SEP	18
Title	Spider Middleware v2
Author	Insophia Team
Created	2010-06-20
Status	Draft (in progress)

SEP-018: Spider middleware v2

This SEP introduces a new architecture for spider middlewares which provides a greater degree of modularity to combine functionality which can be plugged in from different (reusable) middlewares.

The purpose of !SpiderMiddleware-v2 is to define an architecture that encourages more re-usability for building spiders based on smaller well-tested components. Those components can be global (similar to current spider middlewares) or per-spider that can be combined to achieve the desired functionality. These reusable components will benefit all Scrapy users by building a repository of well-tested components that can be shared among different spiders and projects. Some of them will come bundled with Scrapy.

Unless explicitly stated, in this document "spider middleware" refers to the **new** spider middleware v2, not the old one.

This document is a work in progress, see Pending Issues below.

New spider middleware API

A spider middleware can implement any of the following methods:

- `process_response(response, request, spider)`
  - Process a (downloaded) response
  - Receives: The `response` to process, the `request` used to download the response (not necessarily the request sent from the spider), and the `spider` that requested it.
  - Returns: A list containing requests and/or items
- `process_error(error, request, spider)`:
  - Process a error when trying to download a request, such as DNS errors, timeout errors, etc.
  - Receives: The `error` caused, the `request` that caused it (not necessarily the request sent from the spider), and then `spider` that requested it.
  - Returns: A list containing request and/or items
- `process_request(request, response, spider)`
  - Process a request after it has been extracted from the spider or previous middleware `process_response()` methods.
  - Receives: The `request` to process, the `response` where the request was extracted from, and the `spider` that extracted it.
    - Note: `response` is `None` for start requests, or requests injected directly (through `manager.scraper.process_request()` without specifying a response (see below)
  - Returns: A `Request` object (not necessarily the same received), or `None` in which case the request is dropped.
- `process_item(item, response, spider)`
  - Process an item after it has been extracted from the spider or previous middleware `process_response()` methods.
  - Receives: The `item` to process, the `response` where the item was extracted from, and the `spider` that extracted it.
  - Returns: An `Item` object (not necessarily the same received), or `None` in which case the item is dropped.
- `next_request(spider)`
  - Returns a the next request to crawl with this spider. This method is called when the spider is opened, and when it gets idle.
  - Receives: The `spider` to return the next request for.
  - Returns: A `Request` object.
- `open_spider(spider)`
  - This can be used to allocate resources when a spider is opened.
  - Receives: The `spider` that has been opened.
  - Returns: nothing
- `close_spider(spider)`
  - This can be used to free resources when a spider is closed.
  - Receives: The `spider` that has been closed.
  - Returns: nothing

Changes to core API

Injecting requests to crawl

To inject start requests (or new requests without a response) to crawl, you used before:

- `manager.engine.crawl(request, spider)`

Now you'll use:

- `manager.scraper.process_request(request, spider, response=None)`

Which (unlike the old `engine.crawl` will make the requests pass through the spider middleware `process_request()` method).

Scheduler middleware to be removed

We're gonna remove the Scheduler Middleware, and move the duplicates filter to a new spider middleware.

Scraper high-level API

There is a simpler high-level API - the Scraper API - which is the API used by the engine and other core components. This is also the API implemented by this new middleware, with its own internal architecture and hooks. Here is the Scraper API:

- `process_response(response, request, spider)`
    - returns iterable of items and requests
- `process_error(error, request, spider)`
    - returns iterable of items and requests
- `process_request(request, spider, response=None)`
    - injects a request to crawl for the given spider
- `process_item(item, spider, response)`
    - injects a item to process with the item processor (typically the item pipeline)
- `next_request(spider)`
    - returns the next request to process for the given spider
- `open_spider(spider)`
    - opens a spider
- `close_spider(spider)`
    - closes a spider

How it works

The spider middlewares are defined in certain order with the top-most being the one closer to the engine, and the bottom-most being the one closed to the spider.

Example:

- Engine
- Global spider Middleware 3
- Global spider Middleware 2
- Global spider Middleware 1
- Spider-specific middlewares (defined in `Spider.middlewares`)
    - Spider-specific middleware 3
    - Spider-specific middleware 2
    - Spider-specific middleware 1
- Spider

The data flow with Spider Middleware v2 is as follows:

1. When a response arrives from the engine, it it passed through all the spider middlewares (in descending order). The result of each middleware `process_response` is kept and then returned along with the spider callback result
2. Each item of the aggregated result from previous point is passed through all middlewares (in ascending order) calling the `process_request` or `process_item` method accordingly, and their results are kept for passing to the following middlewares

One of the spider middlewares (typically - but not necessarily - the last spider middleware closer to the spider, as shown in the example) will be a "spider-specific spider middleware" which would take care of calling the additional spider middlewares defined in the `Spider.middlewares` attribute, hence providing support for per-spider middlewares. If the middleware is well written, it should work both globally and per-spider.

Spider-specific middlewares

You can define in the spider itself a list of additional middlewares that will be used for this spider, and only this spider. If the middleware is well written, it should work both globally and per spider.

Here's an example that combines functionality from multiple middlewares into the same spider:

```
#!python
class MySpider(BaseSpider):

    middlewares = [RegexLinkExtractor(), CallbackRules(), CanonicalizeUrl(),
                   ItemIdSetter(), OffsiteMiddleware()]

    allowed_domains = ['example.com', 'sub.example.com']

    url_regexes_to_follow = ['/product.php?.*']

    callback_rules = {
        '/product.php.*': 'parse_product',
        '/category.php.*': 'parse_category',
    }

    canonicalization_rules = ['sort-query-args', 'normalize-percent-encoding', ...]

    id_field = 'guid'
    id_fields_to_hash = ['supplier_name', 'supplier_id']

    def parse_product(self, item):
        # extract item from response
        return item

    def parse_category(self, item):
        # extract item from response
        return item
```

The Spider Middleware that implements spider code

There's gonna be one middleware that will take care of calling the proper spider methods on each event such as:

- call `Request.callback` (for 200 responses) or `Request.errback` for non-200 responses and other errors. this behaviour can be changed through the `handle_httpstatus_list` spider attribute.
    - if `Request.callback` is not set it will use `Spider.parse`
    - if `Request.errback` is not set it will use `Spider.errback`
- call additional spider middlewares defined in the `Spider.middlewares` attribute
- call `Spider.next_request()` and `Spider.start_requests()` on `next_request()` middleware method (this would implicitly support backward compatibility)

Differences with Spider middleware v1

- adds support for per-spider middlewares through the `Spider.middlewares` attribute
- allows processing initial requests (those returned from `Spider.start_requests()`)

Use cases and examples

This section contains several examples and use cases for Spider Middlewares. Imports are intentionally removed for conciseness and clarity.

Regex (HTML) Link Extractor

A typical application of spider middlewares could be to build Link Extractors. For example:

```
#!python
class RegexHtmlLinkExtractor(object):

    def process_response(self, response, request, spider):
        if isinstance(response, HtmlResponse):
            allowed_regexes = spider.url_regexes_to_follow
            # extract urls to follow using allowed_regexes
            return [Request(x) for x in urls_to_follow]

# Example spider using this middleware
class MySpider(BaseSpider):

    middlewares = [RegexHtmlLinkExtractor()]
    url_regexes_to_follow = ['/product.php?.*']

    # parsing callbacks below
```

RSS2 link extractor

```
#!python
class Rss2LinkExtractor(object):

    def process_response(self, response, request, spider):
        if response.headers.get('Content-type') 'application/rss+xml':
            xs = XmlXPathSelector(response)
            urls = xs.select("//item/link/text()").extract()
            return [Request(x) for x in urls]
```

Callback dispatcher based on rules

Another example could be to build a callback dispatcher based on rules:

```
#!python
class CallbackRules(object):

    def __init__(self):
        self.rules = {}
        dispatcher.connect(signals.spider_opened, self.spider_opened)
        dispatcher.connect(signals.spider_closed, self.spider_closed)

    def spider_opened(self, spider):
        self.rules[spider] = {}
        for regex, method_name in spider.callback_rules.items():
            r = re.compile(regex)
            m = getattr(self.spider, method_name, None)
            if m:
                self.rules[spider][r] = m

    def spider_closed(self, spider):
        del self.rules[spider]

    def process_response(self, response, request, spider):
        for regex, method in self.rules[spider].items():
            m = regex.search(response.url)
            if m:
                return method(response)
        return []

# Example spider using this middleware
class MySpider(BaseSpider):

    middlewares = [CallbackRules()]
    callback_rules = {
        '/product.php.*': 'parse_product',
        '/category.php.*': 'parse_category',
    }

    def parse_product(self, response):
        # parse response and populate item
        return item
```

URL Canonicalizers

Another example could be for building URL canonicalizers:

```
#!python
class CanonializeUrl(object):

    def process_request(self, request, response, spider):
        curl = canonicalize_url(request.url,
                                rules=spider.canonicalization_rules)
        return request.replace(url=curl)

# Example spider using this middleware
class MySpider(BaseSpider):

    middlewares = [CanonicalizeUrl()]
    canonicalization_rules = ['sort-query-args',
                              'normalize-percent-encoding', ...]

    # ...
```

Setting item identifier

Another example could be for setting a unique identifier to items, based on certain fields:

```
#!python
class ItemIdSetter(object):
```

```
        def process_item(self, item, response, spider):
            id_field = spider.id_field
            id_fields_to_hash = spider.id_fields_to_hash
            item[id_field] = make_hash_based_on_fields(item, id_fields_to_hash)
            return item

    # Example spider using this middleware
    class MySpider(BaseSpider):

        middlewares = [ItemIdSetter()]
        id_field = 'guid'
        id_fields_to_hash = ['supplier_name', 'supplier_id']

        def parse(self, response):
            # extract item from response
            return item
```

robots.txt exclusion

A spider middleware to avoid visiting pages forbidden by robots.txt:

```
    #!python
    class SpiderInfo(object):

        def __init__(self, useragent):
            self.useragent = useragent
            self.parsers = {}
            self.pending = defaultdict(list)


    class AllowAllParser(object):

        def can_fetch(useragent, url):
            return True


    class RobotsTxtMiddleware(object):

        REQUEST_PRIORITY = 1000

        def __init__(self):
            self.spiders = {}
            dispatcher.connect(self.spider_opened, signal=signals.spider_opened)
            dispatcher.connect(self.spider_closed, signal=signals.spider_closed)

        def process_request(self, request, response, spider):
            return self.process_start_request(self, request)

        def process_start_request(self, request, spider):
            info = self.spiders[spider]
            url = urlparse_cached(request)
            netloc = url.netloc
            if netloc in info.parsers:
                rp = info.parsers[netloc]
                if rp.can_fetch(info.useragent, request.url):
                    res = request
                else:
                    spider.log("Forbidden by robots.txt: %s" % request)
                    res = None
            else:
                if netloc in info.pending:
                    res = None
                else:
                    robotsurl = "%s://%s/robots.txt" % (url.scheme, netloc)
                    meta = {'spider': spider, {'handle_httpstatus_list': [403, 404, 500]}
                    res = Request(robotsurl, callback=self.parse_robots,
                        meta=meta, priority=self.REQUEST_PRIORITY)
                info.pending[netloc].append(request)
            return res

        def parse_robots(self, response):
            spider = response.request.meta['spider']
            netloc urlparse_cached(response).netloc
            info = self.spiders[spider]
            if response.status 200;
                rp = robotparser.RobotFileParser(response.url)
                rp.parse(response.body.splitlines())
                info.parsers[netloc] = rp
            else:
                info.parsers[netloc] = AllowAllParser()
```

```
        return info.pending[netloc]

    def spider_opened(self, spider):
        ua = getattr(spider, 'user_agent', None) or settings['USER_AGENT']
        self.spiders[spider] = SpiderInfo(ua)

    def spider_closed(self, spider):
        del self.spiders[spider]
```

Offsite middleware

This is a port of the Offsite middleware to the new spider middleware API:

```
#!python
class SpiderInfo(object):

    def __init__(self, host_regex):
        self.host_regex = host_regex
        self.hosts_seen = set()


class OffsiteMiddleware(object):

    def __init__(self):
        self.spiders = {}
        dispatcher.connect(self.spider_opened, signal=signals.spider_opened)
        dispatcher.connect(self.spider_closed, signal=signals.spider_closed)

    def process_request(self, request, response, spider):
        return self.process_start_request(self, request)

    def process_start_request(self, request, spider):
        if self.should_follow(request, spider):
            return request
        else:
            info = self.spiders[spider]
            host = urlparse_cached(x).hostname
            if host and host not in info.hosts_seen:
               spider.log("Filtered offsite request to %r: %s" % (host, request))
               info.hosts_seen.add(host)

    def should_follow(self, request, spider):
        info = self.spiders[spider]
        # hostname can be None for wrong urls (like javascript links)
        host = urlparse_cached(request).hostname or ''
        return bool(info.regex.search(host))

    def get_host_regex(self, spider):
        """Override this method to implement a different offsite policy"""
        domains = [d.replace('.', r'\.') for d in spider.allowed_domains]
        regex = r'^(.*\.)?(%s)$' % '|'.join(domains)
        return re.compile(regex)

    def spider_opened(self, spider):
        info = SpiderInfo(self.get_host_regex(spider))
        self.spiders[spider] = info

    def spider_closed(self, spider):
        del self.spiders[spider]
```

Limit URL length

A middleware to filter out requests with long urls:

```
#!python

class LimitUrlLength(object):

    def __init__(self):
        self.maxlength = settings.getint('URLLENGTH_LIMIT')

    def process_request(self, request, response, spider):
        return self.process_start_request(self, request)

    def process_start_request(self, request, spider):
         if len(request.url) <= self.maxlength:
            return request
         spider.log("Ignoring request (url length > %d): %s " % (self.maxlength, request.url))
```

Set Referer

A middleware to set the Referer:

```
#!python
class SetReferer(object):

    def process_request(self, request, response, spider):
        request.headers.setdefault('Referer', response.url)
        return request
```

Set and limit crawling depth

A middleware to set (and limit) the request/response depth, taken from the start requests:

```
#!python
class SetLimitDepth(object):

    def __init__(self, maxdepth=0):
        self.maxdepth = maxdepth or settings.getint('DEPTH_LIMIT')

    def process_request(self, request, response, spider):
        depth = response.request.meta['depth'] + 1
        request.meta['depth'] = depth
        if not self.maxdepth or depth <= self.maxdepth:
            return request
        spider.log("Ignoring link (depth > %d): %s " % (self.maxdepth, request)

    def process_start_request(self, request, spider):
        request.meta['depth'] = 0
        return request
```

Filter duplicate requests

A middleware to filter out requests already seen:

```
#!python
class FilterDuplicates(object):

    def __init__(self):
        clspath = settings.get('DUPEFILTER_CLASS')
        self.dupefilter = load_object(clspath)()
        dispatcher.connect(self.spider_opened, signal=signals.spider_opened)
        dispatcher.connect(self.spider_closed, signal=signals.spider_closed)

    def enqueue_request(self, spider, request):
        seen = self.dupefilter.request_seen(spider, request)
        if not seen or request.dont_filter:
            return request

    def spider_opened(self, spider):
        self.dupefilter.open_spider(spider)

    def spider_closed(self, spider):
        self.dupefilter.close_spider(spider)
```

Scrape data using Parsley

A middleware to Scrape data using Parsley as described in UsingParsley

```
#!python
from pyparsley import PyParsley

class ParsleyExtractor(object):

    def __init__(self, parslet_json_code):
        parslet = json.loads(parselet_json_code)
        class ParsleyItem(Item):
            def __init__(self, *a, **kw):
                for name in parslet.keys():
                    self.fields[name] = Field()
            super(ParsleyItem, self).__init__(*a, **kw)
        self.item_class = ParsleyItem
        self.parsley = PyParsley(parslet, output='python')

    def process_response(self, response, request, spider):
        return self.item_class(self.parsly.parse(string=response.body))
```

Pending issues

Resolved:

- how to make `start_requests()` output pass through spider middleware `process_request()`?

    - Start requests will be injected through `manager.scraper.process_request()` instead of `manager.engine.crawl()`

- should we support adding additional start requests from a spider middleware?
    - Yes - there is a spider middleware method (`start_requests`) for that
- should `process_response()` receive a `request` argument with the `request` that originated it?. `response.request` is the latest request, not the original one (think of redirections), but it does carry the `meta` of the original one. The original one may not be available anymore (in memory) if we're using a persistent scheduler., but in that case it would be the deserialized request from the persistent scheduler queue.

    - No - this would make implementation more complex and we're not sure it's really needed

- how to make sure `Request.errback` is always called if there is a problem with the request?. Do we need to ensure that?. Requests filtered out (by returning `None`) in the `process_request()` method will never be callback-ed or even errback-ed. this could be a problem for spiders that want to be notified if their requests are dropped. should we support this notification somehow or document (the lack of) it properly?

    - We won't support notifications of dropped requests, because: 1. it's hard to implement and unreliable, 2. it's against not friendly with request persistence, 3. we can't come up with a good api.

- should we make the list of default spider middlewares empty? (or the "per-spider" spider middleware alone)

    - No - there are some useful spider middlewares that it's worth enabling by default like referer, duplicates, robots2

- should we allow returning deferreds in spider middleware methods?
    - Yes - we should build a Deferred with the spider middleware methods as callbacks and that would implicitly support returning Deferreds
- should we support processing responses before they're processed by the spider, because `process_response` runs "in parallel" to the spider callback, and can't stop from running it.

    - No - we haven't seen a practical use case for this, so we won't add an additional hook. It should be trivial to add it later, if needed.

- should we make a spider middleware to handle calling the request and spider callback, instead of letting the Scraper component do it?

    - Yes - there's gonna a spider middleware for execution spider-specific code such as callbacks and also custom middlewares