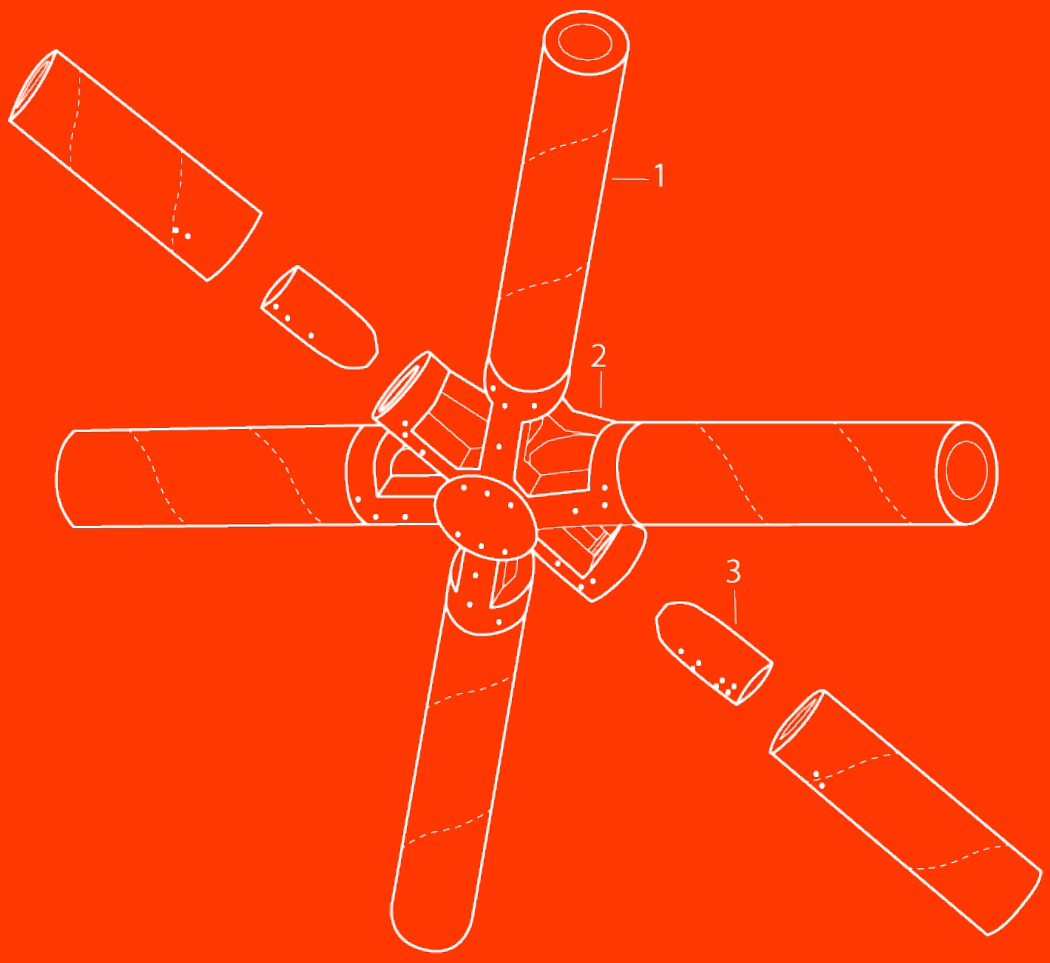
Web API Design

Crafting Interfaces that Developers Love



apigee

Brian Mulloy

Table of Contents

Web API Design - Crafting Interfaces that Developers Love

Introduction

If you're reading this, chances are that you care about designing Web APIs that developers will love and that you're interested in applying proven design principles and best practices to your Web API.

One of the sources for our design thinking is REST. Because REST is an architectural style and not a strict standard, it allows for a lot of flexibly. Because of that flexibility and freedom of structure, there is also a big appetite for design best practices.

This e-book is a collection of design practices that we have developed in collaboration with some of the leading API teams around the world, as they craft their API strategy through a design workshop that we provide at Apigee.

We call our point of view in API design "pragmatic REST", because it places the success of the developer over and above any other design principle. The developer is the customer for the Web API. The success of an API design is measured by how quickly developers can get up to speed and start enjoying success using your API.

We'd love your feedback – whether you agree, disagree, or have some additional practices and tips to add. The API Craft Google Group is a place where Web API design enthusiasts come together to share and debate design practices – we'd love to see you there.

Are you a Pragmatist or a RESTafarian?

Let's start with our overall point of view on API design. We advocate pragmatic, not dogmatic REST. What do we mean by dogmatic?

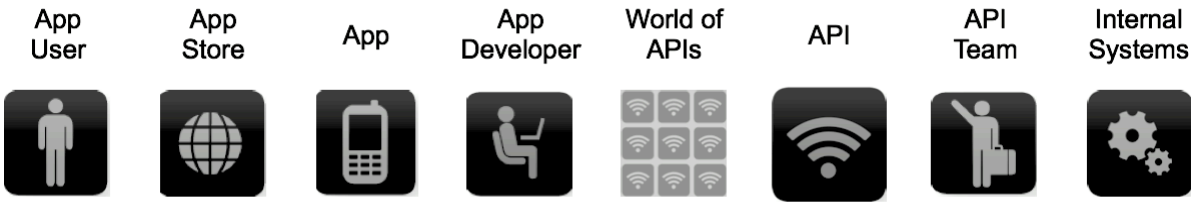
You might have seen discussion threads on true REST - some of them can get pretty strict and wonky. Mike Schinkel sums it up well - defining a RESTafarian as follows:

"A RESTifarian *is a zealous proponent of the REST software architectural style as defined by Roy T. Fielding in Chapter 5 of his PhD. dissertation at UC Irvine. You can find RESTifarians in the wild on the REST-discuss mailing list. But be careful, RESTifarians can be extremely meticulous when discussing the finer points of REST …"*

Our view: approach API design from the 'outside-in' perspective. This means we start by asking - what are we trying to achieve with an API?

The API's job is to make the developer as successful as possible. The orientation for APIs is to think about design choices from the application developer's point of view.

Why? Look at the value chain below – the application developer is the lynchpin of the entire API strategy. The primary design principle when crafting your API should be to maximize developer productivity and success. This is what we call pragmatic REST.



Pragmatic REST is a design problem

You have to get the design right, because design communicates how something will be used. The question becomes - what is the design with optimal benefit for the app developer?

The developer point of view is the guiding principle for all the specific tips and best practices we've compiled.

Nouns are good; verbs are bad

The number one principle in pragmatic RESTful design is: keep simple things simple.

Keep your base URL simple and intuitive

The base URL is the most important design affordance of your API. A simple and intuitive base URL design makes using your API easy.

Affordance is a design property that communicates how something should be used without requiring documentation. A door handle's design should communicate whether you pull or push. Here's an example of a conflict between design affordance and documentation - not an intuitive interface!

A key litmus test we use for Web API design is that there should be only **2 base URLs per resource**. Let's model an API around a simple object or resource, a dog, and create a Web API for it.

The first URL is for a collection; the second is for a specific element in the collection.

/dogs /dogs/1234

Boiling it down to this level will also force the verbs out of your base URLs.

Keep verbs out of your base URLs

Many Web APIs start by using a method-driven approach to URL design. These method-based URLs sometimes contain verbs - sometimes at the beginning, sometimes at the end.

For any resource that you model, like our dog, you can never consider one object in isolation. Rather, there are always related and interacting resources to account for - like owners, veterinarians, leashes, food, squirrels, and so on.

Think about the method calls required to address all the objects in the dogs' world. The URLs for our resource might end up looking something like this.

```
...
/getAllDogs
/verifyLocation
/feedNeeded
/createRecurringWakeUp
/giveDirectOrder
/checkHealth
/getRecurringWakeUpSchedule
/getLocation
/getDog
/newDog
/getNewDogsSince
/getRedDogs
/getSittingDogs
/setDogStateTo
/replaceSittingDogsWithRunningDogs
/saveDog
...
```

```
...
/getAllLeashedDogs
/verifyVeterinarianLocation
/feedNeededFood
/createRecurringMedication
/doDirectOwnerDiscipline
/doExpressCheckupWithVeterinarian
/getRecurringFeedingSchedule
/getHungerLevel
/getSquirrelsChasingPuppies
/newDogForOwner
/getNewDogsAtKennelSince
/getRedDogsWithoutSiblings
/getSittingDogsAtPark
/setLeashedDogStateTo
/replaceParkSittingDogsWithRunningDogs
/saveMommaDogsPuppies
...
```

It's a slippery slope - soon you have a long list of URLs and no consistent pattern making it difficult for developers to learn how to use your API.

Use HTTP verbs to operate on the collections and elements.

For our dog resources, we have two base URLs that use nouns as labels, and we can operate on them with HTTP verbs. Our HTTP verbs are **POST**, **GET**, **PUT**, and **DELETE**. (We think of them as mapping to the acronym, **CRUD** (**C**reate-**R**ead-**U**pdate-**D**elete).)

With our two resources (`/dogs` and `/dogs/1234`) and the four HTTP verbs, we have a rich set of capability that's intuitive to the developer. Here is a chart that shows what we mean for our dogs.

Resource	POST create	GET read	PUT update	DELETE delete
/dogs	Create a new dog	List dogs	Bulk update dogs	Delete all dogs
/dogs/1234	Error	Show Bo	If exists update Bo If not error	Delete Bo

The point is that developers probably don't need the chart to understand how the API behaves. They can experiment with and learn the API without the documentation.

In summary:

Use two base URLs per resource.

Keep verbs out of your base URLs.

Use HTTP verbs to operate on the collections and elements.

Plural nouns and concrete names

Let's explore how to pick the nouns for your URLs.

Should you choose singular or plural nouns for your resource names? You'll see popular APIs use both. Let's look at a few examples:

Foursquare	**GroupOn**	**Zappos**
/checkins	/deals	/Product

Given that the first thing most people probably do with a RESTful API is a GET, we think it reads more easily and is more intuitive to use plural nouns. But above all, avoid a mixed model in which you use singular for some resources, plural for others. Being consistent allows developers to predict and guess the method calls as they learn to work with your API.

Concrete names are better than abstract
Achieving pure abstraction is sometimes a goal of API architects. However, that abstraction is not always meaningful for developers.

Take for example an API that accesses content in various forms - blogs, videos, news articles, and so on.

An API that models everything at the highest level of abstraction - as `/items` or `/assets` in our example - loses the opportunity to paint a tangible picture for developers to know what they can do with this API. It is more compelling and useful to see the resources listed as blogs, videos, and news articles.

The level of abstraction depends on your scenario. You also want to expose a manageable number of resources. Aim for concrete naming and to keep the number of resources between 12 and 24.

In summary, an intuitive API uses plural rather than singular nouns, and concrete rather than abstract names.

Simplify associations - sweep complexity under the '?'

In this section, we explore API design considerations when handling associations between resources and parameters like states and attributes.

Associations
Resources almost always have relationships to other resources. What's a simple way to express these relationships in a Web API?

Let's look again at the API we modeled in nouns are good, verbs are bad - the API that interacts with our **dogs** resource. Remember, we had two base URLs: /dogs and dogs/1234.

We're using HTTP verbs to operate on the resources and collections. Our dogs belong to owners. To get all the dogs belonging to a specific owner, or to create a new dog for that owner, do a `GET` or a POST:

```
GET /owners/5678/dogs
```

```
POST /owners/5678/dogs
```

Now, the relationships can be complex. Owners have relationships with veterinarians, who have relationships with dogs, who have relationships with food, and so on. It's not uncommon to see people string these together making a URL 5 or 6 levels deep. Remember that once you have the primary key for one level, you usually don't need to include the levels above because you've already got your specific object. In other words, you shouldn't need too many cases where a URL is deeper than what we have above `/resource/identifier/resource.`

Sweep complexity behind the '?'
Most APIs have intricacies beyond the base level of a resource. Complexities can include many states that can be updated, changed, queried, as well as the attributes associated with a resource.

Make it simple for developers to use the base URL by putting optional states and attributes behind the HTTP question mark. To get all red dogs running in the park:

```
GET /dogs?color=red&state=running&location=park
```

In summary, keep your API intuitive by simplifying the associations between resources, and sweeping parameters and other complexities under the rug of the HTTP question mark.

Handling errors

Many software developers, including myself, don't always like to think about exceptions and error handling but it is a very important piece of the puzzle for any software developer, and especially for API designers.

Why is good error design especially important for API designers?

From the perspective of the developer consuming your Web API, everything at the other side of that interface is a black box. Errors therefore become a key tool providing context and visibility into how to use an API.

First, developers learn to write code through errors. The "test-first" concepts of the extreme programming model and the more recent "test driven development" models represent a body of best practices that have evolved because this is such an important and natural way for developers to work.

Secondly, in addition to when they're developing their applications, developers depend on well-designed errors at the critical times when they are troubleshooting and resolving issues after the applications they've built using your API are in the hands of their users.

How to think about errors in a pragmatic way with REST?

Let's take a look at how three top APIs approach it.

Facebook

```
HTTP Status Code: 200

{"type" : "OauthException", "message":"(#803) Some of the
aliases you requested do not exist: foo.bar"}
```

Twilio

```
HTTP Status Code: 401

{"status" : "401", "message":"Authenticate","code": 20003, "more
info": "http://www.twilio.com/docs/errors/20003"}
```

SimpleGeo

```
HTTP Status Code: 401

{"code" : 401, "message": "Authentication Required"}
```

Facebook
No matter what happens on a Facebook request, you get back the 200-status code - everything is OK. Many error messages also push down into the HTTP response. Here they also throw an #803 error but with no information about what #803 is or how to react to it.

Twilio
Twilio does a great job aligning errors with HTTP status codes. Like Facebook, they provide a more granular error message but with a link that takes you to the documentation. Community commenting and discussion on the documentation helps to build a body of information and adds context for developers experiencing these errors.

SimpleGeo
SimpleGeo provides error codes but with no additional value in the payload.

A couple of best practices

Use HTTP status codes
Use HTTP status codes and try to map them cleanly to relevant standard-based codes.

There are over 70 HTTP status codes. However, most developers don't have all 70 memorized. So if you choose status codes that are not very common you will force application developers away from building their apps and over to Wikipedia to figure out what you're trying to tell them.

Therefore, most API providers use a small subset. For example, the Google GData API uses only 10 status codes; Netflix uses 9, and Digg, only 8.

Google GData
200 201 304 400 401 403 404 409 410 500

Netflix
200 201 304 400 401 403 404 412 500

Digg
200 400 401 403 404 410 500 503

How many status codes should you use for your API?

When you boil it down, there are really only 3 outcomes in the interaction between an app and an API:

- Everything worked - success
- The application did something wrong – client error
- The API did something wrong – server error

Start by using the following 3 codes. If you need more, add them. But you shouldn't need to go beyond 8.

- 200 - OK
- 400 - Bad Request
- 500 - Internal Server Error

If you're not comfortable reducing all your error conditions to these 3, try picking among these additional 5:

- 201 - Created
- 304 - Not Modified
- 404 – Not Found
- 401 - Unauthorized
- 403 - Forbidden

(Check out this good Wikipedia entry for all HTTP Status codes.)

It is important that the code that is returned can be consumed and acted upon by the application's business logic - for example, in an if-then-else, or a case statement.

Make messages returned in the payload as verbose as possible.

Code for code

```
200 – OK
401 – Unauthorized
```

Message for people

```
{"developerMessage" : "Verbose, plain language description of
the problem for the app developer with hints about how to fix
it.", "userMessage":"Pass this message on to the app user if
needed.", "errorCode" : 12345, "more info":
"http://dev.teachdogrest.com/errors/12345"}
```

In summary, be verbose and use plain language descriptions. Add as many hints as your API team can think of about what's causing an error.

We highly recommend you add a link in your description to more information, like Twilio does.

Tips for versioning

Versioning is one of the most important considerations when designing your Web API.

Never release an API without a version and make the version mandatory.

Let's see how three top API providers handle versioning.

Twilio	/2010-04-01/Accounts/
salesforce.com	/services/data/v20.0/sobjects/Account
Facebook	?v=1.0

Twilio uses a timestamp in the URL (note the European format).

At compilation time, the developer includes the timestamp of the application when the code was compiled. That timestamp goes in all the HTTP requests.

When a request arrives, Twilio does a look up. Based on the timestamp they identify the API that was valid when this code was created and route accordingly.

It's a very clever and interesting approach, although we think it is a bit complex. For example, it can be confusing to understand whether the timestamp is the compilation time or the timestamp when the API was released.

Salesforce.com uses v20.0, placed somewhere in the middle of the URL.

We like the use of the **v.** notation. However, we don't like using the **.0** because it implies that the interface might be changing more frequently than it should. The logic behind an interface can change rapidly but the interface itself shouldn't change frequently.

Facebook also uses the v. notation but makes the version an optional parameter.

This is problematic because as soon as Facebook forced the API up to the next version, all the apps that didn't include the version number broke and had to be pulled back and version number added.

How to think about version numbers in a pragmatic way with REST?

Never release an API without a version. Make the version mandatory.

Specify the version with a **'v'** prefix. Move it all the way to the left in the URL so that it has the highest scope (e.g. `/v1/dogs`).

Use a simple ordinal number. Don't use the dot notation like v1.2 because it implies a granularity of versioning that doesn't work well with APIs--it's an interface not an implementation. Stick with v1, v2, and so on.

How many versions should you maintain? Maintain at least one version back.

For how long should you maintain a version? Give developers at least one cycle to react before obsoleting a version.

Sometimes that's 6 months; sometimes it's 2 years. It depends on your developers' development platform, application type, and application users. For example, mobile apps take longer to rev' than Web apps.

Should version and format be in URLs or headers?

There is a strong school of thought about putting format and version in the header.

Sometimes people are forced to put the version in the header because they have multiple inter-dependent APIs. That is often a symptom of a bigger problem, namely, they are usually exposing their internal mess instead of creating one, usable API facade on top.

That's not to say that putting the version in the header is a symptom of a problematic API design. It's not!

In fact, using headers is more correct for many reasons: it leverages existing HTTP standards, it's intellectually consistent with Fielding's vision, it solves some hard real-world problems related to inter-dependent APIs, and more.

However, we think the reason most of the popular APIs do not use it is because it's less fun to hack in a browser.

Simple rules we follow:

If it changes the logic you write to handle the response, put it in the URL so you can see it easily.

If it doesn't change the logic for each response, like OAuth information, put it in the header.

These for example, all represent the same resource:

```
dogs/1
Content-Type: application/json

dogs/1
Content-Type: application/xml

dogs/1
Content-Type: application/png
```

The code we would write to handle the responses would be very different.

There's no question the header is more correct and it is still a very strong API design.

Pagination and partial response

Partial response allows you to give developers just the information they need.

Take for example a request for a tweet on the Twitter API. You'll get much more than a typical twitter app often needs - including the name of person, the text of the tweet, a timestamp, how often the message was re-tweeted, and a lot of metadata.

Let's look at how several leading APIs handle giving developers just what they need in responses, including Google who pioneered the idea of **partial response**.

LinkedIn

```
/people:(id,first-name,last-name,industry)
```

This request on a person returns the ID, first name, last name, and the industry.

LinkedIn does partial selection using this terse `:(...)` syntax which isn't self-evident. Plus it's difficult for a developer to reverse engineer the meaning using a search engine.

Facebook

```
/joe.smith/friends?fields=id,name,picture
```

Google

```
?fields=title,media:group(media:thumbnail)
```

Google and Facebook have a similar approach, which works well.

They each have an optional parameter called fields after which you put the names of fields you want to be returned.

As you see in this example, you can also put sub-objects in responses to pull in other information from additional resources.

Add optional fields in a comma-delimited list

The Google approach works extremely well.

Here's how to get just the information we need from our dogs API using this approach:

```
/dogs?fields=name,color,location
```

It's simple to read; a developer can select just the information an app needs at a given time; it cuts down on bandwidth issues, which is important for mobile apps.

The partial selection syntax can also be used to include associated resources cutting down on the number of requests needed to get the required information.

Make it easy for developers to paginate objects in a database

It's almost always a bad idea to return every resource in a database.

Let's look at how Facebook, Twitter, and LinkedIn handle pagination. Facebook uses **offset** and **limit**. Twitter uses page and **rpp** (records per page). LinkedIn uses **start** and **count**

Semantically, Facebook and LinkedIn do the same thing. That is, the LinkedIn start & count is used in the same way as the Facebook offset & limit.

To get records 50 through 75 from each system, you would use:

- Facebook - **offset 50** and **limit 25**
- Twitter - **page 3** and **rpp 25** (records per page)
- LinkedIn - **start 50** and **count 25**

Use limit and offset

We recommend limit and offset. It is more common, well understood in leading databases, and easy for developers.

```
/dogs?limit=25&offset=50
```

Metadata

We also suggest including metadata with each response that is paginated that indicated to the developer the total number of records available.

What about defaults?

My loose rule of thumb for default pagination is **limit=10** with **offset=0.** (`limit=10&offset=0`)

The pagination defaults are of course dependent on your data size. If your resources are large, you probably want to limit it to fewer than 10; if resources are small, it can make sense to choose a larger limit.

In summary:

Support **partial response** by adding optional fields in a **comma delimited list.**

Use **limit and offset** to make it easy for developers to paginate objects.

What about responses that don't involve resources?

API calls that send a response that's not a resource ***per se*** are not uncommon depending on the domain. We've seen it in financial services, Telco, and the automotive domain to some extent.

Actions like the following are your clue that you might not be dealing with a "resource" response.

Calculate
Translate
Convert

For example, you want to make a simple algorithmic calculation like how much tax someone should pay, or do a natural language translation (one language in request; another in response), or convert one currency to another. None involve resources returned from a database.

In these cases:

Use verbs not nouns

For example, an API to convert 100 euros to Chinese Yen:

```
/convert?from=EUR&to=CNY&amount=100
```

Make it clear in your API documentation that these "non-resource" scenarios are different.

Simply separate out a section of documentation that makes it clear that you use verbs in cases like this – where some action is taken to generate or calculate the response, rather than returning a resource directly.

Supporting multiple formats

We recommend that you support more than one format - that you push things out in one format and accept as many formats as necessary. You can usually automate the mapping from format to format.

Here's what the syntax looks like for a few key APIs.

Google Data

```
?alt=json
```

Foursquare

```
/venue.json
```

Digg*

```
Accept: application/json
?type=json
```

* The type argument, if present, overrides the `Accept` header.

Digg allows you to specify in two ways: in a pure RESTful way in the Accept header or in the type parameter in the URL. This can be confusing - at the very least you need to document what to do if there are conflicts.

We recommend the Foursquare approach.

To get the JSON format from a collection or specific element:

```
dogs.json
```

```
/dogs/1234.json
```

Developers and even casual users of any file system are familiar to this dot notation. It also requires just one additional character (the period) to get the point across.

What about default formats?

In my opinion, JSON is winning out as the default format. JSON is the closest thing we have to universal language. Even if the back end is built in Ruby on Rails, PHP, Java, Python etc., most projects probably touch JavaScript for the front-end. It also has the advantage of being terse - less verbose than XML.

What about attribute names?

In the previous section, we talked about formats - supporting multiple formats and working with JSON as the default.

This time, let's talk about what happens when a response comes back.

You have an object with data attributes on it. How should you name the attributes?

Here are API responses from a few leading APIs:

Twitter
```
"created_at": "Thu Nov 03 05:19;38 +0000 2011"
```

Bing
```
"DateTime": "2011-10-29T09:35:00Z"
```

Foursquare
```
"createdAt": 1320296464
```

They each use a different code convention. Although the Twitter approach is familiar to me as a Ruby on Rails developer, we think that Foursquare has the best approach.

How does the API response get back in the code? You parse the response (JSON parser); what comes back populates the Object. It looks like this

```
var myObject = JSON.parse(response);
```

If you chose the Twitter or Bing approach, your code looks like this. Its not JavaScript convention and looks weird - looks like the name of another object or class in the system, which is not correct.

```
timing = myObject.created_at;

timing - myObject.DateTime;
```

Recommendations

- Use JSON as default
- Follow JavaScript conventions for naming attributes

  - Use medial capitalization (aka CamelCase)

  - Use uppercase or lowercase depending on type of object

This results in code that looks like the following, allowing the JavaScript developer to write it in a way that makes sense for JavaScript.

```
"createdAt": 1320296464

timing = myObject.createdAt;
```

Tips for search

While a simple search could be modeled as a resourceful API (for example, `dogs/?q=red`), a more complex search across multiple resources requires a different design.

This will sound familiar if you've read the topic about using verbs not nouns when results don't return a resource from the database - rather the result is some action or calculation.

If you want to do a global search across resources, we suggest you follow the Google model:

Global search

```
/search?q=fluffy+fur
```

Here, search is the verb; **?q** represents the query.

Scoped search

To add scope to your search, you can prepend with the scope of the search. For example, search in dogs owned by resource ID 5678

```
/owners/5678/dogs?q=fluffy+fur
```

Notice that we've dropped the explicit search in the URL and rely on the parameter 'q' to indicate the scoped query. (Big thanks to the contributors on the API Craft Google group for helping refine this approach.)

Formatted results

For search or for any of the action oriented (non-resource) responses, you can prepend with the format as follows:

```
/search.xml?q=fluffy+fur
```

Consolidate API requests in one subdomain

We've talked about things that come after the top-level domain. This time, let's explore stuff on the other side of the URL.

Here's how Facebook, Foursquare, and Twitter handle this:

Facebook provides two APIs. They started with api.facebook.com, then modified it to orient around the social graph graph.facebook.com.

```
graph.facebook.com
api.facebook.com
```

Foursquare has one API.

```
api.foursquare.com
```

Twitter has three APIs; two of them focused on search and streaming.

```
stream.twitter.com
api.twitter.com
search.twitter.com
```

It's easy to understand how Facebook and Twitter ended up with more than one API. It has a lot to do with timing and acquisition, and it's easy to reconfigure a CName entry in your DNS to point requests to different clusters.

But if we're making design decisions about what's in the best interest of app developer, we recommend following Foursquare's lead:

Consolidate all API requests under one API subdomain.

It's cleaner, easier and more intuitive for developers who you want to build cool apps using your API.

Facebook, Foursquare, and Twitter also all have dedicated developer portals.

```
developers.facebook.com
developers.foursquare.com
dev.twitter.com
```

How to organize all of this?

Your API gateway should be the top-level domain. For example, `api.teachdogrest.com`

In keeping with the spirit of REST, your developer portal should follow this pattern: `developers.yourtopleveldomain`. For example,

`developers.teachdogrest.com`

Do Web redirects

Then optionally, if you can sense from requests coming in from the browser where the developer really needs to go, you can redirect.

Say a developer types **`api.teachdogrest.com`** in the browser but there's no other information for the GET request, you can probably safely redirect to your developer portal and help get the developer where they really need to be.

api →developers (if from browser)

dev → developers

developer → developers

Tips for handling exceptional behavior

So far, we've dealt with baseline, standard behaviors.

Here we'll explore some of the **exceptions** that can happen - when clients of Web APIs can't handle all the things we've discussed. For example, sometimes clients intercept HTTP error codes, or support limited HTTP methods.

What are ways to handle these situations and work within the limitations of a specific client?

When a client intercepts HTTP error codes

One common thing in some versions of Adobe Flash - if you send an HTTP response that is anything other than HTTP 200 OK, the Flash container intercepts that response and puts the error code in front of the end user of the app.

Therefore, the app developer doesn't have an opportunity to intercept the error code. App developers need the API to support this in some way.

Twitter does an excellent job of handling this.

They have an optional parameter **`suppress_response_codes`**. If **`suppress_response_codes`** is set to true, the HTTP response is always 200.

```
/public_timelines.json?
suppress_response_codes=true
HTTP status code: 200 {"error":"Could not authenticate you."}
```

Notice that this parameter is a big verbose response code. (They could have used something like **`src`**, but they opted to be verbose.)

This is important because when you look at the URL, you need to see that the response codes are being suppressed as it has huge implications about how apps are going to respond to it.

Overall recommendations:

1 - Use suppress_response_codes = true

2 - The HTTP code is no longer just for the code

The rules from our previous Handling Errors section change. In this context, the HTTP code is no longer just for the code - the program - it's now to be ignored. Client apps are never going to be checking the HTTP status code, as it is always the same.

3 - Push any response code that we would have put in the HTTP response down into the response message

In my example below, the response code is 401. You can see it in the response message. Also include additional error codes and verbose information in that message.

Always return OK
```
/dogs?suppress_response_codes = true
```

Code for ignoring
```
200 - OK
```

Message for people & code
```
{response_code" : 401, "message" : "Verbose, plain language
description of the problem with hints about how to fix it."
"more_info" : "http://dev.tecachdogrest.com/errors/12345",
"code" : 12345}
```

When a client supports limited HTTP methods

It is common to see support for GET and POST and not PUT and DELETE.

To maintain the integrity of the four HTTP methods, we suggest you use the following methodology commonly used by Ruby on Rails developers:

Make the method an optional parameter in the URL.

Then the HTTP verb is always a GET but the developer can express rich HTTP verbs and still maintain a RESTful clean API.

Create
```
/dogs?method=post
```

Read
```
/dogs
```

Update
```
/dogs/1234?method=put&location=park
```

Delete
```
/dogs/1234?method=delete
```

WARNING: *It can be dangerous to provide post or delete capabilities using a GET method because if the URL is in a Web page then a Web crawler like the Googlebot can create or destroy lots of content inadvertently. Be sure you understand the implications of supporting this approach for your applications' context.*

Authentication

There are many schools of thought. My colleagues at Apigee and I don't always agree on how to handle authentication - but overall here's my take.

Let's look at these three top services. See how each of these services handles things differently:

PayPal
Permissions Service API

Facebook
OAuth 2.0

Twitter
OAuth 1.0a

Note that PayPal's proprietary three-legged permissions API was in place long before OAuth was conceived.

What should you do?

Use the latest and greatest OAuth - OAuth 2.0 (as of this writing). It means that Web or mobile apps that expose APIs don’t have to share passwords. It allows the API provider to revoke tokens for an individual user, for an entire app, without requiring the user to change their original password. This is critical if a mobile device is compromised or if a rogue app is discovered.

Above all, OAuth 2.0 will mean improved security and better end-user and consumer experiences with Web and mobile apps.

Don't do something *like* OAuth, but different. It will be frustrating for app developers if they can't use an OAuth library in their language because of your variation.

Making requests on your API

Lets take a look at what some API requests and responses look like for our dogs API.

Create a brown dog named Al

```
POST /dogs
name=Al&furColor=brown
```

Response

```
200 OK

{
"dog":{
"id": "1234",
"name": "Al",
"furColor": "brown"
}
}
```

Rename Al to Rover - Update

```
PUT /dogs/1234
name=Rover
```

Response

```
200 OK

{
"dog":{
"id":"1234",
"name": "Rover",
"furColor": "brown"

}
}
```

Tell me about a particular dog

```
GET /dogs/1234
```

Response

```
200 OK

{
"dog":{
"id":"1234",
"name": "Rover",
"furColor": "brown"
}
}
```

Tell me about all the dogs

```
GET /dogs
```

Response

```
200 OK

{
"dogs":
[{"dog":{
"id":"1233",
"name": "Fido",
"furColor": "white"}},
{"dog":{
"id":"1234",
"name": "Rover",
"furColor": "brown"}}]

"_metadata":

[{"totalCount":327,"limit":25,"offset":100}]
}
```

Delete Rover :-(

```
DELETE /dogs/1234
```

Response

```
200 OK
```

Chatty APIs

Let's think about how app developers use that API you're designing and dealing with chatty APIs.

Imagine how developers will use your API
When designing your API and resources, try to imagine how developers will use it to say construct a user interface, an iPhone app, or many other apps.

Some API designs become very chatty - meaning just to build a simple UI or app, you have dozens or hundreds of API calls back to the server.

The API team can sometimes decide not to deal with creating a nice, resource-oriented RESTful API, and just fall back to a mode where they create the 3 or 4 Java-style getter and setter methods they know they need to power a particular user interface.

We don't recommend this. You can design a RESTful API and still mitigate the chattiness.

Be complete and RESTful and provide shortcuts

First design your API and its resources according to pragmatic RESTful design principles and then provide shortcuts.

What kind of shortcut? Say you know that 80% of all your apps are going to need some sort of composite response, then build the kind of request that gives them what they need.

Just don't do the latter instead of the former. First design using good pragmatic RESTful principles!

Take advantage of the partial response syntax

The partial response syntax discussed in a previous section can help.

To avoid creating one-off base URLs, you can use the partial response syntax to drill down to dependent and associated resources.

In the case of our dogs API, the dogs have association with owners, who in turn have associations with veterinarians, and so on. Keep nesting the partial response syntax using dot notation to get back just the information you need.

```
/owners/5678?fields=name,dogs.name
```

Complement with an SDK

It's a common question for API providers - do you need to complement your API with code libraries and software development kits (SDKs)?

If your API follows good design practices, is self consistent, standards-based, and well documented, developers may be able to get rolling without a client SDK. Well-documented code samples are also a critical resource.

On the other hand, what about the scenario in which building a UI requires a lot of domain knowledge? This can be a challenging problem for developers even when building UI and apps on top of APIs with pretty simple domains – think about the Twitter API with it's primary object of 140 characters of text.

You shouldn't change your API to try to overcome the domain knowledge hurdle. Instead, you can complement your API with code libraries and a software development kit (SDK).

In this way, you don't overburden your API design. Often, a lot of what's needed is on the client side and you can push that burden to an SDK.

The SDK can provide the platform-specific code, which developers use in their apps to invoke API operations - meaning you keep your API clean.

Other reasons you might consider complementing your API with an SDK include the following:

Speed adoption on a specific platform. (For example Objective C SDK for iPhone.) Many experienced developers are just starting off with objective C+ so an SDK might be helpful.

Simplify integration effort required to work with your API - If key use cases are complex or need to be complemented by standard on-client processing.

An SDK can help reduce bad or inefficient code that might slow down service for everyone.

As a developer resource - good SDKs are a forcing function to create good source code examples and documentation. Yahoo! and Paypal are good examples:

Yahoo! http://developer.yahoo.com/social/sdk/

Paypal https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/library_download_sdks

To market your API to a specific community - you upload the SDK to a samples or plug-in page on a platform's existing developer community.

The API Façade Pattern

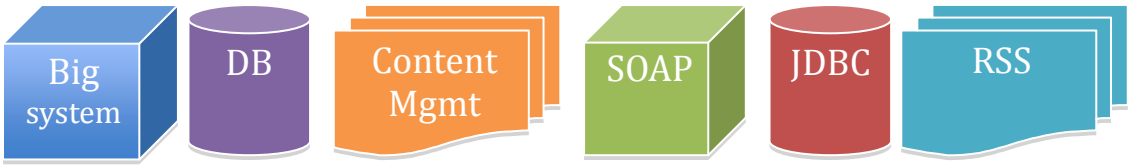
At this point, you may be asking -

What should we be thinking from an architectural perspective?

How do we follow all these best practice guidelines, expose my internal services and systems in a way that's useful to app developers, and still iterate and maintain my API?

Back-end systems of record are often too complex to expose directly to application developers. They are stable (have been hardened over time) and dependable (they are running key aspects of you business), but they are often based on legacy technologies and not always easy to expose to Web standards like HTTP. These systems can also have complex interdependencies and they change slowly meaning that they can't move as quickly as the needs of mobile app developers and keep up with changing formats.

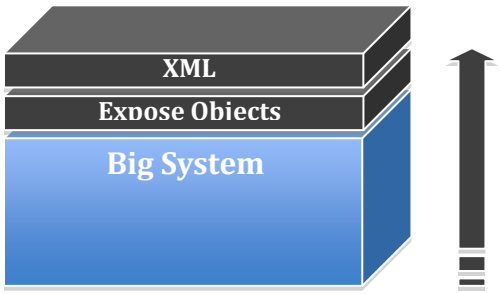
In fact, the problem is not creating an API for just one big system but creating an API for an array of complementary systems that all need to be used to make an API valuable to a developer.



It's useful to talk through a few anti-patterns that we've seen. Let's look at why we believe they don't work well.

The Build Up Approach

In the build-up approach, a developer exposes the core objects of a big system and puts an XML parsing layer on top.

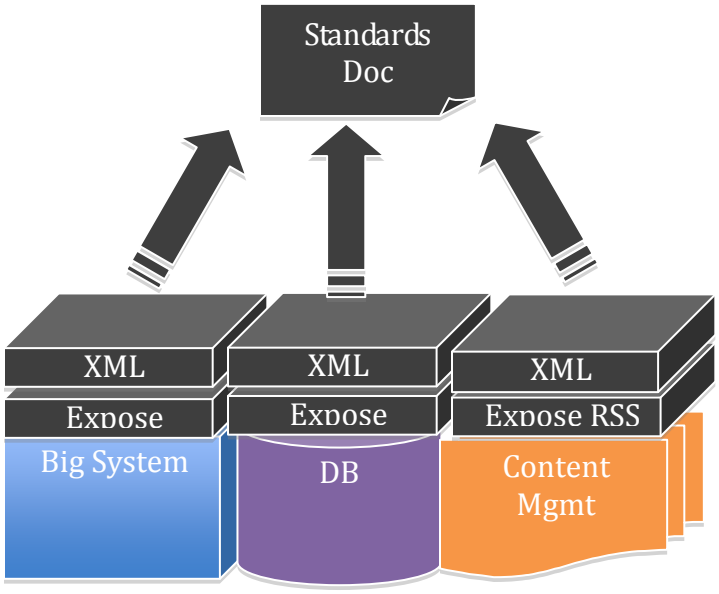


This approach has merit in that it can get you to market with version 1 quickly. Also, your API team members (your internal developers) already understand the details of the system.

Unfortunately, those details of an internal system at the object level are fine grained and can be confusing to external developers. You're also exposing details of internal architecture, which is rarely a good idea. This approach can be inflexible because you have 1:1 mapping to how a system works and how it is exposed to API. In short, building up from the systems of record to the API can be overly complicated.

The Standards Committee Approach

Often the internal systems are owned and managed by different people and departments with different views about how things should work. Designing an API by a standards committee often involves creating a standards document, which defines the schema and URLs and such. All the stakeholders build toward that common goal.

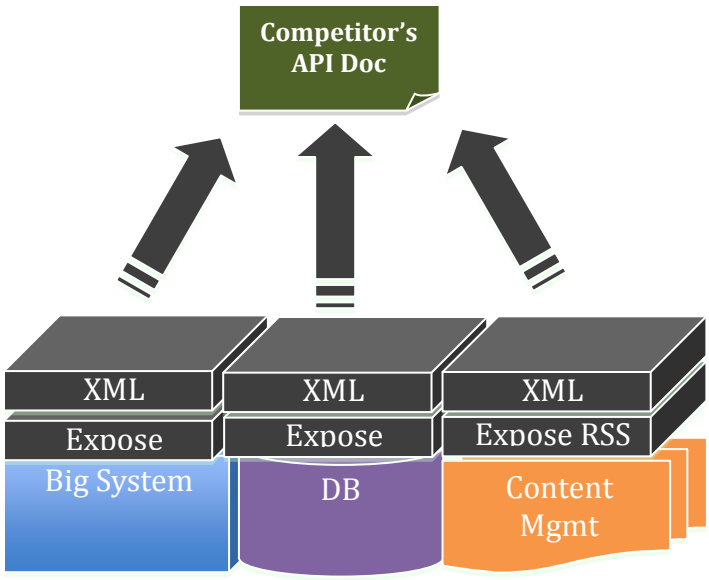


The benefits of this approach include getting to version 1 quickly. You can also create a sense of unification across an organization and a comprehensive strategy, which can be significant accomplishments when you have a large organization with a number of stakeholders and contributors.

A drawback of the standards committee pattern is that it can be slow. Even if you get the document created quickly, getting everybody to implement against it can be slow and can lack adherence. This approach can also lead to a mediocre design as a result of too many compromises.

The Copy Cat Approach

We sometimes see this pattern when an organization is late to market – for example, when their close competitor has already delivered a solution. Again, this approach can get you to version 1 quickly and you may have a built-in adoption curve if the app developers who will use your API are already familiar with your competitor's API.



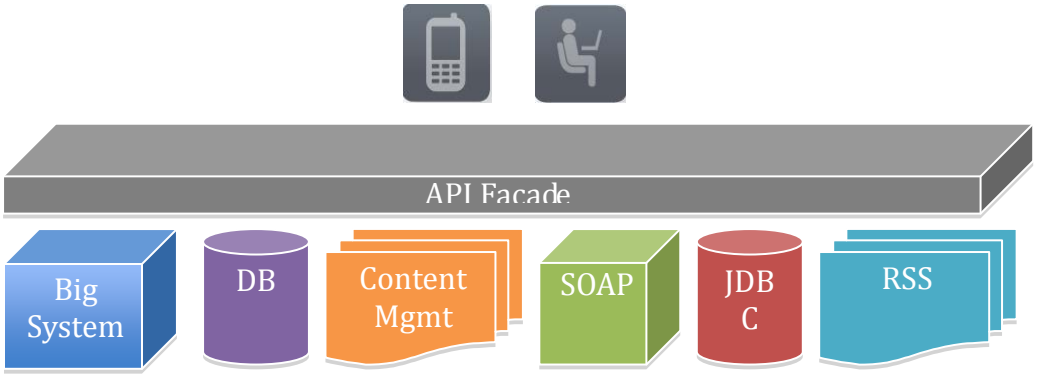
However, you can end up with an undifferentiated product that is considered an inferior offering in the market of APIs. You might have missed exposing your own key value and differentiation by just copying someone else's API design.

Solution – The API façade pattern

The best solution starts with thinking about the fundamentals of product management. Your product (your API) needs to be credible, relevant, and differentiated. Your product manager is a key member of your API team

Once your product manager has decided what the big picture is like, it's up to the architects.

We recommend you implement an **API façade pattern**. This pattern gives you a buffer or virtual layer between the interface on top and the API implementation on the bottom. You essentially create a façade – a comprehensive view of what the API should be and importantly from the perspective of the app developer and end user of the apps they create.



The developer and the app that consume the API are on top. The API façade isolates the developer and the application and the API. Making a clean design in the facade allows you to decompose one really hard problem into a few simpler problems.

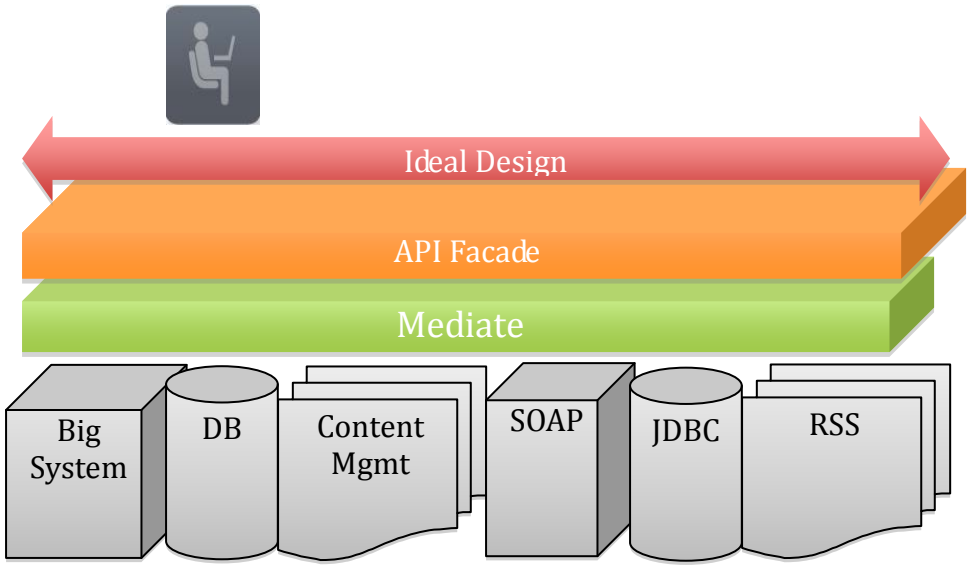
"Use the façade pattern when you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve."
Design Patterns – Elements of Reusable Object-Oriented Software
(Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

Implementing an API façade pattern involves three basic steps.

1 - Design the ideal API – design the URLs, request parameters and responses, payloads, headers, query parameters, and so on. The API design should be self-consistent.

2 - Implement the design with data stubs. This allows application developers to use your API and give you feedback even before your API is connected to internal systems.

3 - Mediate or integrate between the façade and the systems.



Using the three-step approach you've decomposed one big problem to three smaller problems. If you try to solve the one big problem, you'll be starting in code, and trying to build up from your business logic (systems of record) to a clean API interface. You would be exposing objects or tables or RSS feeds from each silo, mapping each to XML in the right format before exposing to the app. It is a machine–to-machine orientation focused around an app and is difficult to get this right.

Taking the façade pattern approach helps shift the thinking from a silo approach in a number of important ways. First, you can get buy in around each of the three separate steps and have people more clearly understand how you're taking a pragmatic approach to the design. Secondly, the orientation shifts from the app to the app developer. The goal becomes to ensure that the app developer can use your API because the design is self-consistent and intuitive.

Because of where it is in the architecture, the façade becomes an interesting gateway. You can now have the façade implement the handling of common patterns (for pagination, queries, ordering, sorting, etc.), authentication, authorization, versioning, and so on, uniformly across the API. (This is a big topic and a full discussion is beyond the scope of this article.)

Other benefits for the API team include being more easily able to adapt to different use cases regardless of whether they are internal developer, partner, or open scenarios. The API team will be able to keep pace with the changing needs of developers, including the ever-changing protocols and languages. It is also easier to extend an API by building out more capability from your enterprise or plugging in additional existing systems.

Resources

Representational State Transfer (REST), Roy Thomas Fielding, 2000

RESTful API Design Webinar, 2nd edition, Brian Mulloy, 2011

Apigee API Tech & Best Practices Blog

API Craft Google Group

About Brian Mulloy

Brian Mulloy, Apigee

Brian has 15 years of experience ranging from enterprise software to founding a Web startup. He co-founded and was CEO of Swivel, a Website for social data analysis. He was President and General Manager of Grand Central, a cloud-based offering for application infrastructure (before we called it the cloud). And was Director of Product Marketing at BEA Systems. Brian holds a degree in Physics from the University of Michigan.

Brian is a frequent contributor on the Apigee API Tech & best practices blog, the Apigee YouTube channel, the API Craft Google Group, and Webinars.

About Apigee

Apigee is the leading provider of API products and technology for enterprises and developers. Hundreds of enterprises like Comcast, GameSpy, TransUnion Interactive, Guardian Life and Constant Contact and thousands of developers use Apigee's technology. Enterprises use Apigee for visibility, control and scale of their API strategies. Developers use Apigee to learn, explore and develop API-based applications. Learn more at http://apigee.com.

Accelerate your API Strategy

Scale Control and Secure your Enterprise

Developers – Consoles for the APIs you ♥