A Guide to Lifecycle Management for APIs

API Monitoring by SMARTBEAR
Contents

Introduction .................................................................................................................. 3

API Modeling .................................................................................................................. 4
- Develop a Model ........................................................................................................ 4
  - Conceptualize .......................................................................................................... 4
  - Define ......................................................................................................................... 4
  - Resource .................................................................................................................... 5
- Define Success .......................................................................................................... 5
- Iterate ......................................................................................................................... 5

Pragmatic REST design – key principles ...................................................................... 6

REST metadata formats – an overview .......................................................................... 8
- apiary.io ..................................................................................................................... 10
- ioDocs ......................................................................................................................... 10
- Swagger ....................................................................................................................... 10
- WADL .......................................................................................................................... 10
- WSDL 2.0 ................................................................................................................... 11
- Selection Criteria ....................................................................................................... 11

Hypermedia APIs – HATEOAS anyone? ...................................................................... 12
- HAL .............................................................................................................................. 12
- JSON Hyper Schema ................................................................................................. 13

API Quality .................................................................................................................... 13
- Functional Testing ..................................................................................................... 13
- Service Simulation (or Mocking) ................................................................................. 14
- Security Testing ......................................................................................................... 14
- Load Testing .............................................................................................................. 14

API Performance Monitoring ....................................................................................... 14
- Re-use functional tests from development ................................................................ 15
- Create tests that mimic your use cases .................................................................... 15
- Be prepared for changing data ................................................................................ 16
- Use a dedicated account ........................................................................................... 16
- Don’t overdo it ........................................................................................................... 16

Moving the needle: Example API metrics .................................................................... 17
- Get early buy-in on the ‘top 3’ .................................................................................. 17
- Track against realistic projections. ............................................................................. 17
- Publish a weekly dashboard, religiously. ................................................................. 18
- Create a metrics ‘pipeline’ ......................................................................................... 18

To Summarize ............................................................................................................... 19
Introduction

Just like many fellow developers and technology geeks, we were avid Lego-builders during our youth. Those small plastic pieces gave us the possibility to create anything our imagination had in stock for us – and the cool thing was (and still is), the more basic the blocks were, the more freedom they gave us. As we got older, we changed, and so did Lego. We can still sit with our friends complaining about the “dark years” of Lego when those basic building blocks turned into pre-molded pieces of wings, buildings, animals – putting a definitive stop to the creative outlet that provided so much joy in our youth. It seems that Lego heard our complaints and has miraculously recovered, not only financially but also “ethically” (from our point of view). Back are the boxes of basic building blocks that set your imagination free – you can even buy them by the gram nowadays – shovel provided!

So what does that short story have to do with APIs? Let's see if we can piece that together. Pun intended.

APIs are the glue that makes modern web applications work. Just like Legos have a universal way of adhering to each other, APIs provide a standardized and (fairly) simple way to connect data sources to each other. By using platform-agnostic technologies like HTTP, JSON and XML, they are the ultimate facilitators for integrating all those data sources into innovative applications and solutions.

Ah – Innovation! This is where things get really exciting. The quality of all the APIs that are used to glue these applications together are pivotal to their success. In this eBook, we will visit some essential concepts around API quality, from design and testing to monitoring post-deployment performance.
Grounded in the premise that the design of an API communicates how app developers will use it and that the API is an extension of a brand, API modeling is a critical concept and discipline.

“The real issue is about design: designing things that have the power required for the job while maintaining understandability, the feeling of control, and the pleasure of accomplishment.”

~Donald Norman

Good design is about supporting rich capabilities while maintaining high usability. When building a cathedral, the design is complex but the outcome is a thing of beauty. The goal of API Modeling is to create seamless designs that are pleasant to use and intuitive.

Develop a Model

Designing API resources and representations begins with a basic understanding of domain modeling. We sometimes use the Unified Modeling Language (UML) to create a visual model of object-oriented software systems. However, even a low-tech napkin sketch can help the conversation get started. Resources, data fields, and relationship mappings are all helpful down the road if discussed and documented up-front.

Conceptualize

To start, gather all the stakeholders involved in the API team, including marketing folks, business analysts, software engineers, and key business people. The goal is to build a conceptual model of the API that will eventually be translated into working software. Include language for any surrounding processes that might be related to this initiative: background jobs, ETL processes, etc.
Define

Develop a common language, like a glossary of terms that will appear in the API. Once that language is established, it's important to use these terms everywhere: in resource names, identifiers, data fields, and documentation. This reduces confusion caused by mental context-switching when speaking to different audiences about the API, including app developers.

Resource

Using a common language, piece together what should be exposed via the API. Resources should emerge naturally from conversations with domain experts. Start with top-level resources and work down the list. Take into consideration all the near-term business projects and don’t forget that some resources may only be exposed to a few individuals with the proper permissions.

Define Success

An API Team defines what makes their API successful. Defining acceptance criteria early ensures the API gains alignment among all business units with a stake in the game. Criteria may be in terms of:

◆ Developer Adoption  ◆ Server Performance
◆ Quality Metrics  ◆ Business Revenue
◆ Usage Metrics  ◆ Sales Leads

Write acceptance criteria with terms that are understood and valued by all members of the API team; ubiquitous language works. This way, criteria may be eligible for automated testing by the technical team, which gives the team something to celebrate and acts as an outline for communicating success to key stakeholders.

Iterate

An API represents your organization and your brand, so make sure all stakeholders are present for key API decisions. Going solo often leads
to a late realization that marketing doesn’t use the same terminology, for example, or the discovery of resource challenges too late in the game. In reality, most problems in modeling are not technical in nature. When business processes are complex, so is the API that supports them.

Continue to iterate. In most cases, a shared language reflects complex relationships that evolve as information is gathered to further the success of your API program and the latest version of your API design.

**Pragmatic REST design – key principles**

Freedom is fantastic until you hit the wall of reality. Your API represents your organization. Make sure your organization is present on Key decisions.

First - let’s start with an overview on API design. First, it’s important to understand the difference between pragmatic and dogmatic REST. You might have seen discussion threads on true REST - some of them can get pretty strict and wonky. Our view is to 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.

Why? Look at the value chain below - the developer is the lynchpin of any API strategy.

![API Value Chain Diagram](image)

Your primary design principle should be to design the API to maximize developer productivity and success. This is what we call pragmatic REST.
The #1 principle in pragmatic RESTful design is: keep simple things simple.

1. 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. A design affordance is a design element 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 for simple API design and pragmatic REST is to limit base URLs to two per resource. If it requires more than two, it’s too complex. As an exercise, let’s model an API around a simple object or resource, and then create a RESTful API that interacts with that resource. Let’s say… dogs. The first base URL is for collections; the second is for a specific element in the collection.

Boiling it down to this level will also force the verbs out of your base URLs, which brings us to the second design principle.

```
/dogs
```

```
/dogs/1234
```
2. Keep verbs out of your base URLs

Many RESTful 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 dog’s world. The URLs for our resource might end up looking something like this:

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

...  ...
/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
...
3. 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. (I think of them as mapping to the old metaphor of CRUD (Create-Read-Update-Delete).)

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 I mean for our dogs.

The point is that a developer probably doesn’t need the chart to grok how the API behaves. They can experiment with and learn the API without having to dig into the documentation.

<table>
<thead>
<tr>
<th>Resource</th>
<th>POST create</th>
<th>GET read</th>
<th>PUT update</th>
<th>DELETE delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dogs</td>
<td>Create a new dog</td>
<td>List dogs</td>
<td>Bulk update dogs</td>
<td>Delete all dogs</td>
</tr>
<tr>
<td>/dogs/1234</td>
<td>Error</td>
<td>Show Bo</td>
<td>If not error</td>
<td>Delete Bo</td>
</tr>
</tbody>
</table>

REST metadata formats – an overview

Although the REST community initially took a stance against metadata for REST APIs, a number of metadata standards have none-the-less emerged over the last couple of years, mainly fueled by the need to document APIs for their consumers. As an added benefit, the same metadata is now often used to generate code (both client and server), create test harnesses, production monitors and perform real-time validation of request and response messages (when applicable). All of these provide a foundation for an improved Quality of Service (QoS) that many
enterprises require as they adopt REST for their information architectures.

Before looking at some of the metadata standards out there, let’s back up a bit and look at what they strive to describe (for those of you that advocate Hypermedia APIs, we’ll try to give some insights on that as well further down).

The common goal of current metadata formats for REST APIs is to specify

- Entry point(s)
- Resource paths
- Methods to access these resources (GET, POST, PUT, etc."
- Parameters that need to be supplied with these methods (Query, Template, HTTP Header, etc.)
- Formats of inbound/outbound messages REPRESENTATIONS (JSON Schema, XML Schema, Relax NG, etc.)
- Status codes and error/fault messages
- Documentary information (descriptions, etc.) for all these

All of these elements are somewhat derived from the principles underlying REST APIs – i.e. those of resources, representations etc. Non-functional aspects of APIs, like for example authentication (Basic, OAuth, SAML, etc.), security (encryption, signatures, etc.) and versioning, are unfortunately still poorly addressed by most REST metadata standards. This is where the WS-* standards "shine", but it is also what many refer to as WS-(death-star) – indicating the complexity of the WS standards that might ultimately lead to their demise.
Let’s have a quick high-level look at what’s out there (in alphabetical order):

**apiary.io**

Apiary has created a [markdown-based metadata](https://apiary.io) called the “API Blueprint” format. The parser itself is open-source and the format is well described on their website and on GitHub. Once you have an API Blueprint for your REST API – the apiary platform can be used to generate documentation, create server mocks, perform validation, etc. Although it is technically possible to use the API Blueprint format itself outside the apiary.io platform, most of the value-added functionality in the apiary.io platform seems to be proprietary.

**ioDocs**

ioDocs is Mashery’s take on REST metadata and documentation; their open-source (node.js based) platform uses JSON-formatted metadata to generate an “interactive documentation” for the described API that can be used to both learn about an API and execute ad-hoc calls against it. As opposed to the other standards described here, ioDocs also includes basic support for signatures and different versions of OAuth. As mentioned, the core ioDocs platform and tools are open-source and free to use outside the Mashery platform, but just like with apiary.io, a large number of value adding features are made available when using the Mashery platform to manage your APIs.

**Swagger**

Swagger is a REST metadata format being developed by Reverb, a spin-off from the Wordnik team. Swagger uses JSON (although it supports XML) and JSON-Schema to describe REST APIs and their parameters and messages. Just like ioDocs, it includes an open-sourced UI (swagger-ui, entirely in HTML/javascript) double-serving as documentation and ad-hoc testing utility, but Swaggers’ strength lies in its ecosystem available at GitHub for generating both code for a number of different languages and Swagger definitions themselves via (for example) java annotations.
WADL

WADL (Web Application Description Language) is an XML vocabulary that has been around for quite some time (it least in internet-time). It was submitted to the W3C by Sun in 2009 but hasn’t been standardized on, perhaps due to lack of adoption. Technically it provides good support of the REST concepts outlined above; its grammar mechanism allows for the use of any standard for describing resource representations (JSON Schema, XML Schema, etc.) and there are a number of tools out there to generate code, tests and documentation from WADL specifications.

WSDL 2.0

You might be surprised to see this on the list, but technically both WSDL 1.1 and WSDL 2.0 support description of REST APIs (although the support is meager in WSDL 1.1). Unfortunately though, the lack of “REST-fulness” in WSDL and its association with SOAP will probably result in its inclusion here being shrugged off as a curiosity as opposed to a valid alternative. That being said, WSDL 2.0 does technically support the description of REST APIs but its core model is not around resources/methods/representations, and the tooling is not as extensive as for the above-mentioned formats.

Selection Criteria

When trying to select one of these metadata formats for your API, make sure you take the following into account:

- Is it easily accessible?
- Are the included tools available for your platform?
- Is there relevant tooling available given your target audience and their needs? (Testing, code generation, etc.)
- Does it support metadata at the level needed in your usage; for example you might decide to create schemas (XML, JSON, etc.) for your resource representations, this is one area where the metadata formats differ.
- Does it support QoS metadata that you might want to make available?
- If it doesn’t support everything you need – is it on their roadmap?
- Is there an associated vendor lock-in for your usage intentions?

This is obviously just scratching the surface, but hopefully gives you an idea of which areas to explore further.

**Hypermedia APIs – HATEOAS anyone?**

As indicated earlier, the REST community has been (and still is) somewhat ambivalent to providing out-of-bounds metadata for REST APIs. Wouldn’t it be nice if the API provided metadata about itself and the actions that are available for a requested resource as part of the response/representation itself? Ultimately you would start with just one endpoint which would return links to all related actions and resources, allowing you to “drill down” into the API and its exposed resource model – just like a visual hypermedia document can be browsed in a web-browser. This is what is commonly referred to as Hypermedia APIs and HATEOAS (“Hypermedia as the Engine of Application State”) and the debate surrounding its adoption vs. metadata can sometimes take on religious proportions – be warned.

Even though the idea of out-of-bounds metadata is abolished in this construct, metadata in itself isn’t; it’s part of the response returned for a certain resource, and as such it needs to adhere to some standard way of doing things. Let’s have a quick look at two of them.

**HAL**

HAL (Hypertext Application Language) is a simple format available for both JSON and XML APIs to provide linking within a response. A multitude of libraries in many languages are available, both for creating and consuming HAL responses.
JSON Hyper Schema

JSON Hyper Schema provides a mechanism for embedding links in JSON documents, achieving the same goals as HAL in a slightly different fashion. It is part of the JSON Schema initiative, which also includes a large number of tools for creating, parsing and validating JSON documents.

Reading through the above might have opened your eyes a bit but perhaps not made things easier for you; should you use metadata at all? Which format seems like your safest bet?

In the end, this decision should be made by putting your end user first when making your choices. Are they invested in some kind of technology? Use that! Do they prefer JSON to XML? JSON it is! Are they legacy and enterprise up to their ears? Perhaps you should use SOAP/WSDL instead of REST (the horror…)! 

Most importantly: make them love you and your APIs, with or without metadata.

API Quality

As part of the Modeling discussion, we mentioned that you should define Acceptance Criteria up front. This should include quality metrics as well. Do you have service level agreements with your customers for performance and uptime? Are there specific calls within the API that must be bug-free in order to drive developer adoption?

When you plan your API quality strategy, you should plan on 4 types of testing (at a minimum):

Functional Testing

Your primary validation should be focused on the functionality the API is intended to deliver. If it is to provide a capability, then you need to test the API calls to make sure that capability is delivered according to
the model you developed with your team. If it is to retrieve data, then validate that the correct data is returned per your model.

**Service Simulation (or Mocking)**

One of the big challenges with API testing is when you are faced with testing against a third-party API for which you don’t have a developer key or an internal API that hasn’t been implemented yet. A good API testing tool lets you create robust tests against an API before it’s implemented or without actually calling it via a valid key.

**Security Testing**

Adding security scans helps you make sure your web services are protected from vulnerabilities. With the increased focus on healthcare APIs, government APIs, and financial services APIs, it’s important to discuss your security requirements as part of the modeling session with your team. If you have security concerns, you should include them as part of the acceptance criteria and plan to actively test for vulnerabilities.

**Load Testing**

Often neglected, load testing can help you ensure that business success doesn’t quickly evolve into business failure. When designing your load tests, be sure to include all stakeholders who can influence traffic to your application, including external developers using your APIs. If your APIs are exposed to partners or the public, the load on your servers can be fairly difficult to predict – running frequent load tests against your APIs can help you determine your breaking point and put the appropriate monitors in place.

**API Performance Monitoring**

Creating monitors for APIs you either provide or consume (or both) should be a cornerstone in your online quality strategy. There are some things to consider when planning your API monitoring strategy:
Reuse functional tests from development

If you’re providing your own APIs then re-using functional API tests created during development and testing for monitoring has several advantages:

◆ Instead of just checking availability, “real” functionality of your API will be scrutinized continuously, providing you with a safety net for continuous deployment practices and infrastructure changes

◆ Given that your functional API tests are set up to assert and provide relevant error messages, the corresponding API monitors will have the ability to give you much more detailed error information for root cause analysis than regular availability monitors.

◆ Given that your functional API monitors mimic expected usage scenarios, their actual structure can tell Operations how your APIs are expected to be used, and help them set up the API infrastructure accordingly.

◆ Obviously, using one tool for creating tests and monitors is lower overhead in maintenance, learning, cost, etc.

Create tests that mimic your use cases

If you are mainly integrating with 3rd party APIs, you need to make sure that you know about their failures before your users notice. Here it is essential that the monitors you create actually mimic how you use that API; for example, if you are using the Flickr API to get the latest photos for a certain group on Flickr, make sure your API monitor does the same thing, and not get a list of popular cameras (or anything else that is easier to set up). Also, make sure you monitor the entire flow of your use cases - don’t just monitor the first API request; monitor them all, in sequence – just like your application uses them.
Be prepared for changing data

This is a tricky one when it comes to monitoring 3rd party APIs. Often your monitor will want to validate some kind of output based on your input; for example, you might validate the coordinates or route-plan you get back from one of the Google Maps APIs to give you the expected results every time. Unfortunately though, Google updates coordinates on the 4th decimal rather frequently, so if your monitor doesn’t take that “volatility” into account, it might fail unnecessarily. The same goes for route-planning; perhaps a traffic jam is making Google return a different “unexpected” result for a limited period of time – something that you need to be prepared for.

On the other hand; if you’re not going to validate the returned data in an API monitor, what are you going to validate to make sure it’s working as required? This is a tough call to make, I recommend you at least be defensive with data-validations; don’t do too many - try to focus on those you think won’t ever change.

Use a dedicated account

Many (most) APIs require you to specify some kind of credentials or access key in your requests; make sure you are using a dedicated account(s) for your monitoring, both for your own APIs and 3rd party ones. There are several reasons for this, including:

◆ for your own APIs, it makes you run your monitors with the same access rights as your users, so you can detect problems that might not affect “super-user” accounts
◆ for 3rd party APIs, it allows you to plan billing and utilize bandwidth separately from your “production” API usage

Don’t overdo it

You probably don’t need to monitor from every location in the world, or exercise all operations in all your APIs with every possible input. Instead, make sure your API monitors are a safety net for catching
problems within the “hot-spots” of your API and those areas that are most vital to your API business. Overdoing things will just result in poorly maintained API monitors, which start to fail and which you will start to ignore, which in the end won’t provide any value to anyone.

**Moving the needle: Example API metrics**

It’s an old cliche, but it’s been said that you can’t move the needle if you can’t *see* the needle. So what are good metrics to measure an API program?

While individual metrics are important - it might be as much about the ‘process around metrics.’ In other words, how are metrics evangelized and used to drive specific parts of the API product development pipeline?

Here are some general guidelines for defining and managing your API metrics:

**Get early buy-in on the ‘top 3’**

Strong API product managers often focus in on 1 to 3 top-level ‘strategic’ metrics and get early, wide agreement from all parts of the extended team - the sponsoring exec, PM, engineering, BD, and operations. If different stakeholders are measuring success with different metrics (say number of developer sign-ups vs. API traffic vs. revenue) this can pull resources in different directions.

**Track against realistic projections.**

Set expectations early by modeling anticipated results and then track actuals against this estimate. For example, pick a ‘comparable’ or com-
petitor’s API to guess developer portal traffic, then model the expected developer sign-ups and conversions (for example, 10% of visitors might ask for a key, 20% of them might built an app, 10% of those apps might drive ongoing traffic, each of those apps might drive a certain volume of traffic, and so on...)

**Publish a weekly dashboard, religiously.**

Proactively call out how product updates and community activities do or don’t move the needle so you can quickly adjust tactics and think of new ideas that might move the needle.

**Create a metrics ‘pipeline’**

How do different metrics measure each stage of the customer conversion process? For example, developer portal traffic might be a good metric to measure the marketing team. But whether or not a developer converts to ask for a key and then converts into an active API user might be a measure of how effective the requirements process is working – does it create a product that developers want to use? User-experienced bugs can measure development and product QA effectiveness, and so on…

Here is an example of a metrics pipeline:

**Awareness (measure of marketing effectiveness)**

- Developer portal traffic: Unique users, page views, and engagement (PVs/UU)
- Top traffic sources (search, direct, referrals)

**Signups (measure of portal messaging effectiveness)**

- Registrations (developer keys issued)

**Adoption (measure of product fit)**
Active developers, partners

Applications (number, by app type, geo, partner ‘tier’)

App end users (such as mobile app users)

Traffic: volume and % API vs. non-API

Developer retention (active developers lost)

Quality (measure of dev process)

User experienced problems (errors returned)

Bugs reported

Critical situations (P1 bugs or blocking bugs)

Community (measure of customer sat)

Community members

Community forum activity and engagement

Number of very active members

Net promoter score

Financial (measure of business model fit)

Revenue

Cost of data served (if licensed)

Profit and margin

Market share

It’s clear that APIs are a critical part of your software planning and should get just as much attention at each stage of the lifecycle as any other feature on your roadmap. Hopefully, the concepts here will help you develop and measure your APIs with growth and sustainability in mind.
To Summarize

- Keep your designs elegant and simple
- Use modeling as a technique to ensure you are building the right API
- Identify success criteria
- Focus on quality
About SmartBear Software

More than one million developers, testers and operations professionals use SmartBear tools to ensure the quality and performance of their APIs, desktop, mobile, Web and cloud-based applications. SmartBear products are easy to use and deploy, are affordable and available for trial at the website. Learn more about the company’s award-winning tools or join the active user community at http://www.smartbear.com, on Facebook or follow us on Twitter @smartbear and Google+.
About Apigee

Apigee is the leading provider of API products and technology for enterprises and developers. Hundreds of companies including AT&T, eBay, Pearson, Gilt Groupe, and Walgreens use Apigee to reach new customers and drive innovation through APIs, apps and data. Apigee’s API Platform is built on enterprise bedrock and designed to meet the challenges of the new mobile, social, cloud marketplace head-on. The API Platform enables businesses and developers to create and deliver well designed, scalable APIs and apps, drive developer adoption, and extract business value from their API ecosystem. Learn more at apigee.com.

Find Best Practices to Accelerate your API Strategy

Scale, Control and Secure your Enterprise

Build Cutting-Edge Apps and Intuitive APIs

Get end-to-end visibility into your business with Apigee Insights