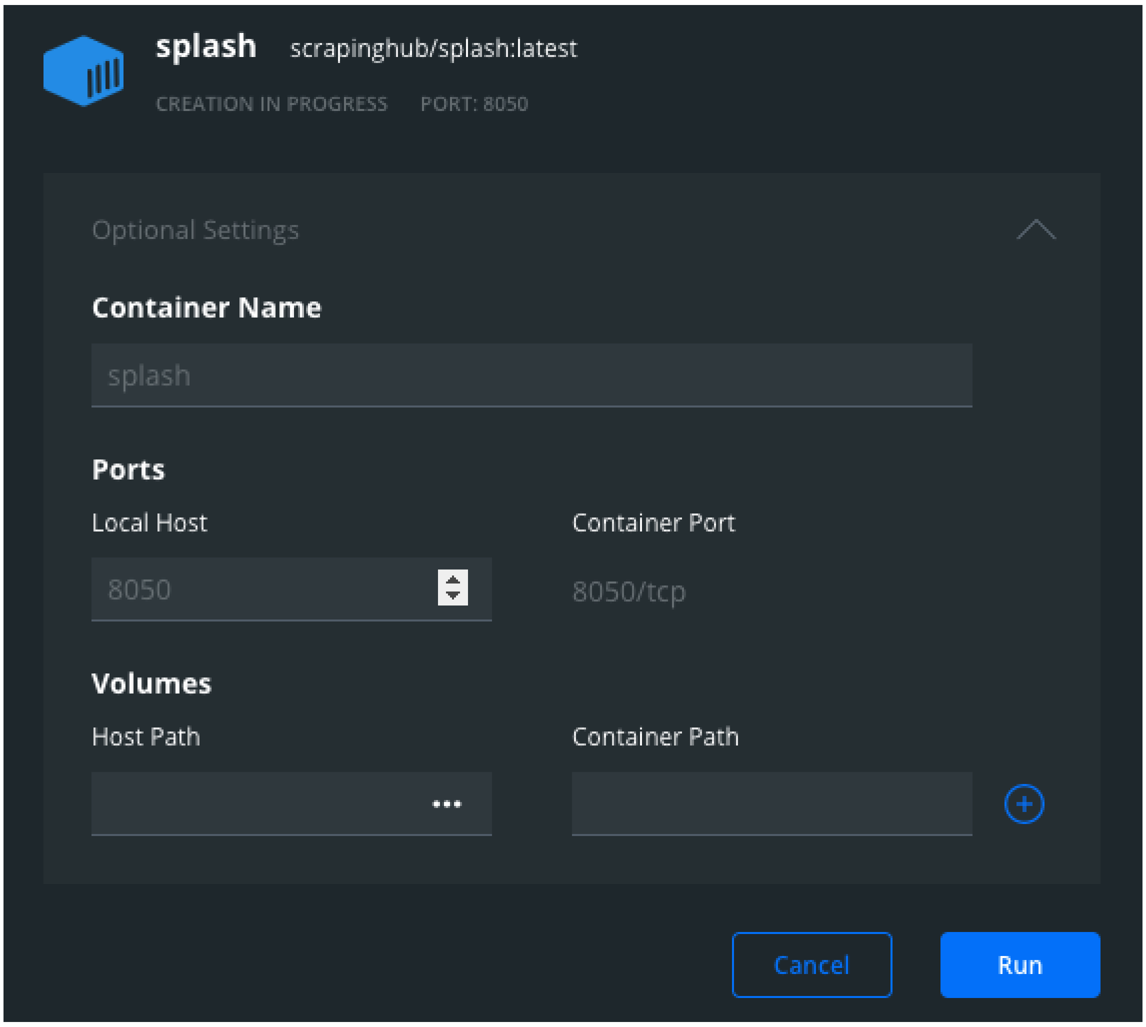
A Guide to Splash

Digital Technologies and Value Creation - Philippe Blaettchen

Introduction and Installation

Splash is a service that renders dynamic websites (essentially, resolving the JavaScript and producing an HTML). Getting Splash to run involves a few steps:

1. Download and install Docker Desktop from https://www.docker.com/get-started
2. Once downloaded, start Docker
3. After you have started Docker the first time, it should now be on your path and accessible from the terminal. Close Docker, then open the terminal and run `docker pull scrapinghub/splash` (this may take a while)
4. Open Docker again and navgate to "Images". Hover over "scrapinghub/splash", where you will see a run button. Click on it, then fill out the the details as in the screenshot:


splash scrapinghub/splash:latest

CREATION IN PROGRESS PORT: 8050

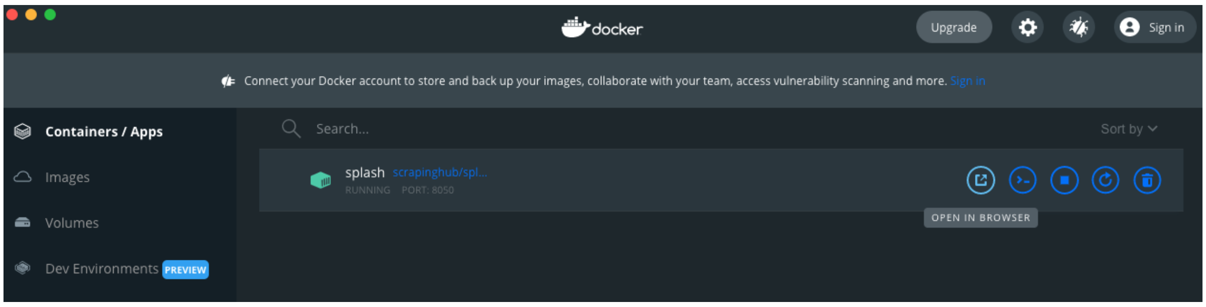
Optional Settings

Container Name: splash

Ports
Local Host: 8050
Container Port: 8050/tcp

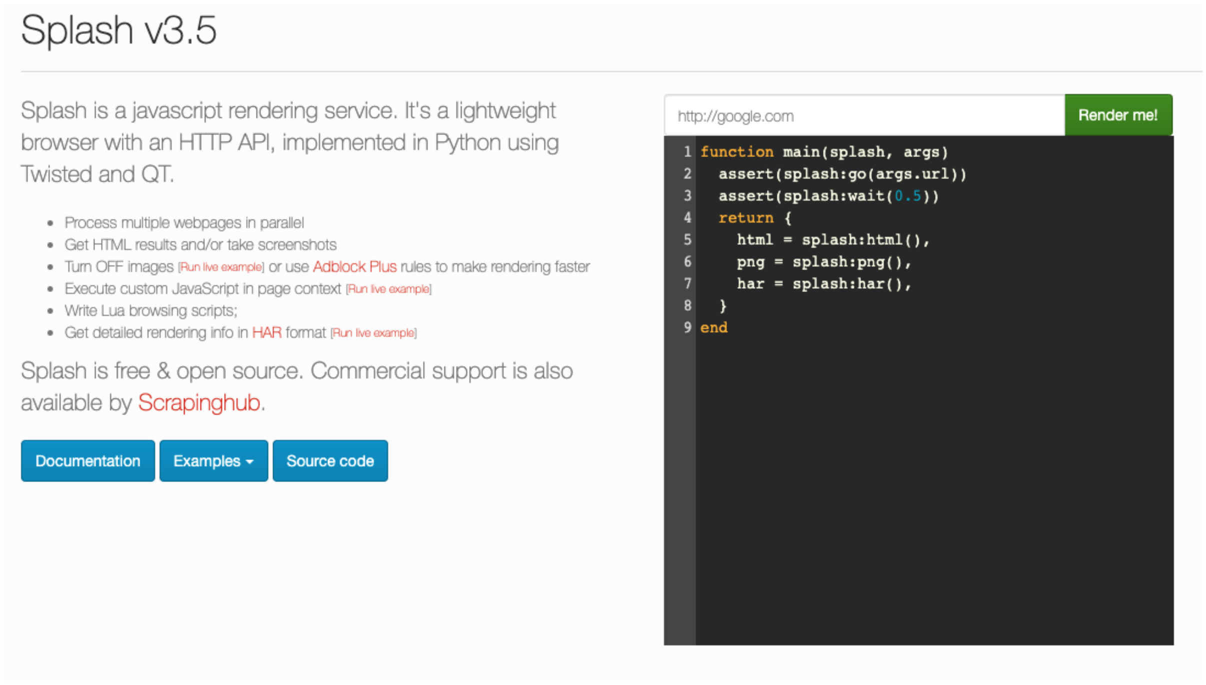
Volumes
Host Path:
Container Path:

1. Click run, then go to "Containers/Apps", where you will see splash running. You can now open it in a browser (instead of clicking, you can also simply navigate to any browser and type in http://localhost:8050):



Connect your Docker account to store and back up your images, collaborate with your team, access vulnerability scanning and more. Sign in

When we open Splash in a Brwoser, we see something like this:



Splash v3.5

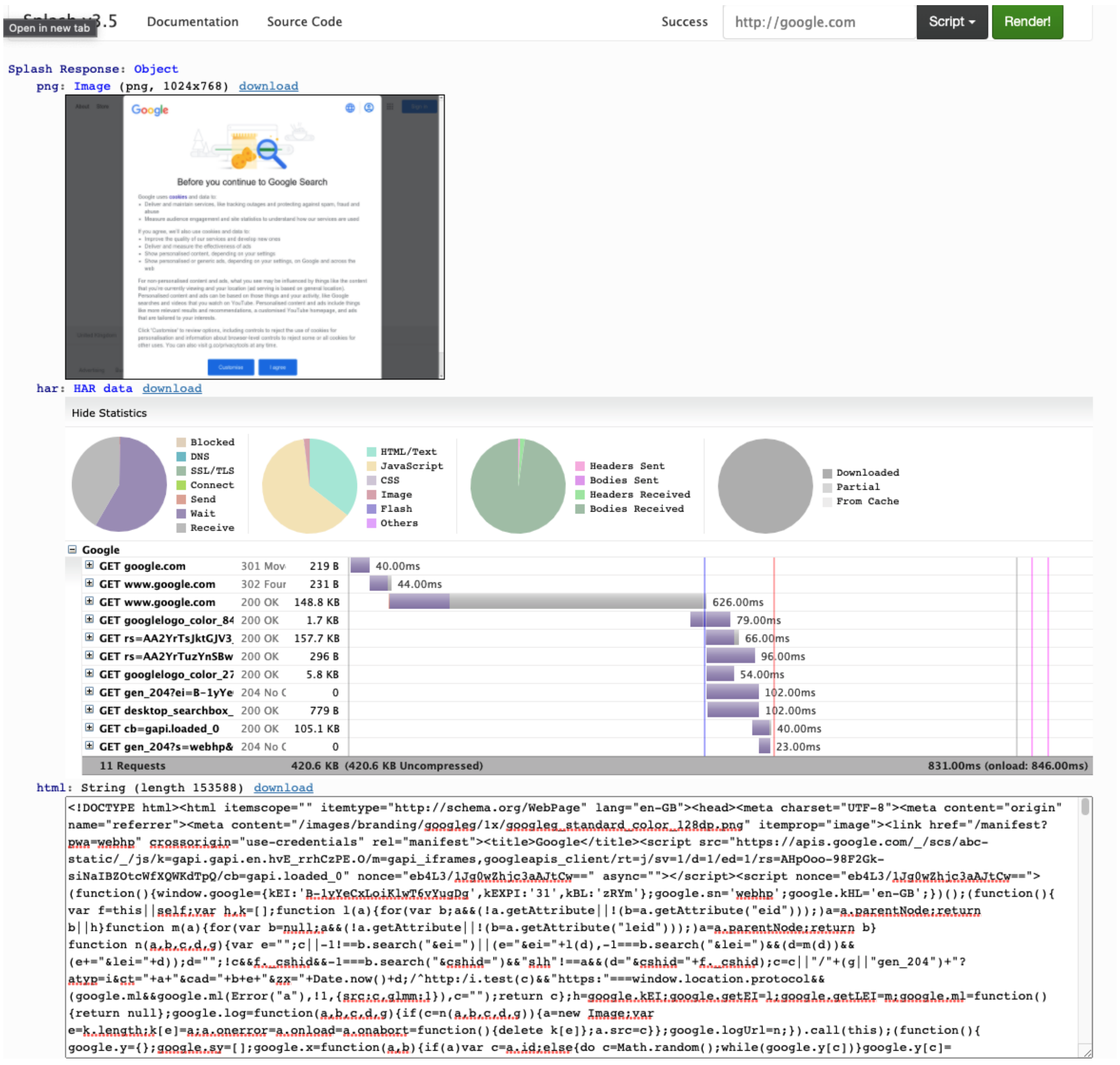
Splash is a javascript rendering service. It's a lightweight browser with an HTTP API, implemented in Python using Twisted and QT.

- Process multiple webpages in parallel
- Get HTML results and/or take screenshots
- Turn OFF images [Run live example] or use Adblock Plus rules to make rendering faster
- Execute custom JavaScript in page context [Run live example]
- Write Lua browsing scripts;
- Get detailed rendering info in HAR format [Run live example]

Splash is free & open source. Commercial support is also available by Scrapinghub.

```lua
function main(splash, args)
  assert(splash:go(args.url))
  assert(splash:wait(0.5))
  return {
    html = splash:html(),
    png = splash:png(),
    har = splash:har(),
  }
end
```

Enter any website and click render. Splash sends a GET request, and renders the dynamic content of the returned website. It then displays an HTML at the bottom of the page:



```
Splash Response: Object
    png: Image (png, 1024x768) download
    har: HAR data download
```

Google			
GET google.com	301 Mov	219 B	40.00ms
GET www.google.com	302 Four	231 B	44.00ms
GET www.google.com	200 OK	148.8 KB	626.00ms
GET googlelogo_color_84	200 OK	1.7 KB	79.00ms
GET rs=AA2YrTsJktGJV3_	200 OK	157.7 KB	66.00ms
GET rs=AA2YrTuzYnSBw	200 OK	296 B	96.00ms
GET googlelogo_color_27	200 OK	5.8 KB	54.00ms
GET gen_204?ei=B-1yYe	204 No C	0	102.00ms
GET desktop_searchbox_	200 OK	779 B	102.00ms
GET cb=gapi.loaded_0	200 OK	105.1 KB	40.00ms
GET gen_204?s=webhp&	204 No C	0	23.00ms
11 Requests		420.6 KB (420.6 KB Uncompressed)	831.00ms (onload: 846.00ms)

```
    html: String (length 153588) download
<!DOCTYPE html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-GB"><head><meta charset="UTF-8"><meta content="origin"
name="referrer"><meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" itemprop="image"><link href="/manifest?
pwa=webhp" crossorigin="use-credentials" rel="manifest"><title>Google</title><script src="https://apis.google.com/_/scs/abc-
static/_/js/k=gapi.gapi.en.hvE_rrhCzPE.O/m=gapi_iframes,googleapis_client/rt=j/sv=1/d=1/ed=1/rs=AHpOoo-98F2Gk-
siNaIBZOtcWfXQWKdTpQ/cb=gapi.loaded_0" nonce="eb4L3/1Jg0wZhjc3aAJtCw==" async=""></script><script nonce="eb4L3/1Jg0wZhjc3aAJtCw==">
(function(){window.google={kEI:'B-1yYeCxLoiKlwT6vYugDg',kEXPI:'31',kBL:'zRYm'};google.sn='webhp';google.kHL='en-GB';})();(function(){
var f=this||self;var h,k=[];function l(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("eid")));)a=a.parentNode;return
b||h}function m(a){for(var b=null;a&&(!a.getAttribute||!(b=a.getAttribute("leid")));)a=a.parentNode;return b}
function n(a,b,c,d,g){var e="";c||-1!==b.search("&ei=")||(e="&ei="+l(d),-1===b.search("&lei=")&&(d=m(d))&&
(e+="&lei="+d));d="";!c&&f._cshid&&-1===b.search("&cshid=")&&"slh"!==a&&(d="&cshid="+f._cshid);c=c||"/"+(g||"gen_204")+"?
atyp=i&ct="+a+"&cad="+b+e+"&zx="+Date.now()+d;/^http:/i.test(c)&&"https:"===window.location.protocol&&
(google.ml&&google.ml(Error("a"),!1,{src:c,glmm:1}),c="");return c};h=google.kEI;google.getEI=l;google.getLEI=m;google.ml=function()
{return null};google.log=function(a,b,c,d,g){if(c=n(a,b,c,d,g)){a=new Image;var
e=k.length;k[e]=a;a.onerror=a.onload=a.onabort=function(){delete k[e]};a.src=c}};google.logUrl=n;}).call(this);(function(){
google.y={};google.sy=[];google.x=function(a,b){if(a)var c=a.id;else{do c=Math.random();while(google.y[c])}google.y[c]=
```

Let's take a step back. Doesn't our browser also render the dynamic content? So why use Splash?

The point is that we don't want to use Splash like this, manually rendering websites. Instead, we want to integrate Splash into our Scrapy projects (so when we come across a website with dynamic content, Splash renders it and returns the HTML to our Scrapy spider)! For that, we need to also use a corresponding Python package:

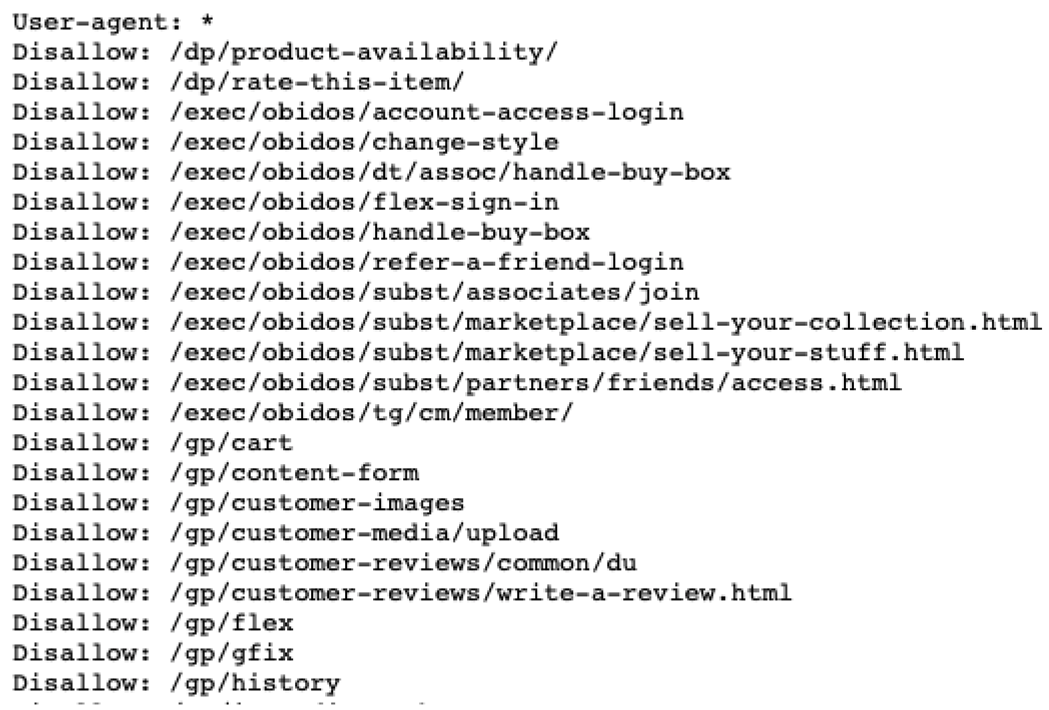
```
pip install scrapy-splash .
```

A look at a dynamic website

As a use case, we will scrape "Trending Deals" from www.amazon.co.uk. If you go to the website, you can scroll down to find the Trending Deals section. There is a button "See all Deals". At the time of writing, it leads to the following url:
https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&pf_rd_p=346c6136-7bc4-49af-8040-6ff24d6ad1c7&pf_rd_r=RMPYZDTBV62JG8PZJDGP&pd_rd_r=7c0634c5-d977-4970-b817-e858b72e99c5&pd_rd_wg=HTY0f&ref_=pd_gw_unk

Note that this url might change (but we could later also adopt our spider to find the correct url...).

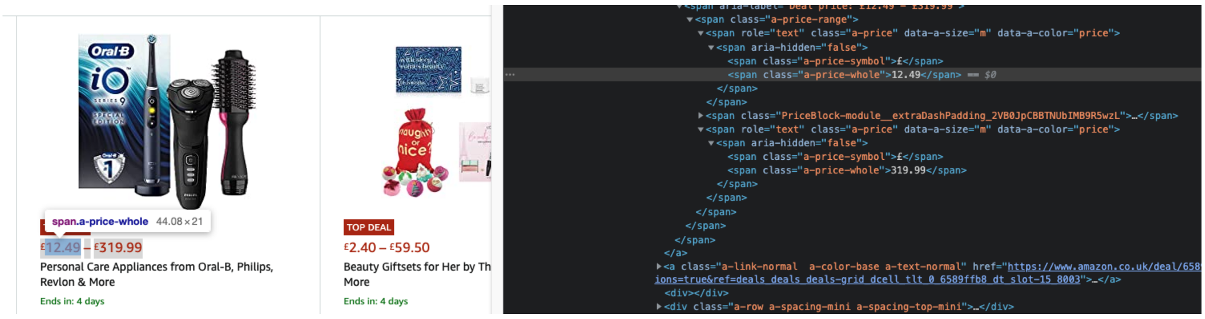
Before starting any scraping, let's first check whether this is okay on Amazon's side. We go to www.amazon.co.uk/robots.txt. We see that many subpages of "/gp/" are disallowed, but the subpage "/gp/deals/" is not!



```
User-agent: *
Disallow: /dp/product-availability/
Disallow: /dp/rate-this-item/
Disallow: /exec/obidos/account-access-login
Disallow: /exec/obidos/change-style
Disallow: /exec/obidos/dt/assoc/handle-buy-box
Disallow: /exec/obidos/flex-sign-in
Disallow: /exec/obidos/handle-buy-box
Disallow: /exec/obidos/refer-a-friend-login
Disallow: /exec/obidos/subst/associates/join
Disallow: /exec/obidos/subst/marketplace/sell-your-collection.html
Disallow: /exec/obidos/subst/marketplace/sell-your-stuff.html
Disallow: /exec/obidos/subst/partners/friends/access.html
Disallow: /exec/obidos/tg/cm/member/
Disallow: /gp/cart
Disallow: /gp/content-form
Disallow: /gp/customer-images
Disallow: /gp/customer-media/upload
Disallow: /gp/customer-reviews/common/du
Disallow: /gp/customer-reviews/write-a-review.html
Disallow: /gp/flex
Disallow: /gp/gfix
Disallow: /gp/history
```

Going by the assumption that not disallowed means allowed, we proceed with our scraping project.

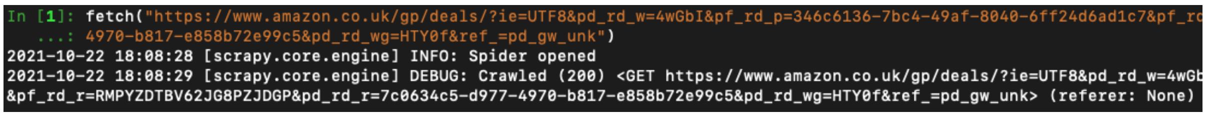
On the Trending Deals site, we see a bunch of items, each with a price range and a time at which the deal ends. Inspection can help us here. For example, the lower end and the upper end of the price range are saved in span-tags of the class "a-price-whole":



span.a-price-whole 44.08 × 21
£12.49 – £319.99
Personal Care Appliances from Oral-B, Philips, Revlon & More
Ends in: 4 days
TOP DEAL
£2.40 – £59.50
Beauty Giftsets for Her by Th
More
Ends in: 4 days

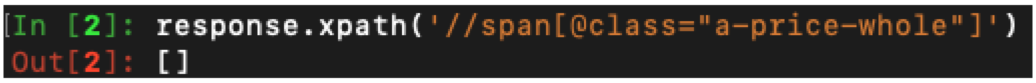
```
▼<span class="a-price-range">
  ▼<span role="text" class="a-price" data-a-size="m" data-a-color="price">
    ▼<span aria-hidden="false">
        <span class="a-price-symbol">£</span>
        <span class="a-price-whole">12.49</span> == $0
      </span>
    </span>
  ▶<span class="PriceBlock-module__extraDashPadding_2VB0JpCBBTNUbIMB9R5wzL">…</span>
  ▼<span role="text" class="a-price" data-a-size="m" data-a-color="price">
    ▼<span aria-hidden="false">
        <span class="a-price-symbol">£</span>
        <span class="a-price-whole">319.99</span>
      </span>
    </span>
  </span>
 </span>
</span>
</a>
▶<a class="a-link-normal  a-color-base a-text-normal" href="https://www.amazon.co.uk/deal/658
ions=true&ref=deals_deals_deals-grid_dcell_tlt_0_6589ffb8_dt_slot-15_8003">…</a>
 <div></div>
▶<div class="a-row a-spacing-mini a-spacing-top-mini">…</div>
```

We know how to read out those tags! So let's kick up our Scrapy shell, and take a look. Remember, we use `scrapy shell` to open the shell from the terminal. We can then fetch the Trending Deals website:



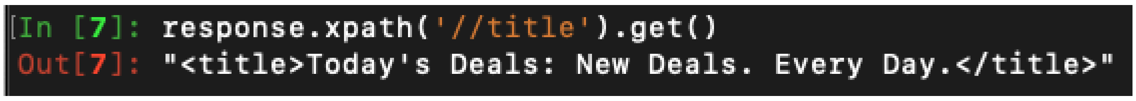
```
In [1]: fetch("https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&pf_rd_p=346c6136-7bc4-49af-8040-6ff24d6ad1c7&pf_rd
   ...: 4970-b817-e858b72e99c5&pd_rd_wg=HTY0f&ref_=pd_gw_unk")
2021-10-22 18:08:28 [scrapy.core.engine] INFO: Spider opened
2021-10-22 18:08:29 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGb
&pf_rd_r=RMPYZDTBV62JG8PZJDGP&pd_rd_r=7c0634c5-d977-4970-b817-e858b72e99c5&pd_rd_wg=HTY0f&ref_=pd_gw_unk> (referer: None)
```

So far, everything looks fine, the website was successfully returned. Let's try to find ourselves some price tags. Recall the XPath syntax: `'//span[@class="a-price-whole"]'` means any span-tag of the class "a-price-whole". However, when we run this we get nothing:



```
In [2]: response.xpath('//span[@class="a-price-whole"]')
Out[2]: []
```

Let's check that we actually got the right side:

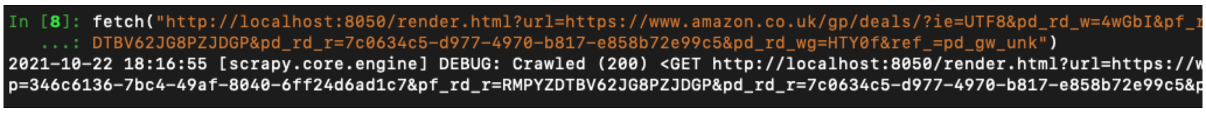


```
In [7]: response.xpath('//title').get()
Out[7]: "<title>Today's Deals: New Deals. Every Day.</title>"
```

There is a title tag which contains the text "Today's Deals: New Deals. Every Day." Sounds promising. So where are the items?

The problem is that the return from Amazon's server does not specify a full HTML document with all items arranged. Why? Because the items shown will vary based on when you are calling up the site, who you are (cookies, IP, device, etc.), and where you are calling the site from. Instead, the return will contain a bunch of JavaScript code that, once executed, gives us a finished HTML.

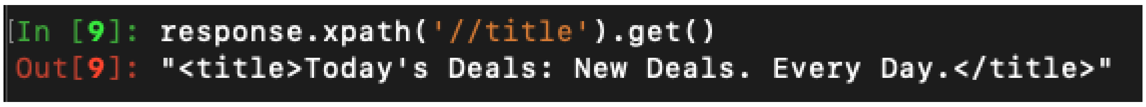
Our challenge now is to first render the dynamic content (with Splash). Luckily, we can do this directly in the shell:



```
In [8]: fetch("http://localhost:8050/render.html?url=https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&pf_r
   ...: DTBV62JG8PZJDGP&pd_rd_r=7c0634c5-d977-4970-b817-e858b72e99c5&pd_rd_wg=HTY0f&ref_=pd_gw_unk")
2021-10-22 18:16:55 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://localhost:8050/render.html?url=https://w
p=346c6136-7bc4-49af-8040-6ff24d6ad1c7&pf_rd_r=RMPYZDTBV62JG8PZJDGP&pd_rd_r=7c0634c5-d977-4970-b817-e858b72e99c5&p
```

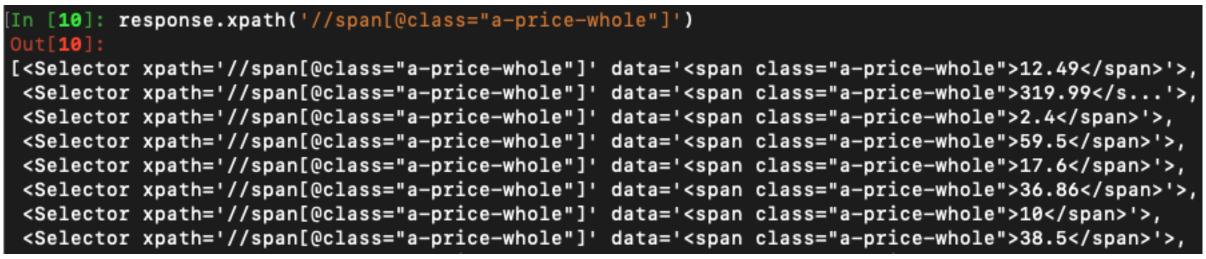
Note here that we are not going directly for the website. Instead, we target our local port 8050 (this is where Splash is sitting at), and use the rendering service provided by Splash. We specify which website Splash should render by giving a url attribute: http://localhost:8050/render.html?url=[Actual_Website_Address]

We see a return value of 200, which is a good sign. We can again check the title to see we are at the right page:



```
In [9]: response.xpath('//title').get()
Out[9]: "<title>Today's Deals: New Deals. Every Day.</title>"
```

Still good. Now, let's try again to find a price tag:

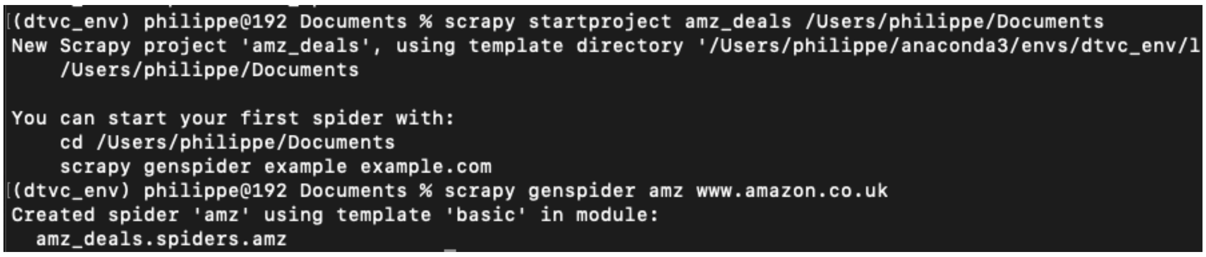


```
In [10]: response.xpath('//span[@class="a-price-whole"]')
Out[10]:
[<Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">12.49</span>'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">319.99</s...'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">2.4</span>'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">59.5</span>'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">17.6</span>'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">36.86</span>'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">10</span>'>,
 <Selector xpath='//span[@class="a-price-whole"]' data='<span class="a-price-whole">38.5</span>'>,
```

This time, there are a lot of price tags! It seems like Splash effectively rendered the website.

Using Splash within a spider

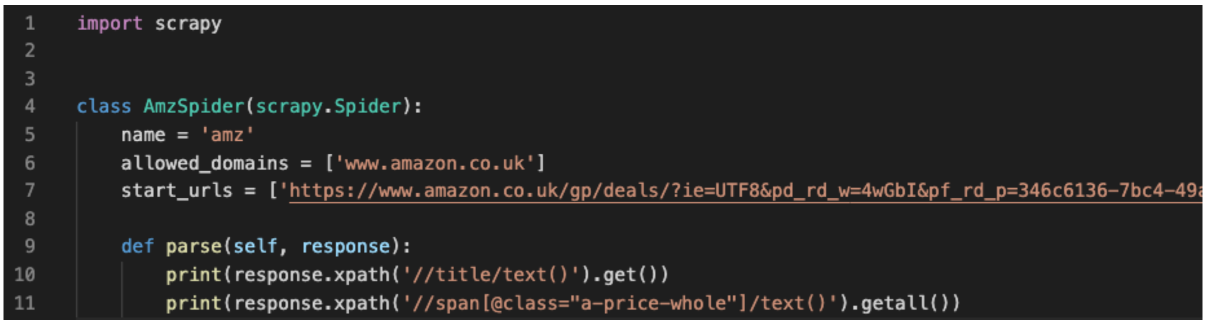
We don't just want to use Splash inside the shell, but instead have spiders that make use of Splash. This is where `scrapy-splash` comes in. Let's exit the shell and start a new project (using `scrapy startproject [NAME] [PATH]`), as well as a basic Spider within the project (cd into the [PATH], then run `scrapy genspider [SPIDERNAME] [DOMAIN]`). This could look as follows:



```
(dtvc_env) philippe@192 Documents % scrapy startproject amz_deals /Users/philippe/Documents
New Scrapy project 'amz_deals', using template directory '/Users/philippe/anaconda3/envs/dtvc_env/l
    /Users/philippe/Documents

You can start your first spider with:
    cd /Users/philippe/Documents
    scrapy genspider example example.com
(dtvc_env) philippe@192 Documents % scrapy genspider amz www.amazon.co.uk
Created spider 'amz' using template 'basic' in module:
  amz_deals.spiders.amz
```

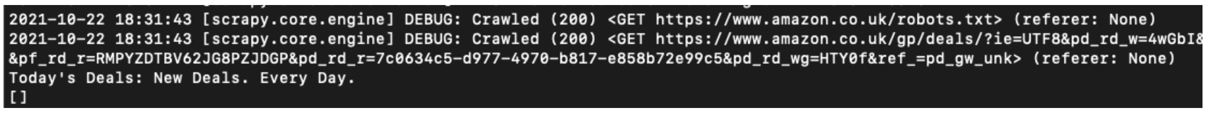
We open the newly created spider and make a few adjustments:

- We change the start-url to the Tranding Deals site
- We print out the title-tag and all the price span-tags (or, more specifically, the text within them).



```python
import scrapy


class AmzSpider(scrapy.Spider):
    name = 'amz'
    allowed_domains = ['www.amazon.co.uk']
    start_urls = ['https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&pf_rd_p=346c6136-7bc4-49a

    def parse(self, response):
        print(response.xpath('//title/text()').get())
        print(response.xpath('//span[@class="a-price-whole"]/text()').getall())
```

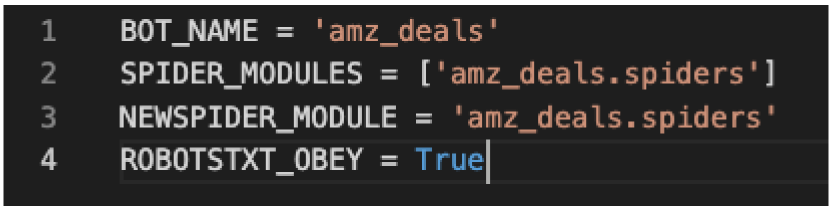
We can save the spider and then run it with `scrapy crawl amz`. It runs perfectly fine:



```
2021-10-22 18:31:43 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.amazon.co.uk/robots.txt> (referer: None)
2021-10-22 18:31:43 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&
&pf_rd_r=RMPYZDTBV62JG8PZJDGP&pd_rd_r=7c0634c5-d977-4970-b817-e858b72e99c5&pd_rd_wg=HTY0f&ref_=pd_gw_unk> (referer: None)
Today's Deals: New Deals. Every Day.
[]
```

However, as we should have known, there were no price tags to print - because the returned site is not yet rendered!

This is where we bring in Splash. Before we turn back to our spider, we have to actually make a few adjustments to the settings of the Scrapy project. In particular, we go to the main project folder and find "settings.py". You'll see a few settings here (some are active, most are commented out). If we delete all the settings that were commented out, this is what remains:



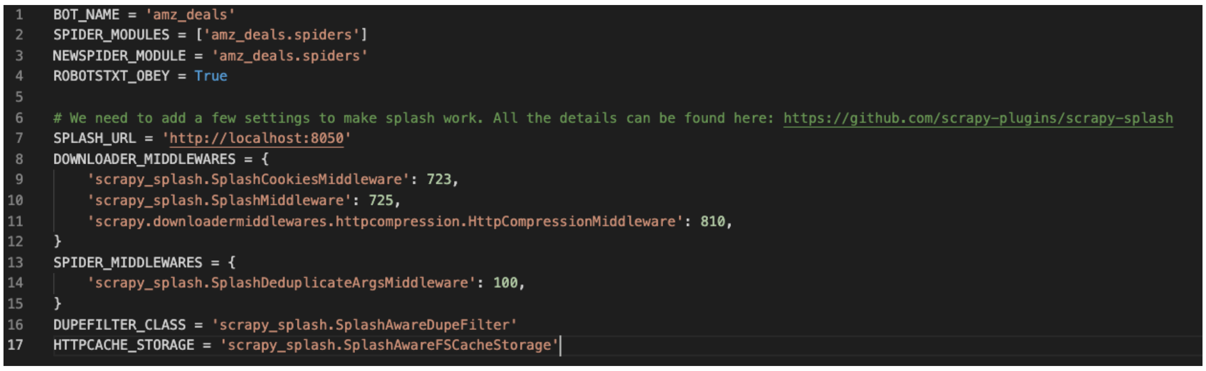
```python
BOT_NAME = 'amz_deals'
SPIDER_MODULES = ['amz_deals.spiders']
NEWSPIDER_MODULE = 'amz_deals.spiders'
ROBOTSTXT_OBEY = True
```

To get Splash to run, we have to add a few lines here. Luckily, we can simply find the lines required by checking the Github-page of scrapy-splash: https://github.com/scrapy-plugins/scrapy-splash

We can find that we need to add the following lines:

```
SPLASH_URL = 'http://localhost:8050'
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddlewar
 810,
}
SPIDER_MIDDLEWARES = {
    'scrapy_splash.SplashDeduplicateArgsMiddleware': 100,
}
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

Note that the Github mentions `SPLASH_URL = 'http://192.168.59.103:8050'` , but this depends on your IP. Just use localhost to make things easier. Our settings file should now look like this:



```
BOT_NAME = 'amz_deals'
SPIDER_MODULES = ['amz_deals.spiders']
NEWSPIDER_MODULE = 'amz_deals.spiders'
ROBOTSTXT_OBEY = True

# We need to add a few settings to make splash work. All the details can be found here: https://github.com/scrapy-plugins/scrapy-splash
SPLASH_URL = 'http://localhost:8050'
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 810,
}
SPIDER_MIDDLEWARES = {
    'scrapy_splash.SplashDeduplicateArgsMiddleware': 100,
}
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

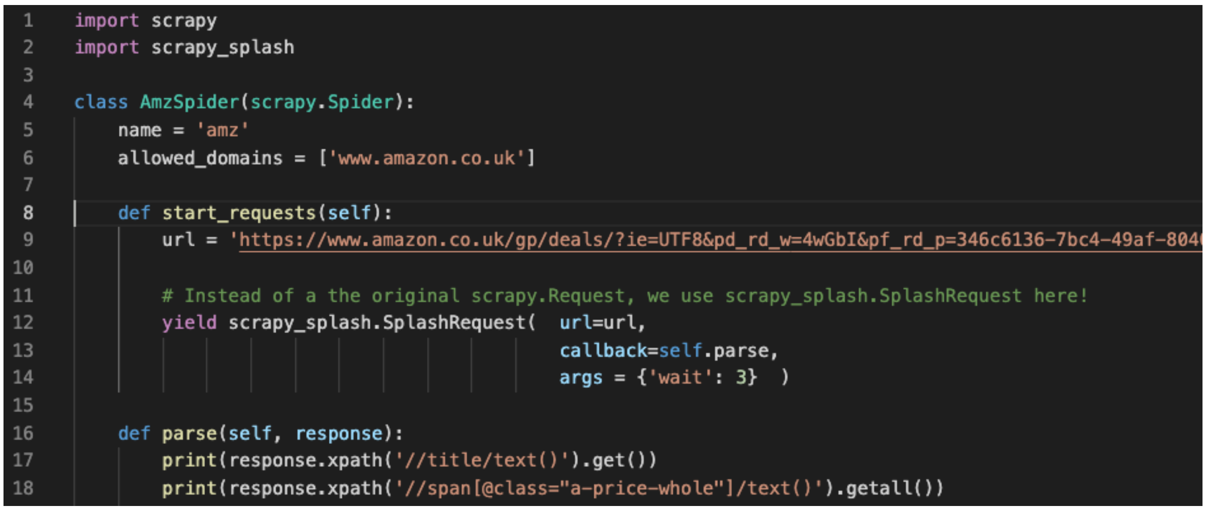
Make sure to save this, then you can close it.

It's time to turn back to our spider. The first thing we need to do is to import `scrapy-splash` (note that when loading the package, it's written `scrapy_splash` due to Python syntax).

The second part we need to change is a bit trickier. Remember how we had a list `start_urls` of websites that are scraped right away when the spider starts? The problem is, this is just a shorthand for issuing a (normal) `scrapy.Request` . But we want to make sure to issue a `scrapy_splash.SplashRequest` instead.

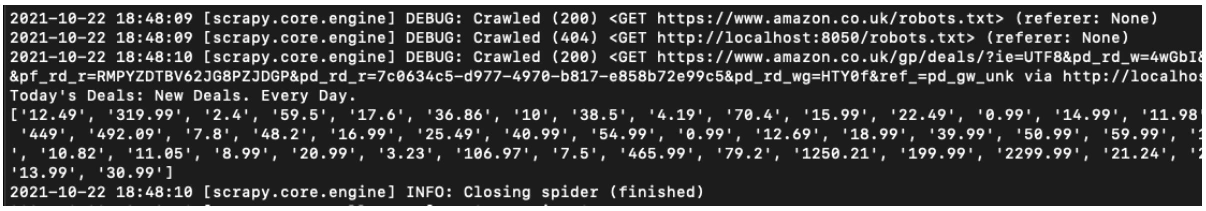
Hence, instead of defining `start_urls` , we define the function `start_requests(self)` , which allows us to manually adjust how our spider starts out. When making the `SplashRequest` , we also ensure to give enough of a wait time for items to actually load. The default value here is 0.5 seconds but this can be too little.

The spider should now look like this:



```python
import scrapy
import scrapy_splash

class AmzSpider(scrapy.Spider):
    name = 'amz'
    allowed_domains = ['www.amazon.co.uk']

    def start_requests(self):
        url = 'https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&pf_rd_p=346c6136-7bc4-49af-804

        # Instead of a the original scrapy.Request, we use scrapy_splash.SplashRequest here!
        yield scrapy_splash.SplashRequest(  url=url,
                                            callback=self.parse,
                                            args = {'wait': 3}  )

    def parse(self, response):
        print(response.xpath('//title/text()').get())
        print(response.xpath('//span[@class="a-price-whole"]/text()').getall())
```

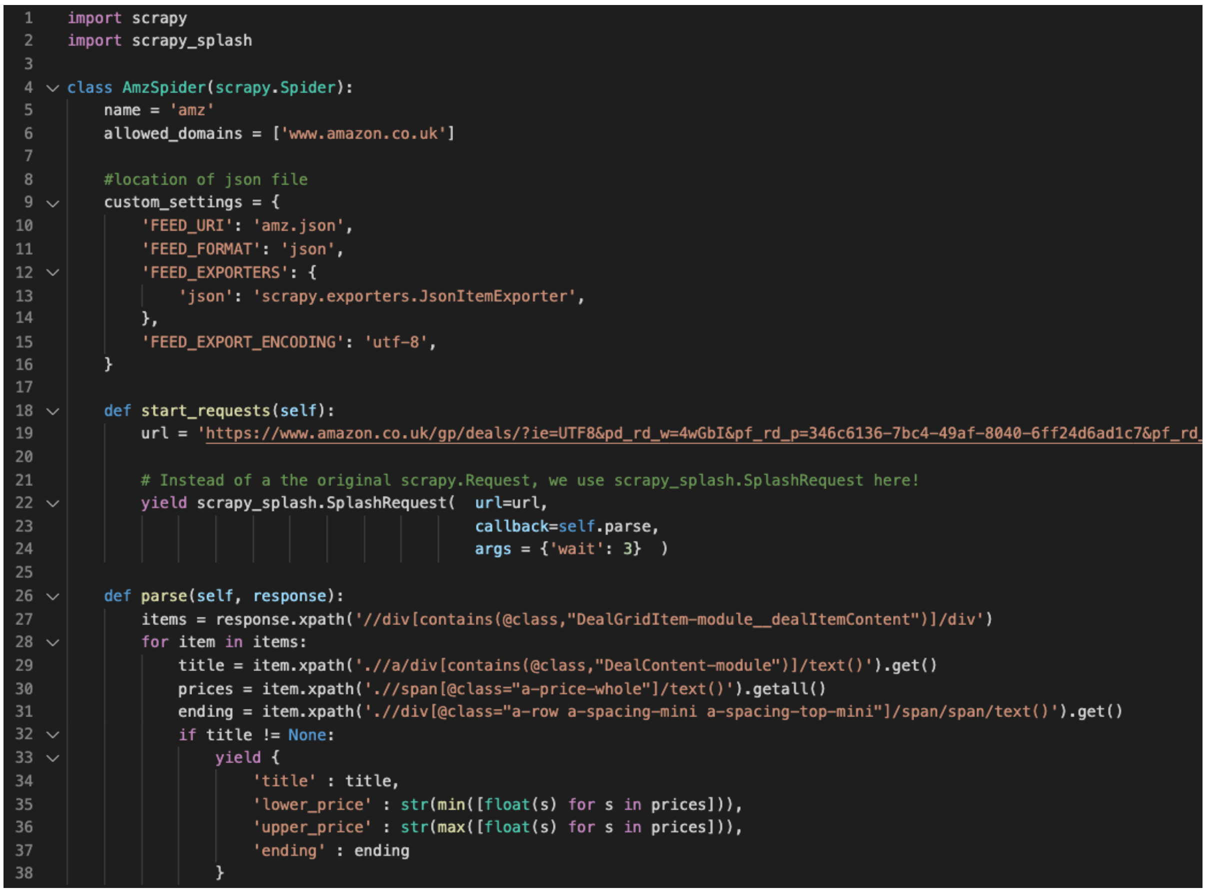
Let's try to run our spider again:



```
2021-10-22 18:48:09 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.amazon.co.uk/robots.txt> (referer: None)
2021-10-22 18:48:09 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://localhost:8050/robots.txt> (referer: None)
2021-10-22 18:48:10 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&
&pf_rd_r=RMPYZDTBV62JG8PZJDGP&pd_rd_r=7c0634c5-d977-4970-b817-e858b72e99c5&pd_rd_wg=HTY0f&ref_=pd_gw_unk via http://localhos
Today's Deals: New Deals. Every Day.
['12.49', '319.99', '2.4', '59.5', '17.6', '36.86', '10', '38.5', '4.19', '70.4', '15.99', '22.49', '0.99', '14.99', '11.98'
 '449', '492.09', '7.8', '48.2', '16.99', '25.49', '40.99', '54.99', '0.99', '12.69', '18.99', '39.99', '50.99', '59.99', '
', '10.82', '11.05', '8.99', '20.99', '3.23', '106.97', '7.5', '465.99', '79.2', '1250.21', '199.99', '2299.99', '21.24', '
'13.99', '30.99']
2021-10-22 18:48:10 [scrapy.core.engine] INFO: Closing spider (finished)
```

We've successfuly crawled a website with dynamic content!

Once we've applauded ourselves sufficiently, we can turn to making this spider a bit more useful. For example, we may come up with something like this:



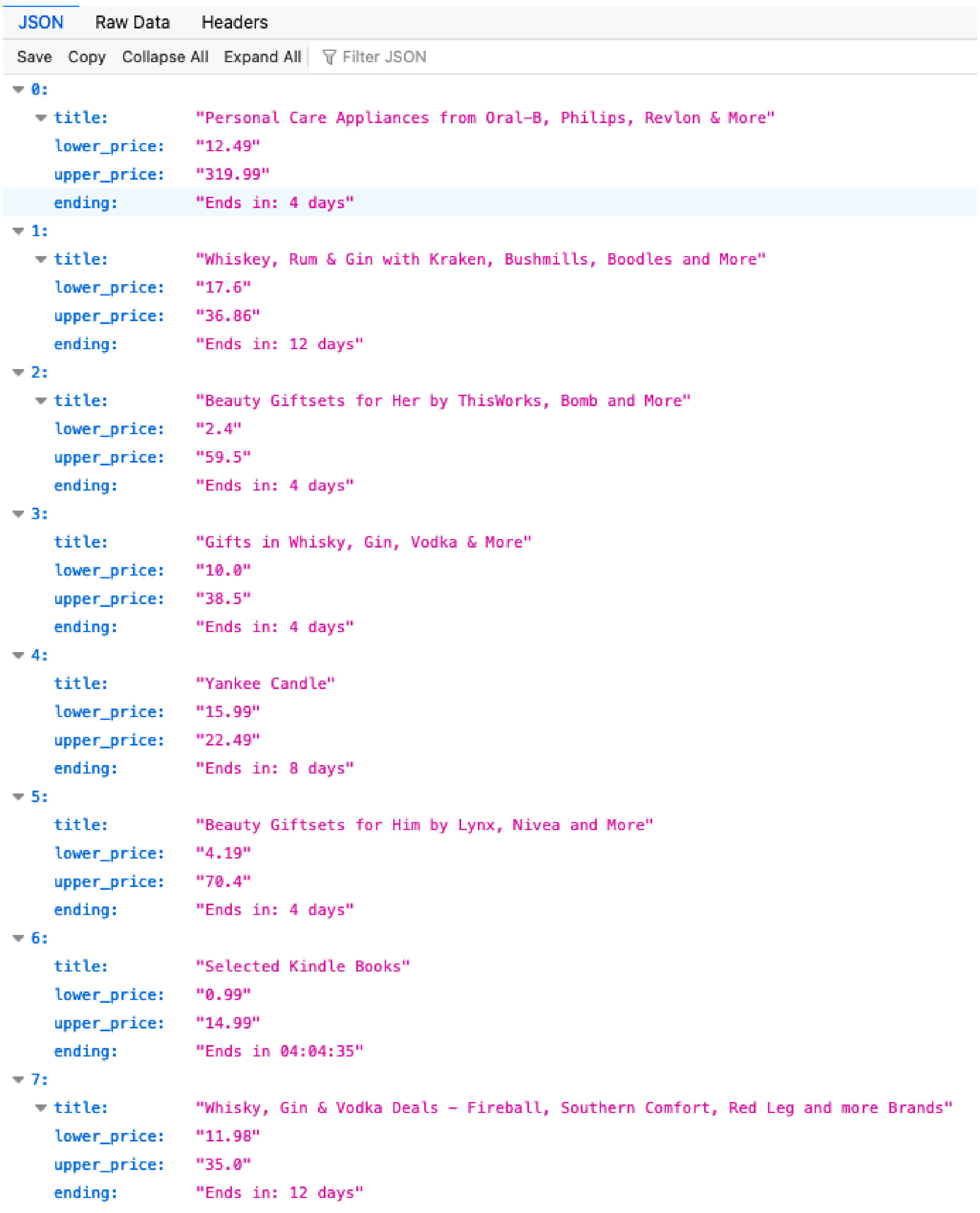
```python
import scrapy
import scrapy_splash

class AmzSpider(scrapy.Spider):
    name = 'amz'
    allowed_domains = ['www.amazon.co.uk']

    #location of json file
    custom_settings = {
        'FEED_URI': 'amz.json',
        'FEED_FORMAT': 'json',
        'FEED_EXPORTERS': {
            'json': 'scrapy.exporters.JsonItemExporter',
        },
        'FEED_EXPORT_ENCODING': 'utf-8',
    }

    def start_requests(self):
        url = 'https://www.amazon.co.uk/gp/deals/?ie=UTF8&pd_rd_w=4wGbI&pf_rd_p=346c6136-7bc4-49af-8040-6ff24d6ad1c7&pf_rd

        # Instead of a the original scrapy.Request, we use scrapy_splash.SplashRequest here!
        yield scrapy_splash.SplashRequest(  url=url,
                                            callback=self.parse,
                                            args = {'wait': 3}  )

    def parse(self, response):
        items = response.xpath('//div[contains(@class,"DealGridItem-module__dealItemContent")]/div')
        for item in items:
            title = item.xpath('.//a/div[contains(@class,"DealContent-module")]/text()').get()
            prices = item.xpath('.//span[@class="a-price-whole"]/text()').getall()
            ending = item.xpath('.//div[@class="a-row a-spacing-mini a-spacing-top-mini"]/span/span/text()').get()
            if title != None:
                yield {
                    'title' : title,
                    'lower_price' : str(min([float(s) for s in prices])),
                    'upper_price' : str(max([float(s) for s in prices])),
                    'ending' : ending
                }
```

What is happening here?

- We added an output stream in the form of a json-file (we've seen this before!)

- We adapted the parse function:
  - Now, we are looking for all items on sale (using the fact that they come within a div-tag with a class "DealGridItem-module__DealItemContent..." - we use `contains` not to have to worry about the rest)
  - Within each of the items, we find the name (div-tag within an a-tag, with class "DealContent-module..."), the range of prices (within span-tags with the class "a-price-whole"), and the ending information (within a div-tag of the class "a-row a-spacing-mini a-spacing-top-mini", and here in the text of a second-level span-tag)
  - We take this information (if it exists), and `yield` it to the higher level (where it gets put into the json file)

The resulting json file looks something like this:


```
JSON    Raw Data    Headers
Save  Copy  Collapse All  Expand All  Filter JSON
▼ 0:
  ▼ title:          "Personal Care Appliances from Oral-B, Philips, Revlon & More"
    lower_price:    "12.49"
    upper_price:    "319.99"
    ending:         "Ends in: 4 days"
▼ 1:
  ▼ title:          "Whiskey, Rum & Gin with Kraken, Bushmills, Boodles and More"
    lower_price:    "17.6"
    upper_price:    "36.86"
    ending:         "Ends in: 12 days"
▼ 2:
  ▼ title:          "Beauty Giftsets for Her by ThisWorks, Bomb and More"
    lower_price:    "2.4"
    upper_price:    "59.5"
    ending:         "Ends in: 4 days"
▼ 3:
    title:          "Gifts in Whisky, Gin, Vodka & More"
    lower_price:    "10.0"
    upper_price:    "38.5"
    ending:         "Ends in: 4 days"
▼ 4:
    title:          "Yankee Candle"
    lower_price:    "15.99"
    upper_price:    "22.49"
    ending:         "Ends in: 8 days"
▼ 5:
    title:          "Beauty Giftsets for Him by Lynx, Nivea and More"
    lower_price:    "4.19"
    upper_price:    "70.4"
    ending:         "Ends in: 4 days"
▼ 6:
    title:          "Selected Kindle Books"
    lower_price:    "0.99"
    upper_price:    "14.99"
    ending:         "Ends in 04:04:35"
▼ 7:
  ▼ title:          "Whisky, Gin & Vodka Deals - Fireball, Southern Comfort, Red Leg and more Brands"
    lower_price:    "11.98"
    upper_price:    "35.0"
    ending:         "Ends in: 12 days"
```

A few final words of warning: if you go through the JSON file, you will see that there are 30 items. However, on the Amazon webpage, there are 60. Careful analysis in the Scrapy shell reveals that the latter 30 items are only loaded as a skeleton (a basic tag is there, but no actual content).

Often, this is the case because items will only be loaded once you scroll down (this is something you can actually do with Splash, but it requies getting into more detail with "Lua scripts"). However, I experimented with the Amazon site, and scrolling does not solve the issue (nor does a longer loading time).

Which leaves the other option: that Amazon recognizes bots and intentionally withholds some of the website information, even when scraping is not disallowed. There are of course ways to circumvent this recognition, but this would be clearly against the intents of the service provider - so not something we will deal with here.

When you're done, don't forget to stop running Splash within Docker