by Andrew Berry // April 15, 2014

Legally Binding your Web APIs

As web architecture evolves towards building distributed, independent applications, and away from single-purpose omnibus websites, Drupal implementations are including services, APIs, and feeds almost by default. While Services, RestWS, and Drupal 8 all simplify creating APIs to distribute content, it's important to step back and think about API design before writing a single line of code.

A quick aside: many prefer to refer to REST APIs as a subtype of the more general Hypermedia API, but for simplicity this article uses the better-known REST acronym.

The Legacy of Common Law APIs

A stock Drupal 7 site contains a large number of HTTP calls that are tightly coupled to specific JavaScript or frontend implementations. Some examples of these include:

- Taxonomy autocomplete
- "Add another" for unlimited count fields
- #ajax for dynamic form interactions

If your site uses Views, there is also pager and search functionality, along with administrative-only AJAX calls for building Views. This is only the tip of the iceberg for a complex Drupal site. It's not unheard-of for a production site to have dozens of these "internal" APIs.

The problem with these sorts of calls isn't that they exist, or that they are meant to be internal to Drupal only. The real issue is that they provide poor examples for new developers when designing modules or site-specific code. As we design and build sites incrementally, from basic Drupal themes with custom JavaScript, through to complete front-end applications using Angular or Backbone, we carry this legacy of coupled implementations with us.

How can we escape this pattern of fragile and one-off APIs? We must:

- Understand different API paradigms
- Design the Object Schema
- Document and Communicate

Civil Law: Define an API paradigm

Ask three developers to design a REST API, and you'll end up with three totally different designs. In practice, it's common for developers to think of REST as meaning "not SOAP" or "over HTTP". Simply

common approaches to an HTTP API that I've seen are:

API AS FEEDS

This paradigm is common in Drupal implementations due to the simplicity of exporting JSON or RSS from Views. I'd go so far as to say that a feed, even in JSON, is not really an "API" but more of a raw data source. You can identify a Feeds-style API if:

- Your API entry mechanism returns a list of recent content with basic filters.
- You have a small number of entry points with query parameters to extract subsets of data. For example if to fetch articles you query `/content?type=article` you might have a Feeds API.
- The primary focus of your API is lists of data and not individual instances of the data itself.

API AS REMOTE PROCEDURE CALLS (RPC)

This paradigm is easy to identify. Almost always, URLs contain the equivalent of method names instead of using HTTP semantics. For example:

- `/api/getUsers` would map to a user load or search method.
- If your API has `/api/user/{id}` as a valid URL, if modifying a user was executed with a POST call to `/api/user/{id}/update`, the API is still an RPC API.
- If the API has a small number "endpoints" (such as with SOAP), it's likely an RPC API. These sorts of APIs would typically have a large number of operations tied to each URL that accepts a POST request, with operational information included in HTTP headers or in the POST body itself.

API AS REST

A REST API tries to exploit the functionality and semantics already defined in HTTP as much as possible. A key distinction with a REST API is the concept of a Resource, which roughly maps to an object instance. Every resource has a unique identifier in the form of it's URL. While we often include numeric IDs for our own sanity, there's nothing that prevents the unique identifier from being a human-readable string. What's important is that the URL is always unique per resource.

- If a GET on `http://example.com/organization/lullabot` returns the Lullabot organization, it is likely that the API is RESTful.
- Likewise, if a POST on the same URL is used to update the Lullabot organization, it's likely that the API is RESTful.
- The API is traversable and discoverable through the API itself when combined with basic HTTP methods.

Many web APIs end up with a mixture of all three paradigms, as functionality is added and modified over time. While developers often prefer REST APIs, what is the most important is that the API is designed and kept to a single paradigm, regardless of what that is.

Setting Jurisdiction: Defining the Object Schema and it's boundaries

Exposing data as JSON and calling it a day doesn't define an API. While message formats (like JSON, XML, and HTML) are important, even more important is the actual structure of objects returned by the API. Where possible, object keys should be common across objects if they have the same meaning. For

Data formats within objects should be defined and controlled as well. A common mistake is to mix date formats between Unix timestamps and ISO dates, or to expose numbers and booleans as strings instead of their raw type. This will not only help to make your API consistent, but will also ensure that it's discoverable and intuitive to API consumers.

Finally, where possible responses should use references instead of composition. Many APIs will embed related objects within a single response to try to reduce the number of HTTP requests. With complicated content models, it's common to end up with circular references between related content. Splitting the objects into separate resources allows clients to decide how deep they want to traverse the object graph. Also, smaller objects allows faster returns to the API client, which in turn unblock the client and gives it the flexibility to run subsequent requests in parallel if it actually needs the data.

For example, when returning an article, don't include an array of every contributor:

```
{
  "type": "article",
  "title": "Legally Binding your Web APIs",
  "contributors": [
    {
      "name": "Andrew Berry",
      "email": "nobody@example.com"
    },
    {
      "name": "Juan Pablo Novillo Requena",
      "email": "nobody@example.ca"
  ]
}
```

Instead return a reference to the author resource:

```
{
  "type": "article",
  "title": "Legally Binding your Web APIs",
  "contributors": [
    "http://example.com/authors/aberry",
    "http://example.ca/authors/juampynr",
  ]
}
```

What about the per-HTTP-request performance hit? It does depend on the scope of the data being returned, but composition might be reasonable if:

- The data is a bounded list, such as a single author instead of a list of authors.
- If the additional data is small, both in the number of properties and the data stored in each property.
- Composition is broadly beneficial to every API consumer's performance.

proxy. This allows the client to aggregate and modify responses tailored exactly to their use case, freeing your application from having to understand the implementation details of API consumers.

Codification and Communication

Without documentation, an API might as well not exist. Anyone who has done ecommerce work or who has worked with proprietary APIs is familiar with the 10MB PDFs or Word documents that typically accompany them. It's this documentation, that describes both the rationale and the implementation of the API that will determine how successful an API is. For web APIs in particular, it makes the most sense for documentation to live on the web where it can be referenced, stubbed, and tested. Some documentation tools I've used in the past include:

- Apiary: A wonderful service for documenting with Markdown, stubbing with JSON, and creating quick API demos.
- Swagger: A tool that can be used to generate and self-host your documentation.
- JSON Schema and Hyper-Schema: A specification for describing the object schema of JSON returns from APIs.

When building your own clients against your own API, it's critical that you use your own documentation as the reference for your implementation. This ensures that your API client isn't a privileged application, and that any other client has access to the same functionality. It's also a great way to do a pass of QA before releasing your API and documentation to the public.

Next Steps: Amendments and Iterations

While we should take great care as API designers to limit API breaks for arbitrary reasons, it's entirely OK to modify and iterate on both the basic assumptions of your API as well as the actual implementation. It's important to give yourself the flexibility to amend and improve your API. After all, best practices are still in such flux that it's unlikely that everything recommended today will stick. Where possible, try to amend your existing API without breaking paradigms or object schemas. If you find that your application or best practices dictate changing those assumptions, consider writing a new, separate API to support in parallel.

PUBLISHED

in

Drupal Development & Mobile

ABOUT THE AUTHOR, ANDREW BERRY

Andrew Berry is a architect and developer who works at the intersection of business and technology.

See more articles by Andrew →