scrapy-poet Documentation

Release 0.22.1

Scrapinghub

Mar 07, 2024

GETTING STARTED

1 Installation 3

2 Basic Tutorial 5

3 Advanced Tutorial 15

4 Pitfalls 19

5 Rules from web-poet 23

6 Stats 29

7 Providers 31

8 Tests for Page Objects 39

9 Settings 41

10 API Reference 45

11 Contributing 53

12 Changelog 55

13 License 65

Python Module Index 67

Index 69

`scrapy-poet` allows to use web-poet Page Objects with Scrapy.

web-poet defines a standard for writing reusable and portable extraction and crawling code; please check its docs to learn more.

By using `scrapy-poet` you'll be organizing the spider code in a different way, which separates extraction and crawling logic from the I/O, and from the Scrapy implementation details as well. It makes the code more testable and reusable. Furthermore, it opens the door to create generic spider code that works across sites. Integrating a new site in the spider is then just a matter of write a bunch of Page Objects for it.

`scrapy-poet` also provides a way to integrate third-party APIs (like Splash and AutoExtract) with the spider, without losing testability and reusability. Concrete integrations are not provided by `web-poet`, but `scrapy-poet` makes them possbile.

To get started, see *Installation* and Scrapy Tutorial.

License is BSD 3-clause.

CHAPTER

ONE

INSTALLATION

1.1 Installing scrapy-poet

`scrapy-poet` is a Scrapy extension that runs on Python 3.8 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 2.6.0 or above is required and it has to be installed separately.

1.2 Configuring the project

To use `scrapy-poet`, enable its middlewares in the `settings.py` file of your Scrapy project:

```
DOWNLOADER_MIDDLEWARES = {
    "scrapy_poet.InjectionMiddleware": 543,
    "scrapy.downloadermiddlewares.stats.DownloaderStats": None,
    "scrapy_poet.DownloaderStatsMiddleware": 850,
}
SPIDER_MIDDLEWARES = {
    "scrapy_poet.RetryMiddleware": 275,
}
REQUEST_FINGERPRINTER_CLASS = "scrapy_poet.ScrapyPoetRequestFingerprinter"
```

1.3 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

BASIC TUTORIAL

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

Note: This tutorial can be followed without reading web-poet docs, but for a better understanding it is highly recommended to check them first.

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"""
```

(continues on next page)

(continued from previous page)

```
        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 WebPage


class BookPage(WebPage):
    """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. In particular, `BookPage` is now independent of Scrapy, and is not doing any I/O.

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 WebPage


class BookPage(WebPage):
    """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,
        }
```

The `BookPage` class we created 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.

See the *Installation* page on how to install and configure `scrapy-poet` in your project.

2.3 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.

Tip: `callback_for()` also supports *async generators*. So if we have the following:

```
class BooksSpider(scrapy.Spider):
    name = "books"
    start_urls = ["http://books.toscrape.com/"]

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

    async def parse_book(self, response: DummyResponse, page: BookPage):
        yield await page.to_item()
```

It could be turned into:

```
class BooksSpider(scrapy.Spider):
    name = "books"
    start_urls = ["http://books.toscrape.com/"]

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

    parse_book = callback_for(BookPage)
```

This is useful when the Page Objects uses additional requests, which rely heavily on `async/await` syntax. More info on this in this tutorial section: *Additional Requests*.

2.4 Final result

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

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


class BookPage(WebPage):
    """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(),
        }
```

(continues on next page)

(continued from previous page)

```
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.

2.5 Single spider - multiple sites

We have seen that using Page Objects is a great way to isolate the extraction logic from the crawling logic. As a side effect, it is now pretty easy to **create a generic spider with a common crawling logic that works across different sites**. The unique missing requirement is to be able to configure different Page Objects for different sites, because the extraction logic surely changes from site to site. This is exactly the functionality that *overrides* provides.

Note that the crawling logic of the `BooksSpider` is pretty simple and straightforward:

1. Extract all books URLs from the listing page
2. For each book URL found in the step 1, fetch the page and extract the resultant item

This logic should work without any change for different books sites because having pages with lists of books and then detail pages with the individual book is such a common way of structuring sites.

Let's refactor the spider presented in the former section so that it also supports extracting books from the page book-page.com/reviews as well.

The steps to follow are:

1. Make our spider generic: move the remaining extraction code from the spider to a Page Object
2. Configure *overrides* for Books to Scrape
3. Add support for another site (Book Page site)

2.5.1 Making the spider generic

This is almost done. The book extraction logic has been already moved to the `BookPage` Page Object, but extraction logic to obtain the list of URL to books is already present in the `parse` method. It must be moved to its own Page Object:

```
from web_poet.pages import WebPage


class BookListPage(WebPage):
```

(continues on next page)

(continued from previous page)

```
    def book_urls(self):
        return self.css(".image_container a")
```

Let's adapt the spider to use this new Page Object:

```
class BooksSpider(scrapy.Spider):
    name = "books_spider"
    parse_book = callback_for(BookPage)  # extract items from book pages

    def start_requests(self):
        yield scrapy.Request("http://books.toscrape.com/", self.parse)

    def parse(self, response, page: BookListPage):
        yield from response.follow_all(page.book_urls(), self.parse_book)
```

Warning: We could've defined our spider as:

```
class BooksSpider(scrapy.Spider):
    name = "books_spider"
    start_urls = ["http://books.toscrape.com/"]
    parse_book = callback_for(BookPage)  # extract items from book pages

    def parse(self, response, page: BookListPage):
        yield from response.follow_all(page.book_urls(), self.parse_book)
```

However, this would result in the following warning message:

> A request has been encountered with callback=None which defaults to the parse() method. On such cases, annotated dependencies in the parse() method won't be built by scrapy-poet. However, if the request has callback=parse, the annotated dependencies will be built.

This means that `page` isn't injected into the `parse()` method, leading to this error:

> TypeError: parse() missing 1 required positional argument: 'page'

This stems from the fact that using `start_urls` would use the predefined `start_requests()` method wherein `scrapy.Request` has `callback=None`.

One way to avoid this is to always declare the callback in `scrapy.Request`, just like in the original example.

See the *Pitfalls* section for more information.

All the extraction logic that is specific to the site is now responsibility of the Page Objects. As a result, the spider is now *site-agnostic* and will work providing that the Page Objects do their work.

In fact, the spider only responsibility becomes expressing the crawling strategy: "fetch a list of item URLs, follow them, and extract the resultant items". The code gets clearer and simpler.

2.5.2 Configure *overrides* for Books to Scrape

It is convenient to create bases classes for the Page Objects given that we are going to have several implementations of the same Page Object (one per each site). The following code snippet introduces such base classes and refactors the existing Page Objects as subclasses of them:

```
from web_poet.pages import WebPage


# ------ Base page objects ------

class BookListPage(WebPage):

    def book_urls(self):
        return []


class BookPage(WebPage):

    def to_item(self):
        return None

# ------ Concrete page objects for books.toscrape.com (BTS) ------

class BTSBookListPage(BookListPage):

    def book_urls(self):
        return self.css(".image_container a::attr(href)").getall()


class BTSBookPage(BookPage):

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

The spider won't work anymore after the change. The reason is that it is using the new base Page Objects and they are empty. Let's fix it by instructing `scrapy-poet` to use the Books To Scrape (BTS) Page Objects for URLs belonging to the domain `toscrape.com`. This must be done by configuring `SCRAPY_POET_RULES` into `settings.py`:

```
SCRAPY_POET_RULES = [
    ApplyRule("toscrape.com", BTSBookListPage, BookListPage),
    ApplyRule("toscrape.com", BTSBookPage, BookPage)
]
```

The spider is back to life! `SCRAPY_POET_RULES` contain rules that overrides the Page Objects used for a particular domain. In this particular case, Page Objects `BTSBookListPage` and `BTSBookPage` will be used instead of `BookListPage` and `BookPage` for any request whose domain is `toscrape.com`.

The right Page Objects will be then injected in the spider callbacks whenever a URL that belongs to the domain `toscrape.com` is requested.

2.5.3 Add another site

The code is now refactored to accept other implementations for other sites. Let's illustrate it by adding support for the books in the page bookpage.com/reviews.

We cannot reuse the Books to Scrape Page Objects in this case. The site is different so their extraction logic wouldn't work. Therefore, we have to implement new ones:

```
from web_poet.pages import WebPage


class BPBookListPage(WebPage):

    def book_urls(self):
        return self.css("article.post h4 a::attr(href)").getall()


class BPBookPage(WebPage):

    def to_item(self):
        return {
            "url": self.url,
            "name": self.css("body div > h1::text").get().strip(),
        }
```

The last step is configuring the overrides so that these new Page Objects are used for the domain `bookpage.com`. This is how `SCRAPY_POET_RULES` should look like into `settings.py`:

```
from web_poet import ApplyRule

SCRAPY_POET_RULES = [
    ApplyRule("toscrape.com", use=BTSBookListPage, instead_of=BookListPage),
    ApplyRule("toscrape.com", use=BTSBookPage, instead_of=BookPage),
    ApplyRule("bookpage.com", use=BPBookListPage, instead_of=BookListPage),
    ApplyRule("bookpage.com", use=BPBookPage, instead_of=BookPage)
]
```

The spider is now ready to extract books from both sites . The full example can be seen here

On the 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.

In the examples above we have been configuring the overrides for a particular domain, but more complex URL patterns are also possible. For example, the pattern `books.toscrape.com/cataloge/category/` is accepted and it would restrict the override only to category pages.

Note: Also see the url-matcher documentation for more information about the patterns syntax.

Manually defining overrides like this would be inconvenient, most especially for larger projects. Fortunately, scrapy-poet already retrieves the rules defined from web-poet's `default_registry`. This is done by setting the default value of the `SCRAPY_POET_RULES` setting as `web_poet.default_registry.get_rules()`.

However, this only works if page objects are annotated using the `web_poet.handle_urls()` decorator. You also need to set the `SCRAPY_POET_DISCOVER` setting so that these rules could be properly imported.

For more info on this, you can refer to these docs:

- `scrapy-poet`'s *Rules from web-poet* Tutorial section.
- External web-poet docs.
  - Specifically, the Rules documentation.

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 the *Rules from web-poet* and *Providers* sections as well as refer to spiders in the "example" folder: https://github.com/scrapinghub/scrapy-poet/tree/master/example/example/spiders

CHAPTER

THREE

ADVANCED TUTORIAL

This section intends to go over the supported features in **web-poet** by **scrapy-poet**:

- `web_poet.HttpClient`
- `web_poet.PageParams`

These are mainly achieved by **scrapy-poet** implementing **providers** for them:

- *`scrapy_poet.HttpClientProvider`*
- *`scrapy_poet.PageParamsProvider`*

3.1 Additional Requests

Using Page Objects using additional requests doesn't need anything special from the spider. It would work as-is because of the readily available *`scrapy_poet.HttpClientProvider`* that is enabled out of the box.

This supplies the Page Object with the necessary `web_poet.HttpClient` instance.

The HTTP client implementation that **scrapy-poet** provides to `web_poet.HttpClient` handles requests as follows:

- Requests go through downloader middlewares, but they do not go through spider middlewares or through the scheduler.
- Duplicate requests are not filtered out.
- In line with the web-poet specification for additional requests, `Request.meta["dont_redirect"]` is set to `True` for requests with the `HEAD` HTTP method.

Suppose we have the following Page Object:

```
import attr
import web_poet


@attr.define
class ProductPage(web_poet.WebPage):
    http: web_poet.HttpClient

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
```

(continues on next page)

(continued from previous page)

```
        }

        # Simulates clicking on a button that says "View All Images"
        response: web_poet.HttpResponse = await self.http.get(
            f"https://api.example.com/v2/images?id={item['product_id']}"
        )
        item["images"] = response.css(".product-images img::attr(src)").getall()
        return item
```

It can be directly used inside the spider as:

```
import scrapy


class ProductSpider(scrapy.Spider):

    custom_settings = {
        "DOWNLOADER_MIDDLEWARES": {
            "scrapy_poet.InjectionMiddleware": 543,
            "scrapy.downloadermiddlewares.stats.DownloaderStats": None,
            "scrapy_poet.DownloaderStatsMiddleware": 850,
        }
    }

    def start_requests(self):
        for url in [
            "https://example.com/category/product/item?id=123",
            "https://example.com/category/product/item?id=989",
        ]:
            yield scrapy.Request(url, callback=self.parse)

    async def parse(self, response, page: ProductPage):
        return await page.to_item()
```

Note that we needed to update the `parse()` method to be an `async` method, since the `to_item()` method of the Page Object we're using is an `async` method as well.

3.2 Page params

Using `web_poet.PageParams` allows the Scrapy spider to pass any arbitrary information into the Page Object.

Suppose we update the earlier Page Object to control the additional request. This basically acts as a switch to update the behavior of the Page Object:

```
import attr
import web_poet


@attr.define
class ProductPage(web_poet.WebPage):
    http: web_poet.HttpClient
```

(continues on next page)

(continued from previous page)

```
    page_params: web_poet.PageParams

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
        }

        # Simulates clicking on a button that says "View All Images"
        if self.page_params.get("enable_extracting_all_images")
            response: web_poet.HttpResponse = await self.http.get(
                f"https://api.example.com/v2/images?id={item['product_id']}"
            )
            item["images"] = response.css(".product-images img::attr(src)").getall()

        return item
```

Passing the `enable_extracting_all_images` page parameter from the spider into the Page Object can be achieved by using `scrapy.Request.meta` attribute. Specifically, any `dict` value inside the `page_params` parameter inside `scrapy.Request.meta` will be passed into `web_poet.PageParams`.

Let's see it in action:

```
import scrapy


class ProductSpider(scrapy.Spider):

    custom_settings = {
        "DOWNLOADER_MIDDLEWARES": {
            "scrapy_poet.InjectionMiddleware": 543,
            "scrapy.downloadermiddlewares.stats.DownloaderStats": None,
            "scrapy_poet.DownloaderStatsMiddleware": 850,
        }
    }

    start_urls = [
        "https://example.com/category/product/item?id=123",
        "https://example.com/category/product/item?id=989",
    ]

    def start_requests(self):
        for url in start_urls:
            yield scrapy.Request(
                url=url,
                callback=self.parse,
                meta={"page_params": {"enable_extracting_all_images": True}}
            )

    async def parse(self, response, page: ProductPage):
        return await page.to_item()
```

CHAPTER

FOUR

PITFALLS

4.1 `scrapy.Request` without callback

Tip: Note that the pitfalls discussed in this section aren't applicable to Scrapy >= 2.8 for most cases.

However, if you have code somewhere which directly adds `scrapy.Request` instances to the downloader, you need to ensure that they don't use `None` as the callback value. Instead, you can use the new `scrapy.http.request.NO_CALLBACK()` value introduced in Scrapy 2.8.

Note: This section *only applies* to specific cases where spiders define a `parse()` method.

The TLDR; recommendation is to simply avoid defining a `parse()` method and instead choose another name.

Scrapy supports declaring `scrapy.Request` instances without setting any callbacks (i.e. `None`). For these instances, Scrapy uses the `parse()` method as its callback.

Let's take a look at the following code:

```
import scrapy


class MySpider(scrapy.Spider):
    name = "my_spider"
    start_urls = ["https://books.toscrape.com"]

    def parse(self, response):
        ...
```

Under the hood, the inherited `start_requests()` method from `scrapy.Spider` doesn't declare any callback value to `scrapy.Request`:

```
for url in self.start_urls:
    yield Request(url, dont_filter=True)
```

Apart from this, there are also some built-in Scrapy < 2.8 features which omit the `scrapy.Request` callback value:

- `scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware`
- `scrapy.pipelines.images.ImagesPipeline`
- `scrapy.pipelines.files.FilesPipeline`

However, omitting the `scrapy.Request` callback value presents *some problems* for **scrapy-poet**.

4.1.1 Skipped Downloads

Note: This subsection is specific to cases wherein a *DummyResponse* annotates the response in a `parse()` method.

Let's take a look at an example:

```
import scrapy
from scrapy_poet import DummyResponse


class MySpider(scrapy.Spider):
    name = "my_spider"
    start_urls = ["https://books.toscrape.com"]

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

In order for the built-in Scrapy < 2.8 features listed above to work properly, **scrapy-poet** chooses to ignore the *DummyResponse* annotation completely. This means that the response is downloaded instead of being skipped.

Otherwise, `scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware` might not work properly and would **not** visit the `robots.txt` file from the website.

Moreover, this **scrapy-poet** behavior avoids the problem of the images or files being missing when the following pipelines are used:

- `scrapy.pipelines.images.ImagesPipeline`
- `scrapy.pipelines.files.FilesPipeline`

Note that the following `UserWarning` is emitted when encountering such scenario:

> A request has been encountered with callback=None which defaults to the parse() method. If the parse() method is annotated with scrapy_poet.DummyResponse (or its subclasses), we're assuming this isn't intended and would simply ignore this annotation.

To avoid the said warning and this **scrapy-poet** behavior from occurring, it'd be best to avoid defining a `parse()` method and instead choose any other name.

4.1.2 Dependency Building

Note: This subsection is specific to cases wherein dependencies are provided by **scrapy-poet** in the `parse()` method.

Let's take a look at the following code:

```
import attrs
import scrapy

from myproject.page_objects import MyPage
```

(continues on next page)

(continued from previous page)

```
class MySpider(scrapy.Spider):
    name = "my_spider"
    start_urls = ["https://books.toscrape.com"]

    def parse(self, response: scrapy.http.Response, page: MyPage):
        ...
```

In the above example, this error would be raised: `TypeError: parse() missing 1 required positional argument: 'page'`.

The reason for this **scrapy-poet** behavior is to prevent the wasted dependency building *(which could be expensive in some cases)* when the `parse()` method is unintentionally used.

For example, if a spider is using the `scrapy.pipelines.images.ImagesPipeline`, **scrapy-poet**'s *`scrapy_poet.downloadermiddlewares.InjectionMiddleware`* could be wasting precious compute resources to fulfill one or more dependencies that won't be used at all. Specifically, the `page` argument to the `parse()` method is not utilized. If there are a million of images to be downloaded, then the `page` instance is created a million times as well.

The following `UserWarning` is emitted on such scenario:

> A request has been encountered with callback=None which defaults to the parse() method. On such cases, annotated dependencies in the parse() method won't be built by scrapy-poet. However, if the request has callback=parse, the annotated dependencies will be built.

As the warning message suggests, this could be fixed by ensuring that the callback is **not** `None`:

```
class MySpider(scrapy.Spider):
    name = "my_spider"

    def start_requests(self):
        yield scrapy.Request("https://books.toscrape.com", callback=self.parse)

    def parse(self, response: scrapy.http.Response, page: MyPage):
        ...
```

The `UserWarning` is only shown when the `parse()` method declares any dependency that is fullfilled by any provider declared in `SCRAPY_POET_PROVIDERS`. This means that the following code doesn't produce the warning nor attempts to skip any dependency from being built because there is none:

```
class MySpider(scrapy.Spider):
    name = "my_spider"
    start_urls = ["https://books.toscrape.com"]

    def parse(self, response: scrapy.http.Response):
        ...
```

Similarly, the best way to completely avoid the said warning and this **scrapy-poet** behavior is to avoid defining a `parse()` method and instead choose any other name.

4.2 Opening a response in a web browser

When using scrapy-poet, the `open_in_browser` function from Scrapy may raise the following exception:

```
TypeError: Unsupported response type: HttpResponse
```

To avoid that, use the `open_in_browser` function from `scrapy_poet.utils` instead:

```
from scrapy_poet.utils import open_in_browser
```

CHAPTER

FIVE

RULES FROM WEB-POET

scrapy-poet fully supports the functionalities of `web_poet.ApplyRule`. It uses the registry from web_poet called `web_poet.RulesRegistry` which provides functionalties for:

- Returning the page object override if it exists for a given URL.
- Returning the page object capable of producing an item for a given URL.

A list of `web_poet.ApplyRule` can be configured by passing it to the `SCRAPY_POET_RULES` setting.

In this section, we go over its `instead_of` parameter for overrides and `to_return` for item returns. However, please make sure you also read web-poet's Rules documentation to see all of the expected behaviors of the rules.

5.1 Overrides

This functionality opens the door to configure specific Page Objects depending on the request URL domain. Please have a look to Scrapy Tutorial to learn the basics about overrides before digging deeper in the content of this page.

Tip: Some real-world examples on this topic can be found in:

- Example 1: shorter example
- Example 2: longer example
- Example 3: rules using `web_poet.handle_urls()` decorator and retrieving them via `web_poet.RulesRegistry.get_rules`

5.1.1 Page Objects refinement

Any `web_poet.pages.Injectable` or page input can be overridden. But the overriding mechanism stops for the children of any already overridden type. This opens the door to refining existing Page Objects without getting trapped in a cyclic dependency. For example, you might have an existing Page Object for book extraction:

```
class BookPage(ItemPage):
    def to_item(self):
        ...
```

Imagine this Page Object obtains its data from an external API. Therefore, it is not holding the page HTML code. But you want to extract an additional attribute (e.g. `ISBN`) that was not extracted by the original Page Object. Using inheritance is not enough in this case, though. No problem, you can just override it using the following Page Object:

```
class ISBNBookPage(WebPage):

    def __init__(self, response: HttpResponse, book_page: BookPage):
        super().__init__(response)
        self.book_page = book_page

    def to_item(self):
        item = self.book_page.to_item()
        item['isbn'] = self.css(".isbn-class::text").get()
        return item
```

And then override it for a particular domain using `settings.py`:

```
SCRAPY_POET_RULES = [
    ApplyRule("example.com", use=ISBNBookPage, instead_of=BookPage)
]
```

This new Page Object gets the original `BookPage` as dependency and enrich the obtained item with the ISBN from the page HTML.

Note: By design overrides rules are not applied to `ISBNBookPage` dependencies as it is an overridden type. If they were, it would end up in a cyclic dependency error because `ISBNBookPage` would depend on itself!

Note: This is an alternative more compact way of writing the above Page Object using `attr.define`:

```
@attr.define
class ISBNBookPage(WebPage):
    book_page: BookPage

    def to_item(self):
        item = self.book_page.to_item()
        item['isbn'] = self.css(".isbn-class::text").get()
        return item
```

5.1.2 Overrides rules

The following example configures an override that is only applied for book pages from `books.toscrape.com`:

```
from web_poet import ApplyRule


SCRAPY_POET_RULES = [
    ApplyRule(
        for_patterns=Patterns(
            include=["books.toscrape.com/cataloge/*index.html|"],
            exclude=["/catalogue/category/"]),
        use=MyBookPage,
        instead_of=BookPage
```

(continues on next page)

(continued from previous page)

```
    )
]
```

Note how category pages are excluded by using a `exclude` pattern. You can find more information about the patterns syntax in the url-matcher documentation.

5.1.3 Decorate Page Objects with the rules

Having the rules along with the Page Objects is a good idea, as you can identify with a single sight what the Page Object is doing along with where it is applied. This can be done by decorating the Page Objects with `web_poet.handle_urls()` provided by web-poet.

Tip: Make sure to read the Rules documentation of web-poet to learn all of its other functionalities that is not covered in this section.

Let's see an example:

```
from web_poet import handle_urls


@handle_urls("toscrape.com", instead_of=BookPage)
class BTSBookPage(BookPage):

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

The `web_poet.handle_urls()` decorator in this case is indicating that the class `BSTBookPage` should be used instead of `BookPage` for the domain `toscrape.com`.

5.1.4 Using the rules in scrapy-poet

scrapy-poet automatically uses the rules defined by page objects annotated with the `web_poet.handle_urls()` decorator by having the default value of the `SCRAPY_POET_RULES` setting set to `web_poet.default_registry.get_rules()`, which returns a `List[ApplyRule]`. Moreover, you also need to set the `SCRAPY_POET_DISCOVER` setting so that these rules could be properly imported.

Note: For more info and advanced features of web-poet's `web_poet.handle_urls()` and its registry, kindly read the web-poet documentation, specifically its Rules documentation.

5.2 Item Returns

scrapy-poet also supports a convenient way of asking for items directly. This is made possible by the `to_return` parameter of `web_poet.ApplyRule`. The `to_return` parameter specifies which item a page object is capable of returning for a given URL.

Let's check out an example:

```
import attrs
import scrapy
from web_poet import WebPage, handle_urls, field
from scrapy_poet import DummyResponse


@attrs.define
class Product:
    name: str


@handle_urls("example.com")
@attrs.define
class ProductPage(WebPage[Product]):

    @field
    def name(self) -> str:
        return self.css("h1.name ::text").get("")


class MySpider(scrapy.Spider):
    name = "myspider"

    def start_requests(self):
        yield scrapy.Request(
            "https://example.com/products/some-product", self.parse
        )

    # We can directly ask for the item here instead of the page object.
    def parse(self, response: DummyResponse, item: Product):
        return item
```

From this example, we can see that:

- Spider callbacks can directly ask for items as dependencies.
- The `Product` item instance directly comes from `ProductPage`.
- This is made possible by the `ApplyRule("example.com", use=ProductPage, to_return=Product)` instance created from the `@handle_urls` decorator on `ProductPage`.

Note: The slightly longer alternative way to do this is by declaring the page object itself as the dependency and then calling its `.to_item()` method. For example:

```
@handle_urls("example.com")
@attrs.define
```

(continues on next page)

(continued from previous page)

```
class ProductPage(WebPage[Product]):
    product_image_page: ProductImagePage

    @field
    def name(self) -> str:
        return self.css("h1.name ::text").get("")

    @field
    async def image(self) -> Image:
        return await self.product_image_page.to_item()


class MySpider(scrapy.Spider):
    name = "myspider"

    def start_requests(self):
        yield scrapy.Request(
            "https://example.com/products/some-product", self.parse
        )

    async def parse(self, response: DummyResponse, product_page: ProductPage):
        return await product_page.to_item()
```

For more information about all the expected behavior for the `to_return` parameter in `web_poet.ApplyRule`, check out web-poet Rules documentation.

CHAPTER

SIX

STATS

scrapy-poet tracks web-poet stats as part of Scrapy stats, prefixed with `poet/stats/`.

6.1 Injector stats

The injector produces some stats also. These are:

- cache stats have the `poet/cache` prefix.
- injected dependencies stats have the `poet/injector` prefix.

CHAPTER

SEVEN

PROVIDERS

Note: This document assumes a good familiarity with `web-poet` concepts; make sure you've read `web-poet` docs.

This page is mostly aimed at developers who want to extend `scrapy-poet`, **not** to developers who are writing extraction and crawling code using `scrapy-poet`.

7.1 Creating providers

Providers are responsible for building dependencies needed by Injectable objects. A good example would be the *scrapy_poet.HttpResponseProvider*, which builds and provides a `web_poet.HttpResponse` instance for Injectables that need it, like the `web_poet.WebPage`.

```
import attr
from typing import Set, Callable

import web_poet
from scrapy_poet.page_input_providers import PageObjectInputProvider
from scrapy import Response


class HttpResponseProvider(PageObjectInputProvider):
    """This class provides ``web_poet.HttpResponse`` instances."""
    provided_classes = {web_poet.HttpResponse}

    def __call__(self, to_provide: Set[Callable], response: Response):
        """Build a ``web_poet.HttpResponse`` instance using a Scrapy ``Response``"""
        return [
            web_poet.HttpResponse(
                url=response.url,
                body=response.body,
                status=response.status,
                headers=web_poet.HttpResponseHeaders.from_bytes_dict(response.headers),
            )
        ]
```

You can implement your own providers in order to extend or override current `scrapy-poet` behavior. All providers should inherit from this base class: *PageObjectInputProvider*.

Please, check the docs provided in the following API reference for more details: *PageObjectInputProvider*.

7.2 Cache Support in Providers

scrapy-poet also supports caching of the provided dependencies from the providers. For example, *HttpResponseProvider* supports this right off the bat. It's able to do this by inheriting the CacheDataProviderMixin and implementing all of its abstractmethods.

So, extending from the previous example we've tackled above to support cache would lead to the following code:

```
import web_poet
from scrapy_poet.page_input_providers import (
    CacheDataProviderMixin,
    PageObjectInputProvider,
)

class HttpResponseProvider(PageObjectInputProvider, CacheDataProviderMixin):
    """This class provides ``web_poet.HttpResponse`` instances."""
    provided_classes = {web_poet.HttpResponse}

    def __call__(self, to_provide: Set[Callable], response: Response):
        """Build a ``web_poet.HttpResponse`` instance using a Scrapy ``Response``"""
        return [
            web_poet.HttpResponse(
                url=response.url,
                body=response.body,
                status=response.status,
                headers=web_poet.HttpResponseHeaders.from_bytes_dict(response.headers),
            )
        ]

    def fingerprint(self, to_provide: Set[Callable], request: Request) -> str:
        """Returns a fingerprint to identify the specific request."""
        # Implementation here

    def serialize(self, result: Sequence[Any]) -> Any:
        """Serializes the results of this provider. The data returned will
        be pickled.
        """
        # Implementation here

    def deserialize(self, data: Any) -> Sequence[Any]:
        """Deserialize some results of the provider that were previously
        serialized using the serialize() method.
        """
        # Implementation here
```

Take note that even if you're using providers that supports the Caching interface, it's only going to be used if the SCRAPY_POET_CACHE has been enabled in the settings.

The caching of provided dependencies is **very useful for local development** of Page Objects, as it lowers down the waiting time for your Responses *(or any type of external dependency for that manner)* by caching them up locally.

Currently, the data is cached using a sqlite database in your local directory. This is implemented using SqlitedictCache.

The cache mechanism that scrapy-poet currently offers is quite different from the HttpCacheMiddleware which

Scrapy has. Although they are quite similar in its intended purpose, `scrapy-poet`'s cached data is directly tied to its appropriate provider. This could be anything that could stretch beyond Scrapy's `Responses` *(e.g. Network Database queries, API Calls, AWS S3 files, etc)*.

Note: The *scrapy_poet.injection.Injector* maintains a `.weak_cache` which stores the instances created by the providers as long as the corresponding `scrapy.Request` instance exists. This means that the instances created by earlier providers can be accessed and reused by latter providers. This is turned on by default and the instances are stored in memory.

7.3 Configuring providers

The list of available providers should be configured in the spider settings. For example, the following configuration should be included in the settings to enable a new provider `MyProvider`:

```
"SCRAPY_POET_PROVIDERS": {MyProvider: 500}
```

The number used as value (*500*) defines the provider priority. See Scrapy Middlewares configuration dictionaries for more information.

Note: The providers in `scrapy_poet.DEFAULT_PROVIDERS`, which includes a provider for `web_poet.HttpResponse`, are always included by default. You can disable any of them by listing it in the configuration with the priority *None*.

7.4 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: DummyResponse, 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
```

(continues on next page)

(continued from previous page)

```
@attr.define
class CachedData:
    key: str
    value: str


class CachedDataProvider(PageObjectInputProvider):
    provided_classes = {CachedData}

    def __call__(self, to_provide: List[Callable], request: scrapy.Request):
        return [
            CachedData(
                key=request.url,
                value=get_cached_content(request.url)
            )
        ]


@attr.define
class MyPageObject(ItemPage):
    content: CachedData

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


class MySpider(scrapy.Spider):
    name = "my_spider"

    def start_requests(self):
        yield scrapy.Request("http://books.toscrape.com/", self.parse_page)

    def parse_page(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.define
class MyResponseData:
```

(continues on next page)

(continued from previous page)

```
    url: str
    html: str


class MyResponseDataProvider(PageObjectInputProvider):
    provided_classes = {MyResponseData}

    def __call__(self, to_provide: Set[Callable], response: Response):
        return [
            MyResponseData(
                url=response.url,
                html=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 start_requests(self):
        yield scrapy.Request("http://books.toscrape.com/", self.parse_page)

    def parse_page(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 `scrapy.http.Response` instances in your Page Objects, use built-in `web_poet.WebPage` — it has `response` attribute with `web_poet.HttpResponse`; no additional configuration is needed, as there is *`HttpResponseProvider`* enabled in `scrapy-poet` by default.

7.4.1 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.

7.5 Attaching metadata to dependencies

Note: This feature requires Python 3.9+.

Providers can support dependencies with arbitrary metadata attached and use that metadata when creating them. Attaching the metadata is done by wrapping the dependency class in `typing.Annotated`:

```
@attr.define
class MyPageObject(ItemPage):
    response: Annotated[HtmlResponse, "foo", "bar"]
```

To handle this you need the following changes in your provider:

```
from andi.typeutils import strip_annotated
from scrapy_poet import PageObjectInputProvider
from web_poet.annotated import AnnotatedInstance


class Provider(PageObjectInputProvider):
    ...

    def is_provided(self, type_: Callable) -> bool:
        # needed so that you can list just the base type in provided_classes
        return super().is_provided(strip_annotated(type_))

    def __call__(self, to_provide):
        result = []
        for cls in to_provide:
```

(continues on next page)

(continued from previous page)

```
            metadata = getattr(cls, "__metadata__", None)
            obj = ...  # create the instance using cls and metadata
            if metadata:
                # wrap the instance into a web_poet.annotated.AnnotatedInstance object
                obj = AnnotatedInstance(obj, metadata)
            result.append(obj)
        return result
```

CHAPTER

EIGHT

TESTS FOR PAGE OBJECTS

`web-poet` provides tools for testing page objects. `scrapy-poet` projects can use a Scrapy command to easily generate tests:

```
scrapy savefixture my_project.pages.MyItemPage 'https://quotes.toscrape.com/page/1/'
```

This will request the provided page, create an instance of the provided page object for this page, request its `to_item()` method and save both the page object dependencies and the resulting item as a test fixture. These fixtures can then be used with the `pytest` plugin provided by `web-poet`.

8.1 Configuring the test location

The `SCRAPY_POET_TESTS_DIR` setting specifies where to create the tests. It can be set in the project settings or with the `-s` command argument.

8.2 Handling time fields

The tests generated by `savefixture` set the frozen_time metadata value to the time of the test creation.

8.3 Using spiders

By default `savefixture` creates a simple spider that uses the project settings and makes one request to the URL provided to the command. It may be needed instead to use a real spider from the project, for example because of its `custom_settings`. In this case you can pass the spider name as the third argument:

```
scrapy savefixture my_project.pages.MyItemPage 'https://quotes.toscrape.com/page/1/'␣
↪toscrape_listing
```

The command will try to run the spider overriding its `start_requests()`, so it should run just one request but it can break on spiders with complicated logic, e.g. ones that use `spider_idle` to schedule requests or modify items returned from `to_item()`. You may need to adapt your spiders to this, for example checking if the special `_SCRAPY_POET_SAVEFIXTURE` setting is set to `True` and using more simple logic in this case.

8.4 Configuring the item adapter

As documented in Item adapters, fixtures can use custom item adapters. The `savefixture` command uses the adapter specified in the `SCRAPY_POET_TESTS_ADAPTER` setting to save the fixture.

CHAPTER

NINE

SETTINGS

Configuring the settings denoted below would follow the usual methods used by Scrapy.

9.1 SCRAPY_POET_PROVIDERS

Default: `{}`

A `dict` wherein the **keys** would be the providers available for your Scrapy project while the **values** denotes the priority of the provider.

More info on this at this section: *Providers*.

9.2 SCRAPY_POET_OVERRIDES

Deprecated. Use `SCRAPY_POET_RULES` instead.

9.3 SCRAPY_POET_RULES

Default: `web_poet.default_registry.get_rules()`

Accepts a `List[ApplyRule]` which sets the rules to use.

> **Warning:** Although `SCRAPY_POET_RULES` already has values set from the return value of `web_poet.default_registry.get_rules()`, make sure to also set the `SCRAPY_POET_DISCOVER` setting below.

There are sections dedicated for this at Scrapy Tutorial and *Rules from web-poet*.

9.4 SCRAPY_POET_DISCOVER

Default: `[]`

A list of packages/modules (i.e. `List[str]`) which scrapy-poet will look for page objects annotated with the `web_poet.handle_urls()` decorator. Each package/module is passed into `web_poet.consume_modules` where each module from a package is recursively loaded.

This ensures that when using the default value of `SCRAPY_POET_RULES` set to `web_poet.default_registry.get_rules()`, it should contain all the intended rules.

Note that it's also possible for `SCRAPY_POET_RULES` to have rules not specified in `SCRAPY_POET_DISCOVER` (e.g. when the annotated page objects are inside your Scrapy project). However, it's recommended to still use `SCRAPY_POET_DISCOVER` to ensure all the intended rules are properly loaded.

9.5 SCRAPY_POET_CACHE

Default: `None`

The caching mechanism in the **providers** can be enabled by either setting this to `True` which configures the file path of the cache into a `.scrapy/` dir in your *local Scrapy project*.

On the other hand, you can also set this as a `str` pointing to any path relative to your *local Scrapy project*.

9.6 SCRAPY_POET_CACHE_ERRORS

Default: `False`

When this is set to `True`, any error that arises when retrieving dependencies from providers would be cached. This could be useful in cases during local development wherein you outright know that retrieving the dependency would fail and would choose to ignore it. Caching such errors would reduce the waiting time when developing Page Objects.

It's *recommended* to set this off into `False` by default since you might miss out on sporadic errors.

9.7 SCRAPY_POET_TESTS_DIR

Default: `fixtures`

Sets the location where the `savefixture` command creates tests.

More info at *Tests for Page Objects*.

9.8 SCRAPY_POET_TESTS_ADAPTER

Default: `None`

Sets the class, or its import path, that will be used as an adapter in the generated test fixtures.

More info at *Configuring the item adapter*.

9.9 SCRAPY_POET_REQUEST_FINGERPRINTER_BASE_CLASS

The default value is the default value of the `REQUEST_FINGERPRINTER_CLASS` setting for the version of Scrapy currently installed (e.g. `"scrapy.utils.request.RequestFingerprinter"`).

You can assign a request fingerprinter class to this setting to configure a custom request fingerprinter class to use for requests.

This class is used to generate a base fingerprint for a request. If that request uses dependency injection, that fingerprint is then modified to account for requested dependencies. Otherwise, the fingerprint is used as is.

Note: Annotations of *annotated dependencies* are serialized with `repr()` for fingerprinting purposes. If you find a real-world scenario where this is a problem, please open an issue.

CHAPTER

TEN

API REFERENCE

10.1 API

scrapy_poet.**callback_for**(*page_or_item_cls: Type*) → Callable

Create a callback for an `web_poet.ItemPage` subclass or an item class.

The generated callback returns the output of the `to_item` method, i.e. extracts a single item from a web page, using a Page Object.

This helper allows to reduce the boilerplate when working with Page Objects. For example, instead of this:

```
class BooksSpider(scrapy.Spider):
    name = "books"
    start_urls = ["http://books.toscrape.com/"]

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

    def parse_book(self, response: DummyResponse, page: BookPage):
        return page.to_item()
```

It allows to write this:

```
class BooksSpider(scrapy.Spider):
    name = "books"
    start_urls = ["http://books.toscrape.com/"]

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

    parse_book = callback_for(BookPage)
```

It also supports producing an async generator callable if the Page Objects's `to_item` method is a coroutine which uses the `async/await` syntax.

So if we have the following:

```
class BooksSpider(scrapy.Spider):
    name = "books"
    start_urls = ["http://books.toscrape.com/"]
```

(continues on next page)

(continued from previous page)

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

    async def parse_book(self, response: DummyResponse, page: BookPage):
        yield await page.to_item()
```

It could be turned into:

```
class BooksSpider(scrapy.Spider):
    name = "books"
    start_urls = ["http://books.toscrape.com/"]

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

    parse_book = callback_for(BookPage)
```

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 attribute (as shown in the example above) if you're planning to use disk queues, because in this case Scrapy is able to serialize your request object.

class scrapy_poet.**DummyResponse**(**args: Any*, ***kwargs: Any*)

This class is returned by the *InjectionMiddleware* when it detects that the download could be skipped. It inherits from `scrapy.http.Response` and signals and stores the URL and references the original `scrapy.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.http.Response`, the *InjectionMiddleware* is going to skip download and provide a *DummyResponse* to your parser instead.

10.2 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.downloadermiddlewares.**InjectionMiddleware**(*crawler: Crawler*)

This is a Downloader Middleware that's supposed to:

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

__init__(*crawler: Crawler*) → None

Initialize the middleware

process_request(*request: Request*, *spider: Spider*) → *DummyResponse* | None

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

If the `scrapy.http.Response` can be ignored, a *DummyResponse* instance is returned on its place. This *DummyResponse* is linked to the original `scrapy.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 Zyte's AutoExtract.

process_response(*request: Request*, *response: Response*, *spider: Spider*) → Generator[Deferred, object, Response]

This method fills `scrapy.Request.cb_kwargs` with instances for the required Page Objects found in the callback signature.

In other words, this method instantiates all `web_poet.Injectable` subclasses declared as request callback arguments and any other parameter with a *PageObjectInputProvider* configured for its type.

10.3 Page Input Providers

The Injection Middleware needs a standard way to build the Page Inputs dependencies that the Page Objects uses to get external data (e.g. the HTML). That's why we have created a colletion of Page Object Input Providers.

The current module implements a Page Input Provider for `web_poet.HttpResponse`, which is in charge of providing the response HTML from Scrapy. You could also implement different providers in order to acquire data from multiple external sources, for example, from scrapy-playwright or from an API for automatic extraction.

class scrapy_poet.page_input_providers.**HttpClientProvider**(*injector*)

This class provides `web_poet.HttpClient` instances.

__call__(*to_provide: Set[Callable]*, *crawler: Crawler*)

Creates an `web_poet.HttpClient` instance using Scrapy's downloader.

class scrapy_poet.page_input_providers.**HttpRequestProvider**(*injector*)

This class provides `web_poet.HttpRequest` instances.

__call__(*to_provide: Set[Callable]*, *request: Request*)

Builds a `web_poet.HttpRequest` instance using a `scrapy.http.Request` instance.

class scrapy_poet.page_input_providers.**HttpResponseProvider**(*injector*)

This class provides `web_poet.HttpResponse` instances.

__call__(*to_provide: Set[Callable]*, *response: Response*)

Builds a `web_poet.HttpResponse` instance using a `scrapy.http.Response` instance.

class scrapy_poet.page_input_providers.**ItemProvider**(*injector*)

async __call__(*to_provide: Set[Callable]*, *request: Request*, *response: Response*) → List[Any]

Call self as a function.

__init__(*injector*)

Initializes the provider. Invoked only at spider start up.

class scrapy_poet.page_input_providers.**PageObjectInputProvider**(*injector*)

This is the base class for creating Page Object Input Providers.

A Page Object Input Provider (POIP) takes responsibility for providing instances of some types to Scrapy callbacks. The types a POIP provides must be declared in the class attribute `provided_classes`.

POIPs are initialized when the spider starts by invoking the `__init__` method, which receives the `scrapy_poet.injection.Injector` instance as argument.

The `__call__` method must be overridden, and it is inside this method where the actual instances must be build. The default `__call__` signature is as follows:

```
def __call__(self, to_provide: Set[Callable]) -> Sequence[Any]:
```

Therefore, it receives a list of types to be provided and return a list with the instances created (don't get confused by the `Callable` annotation. Think on it as a synonym of `Type`).

Additional dependencies can be declared in the `__call__` signature that will be automatically injected. Currently, scrapy-poet is able to inject instances of the following classes:

- `Request`
- `Response`
- `Crawler`
- `Settings`
- `StatsCollector`

Finally, `__call__` function can execute asynchronous code. Just either prepend the declaration with `async` to use futures or annotate it with `@inlineCallbacks` for deferred execution. Additionally, you might want to configure Scrapy `TWISTED_REACTOR` to support `asyncio` libraries.

The available POIPs should be declared in the spider setting using the key `SCRAPY_POET_PROVIDERS`. It must be a dictionary that follows same structure than the Scrapy Middlewares configuration dictionaries.

A simple example of a provider:

```
class BodyHtml(str): pass

class BodyHtmlProvider(PageObjectInputProvider):
    provided_classes = {BodyHtml}

    def __call__(self, to_provide, response: Response):
        return [BodyHtml(response.css("html body").get())]
```

The **provided_classes** class attribute is the `set` of classes that this provider provides. Alternatively, it can be a function with type `Callable[[Callable], bool]` that returns `True` if and only if the given type, which must be callable, is provided by this provider.

__init__(*injector*)

Initializes the provider. Invoked only at spider start up.

is_provided(*type_: Callable*) → bool

Return `True` if the given type is provided by this provider based on the value of the attribute `provided_classes`

class scrapy_poet.page_input_providers.**PageParamsProvider**(*injector*)

This class provides `web_poet.PageParams` instances.

__call__(*to_provide: Set[Callable]*, *request: Request*)

Creates a `web_poet.PageParams` instance based on the data found from the `meta["page_params"]` field of a `scrapy.http.Response` instance.

class scrapy_poet.page_input_providers.**RequestUrlProvider**(*injector*)

This class provides `web_poet.RequestUrl` instances.

__call__(*to_provide: Set[Callable]*, *request: Request*)

Builds a `web_poet.RequestUrl` instance using `scrapy.Request` instance.

class scrapy_poet.page_input_providers.**ResponseUrlProvider**(*injector*)

__call__(*to_provide: Set[Callable]*, *response: Response*)

Builds a `web_poet.RequestUrl` instance using a `scrapy.http.Response` instance.

class scrapy_poet.page_input_providers.**ScrapyPoetStatCollector**(*stats*)

__init__(*stats*)

inc(*key: str*, *value: int | float = 1*) → None

Increment the value of stat *key* by *value*, or set it to *value* if *key* has no value.

set(*key: str*, *value: Any*) → None

Set the value of stat *key* to *value*.

class scrapy_poet.page_input_providers.**StatsProvider**(*injector*)

This class provides `web_poet.Stats` instances.

__call__(*to_provide: Set[Callable]*, *crawler: Crawler*)

Creates an `web_poet.Stats` instance using Scrapy's stat collector.

10.4 Cache

class scrapy_poet.cache.**SerializedDataCache**(*directory: str | PathLike*)

Stores dependencies from Providers in a persistent local storage using *web_poet.serialization.SerializedDataFileStorage*

__init__(*directory: str | PathLike*) → None

10.5 Injection

class scrapy_poet.injection.**Injector**(*crawler: Crawler*, ***, *default_providers: Mapping | None = None*, *registry: RulesRegistry | None = None*)

Keep all the logic required to do dependency injection in Scrapy callbacks. Initializes the providers from the spider settings at initialization.

__init__(*crawler: Crawler*, ***, *default_providers: Mapping | None = None*, *registry: RulesRegistry | None = None*)

build_callback_dependencies(*request: Request*, *response: Response*)

Scan the configured callback for this request looking for the dependencies and build the corresponding instances. Return a kwargs dictionary with the built instances.

build_instances(*request: Request*, *response: Response*, *plan: Plan*)

Build the instances dict from a plan including external dependencies.

build_instances_from_providers(*request: Request*, *response: Response*, *plan: Plan*)

Build dependencies handled by registered providers

build_plan(*request: Request*) → Plan

Create a plan for building the dependencies required by the callback

discover_callback_providers(*request: Request*) → Set[*PageObjectInputProvider*]

Discover the providers that are required to fulfil the callback dependencies

is_scrapy_response_required(*request: Request*)

Check whether Scrapy's `Request`'s `Response` is going to be used.

scrapy_poet.injection.**get_callback**(*request*, *spider*)

Get the `scrapy.Request.callback` of a `scrapy.Request`.

scrapy_poet.injection.**get_injector_for_testing**(*providers: Mapping*, *additional_settings: Dict | None = None*, *registry: RulesRegistry | None = None*) → *Injector*

Return an *Injector* using a fake crawler. Useful for testing providers

scrapy_poet.injection.**get_response_for_testing**(*callback: Callable*) → Response

Return a `scrapy.http.Response` with fake content with the configured callback. It is useful for testing providers.

scrapy_poet.injection.**is_callback_requiring_scrapy_response**(*callback: ~typing.Callable*, *raw_callback: ~typing.Any = <object object>*) → bool

Check whether the request's callback method requires the response. Basically, it won't be required if the response argument in the callback is annotated with *DummyResponse*.

scrapy_poet.injection.**is_class_provided_by_any_provider_fn**(*providers: List[PageObjectInputProvider]*) → Callable[[Callable], bool]

Return a function of type `Callable[[Type], bool]` that return True if the given type is provided by any of the registered providers.

The `is_provided` method from each provider is used.

scrapy_poet.injection.**is_provider_requiring_scrapy_response**(*provider*)

Check whether injectable provider makes use of a valid `scrapy.http.Response`.

10.6 Injection errors

exception scrapy_poet.injection_errors.**InjectionError**

exception scrapy_poet.injection_errors.**MalformedProvidedClassesError**

exception scrapy_poet.injection_errors.**NonCallableProviderError**

exception scrapy_poet.injection_errors.**ProviderDependencyDeadlockError**

This is raised when it's not possible to create the dependencies due to deadlock.

For example:

- Page object named “ChickenPage” require “EggPage” as a dependency.
- Page object named “EggPage” require “ChickenPage” as a dependency.

exception scrapy_poet.injection_errors.**UndeclaredProvidedTypeError**

CHAPTER

ELEVEN

CONTRIBUTING

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

11.1 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

11.2 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.

11.3 Testing

We use tox to run tests with different Python versions:

```
tox
```

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

CHAPTER

TWELVE

CHANGELOG

12.1 0.22.1 (2024-03-07)

- Fixed `scrapy savefixture` not finding page object modules when used outside a Scrapy project.

12.2 0.22.0 (2024-03-04)

- Now requires `web-poet >= 0.17.0` and `time_machine >= 2.7.1`.
- Removed `scrapy_poet.AnnotatedResult`, use `web_poet.annotated.AnnotatedInstance` instead.
- Added support for annotated dependencies to the `scrapy savefixture` command.
- Test improvements.

12.3 0.21.0 (2024-02-08)

- Added a `.weak_cache` to *scrapy_poet.injection.Injector* which stores instances created by providers as long as the `scrapy.Request` exists.
- Fixed the incorrect value of `downloader/response_count` in the stats due to additional counting of *scrapy_poet.api.DummyResponse*.
- Fixed the detection of *scrapy_poet.api.DummyResponse* when some type hints are annotated using strings.

12.4 0.20.1 (2024-01-24)

- `ScrapyPoetRequestFingerprinter` now supports item dependencies.

12.5 0.20.0 (2024-01-15)

- Add `ScrapyPoetRequestFingerprinter`, a request fingerprinter that uses request dependencies in the fingerprint generation.

12.6 0.19.0 (2023-12-26)

- Now requires `andi >= 0.6.0`.
- Changed the implementation of resolving and building item dependencies from page objects. Now `andi` custom builders are used to create a single plan that includes building page objects and items. This fixes problems such as providers being called multiple times.
  - *ItemProvider* is now no-op. It's no longer enabled by default and users should also stop enabling it.
  - `PageObjectInputProvider.allow_prev_instances` and code related to it were removed so custom providers may need updating.
- Fixed some tests.

12.7 0.18.0 (2023-12-12)

- Now requires `andi >= 0.5.0`.
- Add support for dependency metadata via `typing.Annotated` (requires Python 3.9+).

12.8 0.17.0 (2023-12-11)

- Now requires `web-poet >= 0.15.1`.
- `HttpRequest` dependencies are now supported, via *HttpRequestProvider* (enabled by default).
- Enable *StatsProvider*, which provides `Stats` dependencies, by default.
- More robust disabling of *InjectionMiddleware* in the `scrapy savefixture` command.
- Official support for Python 3.12.

12.9 0.16.1 (2023-11-02)

- Fix the bug that caused requests produced by *HttpClientProvider* to be treated as if they need arguments of the `parse` callback as dependencies, which could cause returning an empty response and/or making extra provider calls.

12.10 0.16.0 (2023-09-26)

- Now requires `time_machine >= 2.2.0`.
- `ItemProvider` now supports page objects that declare a dependency on the same type of item that they return, as long as there is an earlier page object input provider that can provide such dependency.
- Fix running tests with Scrapy 2.11.

12.11 0.15.1 (2023-09-15)

- *scrapy-poet stats* now also include counters for injected dependencies (`poet/injector/<dependency import path>`).
- All scrapy-poet stats that used to be prefixed with `scrapy-poet/` are now prefixed with `poet/` instead.

12.12 0.15.0 (2023-09-12)

- Now requires `web-poet >= 0.15.0`.
- Web-poet stats are now *supported*.

12.13 0.14.0 (2023-09-08)

- Python 3.7 support has been dropped.
- Caching is now built on top of web-poet serialization, extending caching support to additional inputs, while making our code simpler, more reliable, and more future-proof.

  This has resulted in a few backward-incompatible changes:

  - The `scrapy_poet.page_input_providers.CacheDataProviderMixin` mixin class has been removed. Providers no longer need to use it or reimplement its methods.
  - The `SCRAPY_POET_CACHE_GZIP` setting has been removed.
- Added `scrapy_poet.utils.open_in_browser`, an alternative to `scrapy.utils.response.open_in_browser` that supports scrapy-poet.
- Fixed some documentation links.

12.14 0.13.0 (2023-05-08)

- Now requires `web-poet >= 0.12.0`.
- The `scrapy savefixture` command now uses the adapter from the `SCRAPY_POET_TESTS_ADAPTER` setting to save the fixture.
- Fix a typo in the docs.

12.15 0.12.0 (2023-04-26)

- Now requires `web-poet >= 0.11.0`.
- The `scrapy savefixture` command can now generate tests that expect that `to_item()` raises a specific exception (only `web_poet.exceptions.PageObjectAction` and its descendants are expected).
- Fixed an error when using `scrapy shell` with `scrapy_poet.InjectionMiddleware` enabled.
- Add a `twine check` CI check.

12.16 0.11.0 (2023-03-17)

- The `scrapy savefixture` command can now generate a fixture *using an existing spider*.

12.17 0.10.1 (2023-03-03)

- More robust time freezing in `scrapy savefixture` command.

12.18 0.10.0 (2023-02-24)

- Now requires `web-poet >= 0.8.0`.
- The `savefixture` command now also saves requests made via the `web_poet.page_inputs.client.HttpClient` dependency and their responses.

12.19 0.9.0 (2023-02-17)

- Added support for item classes which are used as dependencies in page objects and spider callbacks. The following is now possible:

```
import attrs
import scrapy
from web_poet import WebPage, handle_urls, field
from scrapy_poet import DummyResponse

@attrs.define
class Image:
    url: str

@handle_urls("example.com")
class ProductImagePage(WebPage[Image]):
    @field
    def url(self) -> str:
        return self.css("#product img ::attr(href)").get("")

@attrs.define
class Product:
```

(continues on next page)

(continued from previous page)

```
    name: str
    image: Image

@handle_urls("example.com")
@attrs.define
class ProductPage(WebPage[Product]):
    #  NEW: The page object can ask for items as dependencies. An instance
    # of ``Image`` is injected behind the scenes by calling the ``.to_item()``
    # method of ``ProductImagePage``.
    image_item: Image

    @field
    def name(self) -> str:
        return self.css("h1.name ::text").get("")

    @field
    def image(self) -> Image:
        return self.image_item

class MySpider(scrapy.Spider):
    name = "myspider"

    def start_requests(self):
        yield scrapy.Request(
            "https://example.com/products/some-product", self.parse_product
        )

    #  NEW: We can directly use the item here instead of the page object.
    def parse_product(self, response: DummyResponse, item: Product) -> Product:
        return item
```

In line with this, the following new features were made:

  - New *scrapy_poet.page_input_providers.ItemProvider* which makes the usage above possible.
  - An item class is now supported by *scrapy_poet.callback_for()* alongside the usual page objects. This means that it won't raise a `TypeError` anymore when not passing a subclass of `web_poet.pages.ItemPage`.
  - New exception: *scrapy_poet.injection_errors.ProviderDependencyDeadlockError*. This is raised when it's not possible to create the dependencies due to a deadlock in their sub-dependencies, e.g. due to a circular dependency between page objects.

- New setting named `SCRAPY_POET_RULES` having a default value of `web_poet.default_registry.get_rules`. This deprecates `SCRAPY_POET_OVERRIDES`.
- New setting named `SCRAPY_POET_DISCOVER` to ensure that `SCRAPY_POET_RULES` have properly loaded all intended rules annotated with the `@handle_urls` decorator.
- New utility functions in `scrapy_poet.utils.testing`.
- The `frozen_time` value inside the *test fixtures* won't contain microseconds anymore.
- Supports the new `scrapy.http.request.NO_CALLBACK()` introduced in **Scrapy 2.8**. This means that the *Pitfalls* (introduced in `scrapy-poet==0.7.0`) doesn't apply when you're using Scrapy >= 2.8, unless you're using third-party middlewares which directly uses the downloader to add `scrapy.Request` instances with callback set to `None`. Otherwise, you need to set the callback value to `scrapy.http.request.NO_CALLBACK()`.

- Fix the TypeError that's raised when using Twisted <= 21.7.0 since scrapy-poet was using `twisted.internet.defer.Deferred[object]` type annotation before which was not subscriptable in the early Twisted versions.
- Fix the `twisted.internet.error.ReactorAlreadyInstalledError` error raised when using the `scrapy savefixture` command and Twisted < 21.2.0 is installed.
- Fix test configuration that doesn't follow the intended commands and dependencies in these tox environments: `min`, `asyncio-min`, and `asyncio`. This ensures that page objects using `asyncio` should work properly, alongside the minimum specified Twisted version.
- Various improvements to tests and documentation.
- Backward incompatible changes:
  - For the *scrapy_poet.page_input_providers.PageObjectInputProvider* base class:
    - It now accepts an instance of *scrapy_poet.injection.Injector* in its constructor instead of scrapy.crawler.Crawler. Although you can still access the scrapy.crawler.Crawler via the `Injector.crawler` attribute.
    - *scrapy_poet.page_input_providers.PageObjectInputProvider.is_provided()* is now an instance method instead of a class method.
  - The *scrapy_poet.injection.Injector*'s attribute and constructor parameter called `overrides_registry` is now simply called `registry`.
  - Removed the `SCRAPY_POET_OVERRIDES_REGISTRY` setting which overrides the default registry.
  - The `scrapy_poet.overrides` module which contained `OverridesRegistryBase` and `OverridesRegistry` has now been removed. Instead, scrapy-poet directly uses web_poet.rules.RulesRegistry.

    Everything should pretty much the same except for web_poet.rules.RulesRegistry.overrides_for() now accepts str, web_poet.page_inputs.http.RequestUrl, or web_poet.page_inputs.http.ResponseUrl instead of scrapy.http.Request.
  - This also means that the registry doesn't accept tuples as rules anymore. Only web_poet.rules.ApplyRule instances are allowed. The same goes for `SCRAPY_POET_RULES` (and the deprecated `SCRAPY_POET_OVERRIDES`).
  - The following type aliases have been removed:
    - `scrapy_poet.overrides.RuleAsTuple`
    - `scrapy_poet.overrides.RuleFromUser`

12.20 0.8.0 (2023-01-24)

- Now requires `web-poet >= 0.7.0` and `time_machine`.
- Added a `savefixture` command that creates a test for a page object. See *Tests for Page Objects* for more information.

12.21 0.7.0 (2023-01-17)

- Fixed the issue where a new page object containing a new response data is not properly created when `web_poet.exceptions.core.Retry` is raised.
- In order for the above fix to be possible, overriding the callback dependencies created by **scrapy-poet** via `scrapy.http.Request.cb_kwargs` is now unsupported. This is a **backward incompatible** change.
- Fixed the broken `scrapy_poet.page_input_providers.HttpResponseProvider.fingerprint()` which errors out when running a Scrapy job using the `SCRAPY_POET_CACHE` enabled.
- Improved behavior when `spider.parse()` method arguments are supposed to be provided by **scrapy-poet**. Previously, it was causing unnecessary work in unexpected places like `scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware`, `scrapy.pipelines.images.ImagesPipeline` or `scrapy.pipelines.files.FilesPipeline`. It is also a reason `web_poet.page_inputs.client.HttpClient` might not be working in page objects. Now these cases are detected, and a warning is issued.

  As of Scrapy 2.7, it is not possible to fix the issue completely in **scrapy-poet**. Fixing it would require Scrapy changes; some 3rd party libraries may also need to be updated.

  Note: The root of the issue is that when request.callback is `None`, `parse()` callback is assumed normally. But sometimes callback=None is used when `scrapy.http.Request` is added to the Scrapy's downloader directly, in which case no callback is used. Middlewares, including **scrapy-poet**'s, can't distinguish between these two cases, which causes all kinds of issues.

  We recommend all **scrapy-poet** users to modify their code to avoid the issue. Please **don't** define `parse()` method with arguments which are supposed to be filled by **scrapy-poet**, and rename the existing `parse()` methods if they have such arguments. Any other name is fine. It avoids all possible issues, including incompatibility with 3rd party middlewares or pipelines.

  See the new *Pitfalls* documentation for more information.

  There are backwards-incompatible changes related to this issue. They only affect you if you don't follow the advice of not using `parse()` method with **scrapy-poet**.

  - When the `parse()` method has its response argument annotated with *scrapy_poet.api.DummyResponse*, for instance: `def parse(self, response: DummyResponse)`, the response is downloaded instead of being skipped.
  - When the `parse()` method has dependencies that are provided by **scrapy-poet**, the *scrapy_poet.downloadermiddlewares.InjectionMiddleware* won't attempt to build any dependencies anymore.

    This causes the following code to have this error `TypeError: parse() missing 1 required positional argument: 'page'.`:

    ```
    class MySpider(scrapy.Spider):
        name = "my_spider"
        start_urls = ["https://books.toscrape.com"]

        def parse(self, response: scrapy.http.Response, page: MyPage):
            ...
    ```

- *scrapy_poet.injection.is_callback_requiring_scrapy_response()* now accepts an optional `raw_callback` parameter meant to represent the actual callback attribute value of `scrapy.http.Request` since the original `callback` parameter could be normalized to the spider's `parse()` method when the `scrapy.http.Request` has `callback` set to `None`.

- Official support for Python 3.11
- Various updates and improvements on docs and examples.

12.22 0.6.0 (2022-11-24)

- Now requires `web-poet >= 0.6.0`.
  - All examples in the docs and tests now use `web_poet.WebPage` instead of `web_poet.ItemWebPage`.
  - The new `instead_of` parameter of the `@handle_urls` decorator is now preferred instead of the deprecated `overrides` parameter.
  - `scrapy_poet.callback_for` doesn't require an implemented `to_item` method anymore.
  - The new `web_poet.rules.RulesRegistry` is used instead of the old `web_poet.overrides.PageObjectRegistry`.
  - The Registry now uses `web_poet.ApplyRule` instead of `web_poet.OverrideRule`.
- Provider for `web_poet.ResponseUrl` is added, which allows to access the response URL in the page object. This triggers a download unlike the provider for `web_poet.RequestUrl`.
- Fixes the error when using `scrapy shell` while the `scrapy_poet.InjectionMiddleware` is enabled.
- Fixes and improvements on code and docs.

12.23 0.5.1 (2022-07-28)

Fixes the minimum web-poet version being 0.5.0 instead of 0.4.0.

12.24 0.5.0 (2022-07-28)

This release implements support for page object retries, introduced in web-poet 0.4.0.

To enable retry support, you need to configure a new spider middleware in your Scrapy settings:

```
SPIDER_MIDDLEWARES = {
    "scrapy_poet.RetryMiddleware": 275,
}
```

web-poet 0.4.0 is now the minimum required version of web-poet.

12.25 0.4.0 (2022-06-20)

This release is backwards incompatible, following backwards-incompatible changes in web-poet 0.2.0.

The main new feature is support for `web-poet >= 0.2.0`, including support for `async def to_item` methods, making additional requests in the `to_item` method, new Page Object dependencies, and the new way to configure overrides.

Changes in line with `web-poet >= 0.2.0`:

- `web_poet.HttpResponse` replaces `web_poet.ResponseData` as a dependency to use.

- Additional requests inside Page Objects: a provider for `web_poet.HttpClient`, as well as `web_poet.HttpClient` backend implementation, which uses Scrapy downloader.
- `callback_for` now supports Page Objects which define `async def to_item` method.
- Provider for `web_poet.PageParams` is added, which uses `request.meta["page_params"]` value.
- Provider for `web_poet.RequestUrl` is added, which allows to access the request URL in the page object without triggering the download.
- We have these **backward incompatible** changes since the `web_poet.OverrideRule` follow a different structure:
  - Deprecated `PerDomainOverridesRegistry` in lieu of the newer `OverridesRegistry` which provides a wide variety of features for better URL matching.
  - This resuls in a newer format in the `SCRAPY_POET_OVERRIDES` setting.

Other changes:

- New `scrapy_poet/dummy_response_count` value appears in Scrapy stats; it is the number of times `DummyResponse` is used instead of downloading the response as usual.
- `scrapy.utils.reqser` deprecated module is no longer used by scrapy-poet.

Dependency updates:

- The minimum supported Scrapy version is now `2.6.0`.
- The minimum supported web-poet version is now `0.2.0`.

12.26 0.3.0 (2022-01-28)

- Cache mechanism using `SCRAPY_POET_CACHE`
- Fixed and improved docs
- removed support for Python 3.6
- added support for Python 3.10

12.27 0.2.1 (2021-06-11)

- Improved logging message for DummyResponse
- various internal cleanups

12.28 0.2.0 (2021-01-22)

- Overrides support

12.29 0.1.0 (2020-12-29)

- New providers interface
  - One provider can provide many types at once
  - Single instance during the whole spider lifespan
  - Registration is now explicit and done in the spider settings
- CI is migrated from Travis to Github Actions
- Python 3.9 support

12.30 0.0.3 (2020-07-19)

- Documentation improvements
- providers can now access various Scrapy objects: Crawler, Settings, Spider, Request, Response, StatsCollector

12.31 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.

12.32 0.0.1 (2019-08-28)

Initial release.

CHAPTER

THIRTEEN

LICENSE

PYTHON MODULE INDEX

S

scrapy_poet.cache, 49
scrapy_poet.downloadermiddlewares, 46
scrapy_poet.injection, 49
scrapy_poet.injection_errors, 50
scrapy_poet.page_input_providers, 47

INDEX

Symbols

__call__() (*scrapy_poet.page_input_providers.HttpClientProvider method*), 47
__call__() (*scrapy_poet.page_input_providers.HttpRequestProvider method*), 47
__call__() (*scrapy_poet.page_input_providers.HttpResponseProvider method*), 47
__call__() (*scrapy_poet.page_input_providers.ItemProvider method*), 47
__call__() (*scrapy_poet.page_input_providers.PageParamsProvider method*), 48
__call__() (*scrapy_poet.page_input_providers.RequestUrlProvider method*), 49
__call__() (*scrapy_poet.page_input_providers.ResponseUrlProvider method*), 49
__call__() (*scrapy_poet.page_input_providers.StatsProvider method*), 49
__init__() (*scrapy_poet.cache.SerializedDataCache method*), 49
__init__() (*scrapy_poet.downloadermiddlewares.InjectionMiddleware method*), 46
__init__() (*scrapy_poet.injection.Injector method*), 49
__init__() (*scrapy_poet.page_input_providers.ItemProvider method*), 47
__init__() (*scrapy_poet.page_input_providers.PageObjectInputProvider method*), 48
__init__() (*scrapy_poet.page_input_providers.ScrapyPoetStatCollector method*), 49

B

build_callback_dependencies() (*scrapy_poet.injection.Injector method*), 49
build_instances() (*scrapy_poet.injection.Injector method*), 49
build_instances_from_providers() (*scrapy_poet.injection.Injector method*), 50
build_plan() (*scrapy_poet.injection.Injector method*), 50

C

callback_for() (*in module scrapy_poet*), 45

D

discover_callback_providers() (*scrapy_poet.injection.Injector method*), 50
DummyResponse (*class in scrapy_poet*), 46

G

get_callback() (*in module scrapy_poet.injection*), 50
get_injector_for_testing() (*in module scrapy_poet.injection*), 50
get_response_for_testing() (*in module scrapy_poet.injection*), 50

H

HttpClientProvider (*class in scrapy_poet.page_input_providers*), 47
HttpRequestProvider (*class in scrapy_poet.page_input_providers*), 47
HttpResponseProvider (*class in scrapy_poet.page_input_providers*), 47

I

inc() (*scrapy_poet.page_input_providers.ScrapyPoetStatCollector method*), 49
InjectionError, 50
InjectionMiddleware (*class in scrapy_poet.downloadermiddlewares*), 46
Injector (*class in scrapy_poet.injection*), 49
is_callback_requiring_scrapy_response() (*in module scrapy_poet.injection*), 50
is_class_provided_by_any_provider_fn() (*in module scrapy_poet.injection*), 50
is_provided() (*scrapy_poet.page_input_providers.PageObjectInputProvi method*), 48
is_provider_requiring_scrapy_response() (*in module scrapy_poet.injection*), 50
is_scrapy_response_required() (*scrapy_poet.injection.Injector method*), 50

ItemProvider (*class in scrapy_poet.page_input_providers*), 47

M

MalformedProvidedClassesError, 50
module
    scrapy_poet.cache, 49
    scrapy_poet.downloadermiddlewares, 46
    scrapy_poet.injection, 49
    scrapy_poet.injection_errors, 50
    scrapy_poet.page_input_providers, 47

N

NonCallableProviderError, 50

P

PageObjectInputProvider (*class in scrapy_poet.page_input_providers*), 47
PageParamsProvider (*class in scrapy_poet.page_input_providers*), 48
process_request() (*scrapy_poet.downloadermiddlewares.InjectionMiddleware method*), 46
process_response() (*scrapy_poet.downloadermiddlewares.InjectionMiddleware method*), 47
ProviderDependencyDeadlockError, 50

R

RequestUrlProvider (*class in scrapy_poet.page_input_providers*), 49
ResponseUrlProvider (*class in scrapy_poet.page_input_providers*), 49

S

scrapy_poet.cache
    module, 49
scrapy_poet.downloadermiddlewares
    module, 46
scrapy_poet.injection
    module, 49
scrapy_poet.injection_errors
    module, 50
scrapy_poet.page_input_providers
    module, 47
ScrapyPoetStatCollector (*class in scrapy_poet.page_input_providers*), 49
SerializedDataCache (*class in scrapy_poet.cache*), 49
set() (*scrapy_poet.page_input_providers.ScrapyPoetStatCollector method*), 49
StatsProvider (*class in scrapy_poet.page_input_providers*), 49

U

UndeclaredProvidedTypeError, 51