API Caching Using httpantry

CS701 Final Project

Rowen Felt
Middlebury College
Middlebury, VT
rfelt@middlebury.edu

Zebediah Millslagle
Middlebury College
Middlebury, VT
zmillslagle@middlebury.edu

ABSTRACT

Developing and testing software can incur significant monetary, network traffic, and time costs due to repeated API requests. To alleviate these costs, we have created a Python package, `httpantry`, that contains two tools for caching API responses: a proxy server that can be deployed as a standalone process, and a wrapper library for the Python Requests library. The proxy server can be implemented in any language or framework that supports redirection through a proxy server, while the wrapper library can be imported into any Python project with a single line of code. These tools are highly configurable through modification of a simple configuration file, which can be used to specify persistence of the database between sessions, a timeout option to validate responses, a list of remote resources to never cache, and can be used to create custom responses for specific APIs. Both tools utilize the same underlying infrastructure, including the database, configuration file, and custom response file, to resolve, cache, and return responses for API requests. These tools can be installed, configured, and deployed in seconds using a simple command line interface.

Keywords

API; HTTP; Web Development; caching; proxy server

1. INTRODUCTION

Many software products, especially tools like data analytics dashboards, utilize a large number of external resources. Developing this software requires continuous testing, and these tests often involve accessing remote resources. If the remote resource transmits a large amount of data, takes time to render, or costs money, developers can incur a substantial cost during testing in terms of money, time, or bandwidth. In some cases, a developer might circumvent these obstacles by downloading some of the remote resources they need for testing purposes, but this approach can require making substantial changes to software during testing. Developers may also be hindered when they require a specific API endpoint for testing purposes that has not yet been implemented. To help offset these costs, we set out to make a set of API caching tools to store and return previously used responses from remote API requests, and allow a user to easily create and use custom responses for API requests whose endpoints have not yet been implemented. Furthermore, we endeavored to make these tools as easy to install and configure as possible and allow a developer to easily incorporate them into software during development with little or no change to their codebase.

2. PROBLEM STATEMENT

All web applications make use of remote resources stored on servers. These resources could be sources of data such as images, datasets, or html webpages; or, these resources could have computational components, such as messaging services or data processing tools. Software engineers interface with these resources through Application Programming Interfaces (APIs). Most APIs are accessed via HTTP requests sent to servers hosting a given service. In the case of data resources, an API request can be an expensive networking operation, and in the case of computational resources, API requests can be time intensive or monetarily expensive if they are hosted by external commercial services. Software products are optimized to limit API requests to minimize the cost to the consumer.

However, when developing web applications, software is subject to continuous testing. This testing often involves making these API requests continuously, sometimes dozens or hundreds of times a day by a single engineer. If these requests incur a networking cost or monetary cost, hundreds of requests a day by an entire company of engineers can lead to an unnecessarily expensive testing situation.

To offset this cost, we proposed creating a tool that can intercept API requests and cache responses from a given set of services. We identified two ways of accomplishing this goal. One method involves creating a proxy server that can receive any API request and either return a stored response, or resolve the request using the real resource and store the response for later use. The second method involves writing a wrapper class for a common library used to make requests, in this case the Python Requests library, and create instances of common functions that perform the same persistent caching and resolving operations before any request has left the calling application. Both approaches have their merits.

There are many uses for a proxy server, such as examining the body of a request in testing, using a proxy as a testing tool for a set of defined services with determined responses, or examining networking traffic between cache layers. Additionally, if the proxy server runs as a standalone process, then it is possible to accept configuration changes on the fly via cURL requests without having to restart the application being tested. The same cannot be said for a wrapper library. Any configurations of the wrapper library class will be persistent for the lifetime of the application in a given test cycle. However, the wrapper library has the added security

benefit of allowing end-to-end API encryption via HTTPS and eliminates the chance of intercepting data between the application in test and the proxy server. We endeavoured to create both solutions to the described problem.

We successfully created a proxy server capable of resolving any HTTP API request and caching the response. The proxy server is highly configurable and can be configured on-the-fly via a "/config" API. We also created a wrapper library for the Python Requests library that can resolve HTTP and HTTPS requests and cache the return responses. Both the wrapper and proxy server cache responses as binary blobs in a persistent SQLite database, and both services can be customized in a variety of ways, such as preformatted responses, response timeouts, persistence, and uncached URLs.

3. RELATED WORK

Many companies likely have a proprietary set of internal tools that implement the caching behavior we described, and there are several open source iterations in the wild.

Wiremock[1] is a simulator for HTTP based APIs. Wiremock allows users to create custom responses for services that have not been implemented. It can also be used to capture traffic to and from an existing API and cache it for future testing.

Hoverfly[5] is a service virtualization tool that can replace testing provisions in continuous integration environments with on-demand simulations. Hoverfly also allows engineers to test systems that rely on 3rd party APIs.

These "stub servers" perform, among other things, the task we are trying to replicate: returning configured responses to API calls. A similar service known as SpyREST[8] was developed to intercept API requests via a proxy server and uses an example-based model to dynamically generate API documentation.

All of these services are well formulated responses to the same problem we were attempting to solve, and we want to specify that we didn't plan on producing a more robust implementation than these services provide. Our goal was to build a much lighter weight tool than these services provide, ideally a tool easily configured and deployed in minutes. Our goals also differed from these services in that our ideal solution would not impact the code of the application in test in any substantial way. For example, our wrapper code uses the same prototypes as the Requests library implementation of the same functions, so the transition from using our testing cache to production would only involve changing the import of our library to the import of the real Requests library. We also wanted to build this set of tools to learn more about client-server interactions, networking, and building adaptable tools.

4. METHODS

In planning our implementation, we considered a number of approaches. One possibility we considered was creating a command line tool that would simultaneously run an HTTP server and modify the `/etc/hosts` file. A user could modify a configuration file to designate a number of hosts whose APIs will be used. All requests to those hosts would be proctored through our software. If the HTTP request matches a stored API, then the canned responses would be returned by a call to our database. Upon shutdown, the `/etc/hosts` file would be returned to its original state.

This approach was problematic for a number of reasons. As a development tool, the process should be killable without involving any shutdown procedure. Killing this specific process would almost certainly leave the `/etc/hosts` file in an inconsistent state. Although we did not follow this approach, it did lead us towards our true implementation. Our study of Wiremock also gave us some ideas of solutions we might want to offer within our own product, such as the ability to deploy our cache as a standalone process. We later decided that the best solution would be to create a proxy server that could receive HTTP API requests, resolve unknown or unvalidated requests, cache the responses for further use, and return the formatted response object to the client. This approach would take advantage of the proxying feature of most HTTP request libraries. We were able to build this tool using a Flask[7] microframework server.

Flask allowed us to create variable server routes that could redirect any given URL. Once the request is received, the URL and RESTful method are used as a compound primary key for lookup in a persistent database. If a serialized response object is found with a valid timestamp, the deserialized response object is returned. If the query fails or the returned object has an expired timestamp, the request is resolved by querying the specified remote resource. The resource's response is serialized as a binary blob, stored in the persistent database with a current timestamp, and the formatted response object is returned. All API querying within our proxy server is done using the Python Requests[6] library.

We cached our responses in a SQLite database using the target URL and RESTful method type as a compound primary key. The stored fields include timestamp, an epoch integer timestamp, and response, a serialized binary blob containing the response object. We used the python sqlite3[2] library to create an SQLite session within our Flask server. As our application only uses one database schema, we were able to write generic `store()` and `retrieve()` functions that we also used in unit tests for our server. We used pickle[4] to quickly and easily serialize and deserialize the request objects as binary blobs.

To create a more customizable application, we implemented a user configuration system to read a variety of user configurations from a file. We used the Python configparser[3] library to read configurations from a config file or to create a config file with default values if none is present. The values are read and parsed using a configparser object and stored in an instance of a Configuration object we implemented. The config file is parsed every time the proxy server application runs and every time the `httpantry` library is imported, and the returned Configuration object is globally accessible within the main server application. The object contains fields such as:

- persistence: indicates whether or not the cached responses should be stored persistently or in a temporary object.
- uncached apis: a list of APIs that should be resolved upon every request rather than cached.
- port number: the specified port number on which the proxy server should run.

- response file: a path to a file containing pre-configured responses and their associate URLs and RESTful methods encoded as JSON objects.
- timeout: an integer representing a keep-alive time for a given cached response in milliseconds.

The proxy server can also be configured on-the-fly by curling POST requests to the /config API route with a JSON body containing the configuration encoding and parameter. Configuration changes implemented in this manner are not persistent. Persistent configuration changes can only be made by modifying the config file.

Unfortunately, our proxy server was unable to cache HTTPS API requests because HTTPS requires end-to-end encryption. We could work around this by playing key and certificate signing games, but that approach would require the developer to jump through too many hoops. Instead, we returned to one of earlier ideas and developed a wrapper library for the Python Requests library. We developed our own version of the Requests user-facing API methods and utilized Python's `__getattr__()` magic method to redirect unimplemented methods to the true Requests library methods. Our wrapper functions take the same arguments as the Requests library methods, but they also incorporate the same API caching logic, configurations, and custom responses as the proxy server. Because an application using our wrapper library is caching its own requests and responses, the wrapper library allows a developer to cache HTTPS API responses during development.

Although the proxy server does not cache HTTPS responses, it does offer several advantages over the wrapper library. The proxy server is essentially a web server, so it can work with any HTTP request library in any language or framework that supports proxy servers. The wrapper library is limited to the Python Requests library, and even further limited to the specific version of the library supported by our wrapper functions. The proxy server can also be configured dynamically, while the wrapper library can only be configured using the config file at run-time.

We then packaged our project into an easily installable and configurable Python pip package. The pip package offers several ways of interacting with the proxy server and wrapper library through a simple command line interface.

Here is a link to our GitHub repository:
https://github.com/RowenFelt/httpantry
Here is a link to our pip package:
https://pypi.org/project/httpantry

5. RESULTS

We tested our proxy server by making a series of HTTP requests through the Python Requests library, one using the proxy and one without the proxy. We then compared the content of the returned responses. The server logs showed that it resolved each request using the specified URL. We then ran the tests a second time, and the server logs showed that it returned a stored response in each case. The content of the responses returned by the proxy server matched the content of the responses returned by the specified URL. These tests indicate that the proxy server correctly caches and returns responses as intended.

We then tested each configuration of our proxy server.

To test the timeout configuration we made a series of requests through our running proxy server. The server showed that each request was resolved using the specified URL. We then waited an amount of time greater than the specified timeout configuration and performed each request again using the proxy server. The server logs showed that each request resulted in a cache hit, but an invalid timestamp caused each request to resolve using the specified URL. These tests demonstrated that the timeout configuration worked as intended.

To test the persistence configuration, we made a series of requests through our running proxy server, setting the timeout configuration to several minutes. The server logs indicated that each request was resolved to the specified URL. We performed the same requests again, and the server logs showed that it returned a stored response in each case. We then queried the response database directly and found that it did not contain any stored responses. These tests demonstrated that the server was not storing the responses in a persistent database.

To test the uncached APIs configuration, we included the API "http://httpbin.org/image/jpeg" in the config file under uncached APIs. We set persistence to True. We then made a GET request to "http://httpbin.org/image/jpeg". The proxy server returned a JPEG object. We then queried the response database directly and found that it did not contain a stored response for the specified URL. This test demonstrated that the server was not storing responses from APIs indicated in the uncached APIs configuration.

To test the response file configuration, we included a path to a file containing API URLs, methods, and responses stored as JSON objects. We then made a request through the proxy server to an API specified in the response file. The server logs showed that it returned a stored response, and the returned response body matched the response contents of the formatted response in the response file. This test demonstrated that the server was caching and serving responses included in the pre-configured response file.

We also tested these same configurations using curl requests to the /config route in the proxy server. We included print statements in the configuration route to log dynamic changes to the user-configured parameters. We sent a series of requests to the server and noted that the server logs showed each configuration being correctly processed.

We similarly tested the Python Requests wrapper library by making several HTTPS requests. We then queried the requests database correctly for the URL and specified methods. The returned response objects indicated that the wrapper library was correctly caching HTTPS responses. For a demonstration, we created a simple web application that requests a large image from a remote resource (88 MB in size) and timed how long the request took to resolve. We then installed the `httpantry` package and changed the import statement to use our own wrapper library. Originally, the image took 20 seconds to load, but when cached with the wrapper library, the same image could be loaded in less than 1 second.

6. DISCUSSION

Our proxy server worked for any HTTP API request, but its functionality was limited because it could not cache HTTPS responses. HTTPS requests use key and certificate signing to validate a given resource and perform end-to-end encryption on all communications between the client and the requested resource. Therefore, it is not possible to proxy

HTTPs requests without requiring substantial labor on the part of the developer. A developer would have to create a signed certificate for every route to cache and embed that certificate as an exception for each request made. As our entire project is designed for ease of use and to involve as little change in code between testing and production as possible, the steps required to proxy HTTPS requests are intractable.

Our solution to the HTTPS dilemma involved revisiting one of our earliest ideas. We decided to create a wrapper library for the Python Requests library such that HTTPS requests could be sent as the body of an HTTP request to a specific route in the proxy server. The body could then be dissected and an HTTPS request could be made between the proxy and the specified resource using the information contained in the request body. The danger here is that unencrypted data is still being sent over the network between the application and the proxy server. We then realized that if we are creating a wrapper library for the Requests library, we should use the same caching system we already implemented in the proxy server within the wrapper library itself. Using this system, HTTP and HTTPS requests could be resolved and cached using the same persistent database, user configurations, and logic as in the proxy server, but the man-in-the-middle situation can be avoided.

After testing the proxy server and wrapper library, we found that our API caching tool was working as expected. Responses were being returned correctly with fewer requests made to remote resources, the database was only persistent when specified, API responses wouldn't get cached when included in the uncached APIs list, and users were able to load custom responses into the database with a simple JSON file. While the proxy server allowed for broader applications and a few more general features, the wrapper library performed more quickly than using either the normal requests library or the proxy server, with an added guarantee of better security.

While making this tool, we decided early on to use a proxy server because Flask server are easily configured. Had we investigated further before doing so, we would have seen that problems were going to arise when dealing with HTTPS. If we began by creating the wrapper library, we may have ended up with a more robust tool. However, by starting with the proxy server and then introducing the wrapper library later on, we were able to utilize the strengths of each method, while also gaining more context for either solution. This approach provided us with a foundation of knowledge in both areas, which will be useful later on in our careers. Because a main aspect of our project was learning, working on both tools was preferable to working on either tool alone. The project provided us with many learning opportunities, and our final product aligned closely with what we set out to create. This ended up being one of the best parts of the project, it does exactly what we imagined it would do at the beginning and is implemented using methods we brainstormed initially.

While both the proxy server and the wrapper library do their respective jobs well, both have their downsides. The proxy server, while offering a more broadly applicable tool, is limited to only HTTP requests. Because many projects use HTTPS, the uses of this tool are limited. With more time and research, it may be possible to improve the proxy server so it would be able to overcome these problems in a less contrived environment. The wrapper library, while quite efficient and easy to implement, only works for the Requests library in Python. If users want to use a different language, a different library, or a different version of the library, they are unable to continue using the tool. This can be resolved by adding support for other libraries and languages, as well as keeping everything up to date with current versions of those libraries. This would require both more time in development in order to expand the options provided, as well as continual upkeep to ensure everything works with the latest versions of libraries. One other consideration is that none of the response objects in the cache database are encrypted. This could be a consideration for some users, but because our tool is only designed for development environments, we think an unencrypted database is acceptable.

One aspect of this product we prioritized was the user experience. We wanted to make this set of tools as easy to use as possible. In our ideal scenario, the developer would be able to easily install and configure the tool and be able to utilize it while making as few changes to their codebase as possible. We accomplished this by creating a Python pip package that can be easily installed from the command line. The developer could redirect API requests to the proxy server on whichever port they specify using whatever proxying system their framework allows. We also included command line tools to initialize the necessary infrastructure, flush the cache, and clean up the cache directory. Similarly, incorporating the wrapper library into a developers codebase is as easy as importing our library as the Requests library. All Requests methods used in the developer's code will then be processed through our wrapper library or redirected to the original method definitions in the Requests library.

7. ACKNOWLEDGMENTS

We would like to thank Professor Jason Grant for his advice and guidance throughout this project. We would also like to thank Professor Peter Johnson and Ruben Gilbert, who provided valuable direction during implementation, as well our fellow students in CS701 for their input throughout the semester.

8. REFERENCES

[1] T. Akehurst. WireMock, 2019.
[2] P. S. Foundation. 11.13. sqlite3 - DB-API 2.0 interface for SQLite databases, 2019.
[3] P. S. Foundation. configparser - configuration file parser, 2019.
[4] P. S. Foundation. pickle - python object serialization, 2019.
[5] B. Hooper and T. Situ. Hoverfly by SpectoLabs.
[6] K. Reitz. HTTP for Humans, 2019.
[7] A. Ronacher. Flask user guide, 2010.
[8] S. M. Sohan, C. Anslow, and F. Maurer. Spyrest: Automated restful api documentation using an http proxy server (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 271–276. IEEE, 2015.