18F API Standards

18F is a technology team inside the US federal government. 18F is very API-focused: our first project was an API for business opportunities.

This document captures **18F's view of API best practices and standards**. We aim to incorporate as many of them as possible into our work.

APIs, like other web applications, vary greatly in implementation and design, depending on the situation and the problem the application is solving.

This document provides a mix of:

- **High level design guidance** that individual APIs interpret to meet their needs.
- **Low level web practices** that most modern HTTP APIs use.

Design for common use cases

For APIs that syndicate data, consider several common client use cases:

- **Bulk data.** Clients often wish to establish their own copy of the API's dataset in its entirety. For example, someone might like to build their own search engine on top of the dataset, using different parameters and technology than the "official" API allows. If the API can't easily act as a bulk data provider, provide a separate mechanism for acquiring the backing dataset in bulk.
- **Staying up to date.** Especially for large datasets, clients may want to keep their dataset up to date without downloading the data set after every change. If this is a use case for the API, prioritize it in the design.
- **Driving expensive actions.** What would happen if a client wanted to automatically send text messages to thousands of people or light up the side of a skyscraper every time a new record appears? Consider whether the API's records will always be in a reliable unchanging order, and whether they tend to appear in clumps or in a steady stream. Generally speaking, consider the "entropy" an API client would experience.

Using one's own API

The #1 best way to understand and address the weaknesses in an API's design and implementation is to use it in a production system.

Whenever feasible, design an API in parallel with an accompanying integration of that API.

Point of contact

Have an obvious mechanism for clients to report issues and ask questions about the API.

When using GitHub for an API's code, use the associated issue tracker. In addition, publish an email address for direct, non-public inquiries.

Notifications of updates

Have a simple mechanism for clients to follow changes to the API.

Common ways to do this include a mailing list, or a dedicated developer blog with an RSS feed.

API Endpoints

An "endpoint" is a combination of two things:

- The verb (e.g. **`GET`** or **`POST`**)
- The URL path (e.g. /**`articles`**)

Information can be passed to an endpoint in either of two ways:

- The URL query string (e.g. ?**`year=2014`**)
- HTTP headers (e.g. **`X-Api-Key: my-key`**)

When people say "RESTful" nowadays, they really mean designing simple, intuitive endpoints that represent unique functions in the API.

Generally speaking:

- **Avoid single-endpoint APIs.** Don't jam multiple operations into the same endpoint with the same HTTP verb.
- **Prioritize simplicity.** It should be easy to guess what an endpoint does by looking at the URL and HTTP verb, without needing to see a query string.
- Endpoint URLs should advertise resources, and **avoid verbs**.

Some examples of these principles in action:

- FBOpen API documentation
- OpenFDA example query

- Sunlight Congress API methods

Just use JSON

JSON is an excellent, widely supported transport format, suitable for many web APIs.

Supporting JSON and only JSON is a practical default for APIs, and generally reduces complexity for both the API provider and consumer.

General JSON guidelines:

- Responses should be **a JSON object** (not an array). Using an array to return results limits the ability to include metadata about results, and limits the API's ability to add additional top-level keys in the future.
- **Don't use unpredictable keys**. Parsing a JSON response where keys are unpredictable (e.g. derived from data) is difficult, and adds friction for clients.
- **Use under_score case for keys**. Different languages use different case conventions. JSON usesunder_score, not camelCase.

Use a consistent date format

And specifically, use ISO 8601 [xkcd.com/1179/], in UTC.

For just dates, that looks like 2013-02-27. For full times, that's of the form 2013-02-27T10:00:00Z.

This date format is used all over the web, and puts each field in consistent order -- from least granular to most granular.

API Keys

These standards do not take a position on whether or not to use API keys.

But *if* keys are used to manage and authenticate API access, the API should allow some sort of unauthenticated access, without keys.

This allows newcomers to use and experiment with the API in demo environments and with simple `curl`/w`get`/etc. requests.

Consider whether one of your product goals is to allow a certain level of normal production use of the API without enforcing advanced registration by clients.

Error handling

Handle all errors (including otherwise uncaught exceptions) and return a data structure in the same format as the rest of the API.

For example, a JSON API might provide the following when an uncaught exception occurs:

```
{
  "message": "Description of the error.",
  "exception": "[detailed stacktrace]"
}
```

HTTP responses with error details should use a 4XX status code to indicate a client-side failure (such as invalid authorization, or an invalid parameter), and a 5XX status code to indicate server-side failure (such as an uncaught exception).

Pagination

If pagination is required to navigate datasets, use the method that makes the most sense for the API's data.

Parameters

Common patterns:

- **`page`** and **`per_page`**. Intuitive for many use cases. Links to "page 2" may not always contain the same data.
- **`offset`** and **`limit`**. This standard comes from the SQL database world, and is a good option when you need stable permalinks to result sets.
- **`since`** and **`limit.`** Get everything "since" some ID or timestamp. Useful when it's a priority to let clients efficiently stay "in sync" with data. Generally requires result set order to be very stable.

Metadata

Include enough metadata so that clients can calculate how much data there is, and how and whether to fetch the next set of results.

Example of how that might be implemented:

```
{
 "results": [ ... actual results ... ],
  "pagination": {
    "count": 2340,
    "page": 4,
    "per_page": 20
 }
}
```

Always use HTTPS

Any new API should use and require HTTPS encryption (using TLS/SSL). HTTPS provides:

- **Security**. The contents of the request are encrypted across the Internet.

- **Authenticity**. A stronger guarantee that a client is communicating with the real API.
- **Privacy**. Enhanced privacy for apps and users using the API. HTTP headers and query string parameters (among other things) will be encrypted.
- **Compatibility**. Broader client-side compatibility. For CORS requests to the API to work on HTTPS websites -- to not be blocked as mixed content -- those requests must be over HTTPS.

HTTPS should be configured using modern best practices, including ciphers that support forward secrecy, and HTTP Strict Transport Security. **This is not exhaustive**: use tools like SSL Labs to evaluate an API's HTTPS configuration.

For an existing API that runs over plain HTTP, the first step is to add HTTPS support, and update the documentation to declare it the default, use it in examples, etc.

Then, evaluate the viability of disabling or redirecting plain HTTP requests.
See GSA/api.data.gov#34 for a discussion of some of the issues involved with transitioning from HTTP->HTTPS.

Server Name Indication

If you can, use Server Name Indication [en.wikipedia.org/wiki/Server_Name_Indication] (SNI) to serve HTTPS requests.

SNI is an extension to TLS, first proposed in 2003, that allows SSL certificates for multiple domains to be served from a single IP address.

Using one IP address to host multiple HTTPS-enabled domains can significantly lower costs and complexity in server hosting and administration. This is especially true as IPv4 addresses become more rare and costly. SNI is a Good Idea, and it is widely supported.

However, some clients and networks still do not properly support SNI. As of this writing, that includes:

- Internet Explorer 8 and below on Windows XP
- Android 2.3 (Gingerbread) and below.
- All versions of Python 2.x (a version of Python 2.x with SNI is planned [legacy.python.org/dev/peps/pep-0466/]).
- Some enterprise network environments have been configured in some custom way that disables or interferes with SNI support. One identified network where this is the case: the White House.

When implementing SSL support for an API, evaluate whether SNI support makes sense for the audience it serves.

Use UTF-8

Just use UTF-8 [utf8everywhere.org/].

Expect accented characters or "smart quotes" in API output, even if they're not expected.

An API should tell clients to expect UTF-8 by including a charset notation in the Content-Type header for responses.

An API that returns JSON should use:

Content-Type: application/json; charset=utf-8

CORS

For clients to be able to use an API from inside web browsers, the API must enable CORS [enable-cors.org/].

For the simplest and most common use case, where the entire API should be accessible from inside the browser, enabling CORS is as simple as including this HTTP header in all responses:

Access-Control-Allow-Origin: *

It's supported by every modern browser, and will Just Work in many JavaScript clients, like jQuery.

For more advanced configuration, see the W3C spec [www.w3.org/TR/cors/] or Mozilla's guide [developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS].

What about JSONP?

JSONP is not secure or performant [gist.github.com/tmcw/6244497]. If IE8 or IE9 must be supported, use Microsoft's XDomainRequest [blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx?Redirected=true] object instead of JSONP. There are libraries to help with this.