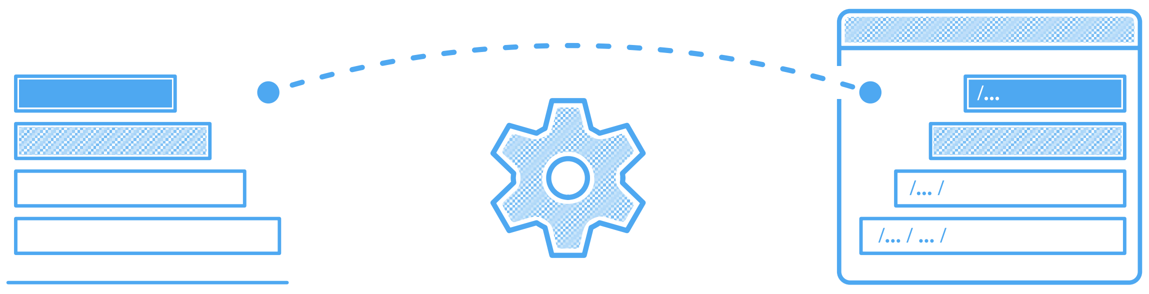
How to Find All URLs on a Domain

by Ziad Shamndy Jan 29, 2025 #Web Crawling #Python



Have you ever wondered how search engines like Google systematically find and index millions of web pages, identifying every URL on a domain? Or perhaps you're managing a website and need to audit your content or analyze its structure. Understanding how to extract all URLs from a domain can unlock invaluable insights for SEO, content strategy, and automation.

In this article, we'll dive deep into the process of finding all URLs on a domain. Whether you're a developer looking for efficient crawling methods or someone exploring no-code options, we've got you covered. By the end, you'll have a clear roadmap for extracting URLs from any domain.

Types of URLs: Understanding the Basics

When it comes to crawling a domain, not all URLs are created equal. Understanding the different types of URLs and their characteristics is key to building an effective crawling strategy.

What is a Domain?

A domain is the primary address of a website (e.g., `scrapfly.io`). It serves as the main identifier for a website, while subdomains like `blog.example.com` may host unique content or serve specific purposes.

Once you understand the concept of a domain, the next step is to explore the types of URLs you'll encounter during crawling, starting with internal and external URLs..

Internal vs. External URLs

When crawling a domain, one of the first distinctions to understand is the difference between internal and external URLs. These two types of links determine whether you're staying within the boundaries of the domain or venturing out to other websites. Let's break it down:

- **Internal URLs**: These are links that point to pages within the same domain. For example, a link like `https://example.com/about-us` is an internal URL if you're crawling `example.com`.
- **External URLs**: These links direct users to other domains, such as `https://another-site.com/resource`.

Understanding the difference between internal and external URLs is essential for planning your crawling strategy. With this distinction clear.

Absolute vs. Relative URLs

The way URLs are written affects how they are interpreted during the crawling process. Absolute URLs are complete and self-contained, while relative URLs require additional processing to resolve. Here's a closer look:

- **Absolute URLs**: These include the full address, with protocol (`https://`), domain name, and path. Example: `https://example.com/page`.
- **Relative URLs**: These are partial links relative to the current domain or page. For example, `/page` refers to a path on the same domain.

Knowing how to handle absolute and relative URLs ensures you don't miss any internal links during crawling. Now that we've covered URL types and formats, we can proceed to the practical task of crawling all URLs effectively.

Crawling All URLs

Crawling is the systematic process of visiting web pages to extract specific data, such as URLs. It's how search engines like Google discover and index web pages, creating a map of the internet. Similarly, you can use crawling techniques to gather all URLs on a domain for SEO analysis, content audits, or other data-driven purposes.

Why Crawl an Entire Domain?

Crawling an entire domain provides valuable insights into the structure, content, and links within a website. There are many reasons to crawl a domain. Here are some key use cases:

- **SEO Analysis**:
  Crawling helps identify broken links, duplicate content, and untapped SEO opportunities. It provides insight into how search engines might view your site.
- **Content Audits**:
  By mapping out the structure of your website, you can assess the organization of your content, identify gaps, and improve user navigation.
- **Security Scans**:
  Crawling can uncover vulnerabilities, outdated software, or sensitive information that may pose security risks.
- **Web Automation**:
  Crawlers are often used to extract data for analysis or reporting, automating repetitive tasks like collecting product details or tracking changes to web pages.

By understanding your goal whether SEO, auditing, or automation you can fine-tune your crawling strategy for the best results.

Next, we'll demonstrate how to build a simple crawler to extract URLs from a domain.

How to Find All URLs on a Domain

Let's look at an example of finding all URLs using Python and the 2 popular libraries:

- httpx for making fast HTTP requests
- beautifulsoup4 for parsing HTML

To start we need a function that reliably tries to return the page retrying connection issues etc.

```python
async def get_page(url, retries=5):
    """Fetch a page with retries for common HTTP and system errors."""
    for attempt in range(retries):
        try:
            async with httpx.AsyncClient(timeout=20) as client:
                response = await client.get(url)
                if response.status_code == 200:
                    return response.text
                else:
                    print(f"Non-200 status code {response.status_code} for {url}")
        except (httpx.RequestError, httpx.HTTPStatusError) as e:
            print(f"Attempt {attempt + 1} failed for {url}: {e}")
        await asyncio.sleep(1)  # Backoff between retries
    return None
```

Then we can use this function to create a crawl loop that:

1. Scrapes a given URL
2. Finds all *unseen* links of the same domain (relative or absolute with same TLD)
3. If crawl limit is not reached, repeat the process for each link

```python
import asyncio
import httpx
from bs4 import BeautifulSoup
from urllib.parse import quote, urljoin, urlparse

# Global configuration variables to track crawled pages and max limit
crawled_pages = set()
max_crawled_pages = 20  # Note: it's always good idea to have a limit to prevent
accidental endless loops

async def get_page(url, retries=5) -> httpx.Response:
    """Fetch a page with retries for common HTTP and system errors."""
    for attempt in range(retries):
        try:
            async with httpx.AsyncClient(timeout=10, follow_redirects=True) as client:
                response = await client.get(url)
                if response.status_code == 200:
                    return response
                else:
                    print(f"Non-200 status code {response.status_code} for {url}")
        except (httpx.RequestError, httpx.HTTPStatusError) as e:
            print(f"Attempt {attempt + 1} failed for {url}: {e}")
        await asyncio.sleep(1)  # Backoff between retries
    return None

async def process_page(response: httpx.Response) -> None:
    """
    Process the HTML content of a page here like store it in a database
    or parse it for content?
    """
    print(f"  processed: {response.url}")
    # ignore non-html results
    if "text/html" not in response.headers.get("content-type", ""):
        return
    safe_filename = quote(response.url, safe="")
    with open(f"{safe_filename}.html", "w") as f:
        f.write(response.text)


async def crawl_page(url: str, limiter: asyncio.Semaphore) -> None:
    """Crawl a page and extract all relative or same-domain URLs."""
    global crawled_pages
    if url in crawled_pages:  # url visited already?
        return
    # check if crawl limit is reached
    if len(crawled_pages) >= max_crawled_pages:
        return

    # scrape the url
    crawled_pages.add(url)
    print(f"crawling: {url}")
    html_content = await get_page(url)
    if not html_content:
        return
    await process_page(html_content)

    # extract all relative or same-domain URLs
    soup = BeautifulSoup(html_content, "html.parser")
    base_domain = urlparse(url).netloc
    urls = []
    for link in soup.find_all("a", href=True):
        href = link["href"]
        absolute_url = urljoin(url, href)
        absolute_url = absolute_url.split("#")[0]  # remove fragment
        if absolute_url in crawled_pages:
            continue
        if urlparse(absolute_url).netloc != base_domain:
            continue
        urls.append(absolute_url)
    print(f"  found {len(urls)} new links")
    # ensure we don't crawl more than the max limit
    _remaining_crawl_budget = max_crawled_pages - len(crawled_pages)
    if len(urls) > _remaining_crawl_budget:
        urls = urls[:_remaining_crawl_budget]

    # schedule more crawling concurrently
    async with limiter:
        await asyncio.gather(*[crawl_page(url, limiter) for url in urls])

async def main(start_url, concurrency=10):
    """Main function to control crawling."""
    limiter = asyncio.Semaphore(concurrency)
    try:
        await crawl_page(start_url, limiter=limiter)
    except asyncio.CancelledError:
        print("Crawling was interrupted")

if __name__ == "__main__":
    start_url = "https://web-scraping.dev/products"
    asyncio.run(main(start_url))
```

▶ **Example Output**

Even basic crawling involves a lot of important steps so lets break down the process:

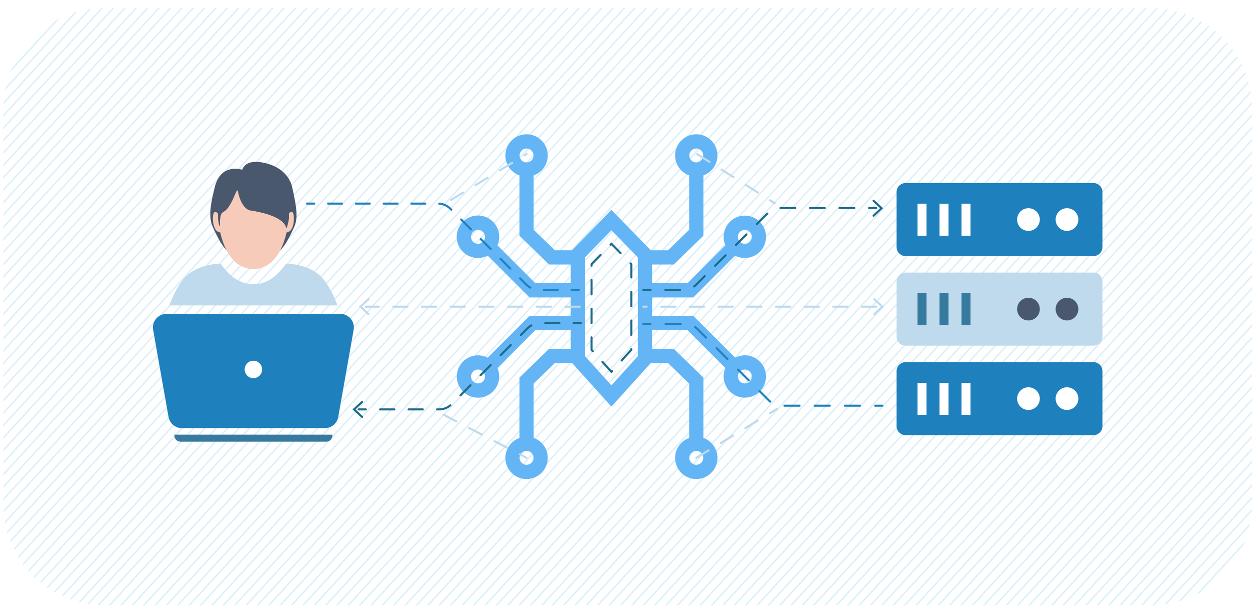
1. To ensure we don't crawl the same pages we keep a set of *seen* URLs and need to clean up urls of fragments and sometimes even query parameters.
2. To avoid crawling too fast we need to implement a limiter using `asyncio.Semaphore` to limit the number of concurrent requests.
3. We might crawl undesired pages like PDF files, images or other media and for that we can check the `Content-Type` header.
4. To prevent endless crawl loops we also set an overall limit of pages to crawl.

This simple crawler example using httpx and beautifulsoup for Python demonstrates how to find and crawl all urls on a domain though for more on crawling challenges see our full introduction to Crawling with Python

Power-Up with Scrapfly

ScrapFly provides web scraping, screenshot, and extraction APIs for data collection at scale.

- Anti-bot protection bypass - scrape web pages without blocking!
- Rotating residential proxies - prevent IP address and geographic blocks.
- JavaScript rendering - scrape dynamic web pages through cloud browsers.
- Full browser automation - control browsers to scroll, input and click on objects.
- Format conversion - scrape as HTML, JSON, Text, or Markdown.
- Python and Typescript SDKs, as well as Scrapy and no-code tool integrations.



Try for FREE!

More on Scrapfly

Using Scrapy for Crawling

Scrapy is a powerful Python framework designed specifically for web crawling and comes with `CrawlSpider` implementation that automatically handles:

- Link extractors that can identify links based on rules config
- Duplicate URL filtering
- Limits and concurrency settings

> Did you know you can access all the advanced web scraping features like **cloud browsers and blocking bypass** of Web Scraping API in your Scrapy spider!

All of this greatly simplifies the crawling process for you automatically and here's what our above crawler would look like when using `scrapy.CrawlSpider`:

```
from urllib.parse import quote

import scrapy
from scrapy.crawler import CrawlerProcess
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule

class SimpleCrawlSpider(CrawlSpider):
    name = "simple_crawler"
    allowed_domains = ["web-scraping.dev"]  # Restrict crawling to this domain
    start_urls = ["https://web-scraping.dev/products"]  # Starting URL

    # Define custom settings for the spider
    custom_settings = {
        "CLOSESPIDER_PAGECOUNT": 20,  # Limit to 20 pages
        "CONCURRENT_REQUESTS": 5,  # Limit concurrent requests
    }

    # Define crawling rules using LinkExtractor
    rules = [
        Rule(
            LinkExtractor(allow_domains="web-scraping.dev"),  # Only follow links
within the domain
            callback="parse_item",
            follow=True,  # Continue crawling links recursively
        )
    ]

    def parse_item(self, response):
        # Process the crawled page
        self.logger.info(f"Crawling: {response.url}")
        safe_filename = quote(response.url, safe="")
        with open(f"{safe_filename}.html", "wb") as f:
            f.write(response.body)

# Run the Scrapy spider
if __name__ == "__main__":
    process = CrawlerProcess()
    process.crawl(SimpleCrawlSpider)
    process.start()
```

In this example, we define a scrapy spider that inherit `CrawlSpider` crawl logic and we defined `rules` attribute for what the crawler should crawl. For our simple rules we lock the domain and inherit default LinkExtractor functionality like avoiding non-html pages.

The `Rule` and `LinkExtractor` object provide a great way to control the crawling process and come with reasonable default configuration so if you're really unfamiliar with crawling then `scrapy.CrawlSpider` is a great place to start.

Advantages of Scrapy

Scrapy is a versatile tool for web scraping, offering powerful features for efficient, large-scale crawling. Here are some key advantages that make it a top choice for developers.

- **Efficient for Large-Scale Crawling:** Scrapy handles concurrent requests and follows links automatically, making it highly efficient for crawling websites with many pages.
- **Built-In Error Handling:** It includes mechanisms to manage common issues such as timeouts, retries, and HTTP errors, ensuring smoother crawling sessions.
- **Respectful Crawling:** Scrapy adheres to `robots.txt` rules by default, helping you scrape ethically and avoid conflicts with website administrators.
- **Extensibility:** It integrates effortlessly with external tools like Scrapfly, enabling advanced features such as JavaScript rendering and proxy rotation for bypassing complex website defenses.

This makes Scrapy a reliable, scalable, and developer-friendly choice for web crawling projects of any size.

Challenges of Crawling URLs

There are clear technical challenges when it comes to crawling like:

- Filtering out unwanted URLs
- Retrying failed requests
- Efficient concurrency management using `asyncio`

Though not only that there are several other challenges that can be encountered in real life web crawling. Here's an overview of common challenges you may encounter during web crawling.

1. Blocking by the Website

One of the most common challenges is getting blocked. Websites use various techniques to detect and block bots, including:

- **IP Address Tracking:** If too many requests come from a single IP address, the website may block or throttle the crawler.
- **User Agent Monitoring:** Websites identify bots by checking the user agent header in requests. If it doesn't mimic a browser or matches known bot patterns, access may be denied.
- **Behavioral Analysis:** Some websites monitor request frequency and patterns. Bots that make rapid or repetitive requests can trigger blocking mechanisms.

2. CAPTCHA Challenges

CAPTCHAs are designed to differentiate between humans and bots by presenting tasks that are easy for humans but difficult for automated systems. There are different types of CAPTCHAs you might encounter:

- **Image-based CAPTCHAs:** Require identifying objects in images (e.g., "select all traffic lights").
- **Text-based CAPTCHAs:** Require entering distorted text shown in an image.
- **Interactive CAPTCHAs:** Involve tasks like sliding puzzles or checkbox interactions ("I am not a robot").

CAPTCHAs are a significant hurdle because they are specifically built to disrupt automated crawling.

3. Rate Limiting

Rate limiting is another common obstacle. Websites often enforce limits on how many requests a single client can make within a given time frame. If you exceed these limits, you may experience:

- **Temporary Bans:** The server may block requests from your IP for a short period.
- **Throttling:** Responses may slow down significantly, delaying crawling progress.
- **Permanent Blocks:** Excessive or aggressive crawling can lead to permanent blacklisting of your IP address.

4. JavaScript-Heavy Websites

Modern websites often rely heavily on JavaScript for rendering content dynamically. This presents two key issues:

- **Hidden Data:** The content may not be present in the initial HTML and requires JavaScript execution to load.
- **Infinite Scrolling:** Some websites use infinite scrolling, dynamically loading content as the user scrolls, making it challenging to reach all URLs.

Traditional crawlers that do not support JavaScript rendering will miss much of the content on such sites.

5. Anti-Bot Measures

Some websites employ sophisticated anti-bot systems to deter automated crawling:

- **Honey Pots:** Hidden links or fields that bots might follow but human users wouldn't, revealing bot activity.
- **Session Validation:** Enforcing user authentication or checking session integrity.
- **Fingerprinting:** Analyzing browser fingerprints (e.g., screen resolution, plugins, and headers) to detect non-human behavior.

6. Dynamic URLs and Pagination

Dynamic URLs, created using parameters (e.g., `?id=123&sort=asc`), can make crawling more complex. Challenges include:

- **Duplicate URLs:** The same content may appear under multiple URLs with slightly different parameters.
- **Navigating Pagination:** Crawlers must detect and follow pagination links to retrieve all data.

Here's a summarized table of the challenges and solutions for crawling URLs:

Challenge	Description	Solution
Blocking	Websites detect bots by monitoring IP addresses, user agents, or request patterns.	Use **proxies** and **IP rotation**, spoof user agents, and randomize request patterns.
CAPTCHA Challenges	CAPTCHAs prevent bots by requiring tasks like solving puzzles or entering text.	Leverage **CAPTCHA-solving tools** (e.g., 2Captcha) or use services like **Scrapfly** for bypassing.
Rate Limiting	Servers restrict the number of requests in a given time frame, causing throttling or bans.	Add **delays** between requests, randomize intervals, and distribute requests across proxies.
JavaScript-Heavy Websites	Content is loaded dynamically through JavaScript or via infinite scrolling.	Use tools like **Puppeteer**, **Selenium**, or Scrapy with **Splash** for JavaScript rendering.
Anti-Bot Measures	Advanced systems detect bots using honeypots, session checks, or fingerprinting.	Mimic human behavior, handle sessions properly, and avoid triggering hidden traps or honeypots.
Dynamic URLs	URLs with parameters can create duplicates or make navigation more complex.	Normalize URLs, remove unnecessary parameters, and avoid duplicate crawling with canonicalization.
Pagination Issues	Navigating through pages of content can lead to missed data or endless loops.	Write logic to detect and follow pagination links, ensuring no pages are skipped or revisited.

This table provides a clear, concise overview of crawling challenges and their corresponding solutions, making it easy to reference while building robust web crawlers.

Addressing these challenges is essential for building resilient crawlers. Tools like **Scrapfly** can simplify the process and enhance your scraping capabilities. Let's explore how Scrapfly can power up your efforts.

FAQ

To wrap up this guide, here are answers to some frequently asked questions about Crawling Domains.

Is it legal to crawl a website?

Yes, generally crawling publicly available web data is legal in most countries around the world, though it can vary by use case and location. For more on that, see our guide on Is Web Scraping Legal?.

Can my crawler be blocked and how to avoid blocking?

Yes, crawlers are often blocked by websites using various tracking techniques. To avoid blocking, first start by ensuring rate limits are set on your crawler. If that doesn't help, various bypass tools like proxies and headless browsers might be necessary. For more, see our intro on web crawling blocking and how to bypass it.

What are the best libraries for crawling in Python?

For HTTP connections, httpx is a great choice as it allows for easy asynchronous requests. Beautifulsoup and Parsel are great for HTML parsing. Finally, Scrapy is a great all-in-one solution for crawling.

Conclusion

In this guide, we've covered how to find all pages on a website using web crawling. Here's a quick recap of the key takeaways:

- **Understanding URL Types**: Differentiate between internal, external, absolute, and relative URLs.
- **Building a Crawler**: Use tools like `BeautifulSoup` or Scrapy to extract URLs.
- **Overcoming Challenges**: Tackle rate-limiting, JavaScript, and anti-bot measures with proxies, delays, and rendering tools.
- **Leveraging Tools**: Streamline crawling with Scrapfly for CAPTCHA bypass, JavaScript rendering, and proxy rotation.
- **Ethical Crawling**: Follow `robots.txt` rules and comply with legal guidelines.

Whether you're a developer or prefer no-code solutions, this guide equips you with the knowledge to crawl domains responsibly and efficiently.

Check out ScrapFly Python SDK

Try ScrapFly for FREE!

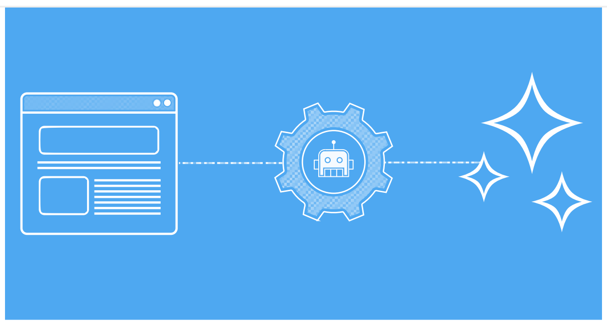
Related Questions

- How to get file type of an URL in Python?
- How to find all links using BeautifulSoup and Python?
- How to ignore non HTML URLs when web crawling?
- What's the difference between Web Scraping and Crawling?

More >

WEB CRAWLING PYTHON

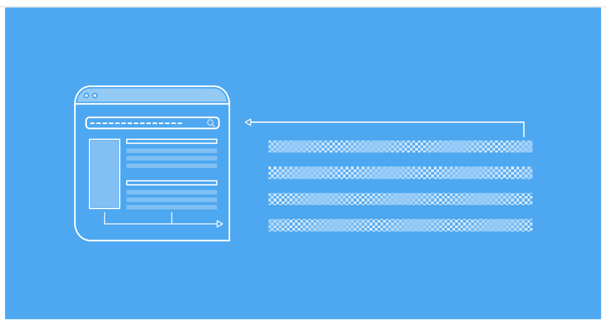
Related Posts



Mar 20, 2025

GPT Crawler: The AI Training Data Collection Guide

Learn how to use GPT Crawler to collect web data for AI training. A developer's guide with setup tips, configuration steps, and best practices.



Mar 10, 2025

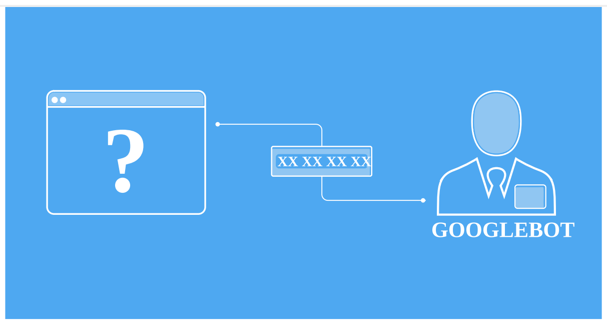
Guide to List Crawling: Everything You Need to Know

In-depth look at list crawling - how to extract valuable data from list-formatted content like tables, listicles and paginated pages.

AI WEB CRAWLING

WEB CRAWLING BEAUTIFULSOUP

PYTHON



Jan 29, 2025

What is Googlebot User Agent String?

Learn about Googlebot user agents, how to verify them, block unwanted crawlers, and optimize your site for better indexing and SEO performance.

WEB CRAWLING SEARCH-ENGINE

SEO

Company

Careers
Terms of service
Privacy Policy
Data Processing Agreement
KYC Compliance
Status

Integrations

Zapier
Make
N8n
LlamaIndex
LangChain

Social

Tools

Convert cURL commands to Python code
JA3/TLS Fingerprint
HTTP2 Fingerprint
Xpath/CSS Selector Tester

Resources

API Documentation
Web Scraping Academy
Is Web Scraping Legal?
Web Scraping Tools