scrapy-poet Documentation

Release 0.0.2

Scrapinghub

May 15, 2020

GETTING STARTED

scrapy-poet easily integrates Page Objects created using web-poet with Scrapy through the configuration of a dependency injection middleware.

web-poet is used to make reusable Page Objects that separates extraction logic from crawling. They could be easily tested and distributed across different projects. Also, they could make use of different backends, for example, acquiring data from Splash and AutoExtract API.

The goal of this project is to provide a bridge between Scrapy Spiders and Page Objects.

Please, see also our *Installation guide* and our *Tutorial* for a quick start.

License is BSD 3-clause.

CHAPTER

ONE

INSTALLATION GUIDE

1.1 Installing scrapy-poet

scrapy-poet is a Scrapy extension that runs on Python 3.6 and above.

If you're already familiar with installation of Python packages, you can install scrapy-poet and its dependencies from PyPI with:

```
pip install scrapy-poet
```

Scrapy 2.1.0 or above is required and it has to be installed separately.

1.2 Things that are good to know

scrapy-poet is written in pure Python and depends on a few key Python packages (among others):

- web-poet, core library used for Page Object pattern
- andi, provides annotation-based dependency injection
- parsel, responsible for css and xpath selectors

CHAPTER

TWO

TUTORIAL

In this tutorial, we'll assume that scrapy-poet is already installed on your system. If that's not the case, see *Installation guide*.

We are going to scrape books.toscrape.com, a website that lists books from famous authors.

This tutorial will walk you through these tasks:

1. Writing a spider to crawl a site and extract data
2. Separating extraction logic from the spider
3. Configuring Scrapy project to use scrapy-poet
4. Changing spider to make use of our extraction logic

If you're not already familiar with Scrapy, and want to learn it quickly, the Scrapy Tutorial is a good resource.

2.1 Creating a spider

Create a new Scrapy project and add a new spider to it. This spider will be called `books` and it will crawl and extract data from a target website.

```
import scrapy


class BooksSpider(scrapy.Spider):
    """Crawl and extract books data"""

    name = 'books'
    start_urls = ['http://books.toscrape.com/']

    def parse(self, response):
        """Discover book links and follow them"""
        links = response.css('.image_container a')
        yield from response.follow_all(links, self.parse_book)

    def parse_book(self, response):
        """Extract data from book pages"""
        yield {
            'url': response.url,
            'name': response.css("title::text").get(),
        }
```

2.2 Separating extraction logic

Let's create our first Page Object by moving extraction logic out of the spider class.

```
from web_poet.pages import ItemWebPage


class BookPage(ItemWebPage):
    """Individual book page on books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """

    def to_item(self):
        """Convert page into an item"""
        return {
            'url': self.url,
            'name': self.css("title::text").get(),
        }
```

Now we have a `BookPage` class that implements the `to_item` method. This class contains all logic necessary for extracting an item from an individual book page like http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html, and nothing else.

If we want, we can organize code in a different way, and e.g. extract a property from the `to_item` method:

```
from web_poet.pages import ItemWebPage


class BookPage(ItemWebPage):
    """Individual book page on books.toscrape.com website"""

    @property
    def title(self):
        """Book page title"""
        return self.css("title::text").get()

    def to_item(self):
        return {
            'url': self.url,
            'name': self.title,
        }
```

2.3 Configuring the project

To use scrapy-poet, enable its downloader middleware in `settings.py`:

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy_poet.InjectionMiddleware': 543,
}
```

BookPage class we created previously can be used without scrapy-poet, and even without Scrapy (note that imports were from `web_poet` so far).

`scrapy-poet` makes it easy to use `web-poet` Page Objects (such as BookPage) in Scrapy spiders.

2.4 Changing spider

To use the newly created BookPage class in the spider, change the `parse_book` method as follows:

```
class BooksSpider(scrapy.Spider):
    # ...
    def parse_book(self, response, book_page: BookPage):
        """Extract data from book pages"""
        yield book_page.to_item()
```

`parse_book` method now has a type annotated argument called `book_page`. scrapy-poet detects this and makes sure a BookPage instance is created and passed to the callback.

The full spider code would be looking like this:

```
import scrapy


class BooksSpider(scrapy.Spider):
    """Crawl and extract books data"""

    name = 'books'
    start_urls = ['http://books.toscrape.com/']

    def parse(self, response):
        """Discover book links and follow them"""
        links = response.css('.image_container a')
        yield from response.follow_all(links, self.parse_book)

    def parse_book(self, response, book_page: BookPage):
        """Extract data from book pages"""
        yield book_page.to_item()
```

You might have noticed that `parse_book` is quite simple; it's just returning the result of the `to_item` method call. We could use *`callback_for()`* helper to reduce the boilerplate.

```
import scrapy
from scrapy_poet import callback_for


class BooksSpider(scrapy.Spider):
    """Crawl and extract books data"""

    name = 'books'
    start_urls = ['http://books.toscrape.com/']
    parse_book = callback_for(BookPage)

    def parse(self, response):
        """Discovers book links and follows them"""
        links = response.css('.image_container a')
        yield from response.follow_all(links, self.parse_book)
```

Note: You can also write something like `response.follow_all(links, callback_for(BookPage))`, without creating an attribute, but currently it won't work with Scrapy disk queues.

2.5 Final result

At the end of our job, the spider should look like this:

```
import scrapy
from web_poet.pages import ItemWebPage
from scrapy_poet import callback_for


class BookPage(ItemWebPage):
    """Individual book page on books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """

    def to_item(self):
        return {
            'url': self.url,
            'name': self.css("title::text").get(),
        }


class BooksSpider(scrapy.Spider):
    """Crawl and extract books data"""

    name = 'books'
    start_urls = ['http://books.toscrape.com/']
    parse_book = callback_for(BookPage)  # extract items from book pages

    def parse(self, response):
        """Discover book links and follow them"""
        links = response.css('.image_container a')
        yield from response.follow_all(links, self.parse_book)
```

It now looks similar to the original spider, but the item extraction logic is separated from the spider.

On a surface, it looks just like a different way to organize Scrapy spider code - and indeed, it *is* just a different way to organize the code, but it opens some cool possibilities.

2.6 Next steps

Now that you know how scrapy-poet is supposed to work, what about trying to apply it to an existing or new Scrapy project?

Also, please check *Advanced Usage* and refer to spiders in the "example" folder: https://github.com/scrapinghub/scrapy-poet/tree/master/example/example/spiders

CHAPTER

THREE

ADVANCED USAGE

3.1 Creating providers

Providers are responsible for building dependencies needed by Injectable objects. A good example would be the `ResponseDataProvider`, which builds and provides a `ResponseData` instance for Injectables that need it, like the `ItemWebPage`.

```
@attr.s(auto_attribs=True)
class ResponseData:
    """Represents a response containing its URL and HTML content."""
    url: str
    html: str


@provides(ResponseData)
class ResponseDataProvider(PageObjectInputProvider):
    """This class provides ``web_poet.page_inputs.ResponseData`` instances."""

    def __init__(self, response: Response):
        """This class receives a Scrapy ``Response`` as a dependency."""
        self.response = response

    def __call__(self):
        """This method builds a ``ResponseData`` instance using a Scrapy
        ``Response``.
        """
        return ResponseData(
            url=self.response.url,
            html=self.response.text
        )
```

You are able to implement your own providers in order to extend or override current scrapy-poet behavior.

Warning: Currently, scrapy-poet is only able to inject `Response` and *`DummyResponse`* instances as *provider* dependencies.

3.2 Ignoring requests

Sometimes requests could be skipped, for example, when you're fetching data using a third-party API such as Auto Extract or querying a database.

In cases like that, it makes no sense to send the request to Scrapy's downloader as it will only waste network resources. But there's an alternative to avoid making such requests, you could use *DummyResponse* type to annotate your response arguments.

That could be done in the spider's parser method:

```
def parser(self, response: DummyRequest, page: MyPageObject):
    pass
```

Spider method that has its first argument annotated as *DummyResponse* is signaling that it is not going to use the response, so it should be safe to not download scrapy Response as usual.

This type annotation is already applied when you use the *callback_for()* helper: the callback which is created by `callback_for` doesn't use Response, it just calls page object's `to_item` method.

If neither spider callback nor any of the input providers are using `Response`, *InjectionMiddleware* skips the download, returning a *DummyResponse* instead. For example:

```
def get_cached_content(key: str):
    # get cached html response from db or other source
    pass


@attr.s(auto_attribs=True)
class CachedData:

    key: str
    value: str


@provides(CachedData)
class CachedDataProvider(PageObjectInputProvider):

    def __init__(self, response: DummyResponse):
        self.response = response

    def __call__(self):
        return CachedData(
            key=self.response.url,
            value=get_cached_content(self.response.url)
        )


class MyPageObject(ItemPage):

    content: CachedData

    def to_item(self):
        return {
            'url': self.content.key,
            'content': self.content.value,
        }
```

(continues on next page)

(continued from previous page)

```
class MySpider(scrapy.Spider):

    name = 'my_spider'

    def parse(self, response: DummyResponse, page: MyPageObject):
        # request will be IGNORED because neither spider callback
        # not MyPageObject seem like to be making use of its response
        yield page.to_item()
```

Although, if the spider callback is not using `Response`, but the Page Object uses it, the request is not ignored, for example:

```
def parse_content(html: str):
    # parse content from html
    pass


@attr.s(auto_attribs=True)
class MyResponseData:

    url: str
    html: str


@provides(MyResponseData)
class MyResponseDataProvider(PageObjectInputProvider):

    def __init__(self, response: Response):
        self.response = response

    def __call__(self):
        return MyResponseData(
            url=self.response.url,
            html=self.response.content,
        )


class MyPageObject(ItemPage):

    response: MyResponseData

    def to_item(self):
        return {
            'url': self.response.url,
            'content': parse_content(self.response.html),
        }


class MySpider(scrapy.Spider):

    name = 'my_spider'

    def parse(self, response: DummyResponse, page: MyPageObject):
        # request will be PROCESSED because spider callback is not
        # making use of its response, but MyPageObject seems like to be
        yield page.to_item()
```

Note: The code above is just for example purposes. If you need to use `Response` instances in your Page Objects, use built-in `ItemWebPage` - it has `response` attribute with `ResponseData`; no additional configuration is needed, as there is `ResponseDataProvider` enabled in scrapy-poet by default.

3.2.1 Requests concurrency

DummyRequests are meant to skip downloads, so it makes sense not checking for concurrent requests, delays, or auto throttle settings since we won't be making any download at all.

By default, if your parser or its page inputs need a regular Request, this request is downloaded through Scrapy, and all the settings and limits are respected, for example:

- `CONCURRENT_REQUESTS`
- `CONCURRENT_REQUESTS_PER_DOMAIN`
- `CONCURRENT_REQUESTS_PER_IP`
- `RANDOMIZE_DOWNLOAD_DELAY`
- all AutoThrottle settings
- `DownloaderAwarePriorityQueue` logic

But be aware when using third-party libraries to acquire content for a page object. If you make an HTTP request in a provider using some third-party async library (aiohttp, treq, etc.), `CONCURRENT_REQUESTS` option will be respected, but not the others.

To have other settings respected, in addition to `CONCURRENT_REQUESTS`, you'd need to use `crawler.engine.download` or something like that. Alternatively, you could implement those limits in the library itself.

CHAPTER

FOUR

API REFERENCE

4.1 Utils

scrapy_poet.**callback_for**(*page_cls: Type[web_poet.pages.ItemPage]*) → Callable

This function is a helper for creating callbacks for `ItemPage` sub-classes. The generated callback should return the result of the call to the `ItemPage.to_item` method.

The generated callback could be used as a spider instance method or passed as an inline/anonymous argument. Make sure to define it as a spider argument if you're planning to use disk queues because in this case, Scrapy should be able to serialize your request object.

Example:

```
class BooksSpider(scrapy.Spider):

    name = 'books'
    start_urls = ['http://books.toscrape.com/']
    parse_book = callback_for(BookPage)

    def parse(self, response):
        links = response.css('.image_container a')
        yield from response.follow_all(links, self.parse_book)
```

class scrapy_poet.utils.**DummyResponse**(*url: str*, *request=typing.Union[scrapy.http.request.Request, NoneType]*)

This class is returned by the `InjectionMiddleware` when it detects that the download could be skipped. It inherits from Scrapy `Response` and signals and stores the URL and references the original `Request`.

If you want to skip downloads, you can type annotate your parse method with this class.

```
def parse(self, response: DummyResponse):
    pass
```

If there's no Page Input that depends on a Scrapy `Response`, the `InjectionMiddleware` is going to skip download and provide a `DummyResponse` to your parser instead.

If your `PageObjectInputProvider` doesn't need a request, you simply don't need to list it as a dependency. But if you need, for example, the original request's URL, you can use `DummyResponse` instead of `Response`:

```
@provides(ResponseData)
class ResponseDataProvider(PageObjectInputProvider):

    def __init__(self, response: DummyResponse):
```

(continues on next page)

(continued from previous page)

```
        self.response = response

    async def __call__(self):
        data = await self.get_data()
        return ResponseData(
            url=self.response.url,
            html=data
        )

    async def get_data(self):
        # make an api call
        # make a database query
        # read from disk
        # ...
        pass
```

4.2 Injection Middleware

An important part of scrapy-poet is the Injection Middleware. It's responsible for injecting Page Input dependencies before the request callbacks are executed.

class scrapy_poet.middleware.**InjectionMiddleware**

This is a Downloader Middleware that's supposed to:

- check if request downloads could be skipped
- inject dependencies before request callbacks are executed

process_request(*request: scrapy.http.request.Request*, *spider: scrapy.spiders.Spider*)

This method checks if the request is really needed and if its download could be skipped by trying to infer if a `Response` is going to be used by the callback or a Page Input.

If the `Response` can be ignored, a `utils.DummyResponse` object is returned on its place. This `DummyResponse` is linked to the original `Request` instance.

With this behavior, we're able to optimize spider executions avoiding unnecessary downloads. That could be the case when the callback is actually using another source like external APIs such as Scrapinghub's Auto Extract.

process_response(*request: scrapy.http.request.Request*, *response: scrapy.http.response.Response*, *spider: scrapy.spiders.Spider*)

This method instantiates all `Injectable` sub-classes declared as request callback arguments and any other parameter with a provider for its type. Otherwise, this middleware doesn't populate `request.cb_kwargs` for this argument.

> **Warning:** We should be able to inject any type into classes that inherit from `web_poet.pages.Injectable`, but currently, we're only able to build and inject `scrapy.Response` instances.

4.3 Page Input Providers

The Injection Middleware needs a standard way to build dependencies for the Page Inputs used by the request callbacks. That's why we have created a repository of `PageObjectInputProviders`.

You could implement different providers in order to acquire data from multiple external sources, for example, Splash or Auto Extract API.

class scrapy_poet.page_input_providers.**PageObjectInputProvider**
This is an abstract class for describing Page Object Input Providers.

abstract __call__()
This method is responsible for building Page Input dependencies.

__init__()
You can override this method to receive external dependencies.

class scrapy_poet.page_input_providers.**ResponseDataProvider**(*response: scrapy.http.response.Response*)
This class provides `web_poet.page_inputs.ResponseData` instances.

__call__()
This method builds a `ResponseData` instance using a Scrapy `Response`.

__init__(*response: scrapy.http.response.Response*)
This class receives a Scrapy `Response` as a dependency.

scrapy_poet.page_input_providers.**provides**(*provided_class: Type*)
This decorator should be used with classes that inherits from `PageObjectInputProvider` in order to automatically register them as providers.

See `ResponseDataProvider`'s implementation for an example.

scrapy_poet.page_input_providers.**register**(*provider_class: Type[*scrapy_poet.page_input_providers.PageObjectInpu
provided_class: Type)
This method registers a Page Object Input Provider in the providers registry. It could be replaced by the use of the `provides` decorator.

Example:

register(ResponseDataProvider, ResponseData)

CHAPTER

FIVE

CONTRIBUTING

scrapy-poet is an open-source project. Your contribution is very welcome!

5.1 Issue Tracker

If you have a bug report, a new feature proposal or simply would like to make a question, please check our issue tracker on Github: https://github.com/scrapinghub/scrapy-poet/issues

5.2 Source code

Our source code is hosted on Github: https://github.com/scrapinghub/scrapy-poet

Before opening a pull request, it might be worth checking current and previous issues. Some code changes might also require some discussion before being accepted so it might be worth opening a new issue before implementing huge or breaking changes.

5.3 Testing

We use tox to run tests with different Python versions:

```
tox
```

The command above also runs type checks; we use mypy.

CHAPTER

SIX

CHANGELOG

6.1 0.0.2 (2020-04-28)

The repository is renamed to `scrapy-poet`, and split into two:

- `web-poet` (https://github.com/scrapinghub/web-poet) contains definitions and code useful for writing Page Objects for web data extraction - it is not tied to Scrapy;
- `scrapy-poet` (this package) provides Scrapy integration for such Page Objects.

API of the library changed in a backwards incompatible way; see README and examples.

New features:

- `DummyResponse` annotation allows to skip downloading of scrapy Response.
- `callback_for` works for Scrapy disk queues if it is used to create a spider method (but not in its inline form)
- Page objects may require page objects as dependencies; dependencies are resolved recursively and built as needed.
- InjectionMiddleware supports `async def` and asyncio providers.

6.2 0.0.1 (2019-08-28)

Initial release.

CHAPTER

SEVEN

LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of ScrapingHub nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

PYTHON MODULE INDEX

S

INDEX

Symbols

C

D

I

M

P

R

S