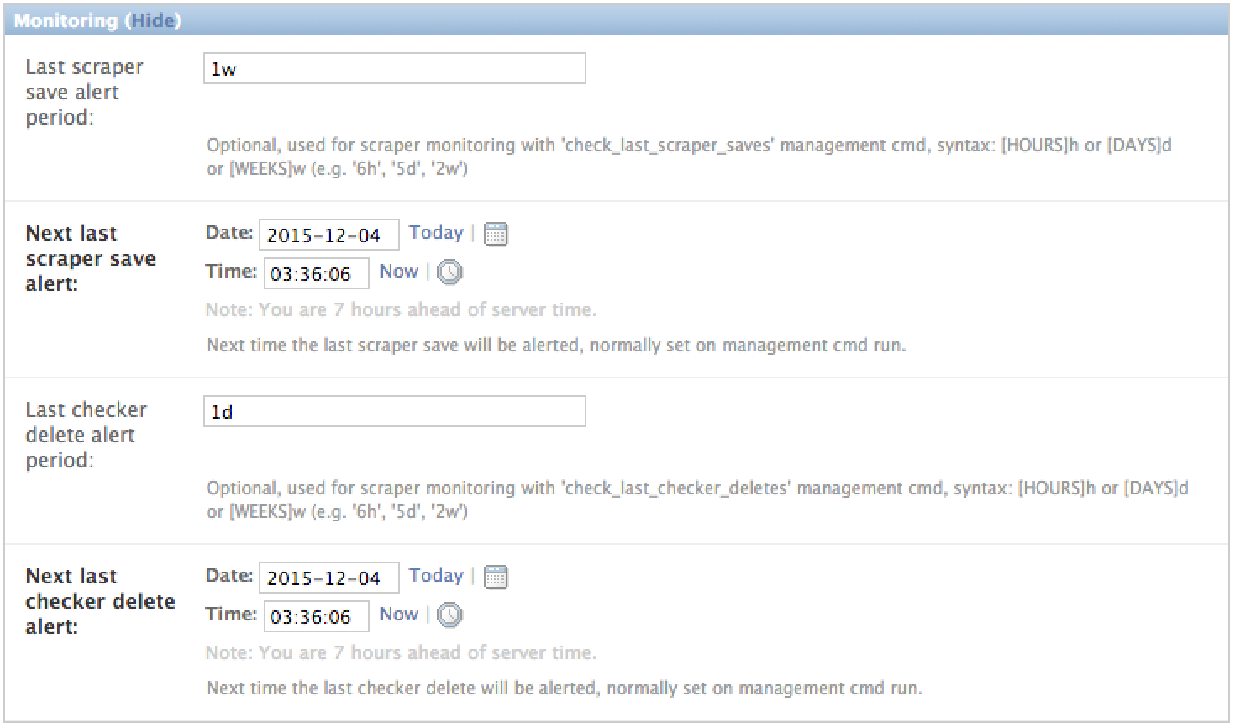
Configuration

You can configure DDS logging behaviour by providing some settings in your *settings.py* configuration file (see *Settings*).

2.5.2 Monitoring

Configuration

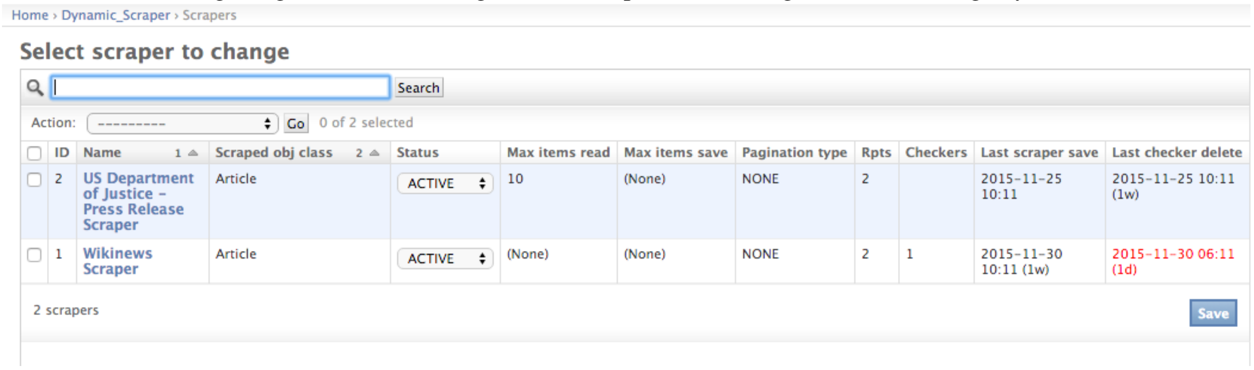
There is a montoring section in the DDS scraper admin form with basic settings which can be used to monitor scraper/checker functionality by checking when the last_scraper_save or last_checker_delete occurred:



Monitoring (Hide)

Last scraper save alert period:	1w 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:	Date: 2015-12-04 Today | Time: 03:36:06 Now | Note: You are 7 hours ahead of server time. Next time the last scraper save will be alerted, normally set on management cmd run.
Last checker delete alert period:	1d 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:	Date: 2015-12-04 Today | Time: 03:36:06 Now | Note: You are 7 hours ahead of server time. Next time the last checker delete will be alerted, normally set on management cmd run.

If last_scraper_save_alert_period or last_checker_delete_alert_period is set with an alert period in the format demanded it is indicated by red timestamps on the admin scraper overview page if a scraper save or checker delete is getting too old, indicating that the scraper/checker might not be working any more.



Home › Dynamic_Scraper › Scrapers

Select scraper to change

	ID	Name 1	Scraped obj class 2	Status	Max items read	Max items save	Pagination type	Rpts	Checkers	Last scraper save	Last checker delete
☐	2	US Department of Justice - Press Release Scraper	Article	ACTIVE	10	(None)	NONE	2		2015-11-25 10:11	2015-11-25 10:11 (1w)
☐	1	Wikinews Scraper	Article	ACTIVE	(None)	(None)	NONE	2	1	2015-11-30 10:11 (1w)	2015-11-30 06:11 (1d)

2 scrapers

Monitoring Automation

You can use the following Django management commands to monitor your scrapers and checkers on a regular basis:

```
python manage.py check_last_scraper_saves [--send-admin-mail] [--with-next-alert]
python manage.py check_last_checker_deletes [--send-admin-mail] [--with-next-alert]
```

Standard behaviour of the commands is to check, if the last scraper save or last checker delete occured is older than the corresponding alert period set (see configuration section above). If the `--send-admin-mail` flag is set an alert mail will be send to all admin users defined in the Django `settings.py` file. Additionally the next alert timestamps (see Django admin form) will be set to the current timestamp.

Practically this leads to a lot of alerts/mails (depending on the frequency of your cronjobs) once an alert situation triggers. If you want to switch from a `Report-Always` to a `Report-Once` (more or less) behaviour you can set the `--with-next-alert` flag.

This will run alert checks only for scrapers where the corresponding next alert timestamp has passed. The timestamp is then updated by the alert period set as the earliest time for a new alert.

An alert for a scraper with an alert period of 2 weeks will then trigger first after the last item was scraped more than 2 weeks ago. With the above flag, the next alert will then be earliest 2 weeks after the first alert.

Note: Using the `--with-next-alert` flag only makes sense if your periods for your alerts are significantly longer (e.g. 1 week+) than your cronjob frequency (e.g. every day).

2.6 Reference

2.6.1 Settings

For the configuration of Django Dynamic Scraper you can use all the basic settings from Scrapy, though some settings may not be useful to change in the context of DDS. In addition DDS defines some extra settings with the prefix `DSCRAPER`. You can also place these settings in the Scrapy `settings.py` configuration file. At the moment this is the only way to define DDS settings and you can't change DDS settings via command line parameters.

DSCRAPER_IMAGES_STORE_FORMAT

Default: `FLAT`

Store format for images (see *Scraping images/screenshots* for more info).

FLAT	Storing only either original or one thumbnail image, no sub folders
ALL	Storing original (in `full/`) and thumbnail images (e.g. in `thumbs/small/`)
THUMBS	Storing only the thumbnail images (e.g. in `thumbs/small/`)

DSCRAPER_SPLASH_ARGS

Default: `{ 'wait': 0.5 }`

Customize `Splash` args when `ScrapyJS/Splash` is used for Javascript rendering.

DSCRAPER_LOG_ENABLED

Default: `True`

Enable/disable the DDS logging (see *Logging / Log Markers* for more info).

DSCRAPER_LOG_LEVEL

Default: `ERROR`

Set the log level for DDS logging. Possible values are CRITICAL, ERROR, WARNING, INFO and DEBUG.

DSCRAPER_LOG_LIMIT

Default: `250`

The number of log entries in the Django database table.

DSCRAPER_MAX_SPIDER_RUNS_PER_TASK

Default: `10`

Maximum number of spider runs executed per task run.

DSCRAPER_MAX_CHECKER_RUNS_PER_TASK

Default: `25`

Maximum number of checker runs executed per task run.

2.6.2 Django Model Reference

TODO

ScrapedObjClass

TODO

ScrapedObjAttr

TODO

Scraper

`status`

Status of the scraper, influencing in which context the scraper is executed.

ACTIVE	Scraper can be run manually and is included on scheduled task execution
MANUAL	Scraper can only be run manually and is ignored on scheduled task execution
PAUSE	Scraper is not executed, use for short pausing
INACTIVE	Scraper is not executed, use for longer interruption of scraper use

ScraperElem

TODO

SchedulerRuntime

TODO

2.6.3 API Reference

TODO

DjangoSpider

TODO

DjangoChecker

TODO

2.6.4 Processors

General Functionality

Attribute Placeholders

Processors can use placeholders referencing other scraped attributes in the form of `{ATTRIBUTE_NAME}`. These placeholders are then replaced with the other scraped attribute string after all other processing steps (scraping, regex, processors).

Attribute placeholders can also be used to form **detail page URLs**. This can be used for more flexible detail page creation, e.g. by defining a non-saved help attribute `tmp_attr_1` in your `ScrapedObjClass` definition and using a `pre_url` processor like `'pre_url': 'http://someurl.org/{tmp_attr_1}'`.

Note: Placeholders for detail page URLs can only be used with attributes scraped from the main page!

Processor Description

string_strip

Description	Applies the python strip function to remove leading and trailing characters
Usable with other processors	Yes
Context definition (Example)	`'string_strip': '   .!'` (optional, default: ‘ ntr’)
Result (Example)	” ... Example Text!!!” -> “Example Text”

remove_chars

Description	Removing of characters or character pattern using the python re.sub function by providing a regex pattern
Usable with other processors	Yes
Context definition (Example)	`'remove_chars': '[-\.]+'`
Result (Example)	"Example... Text–!–!!" -> "Example Text!!!"

pre_string

Description	Adds a string before the scraped text
Usable with other processors	Yes
Context definition (Example)	`'pre_string': 'BEFORE_'`
Result (Example)	"Example Text" -> "BEFORE_Example Text"

post_string

Description	Appends a string after the scraped text
Usable with other processors	Yes
Context definition (Example)	`'post_string': '_AFTER'`
Result (Example)	"Example Text" -> "Example Text_AFTER"

pre_url

Description	Adding a domain to scraped url paths, works like pre_string with some url specific enhancements (throwing away defined domain when scraped text has a leading "http://" e.g.)
Usable with other processors	Yes
Context definition (Example)	`'pre_url': 'http://example.org/'`
Result (Example)	"/path/to/page.html" -> "http://example.org/path/to/page.html"

replace

Description	When the scraper succeeds in scraping the attribute value, the text scraped is replaced with the replacement given in the processor context.
Usable with other processors	No
Context definition (Example)	`'replace': 'This is a replacement'`
Result (Example)	"This text was scraped" -> "This is a replacement"

static

Description	No matter if the scraper succeeds in scraping the attribute value or not, the static value is used as an attribute value. This processor is also useful for testing for not relying on too many x_path values having to succeed at once.
Usable with other processors	No
Context definition (Example)	`'static': 'Static text'`
Result (Example)	"No matter if this text was scraped or not" -> "Static text"

date

Description	Tries to parse a date with Python's strptime function (extra sugar: recognises 'yesterday', 'gestern', 'today', 'heute', 'tomorrow', 'morgen')
Usable with other processors	Yes
Context definition (Example)	`'date': '%d.%m.%Y'`
Result (Example)	"04.12.2011" -> "2011-12-04"

time

Description	Tries to parse a time with Python's strptime function
Usable with other processors	Yes
Context definition (Example)	`'time': '%H hours %M minutes'`
Result (Example)	"22 hours 15 minutes" -> "22:15"

ts_to_date

Description	Tries to extract the local date of a unix timestamp
Usable with other processors	Yes
Context definition (Example)	No context definition
Result (Example)	"1434560700" -> "2015-06-17"

ts_to_time

Description	Tries to extract the local time of a unix timestamp
Usable with other processors	Yes
Context definition (Example)	No context definition
Result (Example)	"1434560700" -> "19:05:00"

duration

Description	Tries to parse a duration, works like time processor but with time unit overlap breakdown
Usable with other processors	Yes
Context definition (Example)	`'duration': '%M Minutes'`
Result (Example)	“77 Minutes” -> “01:17:00”

2.7 Development

2.7.1 How to contribute

You can contribute to improve Django Dynamic Scraper in many ways:

- If you stumbled over a bug or have suggestions for an improvements or a feature addition report an issue on the GitHub page with a good description.
- If you have already fixed the bug or added the feature in the DDS code you can also make a pull request on GitHub. While I can’t assure that every request will be taken over into the DDS source I will look at each request closely and integrate it if I fell that it’s a good fit!
- Since this documentation is also available in the Github repository of DDS you can also make pull requests for documentation!

Here are some topics for which suggestions would be especially interesting:

- If you worked your way through the documentation and you were completely lost at some point, it would be helpful to know where that was.
- If there are unnecessary limitations of the Scrapy functionality in the DDS source which could be eliminated without adding complexity to the way you can use DDS that would be very interesting to know.

And finally: please let me know about how you are using Django Dynamic Scraper!

2.7.2 Running the test suite

Overview

Tests for `DDS` are organized in a separate `tests` Django project in the root folder of the repository. Due to restrictions of Scrapy’s networking engine Twisted, `DDS` test cases directly testing scrapers have to be run as new processes and can’t be executed sequentially via *python manage.py test*.

For running the tests first go to the *tests* directory and start a test server with:

```
./testserver.sh
```

Then you can run the test suite with:

```
./run_tests.sh
```

Note: If you are testing for DDS Django/Scrapy version compatibility: there might be 2-3 tests generally not working properly, so if just a handful of tests don’t pass have a closer look at the test output.

Django test apps

There are currently two Django apps containing tests. The `basic` app testing scraper unrelated functionality like correct processor output or scheduling time calculations. These tests can be run on a per-file-level:

```
python manage.py test basic.processors_test.ProcessorsTest
```

The `scraper` app is testing scraper related functionality. Tests can either be run via shell script (see above) or on a per-test-case level like this:

```
python manage.py test scraper.scraper_run_test.ScraperRunTest.test_scraper #Django 1.6+
python manage.py test scraper.ScraperRunTest.test_scraper #Django up to 1.5
```

Have a look at the `run_tests.sh` shell script for more examples!

Running ScrapyJS/Splash JS rendering tests

Unit tests testing `ScrapyJS/Splash` Javascript rendering functionality need a working `ScrapyJS/Splash` (docker) installation and are therefor run separately with:

```
./run_js_tests.sh
```

Test cases are located in `scraper.scraper_js_run_test.ScraperJSRunTest`. Some links:

- Splash Documentation
- ScrapyJS GitHub
- Installation of Docker on OS X with Homebrew

`SPLASH_URL` in `scraper.settings.base_settings.py` has to be adopted to your local installation to get this running!

Docker container can be run with:

```
docker run -p 5023:5023 -p 8050:8050 -p 8051:8051 -d scrapinghub/splash
```

Note: For rendering websites served on `localhost` from within `Docker/Splash`, you can connect to `localhost` outside the `Docker container` via `http://10.0.2.2` (see e.g. Stackoverflow).

2.7.3 Release Notes

Changes in version 0.11.0-beta (2016-05-13)

- First major release version with support for new `Scrapy 1.0+` structure (only `Scrapy 1.1` officially supported)
- From this release on older Scrapy versions like `0.24` are not supported any more, please update your Scrapy version!
- Beta `Python 3` support
- Support for `Django 1.9`
- The following manual adoptions in your project are necessary:

  - Scrapy's `DjangoItem` class has now moved from `scrapy.contrib.djangoitem` to a separate repository `scrapy-djangoitem` (see Scrapy docs). The package has to be separately installed with `pip install scrapy-djangoitem` and the import in your `models.py` class has to be changed to `from scrapy_djangoitem import DjangoItem` (see: *Creating your Django models*)
  - Due to Scrapy's switch to Python's build-in logging functionality the logging calls in your custom pipeline class have to be slightly changed, removing the `from scrapy import log` import and changing the `log.[LOGLEVEL]` attribute handover in the log function call to `logging.[LOGLEVEL]` (see: *Adding the pipeline class*)
  - Change `except IntegrityError, e:` to `except IntegrityError as e:` in your custom `pipelines.py` module (see: *Adding the pipeline class*)
- Following changes have been made:
  - Changed logging to use Python's build-in `logging` module
  - Updated import paths according to Scrapy release documentation
  - Running most of the unit tests in parallel batches (when using the shell scripts) to speed up test runs
  - Updated `django-celery` version requirement to `3.1.17` to work with `Django 1.9`
  - Updated open_news example fixture, introduction of versioned fixture data dumps
  - Removed dependency on `scrapy.xlib.pydispatch` being removed in `Scrapy 1.1` (former `DDS v.0.10` releases will break with `Scrapy 1.1`)
- If you use `Scrapy/Splash` for `Javascript` rendering:
  - Updated dependencies, replaced `scrapyjs` with `scrapy-splash` (renaming), please update your dependencies accordingly!
- Bugfixes:
  - Fixed bug with `DSCRAPER_IMAGES_STORE_FORMAT` set to `THUMBS` not working correctly

Changes in version 0.10.0-beta EXPERIMENTAL (2016-01-27)

- Experimental release branch no longer maintained, please see release notes for `0.11`.

Changes in version 0.9.6-beta (2016-01-26)

- Fixed a severe bug causing scrapers to break when scraping unicode text
- Making unicode text scraping more robust
- Added several unit tests testing unicode string scraping/usage in various contexts
- Reduce size of textarea fields in scraper definitions
- Added order attribute for scraped object attributes for convenience when editing scrapers (see: *Defining the object to be scraped*)
- New migration `0017`, run Django `migrate` command

Changes in version 0.9.5-beta (2016-01-18)

- Fixed a severe bug when using non-saved detail page URLs in scrapers

Changes in version 0.9.4-beta (2016-01-15)

- Fixed a critical bug when using non-saved fields for scraping leading to incorrect data attribution to items

Changes in version 0.9.3-beta (2016-01-14)

- New command line options `output_num_mp_response_bodies` and `output_num_dp_response_bodies` for logging the complete response bodies of the first {Int} main/detail page responses to the screen for debugging (for the really hard cases :-)) (see: *Running/Testing your scraper*)

Changes in version 0.9.2-beta (2016-01-14)

- New processor `remove_chars` (see: *Processors*) for removing one or several type of chars from a scraped string

Changes in version 0.9.1-beta (2016-01-13)

- Allowing empty `x_path` scraper attribute fields for easier appliance of `static` processor to fill in static values
- Enlargening `x_path`, `reg_exp` and `processor` fields in Django admin scraper definition from `CharField` to `TextField` for more extensive `x_path`, `reg_exp` and `processor` definitions and more comfortable input/editing
- New command line option `max_pages_read` for limiting the number of pages read on test runs (see: *Running/Testing your scraper*)
- New migration `0016`, run Django `migrate` command

Changes in version 0.9.0-beta (2016-01-11)

- BREAKING!!! This release slighly changes the semantics of the internal `ValidationPipeline` class in `dynamic_scraper/pipelines.py` to also pass items to your custom user pipeline when the `do_action` command line parameter (see: *Running/Testing your scraper*) is not set. This creates the need of an additional `if spider.conf['DO_ACTION']:` restriction in your custom user pipeline function (see: *Adding the pipeline class*). Make sure to add this line, otherwise you will get unwanted side effects. If you do more stuff in your custom pipeline class also have a broader look if this new behaviour changes your processing (you should be save though if you apply the `if` restriction above to all of your code in the classs).
- Decoupling of `DDS Django` item save mechanism for the pipeline processing to allow the usage of Scrapy's build-in output options `--output=FILE` and `--output-format=FORMAT` to scrape items into a file instead of the DB (see: *Running/Testing your scraper*).
- The above is the main change, not touching too much code. Release number nevertheless jumped up a whole version number to indicate a major breaking change in using the library!
- Another reason for the new `0.9` version number is the amount of new features being added throuhout minor `0.8` releases (more flexible checker concept, monitoring functionality, attribute placeholders) to point out the amount of changes since `0.8.0`.

Changes in version 0.8.13-beta (2016-01-07)

- Expanded detail page URL processor placeholder concept to generic attribute placeholders (*Attribute Placeholders*)
- Unit test for new functionality

Changes in version 0.8.12-beta (2016-01-06)

- Fixed `Clone Scraper` Django admin action omitting the creation of `RequestPageType` and `Checker` objects introduced in the `0.8` series
- Narrowing the requirements for `Pillow` to `3.x` versions to reduce possible future side effects

Changes in version 0.8.11-beta (2016-01-05)

- New *Attribute Placeholders* (previously: detail page URL placeholder) which can be used for more flexible detail page URL creation
- Unit test for new functionality

Changes in version 0.8.10-beta (2015-12-04)

- New `--with-next-alert` flag for monitoring management cmds to reduce amount of mail alerts, see updated *Monitoring* section for details
- More verbose output for monitoring management cmds
- New migration `0015`, run Django `migrate` command

Changes in version 0.8.9-beta (2015-12-01)

- Minor changes

Changes in version 0.8.8-beta (2015-12-01)

- Fixed a bug in `Django admin` from previous release

Changes in version 0.8.7-beta (2015-12-01)

- New syntax/semantics of management commands `check_last_checker_deletes` and `check_last_scraper_saves`
- Added `last_scraper_save_alert_period` and `last_checker_delete_alert_period` alert period fields for scraper, new migration `0014`, run Django `migrate` command
- New fields are used for providing time periods for the lowest accepted value for last scraper saves and checker deletes, these values are then checked by the management commands above (see: *Monitoring*)
- Older timestamps for current values of a scraper for `last_scraper_save` and `last_checker_delete` also trigger a visual warning indication in the Django admin scraper overview page

Changes in version 0.8.6-beta (2015-11-30)

- Two new management commands `check_last_checker_deletes` and `check_last_scraper_saves` which can be run as a cron job for basic scraper/checker monitoring (see: *Monitoring*)

Changes in version 0.8.5-beta (2015-11-30)

- New `last_scraper_save`, `last_checker_delete datetime` attributes for `Scraper` model for monitoring/ statistis purposes (can be seen on `Scraper` overview page in `Django admin`)
- New migration `0013`, run Django `migrate` command

Changes in version 0.8.4-beta (2015-11-27)

Starting update process for `Python 3` support with this release (not there yet!)

- Fixed severe bug in `task_utils.py` preventing checker scheduling to work
- New dependency on Python-Future 0.15+ to support integrated `Python 2/3` code base, please install with `pip install future`
- Updating several files for being `Python 2/3` compatible

Changes in version 0.8.3-beta (2015-10-01)

- More flexible checker concept now being an own `Checker` model class and allowing for more than one checker for a single scraper. This allows checking for different URLs or xpath conditions.
- Additional comment fields for `RequestPageTypes` and `Checkers` in admin for own notes
- Adopted unit tests to reflect new checker structure
- `self.scrape_url = self.ref_object.url` assignment in checker python class not used any more (see: *Creating a checker class*), you might directly want to remove this from your project class definition to avoid future confusion

- Some docs rewriting for Checker creation (see: *Defining item checkers*)
- New migrations `0011`, `0012`, run Django `migrate` command

Changes in version 0.8.2-beta (2015-09-24)

- Fixed bug preventing checker tests to work
- Added Javascript rendering to checkers
- Fixed a bug letting checkers/checker tests choose the wrong detail page URL for checking under certain circumstances

Changes in version 0.8.1-beta (2015-09-22)

- Fixed packaging problem not including custom static Django admin JS file (for `RequestPageType` admin form collapse/expand)

Changes in version 0.8.0-beta (2015-09-22)

- New request page types for main page and detail pages of scrapers (see: *Adding corresponding request page types*):
  - Cleaner association of request options like content or request type to main or detail pages (see: *Advanced Request Options*)
  - More flexibility in using different request options for main and detail pages (rendering Javascript on main but not on detail pages, different HTTP header or body values,...)
  - Allowance of several detail page URLs per scraper
  - Possibility for not saving the detail page URL used for scraping by unchecking corresponding new `ScrapedObjClass` attribute `save_to_db`
- ATTENTION! This release comes with heavy internal changes regarding both DB structure and scraping logic. Unit tests are running through, but there might be untested edge cases. If you want to use the new functionality in a production environment please do this with extra care. You also might want to wait for 2-3 weeks after release and/or for a following 0.8.1 release (not sure if necessary yet). If you upgrade it is HIGHLY RECOMMENDED TO BACKUP YOUR PROJECT AND YOUR DB before!
- Replaced Scrapy `Spider` with `CrawlSpider` class being the basis for `DjangoBaseSpider`, allowing for more flexibility when extending
- Custom migration for automatically creating new `RequestPageType` objects for existing scrapers
- Unit tests for new functionality
- Partly restructured documentation, separate *Installation* section
- Newly added `static` files, run Django `collectstatic` command (collaps/expand for `RequestPageType` inline admin form)
- New migrations `0008`, `0009`, `0010`, run Django `migrate` command

Changes in version 0.7.3-beta (2015-08-10)

- New attribute `dont_filter` for `Scraper` request options (see: *Advanced Request Options*), necessary for some scenarios where `Scrapy` falsely marks (and omits) requests as being duplicate (e.g. when scraping uniform URLs together with custom HTTP header pagination)
- Fixed bug preventing processing of `JSON` with non-string data types (e.g. `Number`) for scraped attributes, values are now automatically converted to `String`
- New migration `0007`, run Django `migrate` command

Changes in version 0.7.2-beta (2015-08-06)

- Added new `method` attribute to `Scraper` not binding HTTP method choice (`GET`/`POST`) so strictly to choice of `request_type` (allowing e.g. more flexible `POST` requests), see: *Advanced Request Options*
- Added new `body` attribute to `Scraper` allowing for sending custom request `HTTP message body` data, see: *Advanced Request Options*
- Allowing `pagination` for `headers`, `body` attributes
- Allowing of `ScrapedObjectClass` definitions in `Django admin` with no attributes defined as `ID field` (omits double checking process when used)
- New migration `0006`, run Django `migrate` command

Changes in version 0.7.1-beta (2015-08-03)

- Fixed severe bug preventing `pagination` for `cookies` and `form_data` to work properly
- Added a new section in the docs for *Advanced Request Options*
- Unit tests for some scraper request option selections

Changes in version 0.7.0-beta (2015-07-31)

- Adding additional HTTP header attributes to scrapers in Django admin
- Cookie support for scrapers
- Passing Scraper specific Scrapy meta data
- Support for form requests, passing form data within requests
- Pagination support for cookies, form data
- New migration `0005`, run Django `migrate` command
- All changes visible in Scraper form of Django admin
- ATTENTION! While unit tests for existing functionality all passing through, new functionality is not heavily tested yet due to problems in creating test scenarios. If you want to use the new functionality in a production environment please test with extra care. You also might want to wait for 2-3 weeks after release and/or for a following 0.7.1 release (not sure if necessary yet)
- Please report problems/bugs on GitHub.

Changes in version 0.6.0-beta (2015-07-14)

- Replaced implicit and static ID concept of mandatory `DETAIL_PAGE_URL` type attribute serving as ID with a more flexible concept of explicitly setting `ID Fields` for `ScrapedObjClass` in `Django` admin (see: *Defining the object to be scraped*)
- New attribute `id_field` for `ScrapedObjClass`, please run Django `migrate` command (migration `0004`)
- `DETAIL_PAGE_URL` type attribute not necessary any more when defining the scraped object class allowing for more scraping use cases (classic and simple/flat datasets not referencing a certain detail page)
- Single `DETAIL_PAGE_URL` type `ID Field` still necessary for using `DDS` checker functionality (see: *Defining item checkers*)
- Additional form checks for `ScrapedObjClass` definition in `Django` admin

Changes in version 0.5.2-beta (2015-06-18)

- Two new processors `ts_to_date` and `ts_to_time` to extract local date/time from unix timestamp string (see: *Processors*)

Changes in version 0.5.1-beta (2015-06-17)

- Make sure that `Javascript` rendering is only activated for pages with `HTML` content type

Changes in version 0.5.0-beta (2015-06-10)

- Support for creating `JSON/JSONPath` scrapers for scraping `JSON` encoded pages (see: *Scraping JSON content*)
- Added new separate content type choice for detail pages and checkers (e.g. main page in `HTML`, detail page in `JSON`)
- New Scraper model attribute `detail_page_content_type`, please run Django `migration` command (migration `0003`)
- New library dependency `python-jsonpath-rw 1.4+` (see *Requirements*)
- Updated unit tests to support/test `JSON` scraping

Changes in version 0.4.2-beta (2015-06-05)

- Possibility to customize `Splash` args with new setting `DSCRAPER_SPLASH_ARGS` (see: *Setting up ScrapyJS/Splash (Optional)*)

Changes in version 0.4.1-beta (2015-06-04)

- Support for `Javascript` rendering of scraped pages via `ScrapyJS/Splash`
- Feature is optional and needs a working ScrapyJS/Splash deployment, see *Requirements* and *Setting up ScrapyJS/Splash (Optional)*
- New attribute `render_javascript` for `Scraper` model, run `python manage.py migrate dynamic_scraper` to apply (migration `0002`)
- New unit tests for Javascript rendering (see: *Running ScrapyJS/Splash JS rendering tests*)

Changes in version 0.4.0-beta (2015-06-02)

- Support for `Django 1.7/1.8` and `Scrapy 0.22/0.24`. Earlier versions not supported any more from this release on, if you need another configuration have a look at the `DDS 0.3.x` branch (new features won't be back-ported though) (see *Release Compatibility Table*)
- Switched to Django migrations, removed `South` dependency
- Updated core library to work with `Django 1.7/1.8` (`Django 1.6` and older not working any more)
- Replaced deprecated calls logged when run under `Scrapy 0.24` (`Scrapy 0.20` and older not working any more)
- Things to consider when updating Scrapy: new `ITEM_PIPELINES` dict format, standalone `scrapyd` with changed `scrapy.cfg` settings and new deployment procedure (see: *Scrapy Configuration*)
- Adopted `example_project` and `tests` Django projects to work with the updated dependecies
- Updated `open_news.json` example project fixture
- Changed `DDS` status to `Beta`

Changes in version 0.3.14-alpha (2015-05-30)

- Pure documentation update release to get updated `Scrapy 0.20/0.22/.24` compatibility info in the docs (see: *Release Compatibility Table*)

Changes in version 0.3.13-alpha (2015-05-29)

- Adopted test suite to pass through under `Scrapy 0.18` (Tests don't work with `Scrapy 0.16` any more)
- Added `Scrapy 0.18` to release compatibility table (see: *Release Compatibility Table*)

Changes in version 0.3.12-alpha (2015-05-28)

- Added new release compatibility overview table to docs (see: *Release Compatibility Table*)

- Adopted `run_tests.sh` script to run with `Django 1.6`
- Tested `Django 1.5`, `Django 1.6` for compatibility with `DDS v.0.3.x`
- Updated title xpath in fixture for Wikinews example scraper

Changes in version 0.3.11-alpha (2015-04-20)

- Added `only-active` and `--report-only-erros` options to `run_checker_tests` management command (see: *Run checker tests*)

Changes in version 0.3.10-alpha (2015-03-17)

- Added missing management command for checker functionality tests to distribution (see: *Run checker tests*)

Changes in version 0.3.9-alpha (2015-01-23)

- Added new setting `DSCRAPER_IMAGES_STORE_FORMAT` for more flexibility with storing original and/or thumbnail images (see *Scraping images/screenshots*)

Changes in version 0.3.8-alpha (2014-10-14)

- Added ability for `duration` processor to break down and parse second values greater than one hour in total (>= 3600 seconds) (see: *Processors*)

Changes in version 0.3.7-alpha (2014-03-20)

- Improved `run_checker_tests` management command with `--send-admin-mail` flag for usage of command in cronjob (see: *Run checker tests*)

Changes in version 0.3.6-alpha (2014-03-19)

- Added new admin action clone_scrapers to get a functional copy of scrapers easily

Changes in version 0.3.5-alpha (2013-11-02)

- Add super init method to call init method in Scrapy BaseSpider class to DjangoBaseSpider init method (see Pull Request #32)

Changes in version 0.3.4-alpha (2013-10-18)

- Fixed bug displaying wrong message in checker tests
- Removed `run_checker_tests` celery task (which wasn't working anyway) and replaced it with a simple Django management command `run_checker_tests` to run checker tests for all scrapers

Changes in version 0.3.3-alpha (2013-10-16)

- Making status list editable in Scraper admin overview page for easier status change for many scrapers at once
- Possibility to define `x_path` checkers with blank `checker_x_path_result`, the checker is then succeeding if elements are found on page (before this lead to an error message)

Changes in version 0.3.2-alpha (2013-09-28)

- Fixed the exception when scheduler string was processed (see Pull Request #27)
- Allowed Checker Reference URLs to be longer than the the default 200 characters (DB Migration `0004`, see Pull Request #29)
- Changed `__unicode__` method for `SchedulerRuntime` to prevent `TypeError` (see Google Groups Discussion)
- Refer to `ID` instead of `PK` (see commit in nextlanding repo)

Changes in version 0.3.1-alpha (2013-09-03)

- Possibility to add keyword arguments to spider and checker task method to specify which reference objects to use for spider/checker runs (see: *Defining your tasks*)

Changes in version 0.3-alpha (2013-01-15)

- Main purpose of release is to upgrade to new libraries (Attention: some code changes necessary!)
- `Scrapy 0.16`: The `DjangoItem` class used by DDS moved from `scrapy.contrib_exp.djangoitem` to `scrapy.contrib.djangoitem`. Please update your Django model class accordingly (see: *Creating your Django models*).
- `Scrapy 0.16`: `BOT_VERSION` setting no longer used in Scrapy/DDS `settings.py` file (see: *Setting up Scrapy*)
- `Scrapy 0.16`: Some minor import changes for DDS to get rid of deprecated settings import
- `Django 1.5`: Changed Django settings configuration, please update your Scrapy/DDS `settings.py` file (see: *Setting up Scrapy*)
- `django-celery 3.x`: Simpler installation, updated docs accordingly (see: *Installing/configuring django-celery for DDS*)
- New log output about which Django settings used when running a scraper

Changes in version 0.2-alpha (2012-06-22)

- Substantial API and DB layout changes compared to version 0.1
- Introduction of South for data migrations

Changes in version 0.1-pre-alpha (2011-12-20)

- Initial version

2.7.4 Roadmap

[THIS ROADMAP IS PARTIALLY OUTDATED!]

pre-alpha

Django Dynamic Scraper's pre-alpha phase was meant to be for people interested having a first look at the library and give some feedback if things were making generally sense the way they were worked out/conceptionally designed or if a different approach on implementing some parts of the software would have made more sense.

alpha

DDS is currently in alpha stadium, which means that the library has proven itself in (at least) one production environment and can be (cautiously) used for production purposes. However being still very early in develpment, there are still API and DB changes for improving the lib in different ways. The alpha stadium will be used for getting most parts of the API relatively stable and eliminate the most urgent bugs/flaws from the software.

beta (current)

In the beta phase the API of the software should be relatively stable, though occasional changes will still be possible if necessary. The beta stadium should be the first period where it is save to use the software in production and beeing able to rely on its stability. Then the software should remain in beta for some time.

Version 1.0

Version 1.0 will be reached when the software has matured in the beta phase and when at least 10+ projects are using DDS productively for different purposes.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search