Designing and Building a World-class API

Chad M. Maughan, Principal Engineer, Domo

***Abstract*—A few years ago, Marc Andreessen omnisciently observed, "software is eating the world" [1]. Many envisioned an aristocratic setting, heady with sophistication where software, where software carefully and deliberately consuming an artfully prepared business world. A very pleasant setting.**

In reality, software tends to be more like a marooned sailor, recently rescued and alone in an all-you-can-eat buffet. It's gorged itself and is coming back for seconds. Its voracious appetite won't be satiated for the foreseeable future.

Part of the growth of world-eating software can be attributed to the publishing of programmatic access to various sources of data, or APIs. If the world is being eaten by software, APIs are the utensils facilitating the feast.

***Keywords*—Design, API, OAuth, HTTP, REST.**

Introduction

As the world's first business management platform, Domo eats data. We eat a lot of data (our lawyers won't let me say how much data, so you'll have to trust me, it's a lot). Along with data that is pushed to us by our clients, we consume data from over 500 distinct data sources. Data providers such as Facebook, Salesforce, LinkedIn, Xero, and Google to name a few. We consume data from a wide variety of data providers through a wide variety of APIs.

To facilitate that consumption of data, we've also consumed well over 100 individual OAuth implementations. Each implementation being unique in one way or another.

After five years of working to help businesses break down data silos and get data aggregated into a single location, we've learned a number of things about API design. Published APIs from some data providers are a sheer pleasure to work with. Others are incredibly brittle and onerous. When the task came to release our own official API, we decided it was a good time to share some of the things we've learned as guidelines and best practices.

In this document we'll step through some general requirements then move into some more specific items to consider while designing and building a world-class API.

Fundamentals

Broadly speaking, an API needs just a few, very basic characteristics. Without these, it doesn't stand much chance of being adopted.

- Consistency
- Stability
- Performance

Consistency

Software developers (and machines) love consistency. At some level, the consistency of binary and the reproducibility of computers and software enticed many developers to their future craft. As developers are ultimately the prime audience of an API, consistency is the most fundamental component of an API. This consistency should exist in all levels of an API. There should be consistency in resource naming, authentication, HTTP methods, and even through documentation. Business constraints may force developers to deviate from the guidelines of commonly accepted REST principles. That's understandable. However, when and if the need arises, be consistent.

Stability

As an API developer, you should expect your API to always be accessible. Users expect it to be available when they need it (which will be always). They also expect to be able to programmatically check the status of an API.

Users also expect errors to be truly exceptional. An API that frequently serves up 500 HTTP response status codes is the first indication of an unstable and perhaps poorly implemented API. If the client sends a bad request, one should return a 400 HTTP status code with an explanation of the offending portion of the request.

Performance

As an API developer, there are a number of things you can't control. Client network latency being one of the more obvious. Still, nobody likes a slow API. At Domo, our goal is to keep every response to an average of 100ms. Obviously, not every endpoint or resource of an API can always hit that average. Pragmatism should be accounted for. A great practice to ensure acceptable

performance is to log and frequently review statistical performance metrics while continuously working to improve your worst performing resources.

Design

A great API is designed. Time is taken to think about the overall resource model, the various entry points, and how the represented objects are related to one another. One doesn't simply "code up" an API. There are a number of strategies as well as decisions to consider when designing an API.

API First Design

In an ideal world, an API is first carefully thought-out and *then* built. Before any coding is done, the domain is thoroughly researched and object relationships are meticulously modeled. The API is constructed first and only after it is finished is a user experience or other consuming clients started. It's a principle called "API First Design". In theory, it's a fantastic approach. If you have the luxury to start a new project with this approach, consider yourself lucky. In the real world, however, this is rarely possible.

There are a number of issues that prevent developers from building an "API First" project. Sometimes the API you're building is for an existing, legacy system where one is exposing functionality. Other times, disparate teams in an organization are working concurrently (and quickly) to build out core functionality. This often isn't conducive to developing a consistent and simple API. Brian Mulloy, vice president of Apigee, an API technology and services company, describe these as "API anti-patterns" that should be avoided if possible [2].

As a developer, what should you do if you can't build "API First?" If you can't employ the "API First" design principle, consider using the API Facade (or Gateway) Pattern. This is a pattern made popular by Netflix [3] and Apigee [2].

Even before being used in an API context, this pattern was popularized by the famous "Gang of Four" (GoF). They recommended to, "use the facade pattern when you want to provide a simple interface to a complex subsystem" [4]. The API facade pattern is very effective in simplifying an API by allowing you to focus specifically on your intended developer audience. The API Facade pattern also provides a buffer to the teams or systems at the core of your project, allowing them to operate at a faster internal velocity.

"Ain't No REST For the Wicked"

Roy Fielding described an architectural style, "Representational State Transfer" (or REST) for the Web in his Ph.D. dissertation [5]. REST is widely-used today with both published and internal APIs that can be accessed by more modern, single page JavaScript applications.

Why REST?

The REST architectural style was born after efforts by Roy Fielding and Tim Berners-Lee to improve the scalability of the web with version HTTP/1.1. Their proposal was published in RFC 2616 [6]. After the introduction of HTTP/1.1, they also formalized the Uniform Resource Identifiers (or URI) in RFC 3986 [7].

Both of these proposals were foundational in the later growth and adoption of REST. Coupled with these proposals, JavaScript frameworks began improving at a rapid pace. These innovations allowed more and more logic to be moved from the server (where it had traditionally been deployed) to the browser. As this happened, server-side logic began to focus more on core business logic with a main focus on object storage and retrieval.

Why not RPC over HTTP?

One of the chief criticisms of REST is that it's inefficient and not well-suited for clients that need data structured in a variety of ways. For example, a strict REST architecture can require a single-page JavaScript application to make many calls to get the required data. There's nothing wrong with RPC over HTTP, indeed it can be much more efficient than REST. RPC over HTTP design should however be limited to internal endpoints and not be exposed to developers in a published API.

URI Formatting

While many RESTful API design principles are subjective, the formatting of URIs is not. There are well established and practiced principles for how to format and structure resources.

Entry points

One should limit the number of published entry points to a RESTful API. Mark Masse, in his comprehensive API rulebook, describes one of the downfalls of advertising all individual service URIs via documentation. He states that excessively publishing entry points "can lead client developers to code tightly coupled clients that do not treat the APIs URIs as opaque identifiers" [8].

Instead, Masse recommends that "client developers should be instructed to make use of the APIs hyperme-

dia" to discover relationships and sub-resources via links. [8]. This is a fantastic goal to aspire to. It provides great flexibility to restructure and move sub-resources. After understanding the relationships and patterns of resources, developers consuming the API typically tightly couple their clients. In fact, it's also been observed where API providers themselves publish their own tightly coupled clients.

Formatting Identifiers

Ideally, all of your resource identifiers will be single words. Sometimes you just can't avoid it.

Don't use underscores in your resource identifiers. They're impossible to distinguish from spaces in any hyperlink. They also require a second keystroke to type. If the universe wanted you to use underscores, Christopher Latham Sholes would have placed it on an early QWERTY keyboard layout instead of the hyphen (or dash).

Also *don't* use camelCase. While camelCase is the correct format for multiple word variables in JSON request/response bodies, it should not be used for URIs. Yes, technically the path portion of a URI is case insensitive [9], using hyphens makes for better looking and more easily readable URI.

OK	Better	Best
/fileUploads	/file_uploads	/file-uploads

Listing 1: Resource Identifier Formatting

Path & query parameters

In most cases, path parameters are typically either scalar IDs or randomly generated UUIDs. Formatting doesn't apply.

Query parameters, like JSON request/response bodies, should follow a camelCase format.

Nouns, not verbs

This is a universally (at least in the REST universe) agreed upon principle. Identifiers should have nouns for their resources and use HTTP methods to direct desired actions. A resource identifier with a verb has been poorly named.

```
POST /v1/users/1234 HTTP/1.1
```

Listing 2: An example of an entity creation identifier

Universal pagination

For all resource collections, a developer friendly API allows the consumer to control the amount of data returned. This is typically controlled via "offset" and "limit" query parameters. Other variations, use a "page" query parameter. When pagination is supported on the request, a response should include metadata describing the current location and the total number of entities the calling client has access to.

```
GET /v1/users?limit=10&offset=10 HTTP/1.1
Accept: application/json
Host: api.domo.com
```

Listing 3: A collection request with pagination query parameters.

```
HTTP/1.1 200 OK
Content-Type: application/json

{ "users":
 [{
  "createdAt": "2013-02-08T00:33:37.704Z",
  "email": "leonhard.euler@domo.com",
  "id": 42,
  "name": "Leonhard Euler",
  "updatedAt": "2013-02-09T12:33:07.201Z"
 }],
 "metadata": {
  "limit": 10,
  "offset": 20,
  "totalCount": 77
 }
}
```

Listing 4: A collection response with metadata information.

Additionally, if using hypermedia, it's best to include a link to the next page so that the retrieval of the collection can be programmatically followed.

```
[
 {
  "rel": "users.next",
  "href": "/v1/users?offset=30&limit=10"
 },
 {
  "rel": "users.previous",
  "href": "/v1/users?offset=10&limit=10"
 }
]
```

Listing 5: Hypermedia example collection response links.

Another technique for paginating entity collections is to use `Range-Unit` and `Content-Range` headers as

described in the HTTP specification [9]. This allows a clean list of the entities in the response body but still makes counts and range information available to the client in the headers.

```
GET /v1/users HTTP/1.1
Range-Unit: users
Content-Range: 0-50
```

Listing 6: A sample collection request with pagination Range headers.

A response to the previous request would include a total count in the `Content-Range` header. The range returned is first specified, then a forward slash, and then the total count.

```
HTTP/1.1 200 OK
Content-Type: application/json
Range-Unit: users
Content-Range: 0-50/77
```

Listing 7: A sample collection response with pagination Range headers.

There are some other, more sophisticated features one can also include when using header pagination techniques that allow the client to control performance. One such option is to allow the client to include a `Prefer: count=none` header which saves the data store count on the server.

Sorting

Resource identifiers for collections should also have the ability to sort with a query parameter. This sorting should support sub-resources as well as both ascending and descending.

Partial Responses

Another feature that is helpful in controlling response latency and the amount of data returned is the partial response filter. A partial response filter allows a calling client to specify only the fields that it is interested in receiving. For example, a client may only be interested in just the id and email of a user.

```
GET /v1/users?fields=id,email HTTP/1.1
```

Listing 8: Partial response HTTP request.

A partial response filter should also incorporate some sensible defaults, and provide an option for retrieving all data. Both of these could be done with the existing query parameter and values as in the following listing.

```
/v1/users?fields=all
/v1/users?fields=default
```

Listing 9: Partial response example default and all data value.

Custom Date Ranges

Any data or collection resource that is sequential in nature should have the ability to limit data based on date ranges.

"Some Needed R & R" (Requests & Responses)

Just as formatting URIs is important in API design, so is using the intended HTTP mechanics for making requests.

HTTP Request Methods

Like URI formatting, the HTTP methods to use for varying operations is widely agreed upon.

Method	Sample	Description
GET	/users	Retrieves a list of users
GET	/users/42	Retrieves user 42
POST	/users	Creates a new user
PUT	/users/42	Updates user 42
PATCH	/users/42	Partial update of user 42
DELETE	/users/42	Deletes user 42

Listing 10: HTTP Request Method Descriptions.

Response Status Codes

There are 39 defined HTTP status codes in the HTTP/1.1 specification [9]. There are also many status codes that have been expanded by vendors. In practice, an API needs only a fraction of them.

- **200 OK**: Successful request
- **201 Created**: Response to a POST creation
- **204 No Content**: Response to a successful request that won't be returning a body (like a DELETE request)
- **304 Not Modified**: Used when HTTP caching headers are in play

- **400 Bad Request**: The request is malformed, such as if the body does not parse
- **401 Unauthorized**: Invalid authentication
- **403 Forbidden**: Authenticated user doesn't have access to the resource
- **404 Not Found**: When requested resource does not exist
- **405 Method Not Allowed**: When an HTTP method does not exist
- ~~**420 Enhance Your Calm**~~: Requests are being rate limited [1]
- **429 Too Many Requests**: A request is rejected due to rate limiting

It is most important to make sure and use the correct HTTP status response code for the correct reason. A number of APIs return a 500 HTTP status code when the root cause is a poorly formatted request.

"Represent, Yo"

Representation design includes defining the message body format, the support representations (for example JSON or to a much lesser extent, XML).

Hypermedia

Hypermedia is a somewhat controversial topic in the REST world. Some, like Masse, think that it protects you as an API writer by giving you the flexibility to change relationship and sub-resource URIs. In reality, most developers are coding to specific URIs without regard for the possibility for change (see versioning section).

Media types

Specifying distinct media types on the request with an `Accept` header allows the client to control the format of the data being retrieved. The most common is JSON and XML (although XML usage is much less common in recent years). Specifying the media type as an `Accept` header is a much better alternative to having it be in the API. Many data providers allow you to append a file extenstion to a URI to specify the format. For example to retrieve a list of users in XML, `/v1/users` would become `/v1/users.xml`. This is a poor choice and `Accept` headers are much more clear and a natural use of the HTTP specification.

[1] The author was disappointed when Twitter changed to use the official 429 status code for rate limiting https://dev.twitter.com/overview/api/response-codes

ETags

ETags are an efficient way to check if an entity has changed and there is a new representational state version. When a request is made, a header is appended with a hash or checksum of the resource representation to retrieve. The `If-None-Match` header is sent and if it contains a matching ETag value generated on the server side, an HTTP status of `304 Not Modified` is sent.

Last-Modified

Similar to ETags, this header uses a timestamp (instead of a hash or checksum) to denote that the entity has changed.

Know Your Limits

Even the most popular and technologically advanced data providers limit their API usage. Doing so protects the data provider and provides corrective, actionable recommendations to clients.

Limits by Levels

A good API provides rate limits on varying levels. For example, a single, registered OAuth application may be limited by the client ID and secret being used. Another rate limit level could be imposed on a user (thus allowing that user to register multiple OAuth applications and control the data access via scopes). A third rate limit could also be registered on a customer level. Each level increasing in limit.

Clear Limit Rules & Instructions

As an API user, there are few things more frustrating (or panic inducing) than not knowing what your quota is or how to request an increase. The best way to inform a user about their quota usage is with HTTP response headers.

```
HTTP/1.1 200 OK
Content-Type: application/json
X-RateLimit: 1000
X-RateLimit-Remaining: 152
X-RateLimit-Reset: 2016-03-08T00:33:37.704Z
```

Listing 11: Sample rate limit HTTP response headers.

Larger Initial Quota

If possible, increase API limits or quotas for newly registered applications. This is especially important if your API is data intensive. Without an initial increased

quota, it often takes many days (or sometimes even weeks) to get the data needed. A higher initial quota allows a baseline retrieval of the data to be done quickly.

Versioning

An API, like all other software, evolves. As such, you need to account for differentiating future versions of your API. A pragmatic approach to versioning benefits the producer, consumer, and maintainer of the API.

Place the Version in the URI

REST purists often argue that a URI should only represent a resource. They argue that placing the version in the URI pollutes the entity. While this may be true (from a purity perspective), pragmatism should win in this scenario. It's more difficult and less apparent what version you're using without verbose logging or request inspections.

Use Simple Numbering

Use a lowercase letter "v" and a simple scalar number. Don't use major and minor versions or any variety of semantic versioning for REST APIs.

Good	Bad
/v1/users	/version1/users
	/ver2/users
	/v1.1/users
	/v1_2/users
	/ver1.2.1/users
	/ver2.1/users

The Left is the Best

An API version number should be placed in the leftmost portion of the URL as it increases hierarchical scope of that version. Doing so offers some practical benefits. For example, if you need to change authentication schemes in the future, having the version as the leftmost portion of your URI allows you to very easily route new versions of the API to a separate cluster of servers using a load balancer.

Don't Use Headers

The REST purist argument typically recommends placing the version of the API as a request header. This just complicates things.

Don't put it in a custom header like the following example.

```
GET /users HTTP/1.1
X-API-Version: 2
```

Listing 12: Incorrect versioning with a header

Also *don't* repurpose the "Accept" header either with a custom version (or a Content-Type header either).

```
GET /users HTTP/1.1
Accept: application/api.v2+json
```

Listing 13: Incorrect versioning overriding "Accept" header

The most simple and pragmatic way to version is in the path. If the REST purist in you is upset, just tell it that it's essentially part of the domain and that the true resource identifier starts *after* the version.

```
GET /v1/users HTTP/1.1
```

Listing 14: "Correct" way to version an API

Version Functionality Independently

Only increase the version of a logical group of functionality, typically by an entry point. If, for example, you need to change the contract on the user's resource from `v1/users` to `v2/users`, only change the version for users and leave the other resources at their current version. There are some valid times when the entire API should be versioned. Changes that impact all resources, such as an authentication change, merit an increase in version across all endpoints. As mentioned earlier, this is one of the reasons why placing the version number in the leftmost portion of the URL is recommended.

Errors

All errors should be descriptive and actionable. Remember that the primary audience of an API is the developer. Small actions can make a developer's task much easier. The best error responses also include actionable information for consumer of the API to learn more about the error. Direct links to documentation in the error are particularly helpful.

```
{
 "message": "Bad request: Invalid user",
 "description": "Missing role, see:
      http://developers.domo.com/api#users",
 "path": "/v1/users",
 "status": 400
}
```

Listing 15: Sample Error Body Response

Data

The large majority of APIs, at a fundamental level, simply provide a user with access to their data. As such, there are a number of features that can be added that will benefit the API consumer and improve efficiency of the API.

Refreshing Backfills

More and more of the data being stored and subsequently distributed by data providers has some element of sequencing, such as time series data. This means that data retrieved on one API call with query parameter date ranges can vary wildly from the next API call with the same parameter range. Data is eventually consistent, ordered, and indexed. However, the retrieval of the data has to be done several times and compared on the client side to identify what has changed.

Data providers with sequenced data can provide options to only retrieve data delta on specific range of past date queries. Instead of having to download all data multiple times just to pick up a few changes, delta data could be packaged and made available.

Data, Domain & UX Consistency

One of the most difficult and time consuming tasks for the connector team at Domo is the careful investigation of the data retrieved via an API in order to verify it matches the data exposed through the data providers' user interface. It's amazing how a small toggle in a query parameter can make reports become wildly different from what is being displayed.

A good API will describe how a user can mimic the logical representation of data done in a product.

Flattened, Raw Data

Fetching sequential data over time in JSON works but is inefficient. Instead of making several thousands of individual, paged requests, an alternative option of a good API allows for the fetching of flattened, raw data. Coupled with correct documentation of how that data is prepared and logically presented (as described above) the availability of raw, flattened data can greatly reduce computation load and bandwidth costs.

Authentication

Of the 500+ data providers Domo connects to, we see a wide variety of authentication schemes. These range from authentication tokens to username and passwords to standard OAuth 1, and OAuth 2.

OAuth 2

OAuth 2 is becoming the defacto authentication scheme for many data providers. Part of what is fueling this adoption is the flexibility of the OAuth 2 specification. There are many `grant types` that can be implemented that allow an API writer to control the complexity. The OAuth specification is defined in RFC 6749 [10].

There are a number of flows or grant types in the specification as well as extensibility to add your own.

- Authorization Grant
- Authorization Code
- Implicit
- Password Credentials
- Client Credentials
- Access Token
- Refresh Token

Hide Architectural Complexities

One unfortunate side effect regarding the flexibility of the OAuth 2 specification is that many OAuth providers have taken the liberty to exposing their architectural complexities as part of their implementation.

Many OAuth 2 providers require that you include a non-standard header or non-standard query parameter that force you to use a custom client library or hand code the HTTP interactions yourself. A good API will hide those complexities and not expose them to the customer either through token enhancement (adding additional data to the token) or associating routing information on a client creation.

Reasonable Access Token Expiration

At Domo, we encounter wide variation of access token expiration policies. They can range from never expiring all the way down to 10 minutes. From a developer perspective, it is very difficult to have a fully-functioning authentication process before we even begin to explore a

new API. We recommend an expiration time of one hour. This allows you, as an API provider, to use non database driven token technology like JWT to provide significant performance improvement.

Conclusion

An API is considered a success if it helps developers achieve their goals of integration and incorporation of the data provider's data. This is done via consistent design and implementation, a highly available infrastructure, and good performance. Building an ecosystem around your data can prove to be immensely valuable.

References

[1] M. Andreessen, "Why software is eating the world." [Online]. Available: http://www.wsj.com/articles/SB10001424053111903480904576512250915629460

[2] B. Mulloy, "API Facade Pattern: A Simple Interface to a Complex System," 2012, [Online; access February 26, 2016]. [Online]. Available: https://pages.apigee.com/api-facade-pattern-ebook.html

[3] B. Christensen, "Optimizing the Netflix API," 2013, [Online; access February 23, 2016]. [Online]. Available: http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

[4] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[5] R. T. Fielding, "Architectural styles and the design of network-based software architectures." [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[6] R. T. F. e. a. Berners-Lee, Tim, "Uniform resource identifier (uri): Generic syntax." [Online]. Available: http://www.rfc-editor.org/rfc/rfc3986.txt

[7] T. Berners-Lee, R. Fielding, and L. Masinter, "Rfc 3986," *Uniform Resource Identifier (URI): Generic Syntax*, 2005.

[8] M. Masse, *REST API design rulebook*. " O'Reilly Media, Inc.", 2011.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1, 1999," *RFC2616*, 2006.

[10] D. Hardt, "The oauth 2.0 authorization framework (rfc6749)."

About Domo Domo is a cloud-based business management platform that transforms the way business is run. Domo gives CEOs and decision makers across the organization the confidence to make faster, more effective decisions and improve business results by giving them one place to easily access all the information they need.

With more than $450 million in funding, Domo is backed by an all-star list of angels and investors including Benchmark, BlackRock, Capital Group, Fidelity Investments, Founders Fund, GGV Capital, Glynn Capital, Greylock Partners, IVP, salesforce.com, TPG Growth, T. Rowe Price, WPP and Zetta Venture Partners, plus CEOs of the world's largest SaaS and Internet businesses.

Domo's founding team consists of some of the most sought after talent in the industry with experience that includes Amazon, American Express, Ancestry.com, eBay, Endeca, Facebook, Google, LinkedIn, MLB.com, Omniture, salesforce.com, SuccessFactors and SAP. For more information, visit www.domo.com.