Advanced Web Scraping: Bypassing "403 Forbidden," captchas, and more

Thu, Mar 16, 2017

ine.com%2fpost%2fadvanced-web-scraping-
%22403%20Forbidden%2c%22%20captchas%2c%20and%20more)

:om%2fpost%2fadvanced-web-scraping-tutorial%2f)

20Forbidden%2c%22%20captchas%2c%20and%20more&url=http%3a%2f%2fsangaline.com%2fpost%2fadvanced-

%2fsangaline.com%2fpost%2fadvanced-web-scraping-
%20%22403%20Forbidden%2c%22%20captchas%2c%20and%20more)

3%20Forbidden%2c%22%20captchas%2c%20and%20more&body=http%3a%2f%2fsangaline.com%2fpost%2fadvanced-

The full code for the completed scraper can be found in the companion repository on github (https://github.com/sangaline/advanced-web-scraping-tutorial).

Introduction

I wouldn't really consider web scraping one of my hobbies or anything but I guess I sort of do a lot of it. It just seems like many of the things that I work on require me to get my hands on data that isn't available any other way. I need to do static analysis of games for Intoli (http://intoli.com) and so I scrape the Google Play Store to find new ones and download the apks. The Pointy Ball (http://pointyball.com) extension requires aggregating fantasy football projections from various sites and the easiest way was to write a scraper. When I think about it, I've probably written about 40-50 scrapers. I'm not quite at the point where I'm lying to my family about how many terabytes of data I'm hoarding away... but I'm close.

I've tried out x-ray (https://github.com/lapwinglabs/x-ray)/cheerio (https://github.com/cheeriojs/cheerio), nokogiri (https://github.com/sparklemotion/nokogiri), and a few others but I always come back to my personal favorite: scrapy (https://github.com/scrapy/scrapy). In my opinion, scrapy is an **excellent** piece of software. I don't throw such unequivocal praise around lightly but it feels incredibly intuitive and has a great learning curve.

You can read The Scrapy Tutorial (https://doc.scrapy.org/en/latest/intro/tutorial.html) and have your first scraper running within minutes. Then, when you need to do something more complicated, you'll most likely find that there's a built in and well documented way to do it. There's *a lot of power* built in but the framework is structured so that it stays out of your way until you need it. When you finally do need something that isn't there by default, say a Bloom filter for deduplication because you're visiting too many URLs to store in memory, then it's usually as simple as subclassing one of the components and making a few small changes. Everything just feels so *easy* and that's really a hallmark of good software design in my book.

I've toyed with the idea of writing an advanced scrapy tutorial for a while now. Something that would give me a chance to show off some of its extensibility while also addressing realistic challenges that come up in practice. As much as I've wanted to do this, I just wasn't able to get past the fact that it seemed like a decidely dick move to publish something that could conceivably result in someone's servers getting hammered with bot traffic.

I can sleep pretty well at night scraping sites that actively try to prevent scraping as long as I follow a few basic rules. Namely, I keep my request rate comparable to what it would be if I were browsing by hand and I don't do anything distasteful with the data. That makes running a scraper basically indistinguishable from collecting data manually in any ways that matter. Even if I were to personally follow these rules, it would still feel like a step too far to do a how-to guide for a specific site that people might actually want to scrape.

And so it remained just a vague idea in my head until I encountered a torrent site called Zipru. It has multiple mechanisms in place that require advanced scraping techniques but its `robots.txt` file allows scraping. Furthermore, there is *no reason to scrape it*. It has a public API that can be used to get all of the same data. If you're interested in getting torrent data then just use the API; it's great for that.

In the rest of this article, I'll walk you through writing a scraper that can handle captchas and various other challenges that we'll encounter on the Zipru site. The code won't work exactly as written because Zipru isn't a real site but the techniques employed are broadly applicable to real-world scraping and the code is otherwise complete. I'm going to assume that you have basic familiarity with python but I'll try to keep this accessible to someone with little to no knowledge of scrapy. If things are going too fast at first then take a few minutes to read The Scrapy Tutorial (https://doc.scrapy.org/en/latest/intro/tutorial.html) which covers the introductory stuff in much more depth.

Setting Up the Project

We'll work within a virtualenv (https://virtualenv.pypa.io/en/stable/) which lets us encapsulate our dependencies a bit. Let's start by setting up a virtualenv in `~/scrapers/zipru` and installing scrapy.

```
mkdir ~/scrapers/zipru
cd ~/scrapers/zipru
virtualenv env
. env/bin/activate
pip install scrapy
```

The terminal that you ran those in will now be configured to use the local virtualenv. If you open another terminal then you'll need to run `. ~/scrapers/zipru/env/bin/active` again (otherwise you may get errors about commands or modules not being found).

You can now create a new project scaffold by running

```
scrapy startproject zipru_scraper
```

which will create the following directory structure.

```
└── zipru_scraper
    ├── zipru_scraper
    │   ├── __init__.py
    │   ├── items.py
    │   ├── middlewares.py
    │   ├── pipelines.py
    │   ├── settings.py
    │   └── spiders
    │       └── __init__.py
    └── scrapy.cfg
```

Most of these files aren't actually used at all by default, they just suggest a sane way to structure our code. From now on, you should think of `~/scrapers/zipru/zipru_scraper` as the top-level directory of the project. That's where any scrapy commands should be run and is also the root of any relative paths.

Adding a Basic Spider

We'll now need to add a spider in order to make our scraper actually do anything. A spider is the part of a scrapy scraper that handles parsing documents to find new URLs to scrape and data to extract. I'm going to lean pretty heavily on the default Spider (https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy-spider) implementation to minimize the amount of code that we'll have to write. Things might seem a little automagical here but much less so if you check out the documentation.

First, create a file named `zipru_scraper/spiders/zipru_spider.py` with the following contents.

```
import scrapy

class ZipruSpider(scrapy.Spider):
    name = 'zipru'
    start_urls = ['http://zipru.to/torrents.php?category=TV']
```

Our spider inherits from `scrapy.Spider` which provides a `start_requests()` method that will go through `start_urls` and use them to begin our search. We've provided a single URL in `start_urls` that points to the TV listings. They look something like this.

Pages: 1 2 3 4 5 6 7 8 9 »

Cat.	File	Added	Size	S.	L.	💬	Uploader
TV	Arrow.S05E16.HDTV.x264-LOL[ettv] Action, Adventure, Crime, Drama, Mystery, Sci-Fi IMDB: 7.9/10	2017-03-16 02:03:05	248.15 MB	5800	3105	1	ettv
TV	The.Flash.2014.S03E16.HDTV.x264-LOL[ettv] Action, Adventure, Drama, Sci-Fi IMDB: 8.1/10	2017-03-15 02:02:14	215.93 MB	5353	1722	1	ettv
TV	Lethal.Weapon.S01E18.HDTV.x264-LOL[ettv] Action, Crime, Drama, Thriller IMDB: 8.0/10	2017-03-16 02:04:15	247.71 MB	2724	983	1	ettv
TV	DCs.Legends.of.Tomorrow.S02E14.HDTV.x264-LOL[ettv] Action, Adventure, Drama, Sci-Fi IMDB: 7.1/10	2017-03-15 03:03:10	315.97 MB	2700	722	--	ettv
TV	The.Walking.Dead.S07E13.WEB-DL.x264-FUM[ettv] Drama, Horror, Thriller IMDB: 8.5/10	2017-03-13 09:01:23	538.93 MB	2367	255	5	ettv

At the top there, you can see that there are links to other pages. We'll want our scraper to follow those links and parse them as well. To do that, we'll first need to identify the links and find out where they point.

The DOM inspector can be a huge help at this stage. If you were to right click on one of these page links and look at it in the inspector then you would see that the links to other listing pages look like this

```
<a href="/torrents.php?...page=2" title="page 2">2</a>
<a href="/torrents.php?...page=3" title="page 3">3</a>
<a href="/torrents.php?...page=4" title="page 4">4</a>
```

Next we'll need to construct selector expressions for these links. There are certain types of searches that seem like a better fit for either css or xpath selectors and so I generally tend to mix and chain them somewhat freely. I highly recommend learning xpath (http://edutechwiki.unige.ch/en/XPath_tutorial_-_basics) if you don't know it, but it's unfortunately a bit beyond the scope of this tutorial. I personally find it to be pretty indispensible for scraping, web UI testing, and even just web development in general. I'll stick with css selectors here though because they're probably more familiar to most people.

To select these page links we can look for `<a>` tags with "page" in the title using `a[title ~= page]` as a css selector. If you press `ctrl-f` in the DOM inspector then you'll find that you can use this css expression as a search query (this works for xpath too!). Doing so lets you cycle through and see all of the matches. This is a good way to check that an expression works but also isn't so vague that it matches other things unintentionally. Our page link selector satisfies both of those criteria.

To tell our spider how to find these other pages, we'll add a `parse(response)` method to `ZipruSpider` like so

```
    def parse(self, response):
        # proceed to other pages of the listings
        for page_url in response.css('a[title ~= page]::attr(href)').extract():
            page_url = response.urljoin(page_url)
            yield scrapy.Request(url=page_url, callback=self.parse)
```

When we start scraping, the URL that we added to `start_urls` will automatically be fetched and the response fed into this `parse(response)` method. Our code then finds all of the links to other listing pages and yields new requests which are attached to the same `parse(response)` callback. These requests will be turned into response objects and then fed back into `parse(response)` so long as the URLs haven't already been processed (thanks to the dupe filter).

Our scraper can already find and request all of the different listing pages but we still need to extract some actual data to make this useful. The torrent listings sit in a `<table>` with `class="list2at"` and then each individual listing is within a `<tr>` with `class="lista2"`. Each of these rows in turn contains 8 `<td>` tags that correspond to "Category", "File", "Added", "Size", "Seeders", "Leechers", "Comments", and "Uploaders". It's probably easiest to just see the other details in code, so here's our updated `parse(response)` method.

```
    def parse(self, response):
        # proceed to other pages of the listings
        for page_url in response.xpath('//a[contains(@title, "page ")]/@href').extract():
            page_url = response.urljoin(page_url)
            yield scrapy.Request(url=page_url, callback=self.parse)

        # extract the torrent items
        for tr in response.css('table.lista2t tr.lista2'):
            tds = tr.css('td')
            link = tds[1].css('a')[0]
            yield {
                'title' : link.css('::attr(title)').extract_first(),
                'url' : response.urljoin(link.css('::attr(href)').extract_first()),
                'date' : tds[2].css('::text').extract_first(),
                'size' : tds[3].css('::text').extract_first(),
                'seeders': int(tds[4].css('::text').extract_first()),
                'leechers': int(tds[5].css('::text').extract_first()),
                'uploader': tds[7].css('::text').extract_first(),
            }
```

Our `parse(response)` method now also yields dictionaries which will automatically be differentiated from the requests based on their type. Each dictionary will be interpreted as an item and included as part of our scraper's data output.

We would be done right now if we were just scraping most websites. We could just run

```
scrapy crawl zipru -o torrents.jl
```

and a few minutes later we would have a nice JSON Lines (http://jsonlines.org/) formatted `torrents.jl` file with all of our torrent data. Instead we get this (along with a lot of other stuff)

```
[scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/m
[scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023
[scrapy.core.engine] DEBUG: Crawled (403) <GET http://zipru.to/robots.txt> (referer: None) ['partia
[scrapy.core.engine] DEBUG: Crawled (403) <GET http://zipru.to/torrents.php?category=TV> (referer:
[scrapy.spidermiddlewares.httperror] INFO: Ignoring response <403 http://zipru.to/torrents.php?cate
[scrapy.core.engine] INFO: Closing spider (finished)
```

Drats! We're going to have to be a little more clever to get our data that we could totally just get from the public API and would never actually scrape.

The Easy Problem

Our first request gets a `403` response that's ignored and then everything shuts down because we only seeded the crawl with one URL. The same request works fine in a web browser, even in incognito mode with no session history, so this has to be caused by some difference in the request headers. We could use tcpdump (https://en.wikipedia.org/wiki/Tcpdump) to compare the headers of the two requests but there's a common culprit here that we should check first: the user agent.

Scrapy identifies as "Scrapy/1.3.3 (+http://scrapy.org)" (http://scrapy.org)") by default and some servers might block this or even whitelist a limited number of user agents. You can find lists of the most common user agents (https://techblog.willshouse.com/2012/01/03/most-common-user-agents/) online and using one of these is often enough to get around basic anti-scraping measures. Pick your favorite and then open up `zipru_scraper/settings.py` and replace

```
# Crawl responsibly by identifying yourself (and your website) on the user-agent
#USER_AGENT = 'zipru_scraper (+http://www.yourdomain.com)'
```

with

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Geck
```

You might notice that the default scrapy settings did a little bit of scrape-shaming there. Opinions differ on the matter but I personally think it's OK to identify as a common web browser if your scraper acts like somebody using a common web browser. So let's slow down the response rate a little bit by also adding

```
CONCURRENT_REQUESTS = 1
DOWNLOAD_DELAY = 5
```

which will create a somewhat realistic browsing pattern thanks to the AutoThrottle extension (https://doc.scrapy.org/en/latest/topics/autothrottle.html). Our scraper will also respect `robots.txt` by default so we're really on our best behavior.

Now running the scraper again with `scrapy crawl zipru -o torrents.jl` should produce

```
[scrapy.core.engine] DEBUG: Crawled (200) <GET http://zipru.to/robots.txt> (referer: None)
[scrapy.downloadermiddlewares.redirect] DEBUG: Redirecting (302) to <GET http://zipru.to/threat_def
[scrapy.core.engine] DEBUG: Crawled (200) <GET http://zipru.to/threat_defense.php?defense=1&r=78213
[scrapy.core.engine] INFO: Closing spider (finished)
```

That's real progress! We got two `200` statuses and a `302` that the downloader middleware knew how to handle. Unfortunately, that `302` pointed us towards a somewhat ominous sounding `threat_defense.php`. Unsurprisingly, the spider found nothing good there and the crawl terminated.

Downloader Middleware

It will be helpful to learn a bit about how requests and responses are handled in scrapy before we dig into the bigger problems that we're facing. When we created our basic spider, we produced `scrapy.Request` objects and then these were somehow turned into `scrapy.Response` objects corresponding to responses from the server. A big part of that "somehow" is downloader middleware.

Downloader middlewares inherit from `scrapy.downloadermiddlewares.DownloaderMiddleware` and implement both `process_request(request, spider)` and `process_response(request, response, spider)` methods. You can probably guess what those do from their names. There are actually a whole bunch of these middlewares enabled by default. Here's what the standard configuration looks like (you can of course disable things, add things, or rearrange things).

```
DOWNLOADER_MIDDLEWARES_BASE = {
    'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware': 300,
    'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware': 350,
    'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware': 400,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': 500,
    'scrapy.downloadermiddlewares.retry.RetryMiddleware': 550,
    'scrapy.downloadermiddlewares.ajaxcrawl.AjaxCrawlMiddleware': 560,
    'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware': 580,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 590,
    'scrapy.downloadermiddlewares.redirect.RedirectMiddleware': 600,
    'scrapy.downloadermiddlewares.cookies.CookiesMiddleware': 700,
    'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.downloadermiddlewares.stats.DownloaderStats': 850,
    'scrapy.downloadermiddlewares.httpcache.HttpCacheMiddleware': 900,
}
```

As a request makes its way out to a server, it bubbles through the `process_request(request, spider)` method of each of these middlewares. This happens in sequential numerical order such that the `RobotsTxtMiddleware` processes the request first and the `HttpCacheMiddleware` processes it last. Then once a response has been generated it bubbles back through the `process_response(request, response, spider)` methods of any enabled middlewares. This happens in reverse order this time so the higher numbers are always closer to the server and the lower numbers are always closer to the spider.

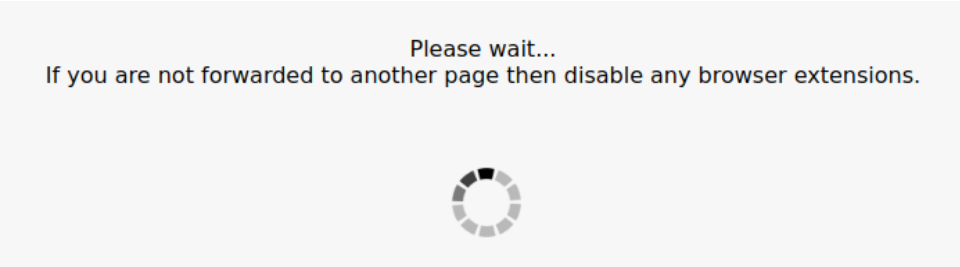
One particularly simple middleware is the `CookiesMiddleware`. It basically checks the `Set-Cookie` header on incoming responses and persists the cookies. Then when a response is on its way out it sets the `Cookie` header appropriately so they're included on outgoing requests. It's a little more complicated (https://github.com/scrapy/scrapy/blob/129421c7e31b89b9b0f9c5f7d8ae59e47df36091/scrapy/downloadermiddl than that because of expirations and stuff but you get the idea.

Another fairly basic one is the `RedirectMiddleware` which handles, wait for it... `3XX` redirects. This one lets any non-`3XX` status code responses happily bubble through but what if there is a redirect? The only way that it can figure out how the server responds to the redirect URL is to create a new request, so that's exactly what it does. When the `process_response(request, response, spider)` method returns a request object instead of a response then the current response is dropped and everything starts over with the new request. That's how the `RedirectMiddleware` handles the redirects and it's a feature that we'll be using shortly.

If it was surprising at all to you that there are so many downloader middlewares enabled by default then you might be interested in checking out the Architecture Overview (https://doc.scrapy.org/en/latest/topics/architecture.html). There's actually kind of a lot of other stuff going on but, again, one of the great things about scrapy is that you don't *have to* know anything about most of it. Just like you didn't even need to know that downloader middlewares existed to write a functional spider, you don't need to know about these other parts to write a functional downloader middleware.

The Hard Problem(s)

Getting back to our scraper, we found that we were being redirected to some `threat_defense.php?defense=1&...` URL instead of receiving the page that we were looking for. When we visit this page in the browser, we see something like this for a few seconds



before getting redirected to a `threat_defense.php?defense=2&...` page that looks more like this

A look at the source of the first page shows that there is some javascript code responsible for constructing a special redirect URL and also for manually constructing browser cookies. If we're going to get through this then we'll have to handle both of these tasks.

Then, of course, we also have to solve the captcha and submit the answer. If we happen to get it wrong then we sometimes redirect to another captcha page and other times we end up on a page that looks like this

where we need to click on the "Click here" link to start the whole redirect cycle over. Piece of cake, right?

All of our problems sort of stem from that initial `302` redirect and so a natural place to handle them is within a customized version of the redirect middleware (https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#module-scrapy.downloadermiddlewares.redirect). We want our middleware to act like the normal redirect middleware in all cases except for when there's a `302` to the `threat_defense.php` page. When it does encounter that special `302`, we want it to bypass all of this threat defense stuff, attach the access cookies to the session, and finally re-request the original page. If we can pull that off then our spider doesn't have to know about any of this business and requests will "just work."

So open up `zipru_scraper/middlewares.py` and replace the contents with

```
import os, tempfile, time, sys, logging
logger = logging.getLogger(__name__)

import dryscrape
import pytesseract
from PIL import Image

from scrapy.downloadermiddlewares.redirect import RedirectMiddleware

class ThreatDefenceRedirectMiddleware(RedirectMiddleware):
    def _redirect(self, redirected, request, spider, reason):
        # act normally if this isn't a threat defense redirect
        if not self.is_threat_defense_url(redirected.url):
            return super()._redirect(redirected, request, spider, reason)

        logger.debug(f'Zipru threat defense triggered for {request.url}')
        request.cookies = self.bypass_threat_defense(redirected.url)
        request.dont_filter = True # prevents the original link being marked a dupe
        return request

    def is_threat_defense_url(self, url):
        return '://zipru.to/threat_defense.php' in url
```

You'll notice that we're subclassing `RedirectMiddleware` instead of `DownloaderMiddleware` directly. This allows us to reuse most of the built in redirect handling and insert our code into `_redirect(redirected, request, spider, reason)` which is only called from `process_response(request, response, spider)` once a redirect request has been constructed. We just defer to the super-class implementation here for standard redirects but the special threat defense redirects get handled differently. We haven't implemented `bypass_threat_defense(url)` yet but we can see that it should return the access cookies which will be attached to the original request and that the original request will then be reprocessed.

To enable our new middleware we'll need to add the following to `zipru_scraper/settings.py`.

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy.downloadermiddlewares.redirect.RedirectMiddleware': None,
    'zipru_scraper.middlewares.ThreatDefenceRedirectMiddleware': 600,
}
```

This disables the default redirect middleware and plugs ours in at the exact same position in the middleware stack. We'll also have to install a few additional packages that we're importing but not actually using yet.

```
pip install dryscrape # headless webkit
pip install Pillow # image processing
pip install pytesseract # OCR
```

Note that all three of these are packages with external dependencies that pip can't handle. If you run into errors then you may need to visit the dryscrape (http://dryscrape.readthedocs.io/en/latest/installation.html), Pillow (http://pillow.readthedocs.io/en/3.4.x/installation.html), and pytesseract (https://github.com/madmaze/pytesseract) installation guides to follow platform specific instructions.

Our middleware should be functioning in place of the standard redirect middleware behavior now; we just need to implement `bypass_thread_defense(url)`. We could parse the javascript to get the variables that we need and recreate the logic in python but that seems pretty fragile and is a lot of work. Let's take the easier, though perhaps clunkier, approach of using a headless webkit instance. There are a few different options but I personally like dryscrape (https://dryscrape.readthedocs.io/en/latest/index.html) (which we already installed).

First off, let's initialize a dryscrape session in our middleware constructor.

```
    def __init__(self, settings):
        super().__init__(settings)

        # start xvfb to support headless scraping
        if 'linux' in sys.platform:
            dryscrape.start_xvfb()

        self.dryscrape_session = dryscrape.Session(base_url='http://zipru.to')
```

You can think of this session as a single browser tab that does all of the stuff that a browser would typically do (*e.g.* fetch external resources, execute scripts). We can navigate to new URLs in the tab, click on things, enter text into inputs, and all sorts of other things. Scrapy supports concurrent requests and item processing but the response processing is single threaded. This means that we can use this single dryscrape session without having to worry about being thread safe.

So now let's sketch out the basic logic of bypassing the threat defense.

```
    def bypass_threat_defense(self, url=None):
        # only navigate if any explicit url is provided
        if url:
            self.dryscrape_session.visit(url)

        # solve the captcha if there is one
        captcha_images = self.dryscrape_session.css('img[src *= captcha]')
        if len(captcha_images) > 0:
            return self.solve_captcha(captcha_images[0])

        # click on any explicit retry links
        retry_links = self.dryscrape_session.css('a[href *= threat_defense]')
        if len(retry_links) > 0:
            return self.bypass_threat_defense(retry_links[0].get_attr('href'))

        # otherwise, we're on a redirect page so wait for the redirect and try again
        self.wait_for_redirect()
        return self.bypass_threat_defense()

    def wait_for_redirect(self, url = None, wait = 0.1, timeout=10):
        url = url or self.dryscrape_session.url()
        for i in range(int(timeout//wait)):
            time.sleep(wait)
            if self.dryscrape_session.url() != url:
                return self.dryscrape_session.url()
        logger.error(f'Maybe {self.dryscrape_session.url()} isn\'t a redirect URL?')
        raise Exception('Timed out on the zipru redirect page.')
```

This handles all of the different cases that we encountered in the browser and does exactly what a human would do in each of them. The action taken at any given point only depends on the current page so this approach handles the variations in sequences somewhat gracefully.

The one last piece of the puzzle is to actually solve the captcha. There are captcha solving services (https://anti-captcha.com/) out there with APIs that you can use in a pinch, but this captcha is simple enough that we can just solve it using OCR. Using pytesseract for the OCR, we can finally add our `solve_captcha(img)` method and complete the `bypass_threat_defense()` functionality.

```
    def solve_captcha(self, img, width=1280, height=800):
        # take a screenshot of the page
        self.dryscrape_session.set_viewport_size(width, height)
        filename = tempfile.mktemp('.png')
        self.dryscrape_session.render(filename, width, height)

        # inject javascript to find the bounds of the captcha
        js = 'document.querySelector("img[src *= captcha]").getBoundingClientRect()'
        rect = self.dryscrape_session.eval_script(js)
        box = (int(rect['left']), int(rect['top']), int(rect['right']), int(rect['bottom']))

        # solve the captcha in the screenshot
        image = Image.open(filename)
        os.unlink(filename)
        captcha_image = image.crop(box)
        captcha = pytesseract.image_to_string(captcha_image)
        logger.debug(f'Solved the Zipru captcha: "{captcha}"')

        # submit the captcha
        input = self.dryscrape_session.xpath('//input[@id = "solve_string"]')[0]
        input.set(captcha)
        button = self.dryscrape_session.xpath('//button[@id = "button_submit"]')[0]
        url = self.dryscrape_session.url()
        button.click()

        # try again if it we redirect to a threat defense URL
        if self.is_threat_defense_url(self.wait_for_redirect(url)):
            return self.bypass_threat_defense()

        # otherwise return the cookies as a dict
        cookies = {}
        for cookie_string in self.dryscrape_session.cookies():
            if 'domain=zipru.to' in cookie_string:
                key, value = cookie_string.split(';')[0].split('=')
                cookies[key] = value
        return cookies
```

You can see that if the captcha solving fails for some reason that this delegates back to the `bypass_threat_defense()` method. This grants us multiple captcha attempts where necessary because we can always keep bouncing around through the verification process until we get one right.

This *should* be enough to get our scraper working but instead it gets caught in an infinite loop.

```
[scrapy.core.engine] DEBUG: Crawled (200) <GET http://zipru.to/robots.txt> (referer: None)
[zipru_scraper.middlewares] DEBUG: Zipru threat defense triggered for http://zipru.to/torrents.php?
[zipru_scraper.middlewares] DEBUG: Solved the Zipru captcha: "UJM39"
[zipru_scraper.middlewares] DEBUG: Zipru threat defense triggered for http://zipru.to/torrents.php?
[zipru_scraper.middlewares] DEBUG: Solved the Zipru captcha: "TQ9OG"
[zipru_scraper.middlewares] DEBUG: Zipru threat defense triggered for http://zipru.to/torrents.php?
[zipru_scraper.middlewares] DEBUG: Solved the Zipru captcha: "KH9A8"
...
```

It at least looks like our middleware is successfully solving the captcha and then reissuing the request. The problem is that the new request is *triggering the threat defense again*. My first thought was that I had some bug in how I was parsing or attaching the cookies but I triple checked this and the code is fine. This is another of those "the only things that could possibly be different are the headers" situation.

The headers for scrapy and dryscrape are obviously both bypassing the initial filter that triggers `403` responses because we're not getting any `403` responses. This must somehow be caused by the fact that their headers *are different*. My guess is that one of the encrypted access cookies includes a hash of the complete headers and that a request will trigger the threat defense if it doesn't match. The intention here might be to help prevent somebody from just copying the cookies from their browser into a scraper but it also just adds one more thing that you need to get around.

So let's specify our headers explicitly in `zipru_scraper/settings.py` like so.

```
DEFAULT_REQUEST_HEADERS = {
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
    'User-Agent': USER_AGENT,
    'Connection': 'Keep-Alive',
    'Accept-Encoding': 'gzip, deflate',
    'Accept-Language': 'en-US,*',
}
```

Note that we're explicitly adding the `User-Agent` header here to `USER_AGENT` which we defined earlier. This was already being added automatically by the user agent middleware but having all of these in one place makes it easier to duplicate the headers in dryscrape. We can do that by modifying our `ThreatDefenceRedirectMiddleware` initializer like so.

```
def __init__(self, settings):
    super().__init__(settings)

    # start xvfb to support headless scraping
    if 'linux' in sys.platform:
        dryscrape.start_xvfb()

    self.dryscrape_session = dryscrape.Session(base_url='http://zipru.to')
    for key, value in settings['DEFAULT_REQUEST_HEADERS'].items():
        # seems to be a bug with how webkit-server handles accept-encoding
        if key.lower() != 'accept-encoding':
            self.dryscrape_session.set_header(key, value)
```

Now, when we run our scraper again with `scrapy crawl zipru -o torrents.jl` we see a steady stream of scraped items and our `torrents.jl` file records it all. We've successfully gotten around all of the threat defense mechanisms!

Wrap it Up

We've walked through the process of writing a scraper that can overcome four distinct threat defense mechanisms:

1. User agent filtering.
2. Obfuscated javascript redirects.
3. Captchas.
4. Header consistency checks.

Our target website Zipru may have been fictional but these are all real anti-scraping techniques that you'll encounter on real sites. Hopefully you'll find the approach we took useful in your own scraping adventures.

← The stories that Hacker News removes from the front page (http://sangaline.com/post/the-stories-that-hacker-news-removes-from-the-front-page/)

Internet Archaeology: Scraping time series data from Archive.org → (http://sangaline.com/post/wayback-machine-scraper/)

Comments

herbst (https://news.ycombinator.com/user?id=herbst)

3 years ago (https://news.ycombinator.com/item?id=13884432)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

I've used antigate for captchas and ether Tor or proxies for 403s before. Usually the browser header alone does not help for long.

tehlike (https://news.ycombinator.com/user?id=tehlike)

3 years ago (https://news.ycombinator.com/item?id=13885048)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Anticaptcha and deatbycaptcha are some others. But it mames me feel sad to use them, as it exploits cheap labor overseas.

herbst (https://news.ycombinator.com/user?id=herbst)

3 years ago (https://news.ycombinator.com/item?id=13885126)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

while I agree, it may also is a option for some unskilled people to get to money they really need. IMO its better than buying cheap clothes, as captcha at least does not kill those people.

To clearify: I lived in some cheap third world cities, and i can certanly see that solving captchas could finance a rather nice life.

homakov (https://news.ycombinator.com/user?id=homakov)

3 years ago (https://news.ycombinator.com/item?id=13885646)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Most of the time they use OCR, humans are unreliable and rarely used.

tyingq (https://news.ycombinator.com/user?id=tyingq)

3 years ago (https://news.ycombinator.com/item?id=13885715)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

That doesn't seem to match the sales pitch at antigate.com, deathbycaptcha, etc.

herbst (https://news.ycombinator.com/user?id=herbst)

3 years ago (https://news.ycombinator.com/item?id=13885746)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

no, at least antigate doesnt. When you hit recaptcha with known proxy urls (or generally hit it a few times per hour) the captchas get so bad that no OCR would be able to solve it, even humans struggle

tehlike (https://news.ycombinator.com/user?id=tehlike)

3 years ago (https://news.ycombinator.com/item?id=13886300)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

yup, exactly. i tried tesarract before (nothing too fancy), it didn't have problems solving it, but at some point it became really hard.

I think part of it is how you crawl (phantomjs, for example, seem to hit captcha almost every time), but things like ip&proxy usage could make this trigger more often.

bdcravens (https://news.ycombinator.com/user?id=bdcravens)

3 years ago (https://news.ycombinator.com/item?id=13886299)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Antigate can solve things that would give OCR fits, like animated letters and the like.

driverdan (https://news.ycombinator.com/user?id=driverdan)

3 years ago (https://news.ycombinator.com/item?id=13895159)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

It's not exploitation.

jordif (https://news.ycombinator.com/user?id=jordif)

3 years ago (https://news.ycombinator.com/item?id=13889809)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Good article! I been doing scraping for the last 10 years and I've seen a lots of differents things to try to avoid us. Also, I'm in the other side protecting websites to ban scrapers, so funny!

skinnymuch (https://news.ycombinator.com/user?id=skinnymuch)

3 years ago (https://news.ycombinator.com/item?id=13890232)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

I'm in the same position for the first time (protecting against scraping) and honestly I'm kind of blind right now. Which is weird because of how much scraping I've done (okay not that much). Any tips or tricks or blogs you know of off the top of your head for protecting your site?

jordif (https://news.ycombinator.com/user?id=jordif)

3 years ago (https://news.ycombinator.com/item?id=13898617)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

After the years, I've arrived at the conclusion that everything can be scrapped. What you have to do is try to put as many walls as you can. But if someone really wants to crawl your site, with the right knowledgement he will able to do it despite of all your walls.

skinnymuch (https://news.ycombinator.com/user?id=skinnymuch)

3 years ago (https://news.ycombinator.com/item?id=13955839)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Yes that's what I've assumed as well

corford (https://news.ycombinator.com/user?id=corford)

3 years ago (https://news.ycombinator.com/item?id=13890696)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Virtually everything can be easily defeated. The only outfit I've consistently seen put up a good fight is Distil. They do it by acting a little like Cloudflare. They put their servers in front of your www facing endpoints and use ML to mine their global client traffic to identify bot signals (aided by some aggressive in-browser javascript fingerprinting).

kbenson (https://news.ycombinator.com/user?id=kbenson)

3 years ago (https://news.ycombinator.com/item?id=13909549)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Yeah, Distil is the first outfit I've encountered where they've got the model to make it really hard to *reliably* bypass. It comes down to "I can spend a significant amount of time trying to bypass this, and I would, but they would

likely identify and block me again within a few weeks at most.", and it's not worth it when it's only *part* of what I need to do to scrap some data, and it's their entire job, and they can afford to hire multiple people.

The economics are in their favor, and I make it a point not to fight economics when I recognize them, it's rarely sustainable.

skinnymuch (https://news.ycombinator.com/user?id=skinnymuch)
3 years ago (https://news.ycombinator.com/item?id=13955851)
Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Distil is really interesting.

skinnymuch (https://news.ycombinator.com/user?id=skinnymuch)
3 years ago (https://news.ycombinator.com/item?id=13955838)
Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Interesting thanks.

eapen (https://news.ycombinator.com/user?id=eapen)
3 years ago (https://news.ycombinator.com/item?id=13910802)
Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Enjoyed learning this and playing with it. What would you recommend storing this sort of data in? Not too keen on going with the traditional MySQL.

thefifthsetpin (https://news.ycombinator.com/user?id=thefifthsetpin)
3 years ago (https://news.ycombinator.com/item?id=13886486)
Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Better solution: pay target-site.com to start building an API for you.

Pros:

* You'll be working with them rather than against them.

* Your solution will be far more robust.

* It'll be way cheaper, supposing you account for the ongoing maintenance costs of your fragile scraper.

* You're eliminating the possibility that you'll have to deal with legal antagonism

* Good anti-scraper defenses are far more sophisticated than what the author dealt with. As a trivial example: he didn't even verify that he was following links that would be visible!

Cons:

* Possible that target-site.com's owners will tell you to get lost, or they are simply unreachable.

* Your competitors will gain access to the API methods you funded, perhaps giving them insight into why that data is valuable.

Alternative better solution for small one-off data collection needs: contract a low-income person to just manually download the data you need with a normal web browser. Provide a JS bookmarklet to speed their process if the data set is a bit too big for that.

lanCal (https://news.ycombinator.com/user?id=lanCal)

3 years ago (https://news.ycombinator.com/item?id=13888018)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

> Better solution: pay target-site.com to start building an API for you.

I'd add to this:

Do you *really* need continuing access? Or just their data occasionally?

Pay them to just get a db dump in some format. For large amounts of data, creating an API then having people scan and run through it is just a massive pain. Having someone paginate through 200M records regularly is a recipe for pain, on both sides.

A supported API might take a significant amount of time to develop, and has on-going support requirements, extra machines, etc. Then you have to have all your infrastructure or long running processes to hammer it and get the data as fast as you can, with network errors and all other kinds of intermittent problems to handle.

A pg_dump > s3 dump might take an engineer an afternoon and will take minutes to download, requiring getting approval from a significantly lower level and having a much easier to estimate cost of providing.

robryan (https://news.ycombinator.com/user?id=robryan)

3 years ago (https://news.ycombinator.com/item?id=13889883)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Many instances of us building scrapers are cases where a partner has data or has tools which are only built into the UI or the UI ones are much more capable.

Rather than waiting potentially years for their IT team to make the required changes we can build a scraper in a matter of days.

hackits (https://news.ycombinator.com/user?id=hackits)

3 years ago (https://news.ycombinator.com/item?id=13890321)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

I can attest to this. From personal experience I found websites that would ignore scrappers and just allow me to access their data on their public web-site easier to deal with code wise and time wise. I make the request, you give me the data I need and then I can piss off.

Web-sites that make it a cluster &&*& to get access to the data do two things. They setup a challenge to break their idiotic `are you a bot?` and secondly it is trivial in most situation just to spin up a vm, and run chrome with selenium and a python script.

Granted I don't use AJAX API or anything like that. Instead I've found developer who nativly have a JSON string along side the data within the HTML to easiest to parse.

Reasons why I've setup bots/scrappers 1) My local city rental market is very competitive and I hate wasting time email landlords who have all-ready signed up a lease. 2) House prices 3) Car prices 4) Stock prices 5) Banking 6) Water

Rates 7) Health insurance comparison 8) Automated billing and payment systems.

brilliantcode (https://news.ycombinator.com/user?id=brilliantcode)

3 years ago (https://news.ycombinator.com/item?id=13889630)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

This can't be a solution for people using web scraping:

The goal for web scrapers is to pay as little as possible for as much data as possible.

selllikesybok (https://news.ycombinator.com/user?id=selllikesybok)

3 years ago (https://news.ycombinator.com/item?id=13893422)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Depends on the scraper. I buy data dumps when I can, if possible. Plus, it can actually be cheaper to enter into a business relationship with the target site than it is to play whack a mile with their anti scraping development efforts over time.

diminoten (https://news.ycombinator.com/user?id=diminoten)

3 years ago (https://news.ycombinator.com/item?id=13888359)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Also known as the tcgplayer.com strategy. Very disappointing to find out about, especially when the margins on hobby-level Magic: the Gathering card selling are already so low.

pjc50 (https://news.ycombinator.com/user?id=pjc50)

3 years ago (https://news.ycombinator.com/item?id=13886576)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

> pay target-site.com to start building an API for you.

When has that ever worked?

benjamincburns (https://news.ycombinator.com/user?id=benjamincburns)

3 years ago (https://news.ycombinator.com/item?id=13887502)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

I can't cite specific examples because the ones I know about formed confidential business relationships, but I can say with confidence that this works All. The. Time.

That said, if you're some small-time researcher who can't offer a compelling business case to make this happen, then it won't be worth their time and they're likely to show you the door. [Note: my implication here is that it's not because you're small time, but it's because by the nature of your work you're not focusing on business drivers which are meaningful to the company/org you're propositioning].

Edit: Also be warned that if you're building a successful business on scraped personal info, you're *begging* to be served w/ a class action lawsuit (though take that well-salted, because IANAL and all that jazz).

GFischer (https://news.ycombinator.com/user?id=GFischer)

3 years ago (https://news.ycombinator.com/item?id=13888085)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

When the money is good enough :) . That is, usually not for startups, yes for established companies with money.

nibbler_death (https://news.ycombinator.com/user?id=nibbler_death)

3 years ago (https://news.ycombinator.com/item?id=13887337)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Never.

thefifthsetpin (https://news.ycombinator.com/user?id=thefifthsetpin)

3 years ago (https://news.ycombinator.com/item?id=13888177)

Y Hacker News (https://news.ycombinator.com/item?id=13884357)

Most of the time, in my (admittedly limited) experience. The two exceptions have been:

A giant site, who already had an API & had deliberately decided to not implement the API calls we wanted. I should add that another giant site have happily added API options for us. (For my client, really; not for me.)

An archaic site. The dev team was gone & the owners were just letting it trickle them revenue until it died -- they didn't even want to think about it anymore.

See other 201 comments on Hacker News (https://news.ycombinator.com/item?id=13884357)

comments.network (https://comments.network/)