Pragmatic Hypermedia: Creating a Generic, Self-Inflating API Client for Production Use

Pete Gamache

Localytics, Inc.
Boston, Massachusetts, USA

pete@gamache.org

ABSTRACT

Hypermedia API design is a method of creating APIs using hyperlinks to represent and publish an API's functionality. Hypermedia-based APIs bring theoretical advantages over many other designs, including the possibility of self-updating, generic API client software. Such hypermedia API clients only lately have come to exist, and the existing hypermedia client space did not compare favorably to custom API client libraries, requiring somewhat tedious manual access to HTTP resources. Nonetheless, the limitations in creating a compelling hypermedia client were few.

This paper describes the design and implementation of HyperResource [19], a fully generic, production-ready Ruby client library for hypermedia APIs. The project leverages the inherent practicality of hypermedia design, demonstrates its immediate usefulness in creating self-generating API clients, enumerates several abstractions and strategies that help in creating hypermedia APIs and clients, and promotes hypermedia API design as the easiest option available to an API programmer.

Categories and Subject Descriptors

H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia — *Architectures*.; D.2.12 [**Software Engineering**]: Interoperability — *Data mapping, interface definition languages*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services — *Web-based services*

Keywords

hypermedia API; generic API client; service-oriented architecture

1. INTRODUCTION

Hypermedia API design is the practice of using hyperlinks to identify server resources and expose API functionality. In a hypermedia API, each response returned by the API contains a set of hyperlinks that specify where to access related resources. The base output format of the API may vary — XML, JSON, and HTML are all viable underpinnings for hypermedia formats — but the defining feature of a hypermedia API is that hyperlinks are used to guide usage of the API, both to client software by removing the need to construct URLs from scratch, and to client users by providing links to only those API features that are accessible to the user's authorization level.

Hypermedia-driven APIs afford several advantages over alternative designs. Foremost, generic API client software may be used to connect to disparate APIs, as long as the APIs support an output format accepted by the client software. This is similar to WSDL [15] in its overall goal; both approaches allow the creation of auto-generated client libraries. In a typical WSDL installation, an API will publish a document in WSDL format formally describing the API's features and how to access them, and end users will run generator software to create a set of API bindings targeted at a particular computer language, which can then be imported into a software project where API access is desired.

The crucial difference between hypermedia and WSDL is that hypermedia APIs and clients are designed to publish and consume these hyperlinks with each API request, turning the cycle of API feature advertisement and client upgrade to a hands-off, closed-loop system. API updates are available instantly, without any update to client library software. Users who read the API documentation and discover the a new feature are able to immediately put it to use. API designers are freed from maintaining custom client software and pushing updates to end users. An API with only hypermedia clients can even change its API structure with a transition period of hours, not months, because they may have confidence that client software can start to use the updated structure immediately.

Despite these very attractive benefits, industry and community adoption of hypermedia designs has been low. In the last several years, several hypermedia formats such as Collection+JSON [16], HAL [22], JSON-LD [10], and Siren [25] have appeared. This bloom of formats has yielded a handful of client libraries supporting them, for instance Ruby's HyperClient [8], JavaScript's HyperAgent [7], and Java's Siren4J [14]. Still, acceptance by API designers at large has been spotty at best. There are examples such as the Github v3 API [6], Amazon's AppStream REST API [3], and MapMyFitness' API [12], but most end users still use traditional, pre-hypermedia API client layers, ignoring the hypermedia data these APIs emit.

Ultimately, the main reason API designers and end users do not yet reach for hypermedia is because it is not yet the easiest thing they can do. On the API side, there is a limited amount of existing software to assist in the formation and serialization of hypermedia resources, but even if custom solutions are involved, it is a problem to be solved only once per API. Much more troublingly, on the consumer side, current client software packages provide unsatisfyingly shallow abstractions over the mundane details of

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04. http://dx.doi.org/10.1145/2567948.2579220.

accessing an HTTP-based API. This creates a problem to be solved by dozens or hundreds of users of each API, not just a few API implementors.

This paper details the conception, strategy, implementation, and debugging of a next-generation generic client for hypermedia APIs, named HyperResource [19]. At each step of its design, the expressiveness and brevity of end-user code was emphasized, in order to produce a generic client with identical or preferable look and feel to hand-built clients. It is the hope of this author that HyperResource will inspire similar clients in other languages, and eventually aid in reducing friction against the greater adoption of hypermedia as a practice.

2. DESIGN

The design of HyperResource began with a number of external factors, as well as baggage from previous experience writing APIs and API clients.

First, this was a client designed to be put into production use almost immediately. The development of HyperResource began when the author was busy creating an API for analytics queries [11] for his employer, Localytics. The API needed a client library written in either Ruby or JavaScript. Ruby was selected as HyperResource's platform in order to allow server-side use in the current Localytics production environment, including on the API itself, a JRuby on Rails application. As a production client, it needed to be reliable, threadsafe, and graceful about error-handling out of the box.

2.1 Serialization Format

Another client design factor, serialization format, originated more from server-side concerns than client-side. In the author's case, this was largely because the implementation of an API provided a sense of urgency and concrete purpose to this client's development. But it is not hard to accept that this lesson of API design dictating a client's input format may apply widely, and in any case a client cannot impose any conditions that an API does not support.

As one of few hypermedia APIs in the wild, and given the sparse selection of hypermedia client libraries of the time, it was important that any format chosen be equally usable in hypermedia and non-hypermedia contexts. In other words, the API needed to be as easy to use by non-hypermedia clients as the best non-hypermedia APIs are. To this end, it was desired for the output format to resemble a "plain old data" response as closely as possible.

It is generally preferable that a serialization format for any API be able to express all features directly in the response body, rather than relying on HTTP headers or other information outside the response body. Proxy servers, in-browser JSON-P, integration with third-party systems, and other factors can challenge any API design, but in most cases the response body has the best chance of making it to the client unmolested.

After reviewing the options at the time, HAL was picked because of its extremely small footprint: it adds only two fields to a response, `_links` and `_embedded`, which can be safely ignored by non-hypermedia clients yet completely encapsulate all hypermedia and embedded resource information. While HAL did not include ways to express concepts like sample forms, default parameter values, input parameter data types, in-band error handling, default HTTP verbs for links, and differentiation between "links" and "actions," the HAL format supported everything necessary to allow full usage of the API by someone armed with a generic client and the API documentation.

Thus, HyperResource had to support at least the HAL format.

2.2 Influential Clients

Given that HyperResource would inevitably be compared to its peers, it was useful to examine the strengths and weaknesses of other API client software on the Ruby platform, and other influential Ruby software in general. Many Ruby programmers are familiar with the ActiveResource [2] library, formerly part of the Rails core project. ActiveResource provided a wrapper around resourceful APIs, offering a look and feel similar to Rails' extremely popular ORM library, ActiveRecord [1]. ActiveResource transforms API responses into native Ruby objects, comparably to ActiveRecord's representation of database tables and rows as classes and instances. An API response is returned as an instance of a predictable class, which led to ActiveResource's key strength: API data could be easily extended with user code by adding methods to the class. Each API endpoint still required manual configuration, though, as with virtually all non-hypermedia clients.

HyperClient is one of the best of Ruby's first-generation hypermedia clients. Its usage pattern is markedly different from ActiveResource; instead of defining named subclasses of `ActiveResource` to represent API data types, all HyperClient API interaction takes place through instances of `HyperClient` itself. This arrangement allows for simple, lightweight, zero-configuration consumption of hypermedia APIs. However, the inability to automatically extend incoming data types with additional code is inconvenient. And, as hinted at earlier, actually using the hypermedia bits of an API is accomplished with a thin veneer over the Faraday [5] HTTP library, rather than the simple method calls of ActiveResource.

It became a chief design intention to marry the killer features of these very different API clients. Given the provenance of this client's feature set and its goals, the name HyperResource was selected, a compact mission statement for the project.

2.3 What Makes a Good Client?

One thread tying together the author's favorite client libraries is an understanding and acceptance of the language and platform on which the client is running. An ideal client library should impedance-match any sort of API (resource-based, RPC-based, etc.) to the client's native platform, whether that means object model, type system, or similar. A good client fits in with its environment; it is best that code using the client does not stick out from its surroundings.

Code that uses HyperResource, then, should look like Ruby code. HyperResource needed to embrace the Ruby object model, a world of single-inheritance and mixins, where everything is an object and they communicate by message-passing-based method invocation. [1]

It is important to note that all access to remote systems brings with it exposure to different kinds of problems absent from local code access — network failure or intermittence, latency, increased security concerns, and non-local concurrency challenges, to name a few. Code which accesses APIs may not be as simple as code accessing functionality which doesn't require a network roundtrip, or even going outside the running process or thread.

In decades past, it was considered unwise by some to hide many of these complexities by providing a unified interface for both local and remote data. [26] However, in today's computing environment, the benefits and drawbacks of service-oriented architecture and distributed systems are more well-understood by programmers at large. Developers consume more external APIs than ever before, and API consumers are required to address these topics no matter what client design they are using. And since HTTP APIs do not refer to local data, unlike the systems criticized in the aforementioned analysis. The line between what requires remote access and what is

[1]Close enough for this working group, anyway.

local is clear. A client need not inconvenience the user just to prove a point about the complexity of distributed systems.

To that point, a strong factor of API client quality is the expressiveness of the finished end-user code. Given something the API can do, is it straightforward to type it out? Has all boilerplate code been factored out? Don't Repeat Yourself, or DRY, is a requirement for a good, modern client. End-user code must be terse, to the limits of its language environment.

2.4 The Central Metaphor

The primary conceit of HyperResource is that a response's hyperlinks can represent the full set of API functions or relations that can be applied to the response. Or, in terms more familiar to a Rubyist, every object comes with its own method list. Responses are treated as objects, links as methods, rels as method names.

Assigning the role of methods to hyperlinks is more powerful than it appears at first glance. At once, it gives a handy guideline for those API implementors applying hypermedia for the first time, and it provides a tidy metaphor for accessing these links in an OO language like Ruby.[2]

A response from a hypermedia API can contain three kinds of data: regular attributes (e.g., `first_name`), links to other API resources, and other API resources embedded within this one. Once committed to the idea of wiring up links as methods on the returned object, and considering that object attributes are in general serviced by accessor methods in Ruby, it becomes apparent that embedded objects might make sense to be accessed via methods as well.

Adopting the structure of methods and objects, however, should not be seen as a suggestion that all, or even many, APIs fit neatly into an object-oriented worldview. It is simply an admission that in the target language, Ruby, objects and methods are the most convenient way to do much of anything. It can be safely stated that HyperResource objects have a one-to-one mapping to API responses; nothing more about API structure is assumed.

2.5 No Religion

In order to be maximally useful to the greatest number of people, HyperResource does not impose any non-essential conduct upon the user. Hypermedia API design is typically associated with the "Representational State Transfer", or REST, described in Roy Fielding's thesis [17]. But hypermedia is just the practice of an API supplying a bunch of named URLs and a client consuming them somehow — nothing more. Hypermedia is extricable from RESTful design, proper use of HTTP verbs, human-readable URLs, and just about every other best practice you can think of.

In a RESTful API, the hyperlinks generally represent formal relationships between two resources. In an RPC-based API, it is likely that the hyperlinks may include not only related resources, but also procedures taking the original resource as input. In many cases, these approaches coëxist in the same API. It is not essential that a hypermedia API client prescribe a usage pattern, only provide a convenient way to access the data at these links' URLs.

Hypermedia, though surrounded by other good ideas pertaining to API design. stands on its own. It depends on nothing but the API knowing its own URLs and publishing that information somehow. Even on its own, hypermedia can support the possibility of generic clients. Many real-world APIs do not map cleanly to a theoretical, ideal, fully RESTful structure. Sometimes an API must support GET or POST in non-semantic situations, and some problems — for example, the analytics API that I put together for Localytics — do not map perfectly to a world consisting of states and transitions.

[2]In fact, given Ruby's facility for dynamic method dispatch via `method_missing`, the implementation almost writes itself. Almost.

As such, HyperResource does not take an official stance on best or worst practices outside of hypermedia itself. As long as an API's hyperlinks are well-formed, everything should work. RPC-over-JSON, page-long URLs, unexpected verbs at unexpected times: all of these can be made to work with little or no effort on the part of the user.

It is the position of the HyperResource project that it is best to consider hypermedia as orthogonal to other axes of API design. This maximizes adoption of hypermedia, a treasured side goal of this project.

2.6 Do The Right Thing

Choosing a single way to access all API features brings benefits and drawbacks. A unified interface goes a long way toward code clarity and a general spirit of Do The Right Thing. One possible problem is namespace collisions; for instance, an API response could contain an attribute named `object_id`, which conflicts with a built-in Ruby method. For these cases, it is easy to provide a fallback mechanism that allows explicit access to attributes, links, or objects.

But in the other cases, as long as the client maximizes the consistency between the different modes of access, APIs could even refactor data between links, embedded objects, and attributes, and the client code could run unchanged! A much less farfetched idea is that making all API access feel as smooth as dealing with "plain old data" is a worthy pursuit, and one of the keys to a generic client library comparing favorably to a hand-rolled client.

Another intriguing way to Do The Right Thing, possible largely thanks to Ruby's `method_missing` catch-all method, is implicit loading of resources. HyperResource opts to implicity load resources when as-yet unknown methods are called on as-yet loaded resources. In this case, an implicit GET could be performed, and the method call repeated on the loaded object. This simple enhancement would be enough to eradicate some very common visual clutter, while remaining predictable in how and when objects are loaded.

These features of programmer convenience and code expressivity are most clearly illustrated through code examples, which follow.

3. SAMPLE CODE

At this point in the development of HyperResource, the author sketched out what sample code using this API client should look like. In most cases, the sketches were compared to equivalent code using HyperClient, to better point out opportunities for optimization of HyperResource. This "docs-driven development" methodology was extremely helpful in cementing design ideas and visualizing the flow of data through typical HyperResource code. With the hope of bringing some of these insights to the reader, here is a working example of using HyperResource to connect to a HAL-based API, almost unchanged from its initial, pre-implementation form.

Consider a hypermedia API whose entry point returns:

```
{ "message": "Hello!",
  "_links": {
    "self": {"href": "/"},
    "users": {
      "href": "/users{?email}",
      "templated": true
    }
  }
}
```

We begin by defining a class to both represent our API and act as a namespace for all resources from this API. If the API provides information about the data type of each resource in some way, then HyperResource can be configured to recognize it, and will instantiate a properly-named class in our namespace. In this example, the API entry point is tagged with the data type `Root`.[3]

```
class MyAPI < HyperResource
  self.root = "https://myapi.example.com/v1"
end
```

Let's imagine our goal is to load a resource representing a user `jdoe@example.com`. We've read the API docs, and we know that we can GET the link named `users` to receive a collection of user objects, and that we can optionally send an email address as a URI query parameter.

```
api = MyAPI.new
jdoe = api.users(email: "jdoe@example.com").first # =>
     #<MyAPI::User:...>
```

That is extremely concise code.[4] HyperResource performs a number of automatic expansions made possible by Ruby's `method_missing` catch-all method, which provides a facility to intercept and manually dispatch unrecognized method calls at runtime.

Any unrecognized method call on a not-yet-loaded resource will load the resource and retry the call. On a loaded resource, methods that don't exist will be cross-referenced by name against any links, embedded objects, or attributes the resource contains. On a link, nonexistent method calls will trigger the link to be loaded, optionally applying values for any given parameters, returning a resource on which the method call is repeated. Each resource provides `links`, `objects`, and `attributes` accessors, which translate method calls to hash access. And finally, as a way of supporting a common case, calling `.first` on a HyperResource will return the first embedded object in the response.

Taking these transformations into account, observe the progressive expansions of the above expression.

```
api.users(email: "jdoe@example.com").first

api.get.users(email: "jdoe@example.com").first

api.get.links.users(email: "jdoe@example.com").first

api.get.links["users"]
  .where(email: "jdoe@example.com").first

api.get.links["users"]
  .where(email: "jdoe@example.com").get.first

api.get.links["users"]
  .where(email: "jdoe@example.com").get
  .objects.first[1][0]
```

Most prior hypermedia clients required end-user code to look somewhat worse than the last line in the series. HyperResource's approach represents a leap in usability. Ruby's ordering of function application, coupled with the simple rules for intercepted method calls we listed above, leaves no ambiguity about when network access occurs, yet the programmer is no longer burdened with expressing anything but intent.

As an example of extending API data types, let us create a convenience method to construct a user's formal name given two attributes in the `User` resource, `first_name` and `last_name`:

```
class MyAPI::User
  def formal_name
    "The Right Honorable #{first_name} #{last_name}"
  end
end

jdoe.formal_name # => "The Right Honorable John Doe"
```

And no suite is complete without POST, PUT, PATCH, and DELETE:

```
jdoe.first_name = "Jane"
jdoe.patch # sends 'first_name' only
jdoe.put   # sends all params

# make a new user
red = api.users.post(first_name: "Red", last_name:
     "Shirt")

# unmake a new user
red.delete
```

4. PROBLEMS AND SOLUTIONS

Once HyperResource hit an acceptable level of implementation and testing, it gained its first production deployment and first non-author user. Of course, this is as essential a part of testing software as any automated verification suite — the proof of the pudding is in the eating. Not surprisingly, there was plenty of room for improvement, and some of these topics are detailed below.

4.1 Cacheability

An important tool in the hypermedia API user's toolbox is a cache. Hypermedia APIs sometimes expose functionality that can't be reached directly from the API root, meaning that users must retrieve certain intermediate resources along the way. For performance reasons, it is often desirable to persist these intermediate values, either in local memory or using an external cache such as Memcached [13]. More broadly, non-local caches are an indispensable part of a modern systems engineer's bag of tricks, and it is highly preferable that a client's objects support them.

It is worth mentioning that hypermedia offers no natural resistance to the difficulty of correct caching. All the usual problems relating to cache lifetime, invalidation, permission, etc. apply to hypermedia APIs. Caching is hard to do properly, no more or less so on hypermedia APIs than on any other data source.

HyperResource's initial implementation defined singleton methods on each API response, one per link or embedded object or attribute. This was intended to speed up method calls by leveraging Ruby's native method resolution. Unfortunately, and much more importantly, this ended up preventing serialization of the objects with Ruby's `Marshal.dump`, in turn preventing storage in an external cache. A solution was devised: methods would instead be properly defined as regular instance methods on the resource class. More on this in a moment.

Another, more interesting problem pertained to newly-launched programs making use of a long-lived external cache. HyperResource often automatically creates Ruby classes at runtime; in the above sample code, `MyAPI` is declared by the programmer, but

[3]There is no standard way to specify this data type information within a HAL document. Possible implementations include an additional field in the response, e.g. `data_type`, or equivalent information as part of the `Content-Type` header. Type annotations are omitted in this example.

[4]It is also quite similar to modern ActiveRecord, and not by coincidence.

MyAPI::Root is created by HyperResource upon encountering a resource tagged with the data type Root. If an object of such an auto-generated class is placed into a cache by one part of a system (e.g., a server you deployed a week ago), other parts of the system (e.g., a server you just deployed) must take care not to load these objects from cache until having created the object class themselves, in the current runtime. Otherwise, Ruby's Marshal will not be able to deserialize the cache object into a nonexistent class. One solution to this problem is to ensure that these errors both freshen the cache and update the current runtime's class hierarchy. Another, less elegant way to avoid this problem is to make a set of API calls early in your program's runtime, to instantiate ahead-of-time all the classes that might be encountered in the cache.

4.2 Persistent vs. Dynamic Methods

The cacheability solution of defining instance methods on classes at runtime had the advantage of using Ruby's built-in method dispatch, as well as retaining the ability to use respond_to?, the Ruby way of testing if an object responds to a given method name without needing to resort to method_missing. It was implemented, and it worked. The only major gotcha is that HyperResource must be careful not to add methods to the HyperResource class or to API namespace classes, only to API data type classes — not a big deal.

But there are deeper problems with persistent method definitions. Ruby and JRuby both employ a cache when resolving precisely what to do when a particular method is called on a particular object or class of object, for performance reasons. Defining methods at runtime, including singleton methods, causes the entire Ruby method resolution cache to be emptied. [24] [23] [5] In the quest to shave away a 0.1 millisecond toll for calling method_missing, in all likelihood connected to a 100 ms network call, HyperResource introduced a 1 ms penalty to the next invocation of any and all methods in the Ruby runtime, until the method resolution cache warms to each and every method in the program again. [20]

Obviously, this is a poor bargain, and in upcoming releases of HyperResource this feature is disabled by default, leaving all dynamic method dispatch to method_missing, and respond_to? is patched to tell the truth. HyperResources behave like Ruby objects to the greatest extent that is practical, leaving fewer surprises for the end developer.

4.3 Structured Data over GET

One notable oversight by the great architects of the Internet is that there is no standard way to serialize data structures, such as arrays or hashes, for inclusion in URIs as URI query parameters. [21] Perhaps the most widely-adopted nonstandard serialization is the one used by jQuery.param. [9] This scheme, also in use by the Rails project, specifies an unsuitably ambiguous format for the data, and was abandoned as a design shortly after the project began.

In order to confront and overcome the fact that there is no standard, default procedure for serializing nested data structures to URL format, HyperResource introduced the outgoing_uri_filter, an overridable method that takes as input a hash of parameter keys and values, and outputs a hash of transformed keys and values. This method is used as a filter for parameter keys and values as URLs are being constructed, and can be used to perform custom serialization of some fields — for example, passing structured data through JSON.dump if the API supports it. A real life example from production code, where the Localytics API's query link takes conditions parameter consisting of structured data, over GET or POST:

```
class LocalyticsAPI < HyperResource
  class Root < LocalyticsAPI
    def outgoing_uri_filter(params)
      if params["conditions"]
        params["conditions"] =
            JSON.dump(params["conditions"])
      end
      params
    end
  end
end

api.get.query(conditions: {day: ["in", "2014-01-01"]},
    ...)
```

If the user chooses to use POST, no intervention is needed, since the request body media type will be application/json, which is quite well-suited to representing structured data. But if the user accesses the API feature through a GET request as in the example above, outgoing_uri_filter will be invoked, and the structured conditions parameter value will be serialized into a JSON string before sending to the API. Since the API is coded to accept this, everyone goes home happy.

For symmetry, outgoing_body_filter was added to filter outgoing params for POST, PUT, and PATCH requests, and incoming_body_filter was added to transform data coming from the API. However, these methods provide only a partial solution to filtering parameters. They are anchored not to a particular link relation (the query part of api.get.query), but to the class of the originating object (the type LocalyticsAPI::Root of the api.get part of api.get.query). If this same link relation is also accessible from other API paths, they will not automatically be able to use this filtering code. This is a candidate for refactoring in future versions of HyperResource.

5. FUTURE WORK

HyperResource is a young project, and there are certainly many unexplored opportunities for improvement. Here are some of the ideas not yet having seen the chance at implementation.

5.1 More Formats

HyperResource was intended from the start to support multiple hypermedia formats, but to date has full support for only one, HAL+JSON. During a fairly major refactoring during 2013, HyperResource's hypermedia format adapter interface was created to provide an abstract definition of the minimum functionality from such an adapter. Three methods must be defined by an adapter implementation: serialize, deserialize, and apply, which applies state from a deserialized API response onto a HyperResource object. This separation works well in the case of the default HAL+JSON adapter, and is holding together well during the present process of adding Siren and Collection+JSON support. Being agnostic to particular hypermedia format is the best way to support all use cases in the future, and it has not hindered the progress of HyperResource at all. More formats are good sense.

5.2 More Auth

HyperResource has mostly ducked the issue of authentication so far. One method is currently supported, and that is HTTP Basic Authentication [18], the simplest (and on its own, least secure) standard authentication mechanism for the web. This is sufficient for production use with many APIs that use HTTPS and Basic Auth

[5]Other actions also cause pre-2.1 Ruby's method resolution cache to be flushed, including Class.new, Object#extend, and and OpenStruct.new.

as an effective mechanism. However, APIs that require OAuth, Amazon-style request signing, or other authentication schemes are left in the cold right now. This is a simple matter of writing more code.

5.3 More Languages

The ideas that constitute HyperResource's functionality are directly applicable in many languages other than Ruby. For instance, a more-or-less direct clone could be written in ECMAScript 6, given the new Proxy API [4] which can perform similar functions to Ruby's `method_missing`. Even without Proxy, a JavaScript hypermedia client could be made to function very much like HyperResource if implicit object loading were abandoned.

It is interesting to consider what shape other implementations of HyperResource's core ideas should take on their respective platforms. Should the JS client use a callback-based interface by default? Promises? Many of these considerations are driven by the wants and desires of a language community, as much as by a language itself. A library should make its users happy, and part of that is fitting in nicely with their idea of a perfect world.

6. CONCLUSION

This paper details the design of HyperResource, a generic client for hypermedia APIs, from the goals of terse, DRY code; extensibility of API data; simplicity of expression; and native look-and- feel on its platform of Ruby. The central concept of an API exposing its entire functionality as links, and using these links as methods on the returned object, was introduced, and demonstrated as a preferable way to access API features. Preëxisting clients were examined, ideal sample code for a new client was imagined, and then a client was implemented to validate this sample code. Several implementation details were described, including a solution for URL-encoding of structured data and production-inspired improvements around cacheability and method resolution.

It is the conclusion of this author that HyperResource proves the concept that generic hypermedia API clients can obtain the same usability and user-comfort of hand-coded client libraries, and that as more tooling catches up to this level of sophistication, hypermedia will gradually supplant non-hypermedia, resource-based JSON APIs as the simplest course of action for API designers and consumers alike.

7. ACKNOWLEDGEMENTS

The author would like to thank his employer, Localytics, for support, encouragement, and the chance to do something right when the alternative wasn't certain peril; HyperResource contributors Ian Asaff, Étienne Barrié, Julien Blanchard, Frank Macneely, James Martelletti, and Jean-Charles d'Anthenaise Sisk; and Benjamin Darfler, Judy Margo, and Joel Rosen for reading drafts of this paper.

8. REFERENCES

[1] Activerecord, `https://github.com/rails/activerecord`

[2] Activeresource, `https://github.com/rails/activeresource`

[3] Amazon appstream rest api documentation, `http://docs.aws.amazon.com/appstream/latest/developerguide/rest-api.html`

[4] Direct proxies, `http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies`

[5] Faraday http library, `https://github.com/lostisland/faraday`

[6] Github v3 api documentation, `http://developer.github.com/v3/`

[7] Hyperagent, `http://weluse.github.io/hyperagent/`

[8] Hyperclient, `https://github.com/codegram/hyperclient`

[9] jquery.param() documentation, `http://api.jquery.com/jquery.param/`

[10] Json for linking data, `http://json-ld.org`

[11] Localytics api version 1 documentation, `https://api.localytics.com/docs`

[12] Mapmyfitness api documentation, `https://developer.mapmyapi.com/docs`

[13] Memcache: a distributed memory object caching system, `http://memcached.org`

[14] Siren4j, `https://code.google.com/p/siren4j/`

[15] Web services description language (wsdl) 1.1, `http://www.w3.org/TR/wsdl`

[16] Amundsen, M.: Collection+json hypermedia type, `http://amundsen.com/media-types/collection/`

[17] Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine, Irvine, CA (2000)

[18] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: Http authentication: Basic and digest access authentication. RFC 2617 (Mar 2012), `http://www.ietf.org/rfc/rfc2617.txt`

[19] Gamache, P.: Hyperresource: A self-inflating ruby client for hypermedia apis, `https://github.com/gamache/hyperresource`

[20] Gamache, P.: Ruby method cache — a benchmark, `http://petegamache.com/ruby-method-cache-a-benchmark/`

[21] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., Orchard, D.: Uri template. RFC 6570 (Proposed Standard) (Mar 2012), `http://www.ietf.org/rfc/rfc6570.txt`

[22] Kelly, M.: Hypertext application language, `http://stateless.co/hal_specification.html`

[23] McCoy, S., Myers, R.: Understanding ruby's method cache (Oct 2013), `http://wickedgoodruby.com/2013/speakers/mccoy_myers`

[24] Somerville, C.: Things that clear ruby's method cache, `https://charlie.bz/blog/things-that-clear-rubys-method-cache`

[25] Swiber, K.: Siren: Structured interface for representing entities, `https://github.com/kevinswiber/siren`

[26] Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing (1994), `http://dl.acm.org/citation.cfm?id=974938`