Item Pipeline

After an item has been scraped by a spider, it is sent to the Item Pipeline which processes it through several components that are executed sequentially.

Each item pipeline component (sometimes referred as just "Item Pipeline") is a Python class that implements a simple method. They receive an item and perform an action over it, also deciding if the item should continue through the pipeline or be dropped and no longer processed.

Typical uses of item pipelines are:

- cleansing HTML data
- validating scraped data (checking that the items contain certain fields)
- checking for duplicates (and dropping them)
- storing the scraped item in a database

Writing your own item pipeline

Each item pipeline component is a Python class that must implement the following method:

process_item(self, item, spider)

This method is called for every item pipeline component. `process_item()` must either: return a dict with data, return an `Item` (or any descendant class) object, return a Twisted Deferred or raise `DropItem` exception. Dropped items are no longer processed by further pipeline components.

Parameters:
- **item** (`Item` object or a dict) – the item scraped
- **spider** (`Spider` object) – the spider which scraped the item

Additionally, they may also implement the following methods:

open_spider(self, spider)

This method is called when the spider is opened.

Parameters: **spider** (`Spider` object) – the spider which was opened

close_spider(self, spider)

This method is called when the spider is closed.

Parameters: **spider** (`Spider` object) – the spider which was closed

from_crawler(cls, crawler)

If present, this classmethod is called to create a pipeline instance from a `Crawler`. It must return a new instance of the pipeline. Crawler object provides access to all Scrapy core components like settings and signals; it is a way for pipeline to access them and hook its functionality into Scrapy.

Parameters: **crawler** (`Crawler` object) – crawler that uses this pipeline

Item pipeline example

Price validation and dropping items with no prices

Let's take a look at the following hypothetical pipeline that adjusts the `price` attribute for those items that do not include VAT (`price_excludes_vat` attribute), and drops those items which don't contain a price:

```
from scrapy.exceptions import DropItem

class PricePipeline(object):

    vat_factor = 1.15

    def process_item(self, item, spider):
        if item['price']:
            if item['price_excludes_vat']:
                item['price'] = item['price'] * self.vat_factor
            return item
        else:
            raise DropItem("Missing price in %s" % item)
```

Write items to a JSON file

The following pipeline stores all scraped items (from all spiders) into a single `items.jl` file, containing one item per line serialized in JSON format:

```
import json

class JsonWriterPipeline(object):

    def open_spider(self, spider):
        self.file = open('items.jl', 'w')

    def close_spider(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        self.file.write(line)
        return item
```

Note

The purpose of JsonWriterPipeline is just to introduce how to write item pipelines. If you really want to store all scraped items into a JSON file you should use the Feed exports.

Write items to MongoDB

In this example we'll write items to MongoDB using pymongo. MongoDB address and database name are specified in Scrapy settings; MongoDB collection is named after item class.

The main point of this example is to show how to use `from_crawler()` method and how to clean up the resources properly.:

```
import pymongo

class MongoPipeline(object):

    collection_name = 'scrapy_items'

    def __init__(self, mongo_uri, mongo_db):
        self.mongo_uri = mongo_uri
        self.mongo_db = mongo_db

    @classmethod
    def from_crawler(cls, crawler):
        return cls(
            mongo_uri=crawler.settings.get('MONGO_URI'),
            mongo_db=crawler.settings.get('MONGO_DATABASE', 'items')
        )

    def open_spider(self, spider):
        self.client = pymongo.MongoClient(self.mongo_uri)
        self.db = self.client[self.mongo_db]

    def close_spider(self, spider):
        self.client.close()

    def process_item(self, item, spider):
        self.db[self.collection_name].insert_one(dict(item))
        return item
```

Take screenshot of item

This example demonstrates how to return Deferred from `process_item()` method. It uses Splash to render screenshot of item url. Pipeline makes request to locally running instance of Splash. After request is downloaded and Deferred callback fires, it saves item to a file and adds filename to an item.

```
import scrapy
import hashlib
from urllib.parse import quote


class ScreenshotPipeline(object):
    """Pipeline that uses Splash to render screenshot of
    every Scrapy item."""

    SPLASH_URL = "http://localhost:8050/render.png?url={}"

    def process_item(self, item, spider):
        encoded_item_url = quote(item["url"])
        screenshot_url = self.SPLASH_URL.format(encoded_item_url)
        request = scrapy.Request(screenshot_url)
        dfd = spider.crawler.engine.download(request, spider)
        dfd.addBoth(self.return_item, item)
        return dfd

    def return_item(self, response, item):
        if response.status != 200:
            # Error happened, return item.
            return item

        # Save screenshot to file, filename will be hash of url.
        url = item["url"]
        url_hash = hashlib.md5(url.encode("utf8")).hexdigest()
        filename = "{}.png".format(url_hash)
        with open(filename, "wb") as f:
            f.write(response.body)

        # Store filename in item.
        item["screenshot_filename"] = filename
        return item
```

Duplicates filter

A filter that looks for duplicate items, and drops those items that were already processed. Let's say that our items have a unique id, but our spider returns multiples items with the same id:

```
from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

    def __init__(self):
        self.ids_seen = set()

    def process_item(self, item, spider):
        if item['id'] in self.ids_seen:
```

```
            raise DropItem("Duplicate item found: %s" % item)
        else:
            self.ids_seen.add(item['id'])
            return item
```

Activating an Item Pipeline component

To activate an Item Pipeline component you must add its class to the `ITEM_PIPELINES` setting, like in the following example:

```
ITEM_PIPELINES = {
    'myproject.pipelines.PricePipeline': 300,
    'myproject.pipelines.JsonWriterPipeline': 800,
}
```

The integer values you assign to classes in this setting determine the order in which they run: items go through from lower valued to higher valued classes. It's customary to define these numbers in the 0-1000 range.

Previous

Next