

Project 2: Web and Content Delivery

Parts 1–3 Due: February 18 11:59pm PT Parts 4–5 Due: February 23 11:59pm PT

Introduction

In this project, you will instrument a headless web browser to collect the **DOM tree** of top websites on the Internet.

From there, you will document the technical services and resources that make up popular websites, ultimately uncovering the technology providers that make up the modern web.

You will leverage the Linux virtual machine (VM) that you were provided in Project 1 to crawl websites using Headless Chromium, a version of the Chromium browser that can be programmatically manipulated. You will need to instrument this browser using existing browser instrumentation tools (i.e., Playwright) to crawl a list of websites we provide you, collect metadata about them, and answer questions about how modern websites function.

Internet Measurement Caveats

As in the previous assignment, you'll find that questions about the Internet do not have a single exact or correct answer, as the real-world Internet is nuanced and constantly changing. Thus, it is important that you and your teammates describe the methodology you use to answer each question in addition to providing your final answer.

The Ethics of Web Crawling. In this project, you will be making *real* requests to websites using a crawling infrastructure that you build. As with any active scanning methodology, there are many ethical considerations at play. For this project, you **MUST ONLY** crawl websites that are in the list we provide to you, as these are very large, popular websites for which the traffic you generate is negligible. You are permitted to scan multiple times, but see that you limit this to the best of your ability; too much unwanted traffic could get your IP address blocked by websites you are trying to reach and may inhibit your ability to complete the assignment.

The Importance of Testing And Starting Early. At one part of this project, you will be asked to run a web crawl on a set of 1000 websites. These crawls may take a significant amount of time—on a single core, running a crawl of 1000 websites with a timeout window of up to 30 seconds per website can take upwards of 3 hours. You want to ensure that your code is generating everything you will need before running on the entire set of 1000 domains. We recommend you test your code thoroughly, and run it through smaller inputs (e.g., the first 25 domains) to make sure you are catching any potential edge cases that you may not anticipate.

Web Background

In this section, we detail the necessary background to understand the components of websites we'll be investigating and the high-level function of web browsers.

Web Pages and Resources

When your browser makes an HTTP request to a website, it will first download the HTML content of the page. The browser abstracts the HTML content as a Document Object Model (DOM), which is a nested tree structure where each object is part of the larger HTML content. Typically, websites will embed many different types of resources in the DOM, such as images, CSS, and JavaScript, and many of these will be served *remotely*, meaning your browser will need to fetch these resources via HTTP to properly render the page. These are called *web resources*.

Web resources are typically retrieved over the Internet, and are thus served from a myriad of ASes and locations around the world. In this project, you will be investigating the types of resources modern websites rely on, where those resources come from, and the implications of your findings on the web at large.

Web Browsers

Web browsers are the primary way that people interact with the web. They handle all the network-side and client-side code required for websites to properly function. Chrome, in particular, exposes an API called the Chrome Development Protocol (CDP) that enables developers to *instrument* the browser to perform automated tasks. For example, through CDP, developers can automatically visit webpages, interact with web elements, and log all requests and responses made during a page visit. Essentially, CDP exposes almost everything you can do as a regular user with your browser, just programmatically.

Part 1: Collecting Web Data

In the first part of the project, you'll need to instrument a web browser to log HTTP response traffic and annotate each resource with metadata, such as the URL it's served on, its content type, and the AS that serves it.

Required Software

For this project, you will be writing Python. Do not be alarmed if you do not have experience programming in this language before, as the assignment does not rely on a deep knowledge of the language itself. We will be using two main software packages for this project:

1. **Playwright**¹ is a library that enables you to instrument a headless web browser (e.g, Chromium) to perform various browser tasks. Playwright is a lightweight wrapper around the Chrome Development Protocol. You can use Playwright to take screenshots of webpages or automate large scale web-application testing. We will be using it to measure the complexity of modern webpages. While the starter code we provide you is written in Python, Playwright is also available for Nodejs, Java, and .NET. If you prefer, you may use those environments instead.
2. **Pyasn**² is a Python library that enables fast lookups from IP addresses (or IP blocks) to the ASes that serve them. It can interface directly with the MRT that you collected from Project 1, or you can use it to download a fresh routing table from the RouteViews project.

Setting up your VM for Web Crawling

To set up your cs249i VM for web crawling, you'll need to install a few packages and libraries.

¹<https://github.com/microsoft/playwright>

²<https://github.com/hadiasghari/pyasn>



Figure 1: **Playwright Flowchart**

First, in order to connect to the internet on your VM without having to run `gobgp`, run:

```
cs249i-student@cs249i-group-XX:~$ sudo ip route add default via 171.67.69.1
```

This will establish a default route to the upstream router without needing to read it in from an active BGP connection. You won't be able to advertise other IPs that belong to you anymore, but that's okay; you only need your own IP for this project.

To begin, download all the prerequisite files onto your VM with the following commands:

```
cs249i-student@cs249i-group-XX:~$ mkdir project2 && cd project2
cs249i-student@cs249i-group-XX:~$ wget https://cs249i.stanford.edu/projects/p2/p2.tar.gz
cs249i-student@cs249i-group-XX:~$ tar -xvf p2.tar.gz
```

These commands download a few files to your VM: `setup.sh`, `starter_code.py`, and `domains.py`. We will explain each in detail below.

To download all the prerequisite software, run the following command from the same directory you downloaded the starter code to:

```
cs249i-student@cs249i-group-XX:~$ chmod +x setup.sh
cs249i-student@cs249i-group-XX:~$ ./setup.sh
```

The setup script will download Playwright, the necessary libraries for the headless browser to function and `Pyasn`. When Playwright is installed, it also downloads a recent version of the Headless Chromium browser to your VM, which is how the library exposes the web crawling APIs it provides. You should ensure you have properly installed all the prerequisite software before proceeding.

Important note for running Python scripts: `setup.sh` installs the Python dependencies you need for this project in a virtual environment in the directory you execute it. So to run `starter_code.py` (or any other Python script you use), first activate the virtual environment by running the Linux command `source bin/activate` within the virtual environment directory. After activating, you can also install additional Python packages with `pip install <package name>`. You will need to activate the environment every time you log into the VM.

Instrumenting the Browser

In order to instrument the browser, you will need to write some Python code that leverages Playwright to collect metadata about the resources loaded when you visit a page. Figure 1 shows how your code will eventually interface with the browser. You will run your code via the Python runtime, which will have direct access to Playwright and the headless Chromium binary. Playwright exposes an API³ that allows you to directly interface with the browser via the Chrome Development Protocol (CDP), which you will use to collect the information you need from page loads. Specifically, you can look in the `HTTPResponse` fields to collect the relevant metadata.

³<https://playwright.dev/python/docs/api/class-playwright>

We provide you with some starter code, `starter_code.py`, which contains a scaffold for browser instrumentation. Your main objective is to extend this script to collect information about every HTTP resource loaded by the pages you visit. The code looks something like this:

```
from playwright.sync_api import sync_playwright
import json

# TODO: Populate domains
domains = ["https://cs249i.stanford.edu/"]

def crawlSite(site):
    with sync_playwright() as p:
        browser = p.chromium.launch()
        context = browser.new_context(
            user_agent="Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36
            (KHTML, like Gecko) Chrome/85.0.4182.0 Safari/537.36"
        )
        page = context.new_page()

        outObj = {}
        outObj["site"] = site

        # TODO: Populate these fields
        outObj["site_ip"] = "TODO"
        outObj["tls_version"] = "TODO"
        outObj["status_code"] = "TODO"
        outObj["time"] = "TODO"
        outObj["resources"] = []
        outObj["sri"] = []

        def handleResponse(response):
            # This event is fired when the browser detects an HTTPResponse
            responseObj = {}
            responseObj["url"] = "TODO"
            responseObj["ip_address"] = "TODO"
            responseObj["headers"] = "TODO"
            outObj["resources"].append(responseObj)

        page.on("response", handleResponse)

        page.goto(site)

        browser.close()
        return outObj

'''
README: runOverDomains() a wrapper function that runs crawlSite on each domain.
So long as the "domains" variable is populated correctly, you should not need
to modify anything in this function in order to get the script to work.
'''

def runOverDomains():
    for site in domains:
        try:
            print(json.dumps(crawlSite(site)))
        except Exception as error:
            print(error)

if __name__ == "__main__":
```

```
runOverDomains()
```

At a high level, the starter code orchestrates the headless browser (Chromium) to visit <https://cs249i.stanford.edu/> and logs information about the page visit. To do this, it loads the Playwright library, and runs two functions, `runOverDomains()` and `crawlSite(site)`.

`runOverDomains()` is a wrapper function that calls `crawlSite` and prints out a JSON blob of the returned result. You should not have to modify this function at all in order to capture the appropriate fields.

`crawlSite(site)` contains the main logic of the script and is where you will have to add additional code. Specifically, you will need to leverage the Playwright API⁴ to fill in the fields marked `TODO` in the starter code. We have already written the code required to capture *when* an `HTTPResponse` event is fired by the browser, all you need to add is how to collect the relevant metadata about each resource.

In summary, your code should collect the following pieces of metadata about the website:

1. The IP address of the root page of the website you are crawling
2. The TLS version used by the root page
3. The HTTP status code of the root page
4. The time it takes to fetch the root page

And for each resource:

1. The URL of the remote resource
2. The IP address of the remote resource
3. The headers of the remote resource

You'll also need to design and implement a methodology for determining when a website uses Subresource Integrity (see Part 4).

Tagging Resources with Autonomous System (AS)

After collecting a JSON object containing resource metadata for <https://cs249i.stanford.edu/>, you will then tag each resource with additional metadata, including the AS that the resource was served from.

For AS tagging, we recommend using `Pyasn`, a simple Python library that contains mappings from IP addresses to ASNs. You can reuse the `asn_names.txt` file we provided you from the previous project to also tag each resource with the name of the AS that serves it. `pyasn` can read the MRT file you collected from the previous project, or you can simply download a recent RIB file from RouteViews⁵ using the tools provided by the `pyasn` library.

Geolocating Web Servers

There exists an entire industry that attempts to map the geographic location of every IP address. While quality of these datasets varies widely, datasets can typically map IPs to the country that uses them accurately. To geolocate where resources are loaded from, you will use a free dataset from `IPInfo` that we have provided in the project directory (`country.mmdb`). To use the database, you can use an `mmdb` library like: <https://github.com/maxmind/MaxMind-DB-Reader-python>. You'll need to geolocate IP addresses of all resources for each site.

⁴<https://playwright.dev/python/docs/api/class-playwright>

⁵<http://www.routeviews.org/routeviews/>

Evaluating Your Final Output

To test your final output, we provide a file, `cs249i_output.jsonl`, that contains the JSON output from our solution crawler on a single domain (the course website). You should check whether or not you received the same resources (though the order and IPs may not match exactly).

Part 2: Crawling the Web

In this second part of the project, you will take the web crawler you built in Part 1 to crawl the top 1000 websites on the Internet, using data from Google Chrome's CrUX dataset. You can download the data from: <https://github.com/zakird/crux-top-lists>.

You should modify your Python crawler to crawl these 1,000 websites. Note that some websites may produce *different* resources when you crawl them multiple times. This is expected, and a product of how dynamic the web has become. You do not need to crawl websites multiple times, but know that doing so may give you slightly different snapshots of the web. Additionally, a small handful of top websites may never load in your browser—this may be due to a number of factors, but for the scope of this project, please use "UNKNOWN" as a placeholder for any data or IP addresses that fail to return. Do not exclude domains from your final output. You will want to log these errors and sanity check them after your crawl has completed (see Part 3 below).

Your objective will be to generate a single **JSON Lines** file called `output.jsonl` that contains full resource metadata for every website in the list, delimited by a newline. This will help with grading and your result will enable you to answer the questions listed in Parts 3 and 4.

Submission

Please upload the following files to Gradescope:

- `output.jsonl`: the file that contains the resources for all 1000 websites.
- `starter_code.py`: your modified starter code.
- Any other scripts you wrote (e.g., Python scripts to tag ASes)
- `README.txt`: A readme file that describes the files you submitted.
- Your write-up for Part 3.

After running your crawler, you should have 1000 valid JSON Lines in your output file. You are responsible for checking that your output parses accordingly.

Part 3: Website Delivery

We will first investigate whether your measurements were successful and start building some basic familiarity with how websites are delivered to you.

1. Active measurements can be error-prone. Your experiment might overload your upstream network, your measurement server might not be able to respond to the data it receives quickly enough, or your measurement instrument might be blocked by destination networks or websites. Let's first validate that the data you collected is reliable.
 - (a) What percent of sites were you able to successfully collect (i.e., received a valid HTTP response)?
 - (b) For the sites that failed to respond over HTTP, what is the breakdown of errors for those sites?
 - (c) Because there's no ground truth in Internet measurement to compare against, we heavily rely on spot checking to try to convince ourselves that the data we have collected is valid. Pick 10–20 of

the sites that failed to load. Do these websites load successfully in a normal web browser? What percentage of the results do you estimate are correct based on your investigation of those sites?

- (d) For each of the error types in (b), do you think that the error is due to: (1) how you performed your measurement, (2) what the website returns to everyone, or (3) a mixture of both? You might have to visit several websites with those errors and/or attempt to perform DNS lookups to understand.
 - (e) Sometimes websites that load successfully over HTTP are unsuccessful *within* the HTTP protocol. For example, you might receive a valid HTTP 400 **Bad Request** error back from a server when you made your request. What is the breakdown of HTTP error codes that you receive back?
 - (f) Do you think that your data is reliable for understanding the web? Why or why not? How would you caveat your results?
2. Now let's look at how the successfully loaded websites are delivered to you. Please describe your methodology.
 - (a) What percentage of websites do you estimate support HTTPS? (Note: you may need to re-crawl some of the sites in the CrUX list with varying HTTP and HTTPS settings depending on how they are listed. For ease of grading, though, please only include the CrUX top 1000 in your submitted output file.)
 - (b) What is the breakdown of TLS versions that websites use?
 - (c) **Content Security Policy (CSP)** is the preeminent method for protecting against XSS attacks. How many websites deploy CSP?

Part 4: Understanding Web Resources

In this section, we will investigate the most common resources that power the modern web. Please share your methodology (a high-level description will suffice) for answering each question.

1. Let's start by looking at the volume of resources delivered to you.
 - (a) How many resources, on average, are loaded per site? How does this compare to the median number? What does this difference tell you about the web ecosystem? (Hint: You can check your intuition by graphing resources per site.)
 - (b) Let's look at the outliers. What are the top ten sites that load the greatest number of resources, and how many resources do those sites load? Why are so many resources loaded for these top sites? Consider the types of resources. Build a table with the site, number of resources, and a short explanation of what's going on.
 - (c) Internet measurement researchers are obsessed with plotting results as CDFs (Cumulative Distribution Function) instead of as histograms or as PDFs (probability density function). Plot the total size (i.e., content length) of websites as a CDF. Based on your CDF, what is the median size of a website? What about the 95th percentile?
2. Next, let's start to look at what all these resources are.
 - (a) Investigate the content type of each resource loaded. What are the most common resource types across all websites in your crawl? What fraction of sites use JavaScript?
 - (b) How commonly do you see a website loading the same resource multiple times? Design a graph that supports your answer: define your axes, and describe what you interpret from the results.
 - (c) What are the top 10 resources (as defined by file name) included by the most websites in the Top 1000, and how frequently do they occur?
 - (d) What function do these most commonly included resources serve?

3. Now let's focus on third-party resources. For this question, consider a third-party domain to be one that is not equivalent to the domain of the website you visited based on the Mozilla Public Suffix List.⁶ For example: `google.com` and `blog.google.com` belong to the same first-party domains, whereas `google.com` and `googledomains.com` are different domains.
 - (a) How many websites depend on third-party content (i.e., resources from a third-party domain)?
 - (b) Investigate the third-party domains most widely depended on. What are the top ten sites relied on for third-party content? Based on the included resources, what role does each of the top ten most relied on sites play in the web ecosystem? Provide a table that includes the depended-on site, the number of websites that rely on it, and its role in the ecosystem.
 - (c) How widely adopted is **Subresource Integrity**?
 - (d) What attack can occur when SRI is not used and resources are loaded from third parties?
4. Now, we'll zoom in specifically on ads and trackers. You can use the lists published at <https://easylist.to/> to help answer this question.
 - (a) What percentage of websites display ads?
 - (b) Who are the ten most popular ad providers?
 - (c) Anecdotally, what patterns do you see emerge in the *types* of websites that deploy a large number of ad providers and/or a large number of ads?
 - (d) What percent of sites deploy tracking code?
 - (e) Who are the top ten trackers used in practice? Based on the sites you have crawled, how much of the web does these trackers have visibility into?

Part 5: Networks and Geographies

In this section, you will investigate the *networks* that the web relies on the most (versus the *domains* most relied on in Part 4). Please share your methodology (a high-level description will suffice) for answering each question.

1. First, we'll look at websites' reliance on major hosting providers.
 - (a) What are the top 10 ASes that are responsible for *hosting* the most websites in your crawl? How many websites does each top AS host? Provide a table with AS number, name, and number of websites hosted. (Hint: what is the IP address of the root page of the website itself?)
 - (b) What is the total percent of websites hosted by Amazon, Cloudflare, Fastly, and Google?
 - (c) What is the average load time for pages on each of the providers in part (b)?
 - (d) What is the average load time for sites *not* hosted by the providers in part (b)? Compare with your answer to (c).
 - (e) Now let's look at third party providers. What are the top 10 third-party ASes that websites rely on to serve the most resources, and what fraction of websites do they appear on? Consider a third-party AS to be one that is not equivalent to the AS that hosts the website.
 - (f) What are the implications of having websites increasingly rely on a small number of ASes to serve and host content? (You may ground part of your answer in your takeaways above, but we encourage you to think more broadly as well.)
2. Third-party resources can also take network traffic to unexpected geographic destinations, particularly for users who live outside of the United States. Using your geolocated IP addresses, let's look at where your network traffic goes.

⁶<https://publicsuffix.org/>

- (a) What percentage of websites' root pages in your dataset are served from the United States?
- (b) What percentage of websites in your dataset have *any* resource served from the United States?
- (c) What percentage of websites have resources served from a different country than the root page?
- (d) What are the top 10 countries that third-party resources are most frequently loaded from?
- (e) What countries are third-party resources loaded from when they're served from a different country than the website?
- (f) What three third-party sites most commonly cause network traffic to go to another country than the main website?
- (g) Find the 10 countries that serve the most websites' root pages. Make a stacked bar chart showing where websites from each of these countries load their resources from. Interpret what you see. (Note: For simplicity, you may want to only plot the top countries using your answer from part (d) plus "Other" as needed.)