RESTful Wrappers

Tony Vu

tony.vu@puppetlabs.com

Abstract

Web APIs are the backbone for our daily Internet life. When functioning smoothly, services go unnoticed as JavaScript, HTML, and CSS paint a magical world on top of them. However, when the services go down, everything comes to a grinding halt. In the fifteen years since Roy Fielding's dissertation on REST [1], there has been significant growth in avenues available for testing RESTful Web APIs. There is now a wealth of products and several competing platforms available to test your own API. However, with this wealth of options comes confusion, as the speed of growth makes today's testing tomorrow's ruinous remains of a once gloriously conceived testing solution.

Assembling and building your own RESTful wrappers for testing services gives you the opportunity to share your tool with other elements of your organization, such as other quality engineers or other development engineers not familiar with your product space. The upfront cost will be higher than implementing a commercial service to test your API, but the resulting library will eliminate the siloing effect of isolated test code and tools, unusable by other members of your organization. For the purposes of this paper, we will consider RESTful wrappers to be a kind of library, and will use the term "“library” and "tool" to mean "RESTful wrapper" unless otherwise specified.

This paper is organized into two sections:

- An argument for quality engineers to create RESTful Wrappers meant to be shared with cross-functional teams.
- An in-depth study of building such a library for RESTful services.

Biography:

Tony began his quality assurance career testing software as a manual tester; after learning how to automate repetitive tasks, he became enamored with the idea of automating all the things. He now spends his work hours as a QA Engineer automating tests at Puppet Labs, and spends his off hours trying to automate everything in his house.

1 Why library building matters

1.1 Quality beyond Testing

It is a virtual guarantee that in any job listing for a Software Quality Engineer you will find the word “test(ing)” peppered throughout the description. “Test Engineer” might be a better title for these people who work under the umbrella of “Quality”; it is commonly believed that it was Microsoft [2] who injected ‘test’ into a now industry standard title, Software Design Engineer in Test--or SDET--to distinguish automated test writers from manual testers. Indeed, this seems to indicate that the most sought after skill of any Quality Assurance (QA) Engineer is to produce and execute tests.

To test the veracity of this proclaimed virtual guarantee, this paper collected 25 jobs listed for a quality engineer during the month of June 2015. The listings came from a variety of sources including indeed.com, craigslist.org, and glassdoor.com. Each descriptions was analyzed for keyword frequency. In 20 out of the 25 listings analyzed, “test(ing)” was the most frequent keyword, and in 24 of the listings “test(ing)” was listed in the top three. Only one listing did not contain “test(ing)” in the top 5 most frequent keywords. This analysis--admittedly rough and course--indicates that the 2015 quality engineer market prioritizes what the American Society for Quality [3] refers to as “verification and validation”.

This should not be interpreted to mean “quality = testing”. In fact, Quality Engineering encompasses much more than “verification and validation”; one does not have to search long outside of software quality job postings to see examples of quality engineers engaged throughout the entire Software Development Life Cycle [4]. How the market/engineering organizations came to prioritize testing is not a focus for this paper. Rather, confirmation of testing’s dominance and prevalence in current software quality practices gives this paper a platform on how to expand out from a testing heavy environment and provide value beyond validation.

This paper explores expanding that value by building common libraries targeted not only for test writers to employ, but for all engineers to utilize. Specifically, this paper examines important design considerations when building wrappers for HTTP services with a RESTful architecture. Included in this examination is how Puppet Labs evaluated those considerations in the construction of their own wrappers.

1.2 Intermediary users before the end user and test result

With the advent of Agile and the rise of User Stories, tests are often conceived with QA Engineers representing the consumer. QA Engineers are almost always first to evaluate development efforts, proxying the end user. Positive test results are a confirmation that the end user’s requirements are met. However, with this emphasis on the relationship between the end user and the software, we neglect the interfaces between separate engineering teams who are often interacting within the same project. Quality testing should be concerned not just with the end user and software, but also these interfaces between disparate teams as well.

The health of these interfaces is crucial in determining the risk of issues arising from integration failures of separate software modules. Well organized engineering projects will immediately see the need for tight integration on separate but highly dependent, linked services; in these scenarios, it is unlikely that two development teams might strongly benefit from a RESTful wrapper, since the integration risk is understood and scoped early on. However, libraries can provide enormous benefits in scenarios where two teams are not highly engaged but still present dependencies that will manifest at an integration level. Despite the best planning, we cannot always foresee these interactions. By building a RESTful wrapper for interaction with other development teams, we can mitigate the cost of these unforeseen collisions. Unless there was a disaster in planning, most of these collisions are shallow in their requirements and should be easily handled with a readily available library.

A library is also an essential way to maintain cohesion as Quality organization. QA Engineers are now commonly embedded on specific engineering teams, often separated physically from other QA engineers. Developing RESTful wrappers as a Quality organization links these QA engineers back together, keeping their methodologies in sync with each other. A common library is also essential to preventing code duplication and establishing and encouraging best practices amongst the QA engineers. The library itself cannot enforce best practices or prevent code duplication, but it does provide an avenue for these behaviors to manifest.

1.3 Assessing if a RESTful Wrapper is the right solution

The previous section of this paper lays out arguments for the development of a RESTful wrapper. However, a RESTful wrapper is not always the right solution. While this paper certainly advocates for their implementation, there are situations where their creation is not warranted.

- **smaller, undivided work streams**
  - Engineering organizations can vary in size greatly; if you have a team with several sub-teams that work out of band with each other, a library may absolutely be appropriate. However, if your organization is small and the number of disparate teams is minimal, then taking the efforts to build a library may not be appropriate. One advantage of a library or tool is that an increasing number of users using it indicates that the initial upfront cost of developing it was worth it; however, this value is severely diminished if only a minimal number of teams get involved with using it.

- **shallow complexity of requirements**
  - One major goal of your tool should be to reduce the complexity for users not intimately engaged with your team. Gauge the complexity of your software module and the needs for an interface into it; there could be insufficient complexity to warrant the building of a library. Also, consider augmenting the current tooling in place; it may be that the current tool chain can be modified to suit your needs, thus not requiring a new dependency. For example, the tool cURL [5] is installed by default on nearly every Linux distribution and thus already part of the tool chain; instead of pulling in a new dependency, you may find that wrapping cURL will be sufficient for your needs.

- **questionable longevity of project**
  - Creating a library or tool requires maintenance and long standing costs; you should assess the longevity of the software module that you are proposing to build a library for. If the software module or project is at risk, you should consider taking on more temporary measures for testing that do not require such a high, upfront technical cost. This may sound like allowing for technical debt to grow. However, sometimes taking on technical debt is reasonable until you have assurances that building out the library will be supported, both financially and by the teams involved.

It should be noted that the RESTful wrappers described in this paper are generally not considered customer facing, though they may be robust enough for customer use. Their intended target audience is inward facing. However, the RESTful wrappers described in this paper could be excellent candidates for a Free and Open Source Software (FOSS) release, meant to facilitate third party integration efforts. There are several libraries available that perform this function. Some are truly third party implementations, such as the koala ruby library [6] for Facebook. Others are more official, such as the python-sdk [7] owned by Facebook itself. Determining if a FOSS release is suitable for your library is a complex decision and outside the scope of this paper's topics, but should nevertheless be considered when creating your RESTful wrapper.

2. Building out RESTful Wrappers

2.1 Choosing your dependencies

Most languages have standard libraries that support sending and receiving HTTP traffic; in some cases, these libraries may be sufficient for your wrapper's needs. In other cases, you may have to use these standard libraries because of restrictions in your test environment that do not allow for non-standard library inclusion. However, in most situations and languages, there are several competing HTTP client libraries that you can incorporate into your wrapper. Determining which of these competing HTTP libraries to build off of should be decided with great care. Consider the following factors:

2.1.1 Ease of use vs. complexity of implementation

Many HTTP client libraries are created simply because of poor interfaces into the standard HTTP libraries of the language itself. Some of the highest praise of these non-standard HTTP client libraries is in the simplicity of their implementation. Indeed, that allure can seem quite promising, encapsulating intricate logic away from your own RESTful wrapper and using a cleaner interface.

On the other hand, some libraries come with very powerful features beyond what the standard libraries can offer. Some of these features include the parallelization of requests or the ability to maintain web state--such as maintaining cookies or localStorage. These libraries tend to not sell themselves on their ease of implementation but on their relative strength and speciality when compared to HTTP libraries shipped as a standard component library.

2.1.2 Extensibility and community involvement

Some HTTP client libraries are extensible by design; other libraries are designed without this requirement. Extensible HTTP client libraries are not necessarily better than those designed to be a single solution, but that extensibility is certainly something to consider when building out your own RESTful wrapper. On one hand, managing an extensible HTTP client library and several extensions may actually be less appealing than an all-in-one, closed solution. On the other hand, choosing an HTTP client library that is extensible allows for a wider range of community support, as extensions can live in separate repositories without affecting the core client code.

Community involvement can be measured in several ways, not just in the number of extensions available. An HTTP client library that has many contributors is likely to have been vetted and approved by several sources. Likewise, an HTTP client library that has been downloaded several times suggests that it has been sufficient for several other entities, and may be sufficient for your own RESTful wrapper.

Ultimately, there is no single question that will illuminate which HTTP client library to use. For instance, it is this author’s sneaking suspicion that some libraries are downloaded frequently simply for their ease of use, and are only kept as a necessity for dependency compatibility; once embedded into a system, removing your HTTP client library will prove difficult to remove, so choose wisely!

2.1.3 HTTP client libraries used at Puppet Labs

The Ruby programming language is the test language of choice at Puppet Labs with RubyGems [8] the standard for package and library sharing. According to The Ruby-Toolbox [9], there are 10 HTTP client libraries hosted by RubyGems with over 1 million downloads. With such a variety of HTTP client libraries available on RubyGems, engineers at Puppet Labs can choose the library that best suits their needs.

Many projects at Puppet utilize the Ruby gem *HTTParty* [10]; that gem is particularly focused on improving the interface into the standard *Net::HTTP* library that comes standard with a Ruby installation, proclaiming “(HTTParty) makes consuming restful web services dead easy.” [11] However, Puppet’s utilization of HTTParty is often not for a library to be shared out with other engineers. Rather, HTTParty is employed as a shallow wrapper, mostly as a vehicle to deliver highly crafted, specific traffic to some service under test.

I initially used *HTTParty* to construct my RESTful wrapper to be consistent with the most common usage at Puppet, but found that its focus on easing simplicity into the standard *Net::HTTP* library of Ruby was no longer a compelling reason for its inclusion. Additionally, the singleton class design of *HTTParty* became constrictive in how I could construct my own wrapper; it worked well as a shallow wrapper, but extending *HTTParty* to behave in more complicated patterns and usages became too complex to feasibly maintain for the lifetime of the RESTful wrapper.

After surveying other popular HTTP client gems, I chose the *Faraday* [12] gem based on its pluggable architecture and wide set of gems compatible with it. Since *Faraday* was designed to to be highly extendable and pluggable, I knew that it was unlikely I would find the same restrictions as I did with *HTTParty*. Out of the box, Faraday lacked some of the features of HTTParty, but third party support more than made up for those absent features.

2.2 Creating the initial wrapper base layer

Once you have determined the HTTP client library you wish to use for your RESTful wrapper, you should begin construction by establishing a foundational wrapper layer of RESTful methods, each method mapping directly to an exposed endpoint of the REST service. These base methods should only expose the endpoint and any optional or necessary parameters required for successful interaction. These base methods should not alter the data returned. The response object should always be true to the response from the server and contain all the information returned from the service: headers, codes, and bodies.

How these base methods manifest in your RESTful wrapper can vary greatly. For instance, you may not wish to embed the route to the endpoint at all, forcing the RESTful wrapper implementation to provide the route as an argument at calltime:

```
//strategy 1: code of an implementation
//requiring the route
client.get('/v1/books')
```

In this example pseudo code, it is up to the caller to understand the routes that the RESTful API has exposed. At the opposite end of the spectrum, we can create RESTful wrappers that completely abstract out the routes:

```
//strategy 2: code of an /implementation requiring
//no knowledge of the underlying REST API
client.get_books
```

In this example pseudo code, the caller is not required to know the routes for the resources in question.

Both implementations listed above are valid strategies. When the implementor is required to provide the routes to the RESTful API, the underlying RESTful wrapper becomes more stable. Changes to the API are handled at the implementation level of the RESTful wrapper and not with the RESTful wrapper itself. One drawback of this implementation is that updating the routes to accommodate an update in the REST API will not follow the principle of “Don’t Repeat Yourself”, or DRY. However, a change in the REST API does not necessarily mean an update to your RESTful wrapper.

When the implementor is able to abstract away the specific routes of the RESTful API, the RESTful wrapper becomes more brittle, subject to changes in the API breaking the wrapper. However, changes to the API can be handled at the RESTful wrapper level, making updates more frequent but potentially far more DRY.

If your RESTful wrapper has a simple, non-volatile REST API it communicates with, you may want to consider strategy 1. While this may push some of the responsibility to users to update their implementations of the wrapper when the underlying REST API changes, a simple, non-volatile API should not require constant maintenance, nor should it continually fluctuate in it’s implementation. This is also a suitable strategy when your RESTful wrapper behaves more like a shim than a library, and doesn’t handle much complexity.

However, if your RESTful wrapper behaves more like a library than a shim, than you may wish to consider strategy 2. In this strategy the RESTful wrapper becomes substantially more than simply a shim; it evaluates logic and extends the interface for the REST service.

2.3 Adding abstraction on top of your base wrapper layer

Layering abstraction almost certainly requires the implementation of strategy two. In theory, it could be accomplished with an implementation of strategy one, though your code would be littered with hardcoded paths throughout all the abstraction, since it would not have a predefined set of base methods available for use and would have to constantly hardcode the paths in.

The most evident use case for an abstracted method is to utilize multiple base wrapper methods to chain together multiple calls to the server. For example, it is common for password resets to occur in two API calls to the server: one call to generate a one-time use token, and a second call passing in that token along with a new password. With only base wrapper methods, we might be forced to write something like:

```
//example 1: Resetting a user's
//password only using base wrapper methods
client_id = client.get_uuid
token = client.get_password_reset_token(client_id)
client.update_password(token, new_password)
```

However, with an abstraction for this, we could simplify it into a one liner:

```
//example 2: Resetting a user's
//password with an abstracted method
client.reset_and_update_password(new_password)
```

Chaining methods in this manner allows for us to eliminate much boilerplate code that would otherwise have to exist if we only used base wrapper methods. However, we should be careful to implement all the calls in base wrapper methods; jumping straight into a more complex method indicates an area of weakness in the testing, where certain endpoints might not have the proper coverage, and are only incidentally covered by these more abstract methods.

Beyond chaining, abstraction allows for the RESTful wrapper to implement arbitrary logic to create methods that might otherwise not exist. For instance, imagine we are dealing with a REST API that does not support filtering query parameters. If we stuck with only the base wrapper methods, we would force the implementor of the RESTful wrapper to search the results themselves:

```
//example 3: Forcing a user to search the results of a GET request
books = client.get_all_books
favorite_book = books.search(“Huckleberry Finn”)
```

A RESTful wrapper could easily support that functionality, implementing the logic to search for the book title in an abstracted method itself:

```
//example 4: Filtering a result on the client side
favorite_book = client.get_book_by_name(“Huckleberry Finn”)
```

These abstract methods don’t have to mimic common RESTful patterns; they could filter on virtually any criteria available.

Abstracted methods proved to be the most popular feature of my RESTful wrapper; more than just simply wrapping a RESTful API in an interface, abstracted methods provided real value to users to make their work easier to get done, eliminating lines of code they would have to otherwise implement on their own. Determining what to abstract can be difficult; it is likely you will have to ask the prospective users of your wrapper what they might need. Alternatively, you could encourage users to write their own abstracted methods and contribute back to the wrapper’s code.

3 Conclusion

RESTful wrappers are an integral part of getting any adoption of a new REST service. While the wrappers are intended to be tools to make that adoption as painless as possible, the wrappers themselves must be well understood and targeted precisely. A poorly thought out wrapper will be disregarded and unused, driving down adoption of that REST API.

Quality Assurance engineers are in the unique position to craft and implement these wrappers. Instead of just proxying the end user, QA engineers can create wrappers that act as bridges between separate engineering teams, easing integration pain points and providing a much needed holistic view.

References

[1] Fielding, Roy. “Representational State Transfer(REST)”, University of California, Irvine. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (accessed July 31 2015).

[2] Wikipedia. “Software Design Engineer in Test”, https://en.wikipedia.org/wiki/Software_Design_Engineer_in_Test (accessed July 31 2015).

[3] American Society for Quality. “Body of Knowledge - Software Quality Engineer Certification - CSQE”, http://asq.org/cert/software-quality-engineer/bok (accessed July 31 2015).

[4] Hutchinson, Allen. Han, Ray. “The difference between QA, QC, and Test Engineering”, Google Testing Blog, http://googletesting.blogspot.com/2007/03/difference-between-qa-qc-and-test.html (accessed July 31 2015).

[5] cURL Library homepage. http://curl.haxx.se/ (accessed July 31 2015).

[6] Koppel, Alex. “Koala Facebook Library”, https://github.com/arsduo/koala (accessesd July 31 2015).

[7] Facebook SDK for Python. https://facebook-sdk.readthedocs.org/en/latest/ (accessed July 31 2015).

[8] RubyGems, https://rubygems.org/ (accessed July 31 2015).

[9] Olzsowka, Christoph. “The Ruby Toolbox - http clients”, The Ruby Toolbox, https://www.ruby-toolbox.com/categories/http_clients (accessed July 31 2015).

[10] Nunemaker, John. “HTTParty”, http://johnnunemaker.com/httparty/ (accessed July 31 2015).

[11] Nunemaker, John. “HTTParty”, https://github.com/jnunemaker/httparty/blob/master/httparty.gemspec (accessed July 31 2015).

[12] Faraday HTTP client library homepage. https://github.com/lostisland/faraday (accessed July 31 2015).

Acknowledgements

A special thanks to Andrea Lemhouse of Puppet Labs who helped collect and analyze quality engineering job postings.