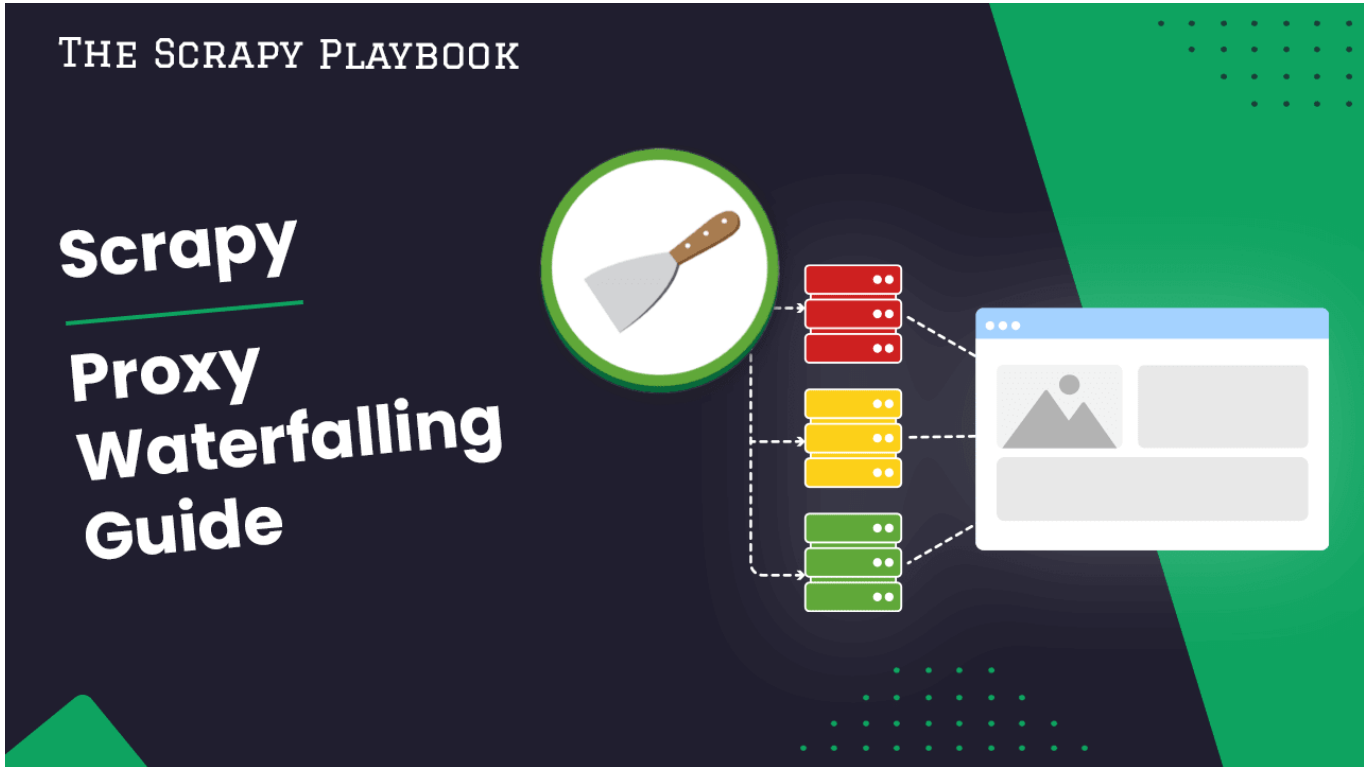

Scrapy Proxy Waterfalling: How to Waterfall Requests Over Multiple Proxy Providers

A very common requirement for many web scrapers is designing a system that can waterfall requests over mutliple proxy providers to increase reliabilty and decrease costs.

Proxies can be tempermental. They might work today for a particular website, but not tomorrow. And can vary massively in price.

So a common strategy most developers use, is to have proxy plans with a number of proxy providers and distribute their requests amongst.

In this guide, we're going to go through how you can build a custom proxy waterfalling middleware for Scrapy:

First, let's discuss our proxy waterfalling strategy.

Our Proxy Waterfalling Strategy

Your proxy waterfalling system can be as complex or as simple as you need it to be based on your own requirements and objectives. In this guide, we're going to focus on a couple key objectives:

1. Use no proxy on first request, to save on proxy costs.
2. Using the cheapest working proxy for a particular website.

3. Increasing reliability by falling back to better (more expensive) proxies if others are failing.
4. Ability to use specific proxies for specific websites.

You could make this middleware more advanced by adding in the functionality to customise proxy parameters (geotargeting, JS rendering, etc.) based on flags set in the request, manage user-agents, or validate responses on the fly. However, for this guide we're going to cover the essentials as you should be able to expand it if you need to.

For this guide we're going to use 3 proxy providers:

- Scrapingdog
- ScraperAPI
- Scrapingbee

With **Scrapingdog** being the cheapest, followed by **ScraperAPI** and then **Scrapingbee**.

Building The Proxy Waterfall Middleware

To build the proxy waterfall middleware we are going to create a new `DownloaderMiddleware` in our projects `middleware.py` file, then complete the following steps:

1. Create Outline of Middleware
2. Setup Proxies Upon Launch
3. Function To Set Proxy On Request
4. Architect Our Proxy Waterfall

Let's get started.

---

1. Create Outline of Middleware

First things first is that we will outline the middleware. Which thanks to Scrapy is already pretty well

defined.

For this middleware, we only care about setting up the middleware on launch and adding proxies to incoming requests so we just need to use the `from_crawler`, `__init__`, and `process_request` functions available in Scrapy middleware classes. We will also create a `api_key_valid` and `add_proxy` to keep our code clean.

```
## middleware.py

class ProxyWaterfallMiddleware:

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.settings)

    def __init__(self, settings):
        # Where we set up the middleware at launch.
        pass

    def api_key_valid(self, api_key):
        # This is a custom function, where we will use this to make sure we have
        # a valid API key for the proxy before using it.
        pass

    def add_proxy(self, request, username, password, host):
        # This is a custom function, where we will use to set a proxy on a request.
        pass

    def process_request(self, request, spider):
        # Event listen function in Scrapy where we will implement the waterfall system.
        pass
```

---

2. Setup Proxies Upon Launch

The first bit of functionality we need to create is to setup the middleware on launch, by loading in our proxy provier API keys from the `settings.py` file and then define the proxy `username`, `password`, and `host` for each proxy provider. We do this in the `__init__` method.

```
## middleware.py

class ProxyWaterfallMiddleware:

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.settings)

    def __init__(self, settings):
        self.scraperapi_api_key = settings.get('SCRAPERAPI_API_KEY')
        self.scrapingbee_api_key = settings.get('SCRAPINGBEE_API_KEY')
        self.scrapingdog_api_key = settings.get('SCRAPINGDOG_API_KEY')

        ## Scrapingbee
        self.scrapingbee_http_address = 'http://proxy.scrapingbee.com:8886'
        self.scrapingbee_username = self.scrapingbee_api_key
        self.scrapingbee_password = 'render_js=False'

        ## ScraperAPI
        self.scraperapi_http_address = 'http://proxy-server.scraperapi.com:8001'
        self.scraperapi_username = 'scraperapi'
        self.scraperapi_password = self.scraperapi_api_key

        ## Scrapingdog
        self.scrapingdog_http_address = 'http://proxy.scrapingdog.com:8081'
        self.scrapingdog_username = 'scrapingdog'
        self.scrapingdog_password = self.scrapingdog_api_key

    def api_key_valid(self, api_key):
        if api_key is None or api_key == '':
            return False
        return True

    def add_proxy(self, request, username, password, host):
```

```
        # This is a custom function, where we will use to set a proxy on a request.
        pass

    def process_request(self, request, spider):
        # Event listen function in Scrapy where we will implement the waterfall system.
        pass
```

Each proxy provider has a slightly different layout in terms of what values up put in the `username`, `password`, and `host` so we've had to define them seperatly.

We also, created the `api_key_valid` method which will be used later on to make sure we have a valid API key before sending a request with a proxy.

---

3. Function To Set Proxy On Request

Next, we're going to create the `add_proxy` which will be used to set a proxy for each request, and import the `base64` library so we can encode the user credentials.

```
## middleware.py
import base64

...

    def add_proxy(self, request, username, password, host):
        user_credentials = '{user}:{passw}'.format(user=username, passw=password)
        basic_authentication = 'Basic ' +
base64.b64encode(user_credentials.encode()).decode()
        request.meta['proxy'] = host
        request.headers['Proxy-Authorization'] = basic_authentication
```

This takes in the with the `username`, `password` and `host` of the proxy we want to use and applies

them to the `Request` we provide.

---

4. Architect Our Proxy Waterfall

Finally, we have everything we need to create our proxy waterfall system.

We're going to architect it as follows:

1. Always use ScraperAPI when scraping Google as they don't charge extra for Google requests.
2. Other then scraping Google, make the request without a proxy to try and reduce costs.
3. Use Scrapingdog for the first 2 retries.
4. Use ScraperAPI for the 3rd & 4th retry.
5. Use Scrapingbee for all other retries.

If one of the proxies isn't enabled (checks if no API key set with `check_api_key_valid`), the the proxy middleware will fallback to the next proxy in the waterfall. The final middleware looks like this:

```
## middleware.py

import base64

class ProxyWaterfallMiddleware:


    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.settings)


    def __init__(self, settings):
        self.scraperapi_api_key = settings.get('SCRAPERAPI_API_KEY')
        self.scrapingbee_api_key = settings.get('SCRAPINGBEE_API_KEY')
        self.scrapingdog_api_key = settings.get('SCRAPINGDOG_API_KEY')

        ## Scrapingbee
```

```
        self.scrapingbee_http_address = 'http://proxy.scrapingbee.com:8886'
        self.scrapingbee_username = self.scrapingbee_api_key
        self.scrapingbee_password = 'render_js=False'

        ## ScraperAPI
        self.scraperapi_http_address = 'http://proxy-server.scraperapi.com:8001'
        self.scraperapi_username = 'scraperapi'
        self.scraperapi_password = self.scraperapi_api_key

        ## Scrapingdog
        self.scrapingdog_http_address = 'http://proxy.scrapingdog.com:8081'
        self.scrapingdog_username = 'scrapingdog'
        self.scrapingdog_password = self.scrapingdog_api_key


    def api_key_valid(self, api_key):
        if api_key is None or api_key == '':
            return False
        return True


    def add_proxy(self, request, username, password, host):
        user_credentials = '{user}:{passw}'.format(user=username, passw=password)
        basic_authentication = 'Basic ' +
base64.b64encode(user_credentials.encode()).decode()
        request.meta['proxy'] = host
        request.headers['Proxy-Authorization'] = basic_authentication


    def process_request(self, request, spider):
        retries = request.meta.get('retry_times', 0)

        ## Always Use ScraperAPI for Google.com
        if 'google.com' in request.url and
self.check_api_key_valid(self.scraperapi_api_key):
            self.add_proxy(request, self.scraperapi_username, self.scraperapi_password,
self.scraperapi_http_address)
            return None
```

```
        ## No Proxy --> Use No Proxy On First Request
        if retries == 0:
            return None

        ## Proxy Tier #1 --> Use Scrapingdog For All Requests For First 2 Retries
        if self.check_api_key_valid(self.scrapingdog_api_key) and retries <= 2:
            self.add_proxy(request, self.scrapingdog_username,
self.scrapingdog_password, self.scrapingdog_http_address)
            return None


        ## Proxy Tier #2 --> Use ScraperAPI For All Requests For 3rd & 4th Retry
        if self.check_api_key_valid(self.scraperapi_api_key) and retries <= 4:
            self.add_proxy(request, self.scraperapi_username, self.scraperapi_password,
self.scraperapi_http_address)
            return None


        ## Proxy Tier #3 --> Use Scrapingbee For All Requests Aftr 5 Retries Or As
Fallback
        if self.check_api_key_valid(self.scrapingbee_api_key):
            self.add_proxy(request, self.scrapingbee_username,
self.scrapingbee_password, self.scrapingbee_http_address)
            return None
```

Now with the `ProxyWaterfallMiddleware` complete, we just need to enable it.

Integrating Our Proxy Waterfall Middleware Into Spiders

Integrating the **ProxyWaterfallMiddleware** we just created is very simple. We just need to update our `settings.py` with our proxy API keys and enable the middleware in our download middlewares.

```
## settings.py
```

```
SCRAPERAPI_API_KEY = 'YOUR_API_KEY'
SCRAPINGBEE_API_KEY = 'YOUR_API_KEY'
SCRAPINGDOG_API_KEY = 'YOUR_API_KEY'

DOWNLOADER_MIDDLEWARES = {
    'proxy_waterfall.middlewares.ProxyWaterfallMiddleware': 350,
}
```

Now, whenever you run a spider your requests will be routed through the Proxy Waterfall middleware and it will select the best proxy based on the criteria you have defined.

More Scrapy Tutorials

That's how you can create your own proxy waterfall system to help you lower proxy costs whilst also increasing reliability. This middleware can be easily customised and extended for your own project requirements.

If you would like to learn more about Scrapy in general, then be sure to check out The Scrapy Playbook.