SEP	16
Title	Leg Spider
Author	Insophia Team
Created	2010-06-03
Status	Superseded by :doc:`sep-018`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scrapy-master\sep\(scrapy-master)(sep)sep-016.rst, line 7); *backlink*

Unknown interpreted text role "doc".

SEP-016: Leg Spider

This SEP introduces a new kind of Spider called `LegSpider` which provides modular functionality which can be plugged to different spiders.

Rationale

The purpose of Leg Spiders is to define an architecture for building spiders based on smaller well-tested components (aka. Legs) 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 (legs) that can be shared among different spiders and projects. Some of them will come bundled with Scrapy.

The Legs themselves can be also combined with sub-legs, in a hierarchical fashion. Legs are also spiders themselves, hence the name "Leg Spider".

`LegSpider` API

A `LegSpider` is a `BaseSpider` subclass that adds the following attributes and methods:

- `legs`
  - legs composing this spider
- `process_response(response)`
  - Process a (downloaded) response and return a list of requests and items
- `process_request(request)`
  - Process a request after it has been extracted and before returning it from the spider
- `process_item(item)`
  - Process an item after it has been extracted and before returning it from the spider
- `set_spider()`
  - Defines the main spider associated with this Leg Spider, which is often used to configure the Leg Spider behavior.

How Leg Spiders work

1. Each Leg Spider has zero or many Leg Spiders associated with it. When a response arrives, the Leg Spider process it with its `process_response` method and also the `process_response` method of all its "sub leg spiders". Finally, the output of all of them is combined to produce the final aggregated output.
2. Each element of the aggregated output of `process_response` is processed with either `process_item` or `process_request` before being returned from the spider. Similar to `process_response`, each item/request is processed with all `process_{request,item}` of the leg spiders composing the spider, and also with those of the spider itself.

Leg Spider examples

Regex (HTML) Link Extractor

A typical application of LegSpider's is to build Link Extractors. For example:

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

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

class MySpider(LegSpider):
```

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

    def parse_response(self, response):
        # parse response and extract items
        return items
```

RSS2 link extractor

This is a Leg Spider that can be used for following links from RSS2 feeds.

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

    def process_response(self, response):
        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(LegSpider):

    def __init__(self, *a, **kw):
        super(CallbackRules, self).__init__(*a, **kw)
        for regex, method_name in self.spider.callback_rules.items():
            r = re.compile(regex)
            m = getattr(self.spider, method_name, None)
            if m:
                self._rules[r] = m

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

class MySpider(LegSpider):

    legs = [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(LegSpider):

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

class MySpider(LegSpider):

    legs = [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(LegSpider):

    def process_item(self, item):
```

```
        id_field = self.spider.id_field
        id_fields_to_hash = self.spider.id_fields_to_hash
        item[id_field] = make_hash_based_on_fields(item, id_fields_to_hash)
        return item

class MySpider(LegSpider):

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

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

Combining multiple leg spiders

Here's an example that combines functionality from multiple leg spiders:

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

    legs = [RegexLinkExtractor(), ParseRules(), CanonicalizeUrl(), ItemIdSetter()]

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

    parse_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 process_product(self, item):
        # extract item from response
        return item

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

Leg Spiders vs Spider middlewares

A common question that would arise is when one should use Leg Spiders and when to use Spider middlewares. Leg Spiders functionality is meant to implement spider-specific functionality, like link extraction which has custom rules per spider. Spider middlewares, on the other hand, are meant to implement global functionality.

When not to use Leg Spiders

Leg Spiders are not a silver bullet to implement all kinds of spiders, so it's important to keep in mind their scope and limitations, such as:

- Leg Spiders can't filter duplicate requests, since they don't have access to all requests at the same time. This functionality should be done in a spider or scheduler middleware.
- Leg Spiders are meant to be used for spiders whose behavior (requests & items to extract) depends only on the current page and not previously crawled pages (aka. "context-free spiders"). If your spider has some custom logic with chained downloads (for example, multi-page items) then Leg Spiders may not be a good fit.

`LegSpider` proof-of-concept implementation

Here's a proof-of-concept implementation of `LegSpider`:

```
#!python
from scrapy.http import Request
from scrapy.item import BaseItem
from scrapy.spider import BaseSpider
from scrapy.utils.spider import iterate_spider_output


class LegSpider(BaseSpider):
    """A spider made of legs"""

    legs = []

    def __init__(self, *args, **kwargs):
        super(LegSpider, self).__init__(*args, **kwargs)
```

```
        self._legs = [self] + self.legs[:]
        for l in self._legs:
            l.set_spider(self)

    def parse(self, response):
        res = self._process_response(response)
        for r in res:
            if isinstance(r, BaseItem):
                yield self._process_item(r)
            else:
                yield self._process_request(r)

    def process_response(self, response):
        return []

    def process_request(self, request):
        return request

    def process_item(self, item):
        return item

    def set_spider(self, spider):
        self.spider = spider

    def _process_response(self, response):
        res = []
        for l in self._legs:
            res.extend(iterate_spider_output(l.process_response(response)))
        return res

    def _process_request(self, request):
        for l in self._legs:
            request = l.process_request(request)
        return request

    def _process_item(self, item):
        for l in self._legs:
            item = l.process_item(item)
        return item
```