import requests && import threading $\Longrightarrow$ import scrapy

Jacopo Notarstefano
jacopo [at] nabla.com

September 10th, 2019

The Problem

Let's suppose that someone asks you to scrape a website containing a certain collection of N articles ($N \approx 10^4$), where each article is a webpage roughly one megabyte in size.

You start thinking about using the Python package `requests`, perhaps paired for performance reasons with the `threading` portion of the standard library, but then you start worrying about the GIL...

The Solution

A Claim

Claim (Notarstefano, 2019)

Any piece of code that uses both `requests` *and* `threading` *should instead be a* `scrapy` *spider.*

A First scrapy Spider

```
from scrapy import Spider

class BlogSpider(Spider):
    name = 'blogspider'
    allowed_domains = ['scrapinghub.com']
    start_urls = ['https://blog.scrapinghub.com']

    def parse(self, response):
        for title in response.css('.post-header > h2'):
            yield {'title': title.css('a ::text').get()}

        for next_page in response.css('a.next-posts-link'):
            yield response.follow(next_page, self.parse)
```

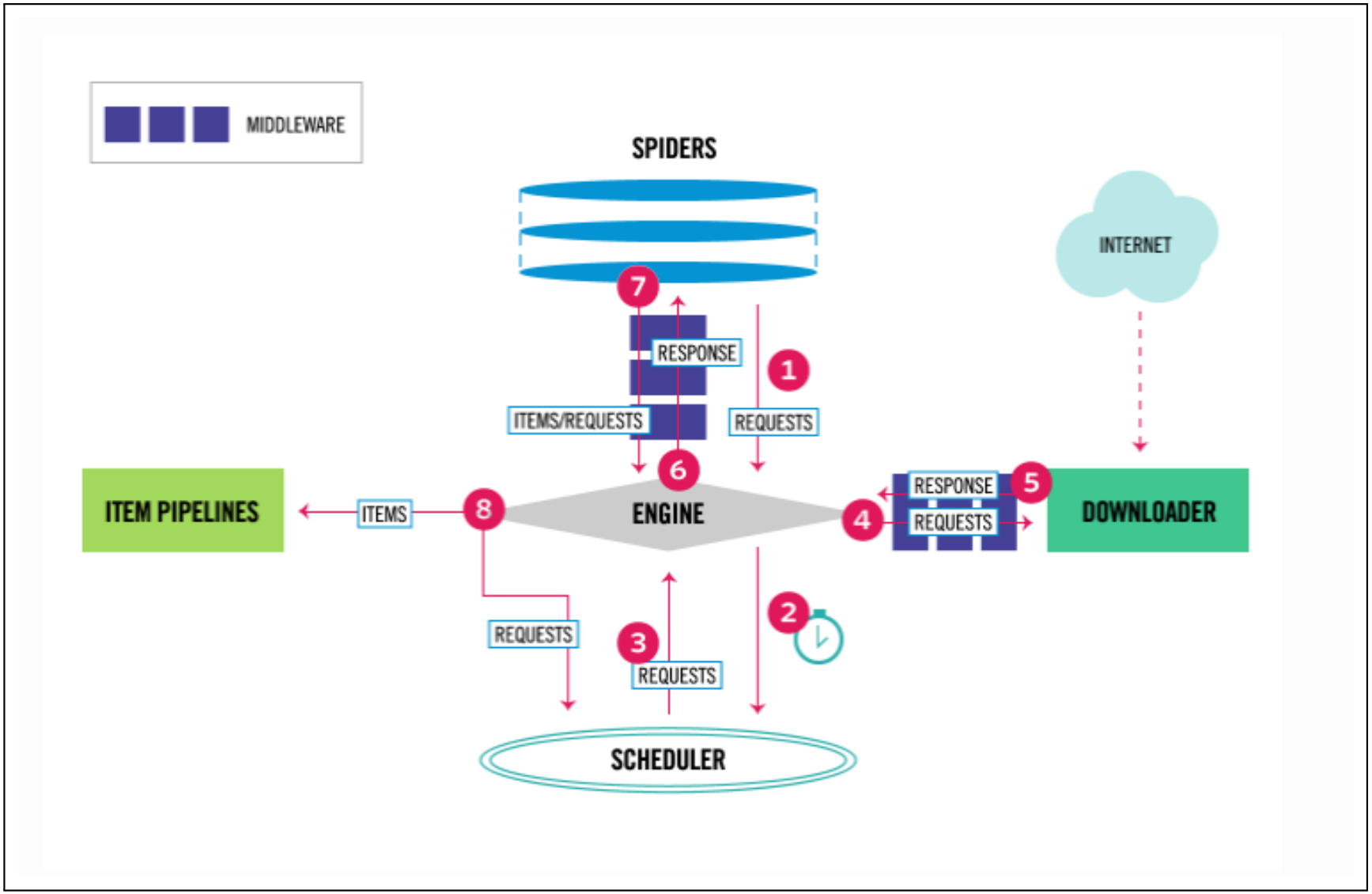
We can invoke it with `scrapy runspider blogspider`.

A First scrapy Project

We can bootstrap a new scrapy project by running the command `scrapy startproject newproject`. We will end up with the following directory structure:

```
$ tree --charset=ascii
.
|-- scrapy.cfg
`-- newproject
    |-- __init__.py
    |-- items.py
    |-- middlewares.py
    |-- pipelines.py
    |-- settings.py
    `-- spiders
        `-- __init__.py
```

Data Flow in scrapy



Items in scrapy

```
from scrapy import Field, Item

class Product(Item):
    name = Field()
    price = Field()
    stock = Field()
    tags = Field()
    last_updated = Field(serializer=str)
```

Item instances are simple containers used to collect scraped data. They also provide a way to specify how data should be serialized.

Pipelines in scrapy, 1/2

```
from scrapy.exceptions import DropItem

class PricePipeline(object):

    vat_factor = 1.15

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

Item Pipelines provide a way of specifying linear workflows that Item instances will go through before being emitted or eventually discarded.

Pipelines in scrapy, 2/2

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

Item Pipelines are configured by modifying the appropriate variable in the `settings.py` file, alongside their priority.

Note however that you typically won't need to define a `JsonWriterPipeline` as in the example, because you can use the predefined Feed Exporter for JSON.

Middlewares in `scrapy`

There are two kinds of Middlewares in `scrapy`: the ones that intercept Request and Responses of Spiders and the ones that do the same for Downloaders.

They are allowed to do pretty much anything, including discarding the Request or generating multiple Responses. Let's see some examples from the documentation.

Once again, they are configured by modifying the `SPIDER_MIDDLEWARES` and `DOWNLOADER_MIDDLEWARES` respectively in the `settings.py` file.

Spiders in `scrapy`

Finally, Spider classes should ideally contain only the logic that deals with extracting the relevant information from the downloaded page into Items, or generating more Requests for other pages.

Observation

It's very easy to fall into the trap of putting all the logic in the Spider class. Instead, HTTP-level logic belongs to Downloader Middlewares, while business logic belongs to Pipelines.

Developing and Debugging Spiders

The `scrapy shell` command is invaluable when developing and debugging. Let's see it in a live example.