Is your API naked?

Roadmap considerations for API product and engineering managers

www.apigee.com

January 2012

This whitepaper is derived from a series of posts from our API technology best practices blog series *"Is your API Naked? 10 considerations for your API roadmap."*

This series is put together based on our collective experience talking to hundreds of product and engineering managers on issues around operating an open API.

Why 'naked' in the title? A customer of ours once said it best - "...most open APIs start out as little more than raw, naked, features exposed to developers, and there is a big gap between a feature and a full-fledged service."

Since we first published this paper, we've been conducting workshops with API teams around the country. These workshops focus on strategy, technical, and marketing best practices for APIs. We call our workshop "RAW or 'Rapid API Workshops" ☺

In this latest edition of the whitepaper, we've included the best of our experience and learning from the RAW discussions (which we've also posted on our blog).

With your continued feedback, we'll keep documenting all the great best practices, insights, and feedback from the innovative API teams we meet.

This whitepaper is about the capabilities 'around the API' that might be needed to 'productize', 'monetize', or 'operationalize' an API-based service. We'll explore issues around traffic management, protecting your organization and customers, scaling performance, monetization, community, and more.

If you own an open API roadmap, these short pieces are meant to give you ammunition and questions to think about for your own roadmap across different areas where your API might need covering up.

And it goes without saying we'd love to talk to you about how Apigee Enterprise API management can help. Or try our free Apigee service today.

Cheers,

The Apigee Team

Table of Contents

Is your API Naked? API Roadmap considerations

1. Is your API naked? (API Visibility)

We're always surprised how almost every company we talk to acknowledges how little they know about their API traffic and usage.

We see most folks sifting through web server logs to understand usage of the API as a product. As the API becomes strategic and you want to make the case for additional investment, this gets more painful.

This often happens when an API starts as an experiment. As we once heard a customer explain – most APIs start out as little more than 'raw naked features.' The folks who ask for forgiveness, not permission, often launch APIs. (You know who you are :-).) Things like metrics or analytics are placed on the back burner until an API either gets off the ground or doesn't.

But the API economy is exploding. Most APIs usually end up getting important more quickly than expected. You, as a product and engineering manager you may start asking:

Who is using the API and how much are they using?

- How many clients, apps and developers are out there?
- How do they break down by type and region?
- How does usage map to existing customer or partner organizations? Or how do developers map to applications and map to customers? (This can be tough with only key or IP based tracking.)
- What parts of the API and service are they using? You probably use web analytics- at some point you need this for APIs as well to see where to invest.
- How does traffic breakdown between your own products and 3rd party products when they are using the same API?
- What the aggregate and per developer/app/customer transaction and data volumes?

How fast and 'good' is your service?

- What latency are customers experiencing, by developer, customer, region, or operation?
- Where are errors and user-experienced bugs happening and how often?
- How is the API delivering vs. any formal SLA agreed to or paid for?
- How can you find out if a customer is having a problem (before they call you)?
- How is the API usage impacting the rest of the platform or web products that also use the same API?

- Can you quickly trap and debug based on a specific message? Based on what is in a cache?

How does the API drive the business?

- Who are the top priority customers? Developers? Partners? Who should you call to up sell to a higher service tier or do a deal with?
- What do you need to show general management to make product strategy (or tactical) decisions?
- Will you need to create audit trails or metering reports for partners that are paying for API access?
- Do you need to create metrics based on a certain data value in the payload? (For example, a specific product SKU.)
- What is the cost of the data that you are serving up? (If you are licensing this data.)
- Are you in line with all the compliance standards that IT enforces for the on-premise apps?

These questions become increasingly important when opening a service as an API. Delivery times are critical when customers, contract terms, and compliance regulations come into play. Knowing these metrics and analytics and how they map to the business will help you be get ahead of the demands of customers and partners, and are critical when you need to make the business case for an API to executives.

(Thanks to Esparta and Naked Cowboy for the pic)

2. Don't have a meltdown: API Traffic Management

Customers come up with far better sound bites than we do. One asked us if we could protect them from "back-end melt down." They had opened an API for customers, but it was the same one their Web apps was using, so bursts of traffic from customer apps had their Web product running hot – and not in a good way.

A back-end melt down will quickly lead to a front-end melt down. That's a major risk to the business.

It's also one reason that traffic management is usually one of the first management features added to an open API. Typically, there are both technical and business reasons for this.

Technical folks want throttling, or rate limiting, to regulate traffic flow. Your customers' developers might inadvertently write some inefficient code. Someone might try to download **all** your data through your API. Rarely, you might get malicious use. Rate limiting or transaction throttling (x transactions per second usually) act like a circuit breaker that protect your own apps and keeps your Web and API customers happy.

Business managers may need to measure consumption against a daily quota. This might be to find growing customers to upsell, or keep track of customer data usage (especially if you are paying for that data). Or they may want to create different or custom *service tiers*.

Pitfalls to avoid with Rate limits and Quotas

We see a few API providers substitute 'quotas' for rate limits – telling developers they can have X transactions per day. Measuring against daily quotas like this can leave your 'back-end exposed' if you get a short burst of traffic. And *per-second* rate limits aren't a great way to create tiers of business-level consumption. You probably need both.

We also see customers that define an SLA only on paper and say "call us if you need more." We had a SaaS customer that started this way but then found it was tough to tell important customers they needed to manage their API usage. They eventually found a good way to ease into the traffic management discussion - by first providing the customer with usage reports. ("Look how all your requests come in Monday at 9 am, we love you but can you help us?")

And you may not consider all requests equal – you may find that some requests contain many transactions or big messages, so a simple "X requests per second" rate limit might not be enough.

Consider these traffic-management questions for your open API roadmap:

What kinds of rate limiting do you need?

- Do you need to rate limit by developer, app, or customer organization?
- Can you rate limit a particular client (key and IP address)?
- Are all messages the same or do you need to adjust based on message size, or records per message, or bandwidth?
- How do you throttle differently for your own web or internal apps vs. API traffic?
- Do you need to buffer or queue requests?

Does your business need quotas on API usage?

- Do you need to keep track of daily or monthly usage to measure business consumption?
- Do you need to define different consumption levels for different service tiers?
- If someone goes over the quota, what is the best business recourse? Call them up and upsell them? Or shut them off?
- If you are paying for the data are you measuring this for billing and pricing?

How do you monitor and respond to traffic management?

- How will you know when someone goes over a rate limit or quota?
- Are quota or rate limit overages a trend or a ‘one time thing’?
- Can you define flexible actions? (I.e. drop requests, do nothing, send an alert?)
- Can you provide customers with data so they can help by metering themselves?

3. Do you need API keys? API Identity, Authorization, & Authentication

What about bad stuff like unauthorized access, Denial-of-Service, or XML specific attacks?

We've seen very few API providers with a completely open API – almost all employ at least one of these:

- Identity - who is making an API request?
- Authentication - are they really are who they say they are?
- Authorization - are they allowed to do what they are trying to do?

Do you need them all? Maybe not! Some APIs only need to establish identity and don't really need to authenticate or authorize.

Compare the Google Maps and Twitter API – Identity vs. Authentication

Take the Yahoo and Google Maps APIs – they are fairly open. They want to know who you are but they aren't concerned what address you are looking up. They use an "API key" to establish identity, but don't authenticate or authorize. If you use someone else's API key, it's not good but it's not a serious security breach.

The API key lets them identify (most likely) who is making an API call so they can limit on the number of requests you can make. Identity is important here to keep service volume under control.

Then take Twitter's API. It's open for looking up public information about a user, but other operations require authentication. In other words, Twitter supports both username/password authentication as well as OAuth. Twitter also has authorization checks in its code, meaning that you cannot "tweet" on behalf of another user without either their password or an OAuth key to their account. This is an example of an API that implements both *authentication* and *authorization.*

The "API Key" – do you need one?

API keys originated with the first public web services, like Yahoo and Google APIs. The developers wanted to have some way to establish *identity* without having the complexity of actually authenticating users with a password, so they came up with the "API key," which is often a UUID or unique string. If the API key doesn't grant access to very sensitive data, it might not be critical to keep secret, so this use of the API key is easy for the consumers of the API to use however they invoke the API.

An API key gives the API provider a way to (most of the time) know the identity of each caller to maintain a log and establish quotas by user (see the previous section on API traffic management).

Usernames and Passwords – again, see Twitter

With more sensitive data a simple API key is not enough, unless you take measures to ensure users keep the key secret. An alternative is username/password authentication, like the authentication scheme supported by the vast majority of secure web sites.

It's easiest to use "HTTP Basic" authentication that most websites use. The advantage of using this technology is that nearly all clients and servers support it. There is no special processing required, as long as the caller takes reasonable precautions to keep the password secret.

Twitter simplifies things for their users by using usernames and passwords for API authentication - especially with human users. Every time a user starts a Twitter client, it either prompts for a username and password, or it fetches them from the disk (where it is somehow scrambled or encrypted where possible). In this case, it makes a lot of sense to have the same username and password for the Twitter API that is used for the web site.

Usernames and passwords also work well for application-to-application communications. The trick is that the password must be stored securely – and if a server is using it, where do you store it? If you are running an application server that uses a database, you've already solved this problem, because the database usually requires a password too. Better application server platforms include a "credential mapper" that can be used to store such passwords relatively securely.

Session-based Authentication – cumbersome with RESTful APIs

Lots of APIs support session-based authentication. In these schemes, the user first has to call a "login" method, which takes the username and password as input and returns a unique session key. The user must include the session key in each request, and call "logout" when they are done. This way, authentication is kept to a single pair of API calls and everything else is based on the session.

This type of authentication maps very naturally to a web site, because users are used to "logging in" when they start working with a particular site and "logging out" when they are done.

However, session-based authentication is much more complex when associated with an API. It requires that the API client keep track of state, and depending on the type of client that can be anything from painful to impossible. Session-based authentication, among other things, makes your API less "RESTful" - an API client can't just make one HTTP request, but now a minimum of three.

Two-Way SSL, X.509, SAML, WS-Security...

Once we leave the world of "plain HTTP" we encounter many other methods of authentication, from SAML, X.509 certificates and two-way SSL, which are based on secure distribution of public keys to individual clients, to the various WS-Security specifications, which build upon SOAP to provide... well, just about everything.

An API that is primarily used by "enterprise" customers – that is, big IT organizations – might consider these other authentication mechanisms such as an X.509 certificate or SAML for more assurance over the authentication process than a simple username/password. Also, a large enterprise may have an easier time accessing an API written to the SOAP standard because those tools can import a WSDL file and generate an API client in a few minutes.

Know your audience. If your audience is a fan of SOAP and WSDL, then consider some of the more SOAP-y authentication mechanisms like SAML and the various WS-Security specifications. (And if you have no idea what I'm talking about, then your audience probably isn't in that category!)

SSL

Most authentication parameters are useless, or even dangerous, without SSL. For instance, in "HTTP Basic" authentication the API must be able to see the password the client used, so the password is encoded – but not encrypted – in a format called "base 64." It's easy turn this back into the real password. The antidote is to use SSL to encrypt everything sent between client and server. This is usually easy to do and does not add as much of a performance penalty as people often think. (With Apigee's products, we see a 10-15 percent drop in throughput when we add SSL encryption.)

Some security schemes, such as OAuth, are designed to be resistant to eavesdropping. For instance, OAuth includes a "nonce" on each request, so that even if a client intercepts an OAuth token on one request, they cannot use it on the next. Still, there is likely other sensitive information in the request as well.

In general, if your API does not use SSL it also potentially exposes everything that your API does.

So Many Authentication Schemes – What Should You Use?

OAuth, OpenID, SAML, HTTP authentication, WS-Security, and the basic API key - it seems that there are as many *other* API authentication schemes as there are APIs. What to use and when?

Read on for recommendations!

4. Don’t roll your own: API Security Recommendations

OAuth, OpenID, SAML, HTTP authentication, WS-Security, and the basic API key - it seems that there are as many *other* API authentication schemes as there are APIs. Amazon Web Services, Facebook, and some Google APIs are among some of the big APIs that combine an API key with both public and secret data, usually through some sort of encryption code, to generate a secure token for each request.

The issue – every new authentication scheme requires API clients to implement it. On the other hand, many tools already support OAuth and HTTP Basic authentication. The big guys may be able to get away with defining their own authentication standards but it’s tough for every API to do things its own way.

Use API Keys for non-sensitive data (only)

If you have an “open” API - one that exposes data you’d make public on the Internet anyway - consider issuing non-sensitive API keys. These are easy to manipulate and still give you a way to identify users. Armed with an API key, you have the option of establishing a quota for your API, or at least monitoring usage by user. Without one, all you have to go on is an IP address, and those are a lot less reliable than you might think. (Why don’t web sites issue *web site keys*? Because they have cookies.)

The Yahoo Maps Geocoding API is a good example of one that issues API keys in order to track its users and establish a quota, but the data that it returns is not sensitive so it’s not critical to keep the key secret.

Use username/password authentication for APIs based on web sites

If your API is based on a web site, support username/password authentication. That way, users can access your site using the API the same way that they access it using their web browser. For example, the Twitter API supports username/password authentication - when you access it using a Twitter API client like Spaz or TweetDeck, you simply enter the same password you use when you use the twitter.com web site.

However, if you do this, you may want to **avoid session-based authentication**, especially if you want people to be able to write API clients in lots of environments. For example, it is very simple to use “curl” to grab some data from the Twitter API because it uses HTTP Basic authentication. It is a lot more complex if I instead have to call “login,” and extract a session ID from some cookie or header, and then pass that to the real API call I want to make.

OAuth for server-to-server APIs that are based on web sites

If your API is based on a web site (so you already have a username/password for each account) and if other sites also use the API, then support OAuth in addition to username/password authentication. This gives users a lot of peace of mind, because

they can authorize another site to access their sensitive data without giving that site their password permanently.

SSL for everything sensitive

Unless your API has only open and non-sensitive data, support SSL, and consider even enforcing it (that is, redirect any API traffic on the non-SSL port to the SSL port). It makes other authentication schemes more secure, and keeps your users' private API data from prying eyes – and it's not very difficult to do.

Questions to ask when constructing your API security roadmap

Distilling what we've discussed in the previous sections and what we've learned in the past couple of years, here are some questions you may want to ask when putting together your API security roadmap:

How valuable is the data exposed by your API?

- If it's not that valuable, or if you plan to make it as open as possible, is an API enough to uniquely identify users?
- If it is valuable, can you reuse username and password scheme for each user?
- Are you using SSL? Many authentication schemes are vulnerable without it.

What other schemes and websites will your API and users want to integrate with?

- If other APIs will call your API programmatically, or if your API is linked to another web site that requires authentication, have you considered OAuth?
- If you have username/password authentication, have you considered OpenID?
- Can you make authorization decisions in a central place?

What other expectations might your customers have?

- If your customers are enterprise customers, do they feel better about SAML or X.509 certificates?
- Can you change or support more than one authentication approach for diverse enterprise customers?
- Do you have an existing internal security infrastructure that you need your API to interact with?

5. Best Practices for OAuth 2.0 vs. OAuth 1.0

Since we first wrote about OAuth 1.0 vs. 2.0, lots has been happening and lots has changed. The OAuth community has made progress and made changes, and an increasing number of API providers have deployed APIs that use OAuth 2.0. (Similarly, the number of new OAuth 1.0-enabled APIs doesn't seem to be growing.)

We've been busy helping our customers implement OAuth-based APIs, and we've also been watching the process develop. Here are some things that we've learned:

OAuth is a solution—not technology

In order for an API provider to support OAuth, a number of things have to happen. There must be a place where the OAuth protocol is actually implemented—all of the different URLs for requesting tokens, issuing tokens, redirecting browsers, and so on. This part is a matter of implementing the standard and doesn't change from implementation to implementation. In this way, it can be delegated to a third-party component like Apigee Enterprise.

In addition, there has to be somewhere to store OAuth tokens. There is no standard for this—some use an RDBMS, some use NoSQL, and others use a cloud-hosted service.

Finally, the OAuth solution has to integrate with a provider's "login page," so that when a user wants to get an OAuth token, they are redirected to a legitimate "login" page with a proper SSL certificate, look and feel, and so on. All of these kinds of things can be different from one provider to another.

At Apigee, we have been working with many companies to deploy APIs built using OAuth. What we've found is that OAuth is not just a technology that you buy or download—it's a bit of technology and a bit of integration, and to implement it at all you need a platform that can flexibly integrate with what you already have, while adding what you might be missing.

Be careful

Since we first blogged about it, OAuth 2.0 has been improved and has changed some basic things—for instance the name on the HTTP "Authorization" header has changed from "OAuth" to "Bearer." This is a big change, and obviously if an API provider just switches from one value to another without warning, all the clients will break.

At the time of writing, OAuth 2.0 was at draft 13, and with every draft it gets closer to being "done"—but it's not done yet and I haven't seen anyone say when it will be. That means that if you implement, say, draft 13, you will likely have to change, and find a way to maintain backward compatibility for existing clients.

If that's too much, then implement OAuth 1.0a—it's as "done" as it's going to get at this

point!

Stick with “bearer tokens” for now in conjunction with SSL

The latest draft includes support for various types of security tokens, and a mechanism to plug in new types. The simplest are “bearer tokens.” A bearer token is just a big, random string that a client must present on every API call. Bearer tokens are simple because there’s no special signature or validation code required on either end. The client is responsible for storing the token in a safe place and sending it with every request. The server is responsible for looking up the token in a database and making sure it’s a valid one—that’s it.

However, bearer tokens provide no security unless used in conjunction with SSL—otherwise, any eavesdropper in a coffee shop can snatch OAuth tokens from the air. (Fortunately, OAuth tokens can be individually revoked without requiring a password reset.)

Get ready to change—or stick with OAuth 1.0a

We said this before but it bears repeating—if you implement OAuth 2.0, it is likely that you will have to change your servers to support newer versions, and also maintain backward compatibility.

One way to ease the pain is to put version numbers in the URLs that you hand out for the various steps in the OAuth flow. So, instead of using:

http://api.foo.com/oauth/authorize

Use:

http://api.foo.com/oauth/13/authorize

Because you know that there will be future versions and that they will be different...

Get ready for more options

Part of the OAuth 2.0 work defines some new security token types other than “bearer.” Right now only “bearer” seems to be sufficiently mature from a spec standard, but this will change over the next few months. Here are some of the options:

Bearer—as we mentioned before, this is just a long, random string. The advantage is that it’s extremely easy to implement on both client and server. The disadvantage is that it requires SSL for all API calls. Also, if API traffic passes through multiple endpoints such as proxies, then the proxies will be able to see the token as it passes by.

Mac—this is equivalent to the token scheme in OAuth 1.0, in that it requires both a key and secret, and uses an “Hmac” algorithm to encrypt some data on each request. The result is that the request is only valid if both client and server have the same keys, and there is no way for an intermediary to re-create the original request without a password.

In other words, it is secure even if SSL is not used or even if a proxy intercepts the request.

SAML—this uses SAML assertions on each request as a way to establish the client's identity. This is helpful if you already have SAML infrastructure for either the client or server in your environment.

See the ongoing discussion about **OAuth on blog.apigee.com**

6. Indecent Exposure: Data Protection

None of us want to see accidental customer data exposure through our APIs (as Paris experienced (an API authentication issue)).

In addition to having strong practices in identity, authentication, and authorization, an important aspect of API management is **data protection** - specifically:

- Encryption – protecting your API data from eavesdropping
- Threat detection – ensuring client API requests won't cause back-end problems
- Data masking – preventing inadvertent data exposure in API responses

First things first - protecting sensitive data requires knowing what and where it is, and how sensitive it is to you and your customers. Be aware of the various regulations and best practices for Social Security numbers, credit card numbers, home addresses, birthdays, and such.

Once you have identified which data is sensitive, you can think about how to best protect it in your API.

Encryption – SSL or XML?

Encryption is a basic technology that must be part of any API with sensitive data. Some relevant technologies include SSL and XML encryption. For most APIs, the most critical encryption mechanism is SSL. It works on every platform, and the overhead is minimal (about 5% through our own API gateway product in benchmark tests.)

The least your API should do is to use SSL to encrypt any sensitive data. Alternatively, you can encrypt either all or part of each message using XML Encryption (a W3C standard implemented by many SOA products). Deploying XML Encryption technology can be complex and has a larger performance impact than SSL, because it requires that you and your clients manage public/private key pairs.

However, XML Encryption is tremendously useful when it's important to manage sensitive data *behind* the API. A good use case is when API data must not only be transmitted securely over the Internet, but also stored in internal systems in encrypted form on a disk or in a database. Otherwise, stick with SSL.

Threat Detection

Any server that receives data over the Internet is subject to attack. Some attacks are more specific to an API and merit additional consideration.

SQL injection. A SQL injection attack takes advantage of internal systems that construct database queries using string concatenation. If there's a way to take data from the client and paste it inside a database query, then there may be a way to compromise the system using SQL injection. The best way to prevent SQL injection is to code the back-end systems so that a SQL injection attempt is not possible. But it's also important to stop SQL injection attempts before they get to the back end.

XML attack An XML attack takes advantage of the flexibility of XML by constructing a document that can cause a problem for a back-end system. Examples include causing the XML software to try to allocate more memory than is available or an XML document that is nested many levels deep, or with extremely large entity names or comments. A simple check to ensure that XML or JSON is *well formed* can save resources on the back-end that would otherwise be devoted to generating and logging error messages.

These attacks aren't always intentional. Ever used an API like StAX to construct an XML document, but forgotten to add all the *end tags*? An invalid XML document that appears to be nested very deep can cause problems for the back-end servers, or at least tie up CPU and memory resources.

Data Masking

Encryption is critical to keep sensitive data private, but in some cases, it may make sense to try and re-use internal services and data.

However, you might need to screen - or mask - some private data for the API. This means using an XML or JSON transformation to either remove or obfuscate certain data from an API response. While this technique must be used with care, there are cases in which only certain API users are authorized to see certain information. For example, there might be an API call that returns a "user" record with all the details when the user him or herself calls this API, but only limited data when a customer service reps access the "user" record using the API.

You can design for a scenario like this by building only one version of the API on a back-end server and adding a data transformation rule that removes the user's home address if the request is coming from a CSR's account. If you have many services, you might consider having a common layer that performs these types of transformations – especially if you may need to add certain data fields as well as masking fields. (More on that in the next section on **API Mediation**.)

Data Protection recommendations for your API roadmap

- Use SSL when the API includes sensitive data, or if the authentication mechanism in use does not include an encryption component. (HTTP basic authentication, for instance, allows an eavesdropper to intercept the password unless SSL is used; OAuth does not.)
- Always defend against SQL injection, either in the back-end server, at the edge of the network, or both.
- If your API accepts XML input via HTTP POST or some other way, then defend against the many types of XML attacks. These include large inputs, payloads or attachments, 'header bombs', replay attacks, message tampering and more.
- Consider using data masking in a common transformation layer if your back end servers may return some data that should not be given out to all users of the API.

(Thanks to carbonnyc for the photo)

7. Pragmatic PCI Compliance and APIs: Just Enough Process

"If the minimum wasn't good enough, it wouldn't be the minimum." - Keith W.

These are wise words from one of my developer friends many years ago. When it comes to tackling PCI Compliance, it is advice well worth taking.

With leaks of sensitive customer information in the news, there's an increased focus on compliance as more services shift to cloud computing and APIs.

If you are a merchant of any kind or deal with customer credit card information then you must be aware of PCI compliance regulations that are designed to protect consumer credit card information from exposure.

PCI compliance gets tricky as apps and services move to cloud services and APIs. If you're heading down the path of PCI compliance or just trying to position yourself, your APIs and your internal systems better for the future, keeping it simple will help you be successful.

First, document your process

The PCI Data Security Standards (PCI DSS) establish the "what" but not the "how" of achieving compliance.

The "how" is up to you. But like most audit and process centric assessments, what is most important is being able to articulate what you do to support a particular DSS item - and then being able to show evidence to support that statement.

Identify all of the process standards that apply to you from the DSS and identify the proper owner of those processes. Put together a simple Process Description Template that everyone uses to document their individual processes and adopt a naming convention that calls out the DSS section. Centralize the storage of those documents and make sure everyone knows where they are.

Just focus on capturing the "how" of your processes in as lean a manner as possible. Your assessment team is not going to evaluate quality of the process or the documentation, only that it meets the requirements of the DSS.

With your processes documented "well enough" and easily mapped to the PCI DSS, you'll discover gaps, strengths and you'll make your assessors life easier and that makes your life easier.

Next, we'll talk about the special challenges with PCI in cloud computing and APIs, and practices that you can apply to reduce your risk.

PCI and APIs in the Cloud

What happens when you inject "The Cloud" into the picture?

PCI Compliance isn't something that someone can sell you and even a PCI compliant environment can be misused - creating a hole in your assessment.

What is special about the cloud from a PCI perspective?

First off, you don't control the physical environment and therefore you are dependent upon your provider's physical security measures to maintain compliance. This doesn't need to be a problem as there are numerous providers available now that can provide "bare metal" that is certified compliant for you to work from.

You still are likely to have the responsibility of maintaining the virtual machine environment, updating operating systems, app servers, frameworks, applications and databases. How do you offload that responsibility even further? PCI is all about the cardholder environment and the protection of Primary Account Numbers (PANs).

Whether deployed in the cloud or on-premise, the guidelines are the same and keeping your exposure minimized is key to simplifying your PCI compliance.

Isolating your cardholder environment and ensuring servers have a single purpose is key area of compliance and if you can leverage a cloud environment and a providers physical security to meet this goal, so much the better.

Know Your Data

You've hardened your processes, separated environments, encrypted tables in your DBs, and trained your developers and IT staff. Then comes your audit and your auditors jump in and run a script against your DB. Left and right you start seeing things that look like Primary Account Numbers! What? Where? How did this happen?

One of the challenges of PCI is that you'll be focused on your cardholder environment and your payment processing tools and applications. Meanwhile, you've got API's, web forms, chat windows, log files, support tickets and any number of other places for data to hide. Your customers, developers and employees will likely have innocently created a PCI Compliance risk.

In what other streams does data enter your system? Do you have a customer support or CRM tool? Often customers, who may not be PCI savvy, will send you information they shouldn't. An email with a credit card number (aka PAN) in it asking for a refund, or a chat window or a comment in an API call are problematic.

You didn't expect this channel of communication to be used to send credit card data, yet

there they are – those 16 digits uncovered at the wrong time, during your assessment.

The antidote?

Know your data. Know what information is flowing through your system, what information is stored and what might be masked on collection or display. If you're an API provider, this can mean watching your APIs for sensitive information passing through. Not only PAN data, but also other data can be useful to avoid storing unencrypted or storing at all. Social Security Numbers are another sticky piece of data you'd like to avoid if possible.

Leveraging tools to help you discover your hidden vulnerabilities is one tactic as is encrypting vulnerable tables. Eliminating those vulnerabilities is a better route. As we shift towards an API economy, knowing the data passing through your APIs grows ever important in achieving and maintaining PCI compliance.

Ask us about Apigee PCI Gateway Policies and how we can help you know your data.

For more on PCI and how it might impact your API strategy, check out our recorded webinar: Does your API need to be PCI Compliant? a*nd the* blog.

8. API Design: Are you a Pragmatist or a RESTafarian?

This starts a series of posts on RESTful API design best practices that we've observed over the last year.

I also covered a lot of the tips and tricks in the RESTful API Design Webinar video.

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

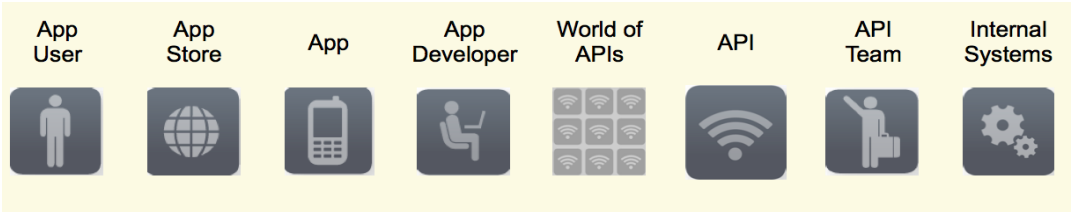
You might have seen discussion threads on true REST - some of them can get pretty strict and wonky. Mike Schinkel sums it up well - defining a RESTafarian.

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

Why? Look at the value chain below - the developer is the lynchpin of our entire API strategy. So design the API to maximize developer productivity and success as the primary design principle.

This is what we call pragmatic REST.



Pragmatic REST is a design problem

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

The design aesthetic for APIs is to think about design choices from the developer point of view.

This is pretty much the guiding principle for all the specific tips that we dig into in the RESTful API Design series.

9. One size doesn't fit all: API Versioning and Mediation

TrueCredit.com tells a story of calculating that they would need thousands of IP addresses for all the different versions and flavors of their open API to account for different variations and versions needed by partners.

Even if you start with a *one-size-fits-all* API, you might need to be able to transform data, mediate terms or customize SLAs without coding each change or creating a new version of the API. Reasons could include:

- **Protocol needs** - We have a SaaS customer with a REST API that had an important deal on the table with a bank. The bank insisted on a SOAP API with WS-Security. On the other hand, some SOAP shops want to offer a RESTful API because it's easier for developers to work with. And we see lots of needs to transform between different syntaxes of SOAP, REST, or REST/JSON, and the list goes on.

- **Monetization** - You might want to sell a premium version of your *one-sized-fits-all* free API. For example, a search API provider wanted to do a BD deal with its free API and needed to insert extra data the partner wanted to pay for.

- **Standardization** - If you have a lot of APIs, built by different teams, one day you might want to enforce consistency. A customer of ours grew from one API to 40, and needed to add some standard fields to each API without needing to coordinate 10 teams to write code.

- **Versioning** - Ever used an API where you get an email every month asking you to upgrade to a new version? TrueCredit wanted to provide API upgrades to customers that needed it while holding the API fixed for everybody else longer to reduce versioning headaches.

In short, you may need to figure out how you can provide and manage different flavors or versions of the same API – or mediate (or transform) API content and syntax.

Alternatives might be to

- Support multiple APIs (painful)

- Hold off as long as possible and push back on customers to snap to a *one-sized-fits-all* model (more painful)

- Create a *mediation* capability or a layer that can transform between different 'shapes' of the API – protocol, data, version, credentials, and so on.

(All these were reasons for TrueCredit (mentioned above) to think about an API gateway for mediation, caching, load balancing, and more.)

Considerations and questions to ask for API roadmap

Will you need to mediate protocols?

- Do you anticipate needing to offer more than one protocol or a different protocol than what you have now? (SOAP for enterprise customers? REST or JSON for developer adoption?)
- Do you anticipate needing to map across different security or credential schemes? (For example, from simple HTTP auth to WS-Security.)
- Do you currently write code to transform between different styles of a particular protocol (SOAP 1.1 vs. 1.2, and so on)?
- How important will it be to reduce the number of APIs versions for development and maintenance?

Version management

- How often will you need to release upgrades to the API? What is your process for asking customers to upgrade and how long will it take to sunset a version?
- If you offer more than one API, do you have a need to standardize elements of the API (header or payload)? Do different teams need to do this or does it make sense to put this capability at one point?

Message enrichment, clipping

- Will you ever need to enrich an API for a particular customer or class of service with more data or functionality? (Do you have a big customer that licenses more data?)
- Will you need to remove or clip certain fields for certain customers or classes of service?
- How fast will you need to turn around these requests for the business vs. your development or product cycle?

(Thanks to collinanderson for the photo)

10. Welcome Aboard: API User Management

Just like you have processes to add and maintain user accounts with your website, you'll need the same for developers using your API. Making it fast and easy for developers to quickly get API account details squared away is critical to reduce barriers to adoption. And good user management can pay off by reducing your support costs and increasing customer satisfaction.

This includes setting developers up with an account and any authentication credentials they need to work with your API. But you'll also need UI and processes to add, maintain (such as reset) and revoke or delete these accounts. You may also need to take into account how to distribute keys or OAuth keys, and even consider an approval process, although you may not need these.

Also, consider whether you can make things simpler by defining rights around standard user profiles. For example, you may have 'bronze' customers and 'gold' customers who each have access to different APIs or requests.

Finally, consider any other information you can give developers that will make it faster and easier to work with you - to get their questions answered without calling you. For example, usage statistics and alerts on versioning upgrades and outages.

Do you need to start from scratch?

Before building or buying anything, consider if you can extend some other existing user management UI and processes to work for developers requesting access to your API.

(For example, we built our own user management system for Apigee quickly using our existing salesforce.com account, with just a couple integration points to our account provisioning functionality.)

Key to this decision is how much integration you need into your existing business processes. Do you need to create accounts in your CRM system, and make sure you have enough developer information to map developers to applications and customers?

Consider the following user management questions for your API roadmap

For on-boarding developers

- Do you already have a way to manage user accounts that you plan to re-use for your API? Do you also wish to offer OAuth keys?
- If you have no user management at all, what does a user need to do in order to sign up to use your API?

- Can they sign up quickly and easily using a web browser?
- Can you simplify things by defining 'tiers' of users?
- What kind of information do they need to have access to?

For maintaining and managing accounts

- Can you re-set their passwords?
- What user interface do you want to create for user management?
- Can users do it using a web site or is there some other way?
- Can you monitor their usage? Can they monitor it themselves?
- Can you revoke user accounts?
- Do you need to implement an approval or screening process?
- Do you need reporting and analytics around users – active developers, engagement and retention rates?

Integrating your API users into the rest of your business

- Does your developer activity need to get mapped into your sales, support, and ERP systems?
- Does your API key structure enable you to map developers to applications, your customers, and their end users?
- Does user data need to be integrated with existing profiles or user data systems? Can you use existing SSO (single sign-on) systems?
- Can you create usage incentives by providing developer access to their own usage data?

11. If you build it will they come? API Community and Audience

You've built a great API, and have all the operational bases covered. But it doesn't matter if nobody knows about it. Awareness and adoption are likely a big reason you might open an API. So if you think of it like throwing a party - you need to make sure people know it's happening, guests show up, have a great time, and make it an interesting.

(And while many APIs are private – such as a SaaS API only for your partners – you still need to do a lot of this.)

But first you must have a cool product.

Check out Dave McClure's excellent developer programs presentation. Dave makes a key point that your product "better be cool." For your developer audience, starting with a great, differentiated API is key.

Developers, developers, developers

Developers make or break any 'openness' strategy. They are your greatest source of feedback, your best advocates, and your toughest critics. Focus on making developers successful and you are building on a very strong foundation.

Your Developer Community

One of the best ways to make developers successful is to put them in touch with each other – so that they can share information, tips and tricks, and creativity in working with your API that you may never figure out on your own. That's why some of the most successful software companies have the most vibrant developer communities.

So you need to put down the tools that help developers work together and discuss and share knowledge about your APIs. Blogs, wikis with documentation, newsgroups, and venues for code examples are all important. One of the best tools for your developers is a code sample that they can incorporate in their own apps. Having a place for not just your own sample apps and widgets but also for developers to share their own is critical.

Don't forget about "Audience"

Think of launching an API like throwing a party. Sure, you need a great venue, food, and music. But how do you make sure you get the word out to make sure lot of guests show up? We see lots of API strategies that focus on the community tools, but tools do not equal a community. If you can't find an *audience, * it's like a party with great catering that nobody knows about.

A big part of this is making sure your API is highly ranked in Google searches on anything related to "API + your business." There might be 10 other APIs in your vertical

space – you need to be near the top of the list. This is another reason why being on all those other community sites, directories and discussion boards is so important, as they give you the links that raise the Google score that gets you found, that gets you used, that gets you found, and do on.

You can't outsource passion

Should you outsource community? Be careful. The best communities have a 'hot core' of very active members and the ringleader is usually an incredibly passionate evangelist that knows each of these folks personally, and is obsessed with tracking and immediately responding to any question or issue, or leaping at any opportunity to talk about how cool your API is.

Can you outsource that passion? We recommend not – find one dedicated, maniacal evangelist in your organization to nurture community.

Finally, consider evangelizing your developers – if they are doing great work with your API or platform, give them props and make them stars.

Your Roadmap: Audience and Community

If you are opening an API, think about the following for your community roadmap and don't forget about "Audience."

Community tools

- Do you have formal documentation? Can you put it on a wiki so that developers can edit, add, and comment?
- Do you have code samples on how to use the API?
- Do you have a place for developers to put their own code samples and showcase their own work and sample apps? (Widgets, mobile apps, etc.)
- Are code libraries needed for important platforms? Should these be open source?
- Do you have a blog for best practices and a way to get in touch with developers on important changes, such as API version updates?

Audience and distribution

- Can you get awareness and distribution through existing developer communities, such as those of vendors (MS, Google IBM), platforms (Ruby, iPhone), and independent directories (ProgrammableWeb)?
- What Web marketing or SEO/SEM - Search Engine Optimization or Search Engine Marketing (Adwords) - can you do to make sure developers find you when searching for "API" + your type of content or business?

Community management

- Should you have a dedicated full-time employee to drive community and evangelize your product and best developers?
- Are there any offline events or meetups you should be at?
- How can you recognize and promote your hardcore community members? Do you have an evangelist that knows these folks personally?

12. Developer Segmentation: who are they and how to reach them?

Why think about developer segmentation for your API strategy? Very simply, developers are your new channels. In the way that retail stores were a channel for your packaged goods, in the new world of APIs, apps, and Internet services, developers are your channels. It important to understand developers' goals, needs, and approaches in order to target them effectively, have them adopt your API, and make both them and you successful.

Let's start with how we define a developer? Perhaps because developers of one kind or another are part of so many different businesses, there end up being a large number of definitions. Because people mean and understand different things by the term **developer**, in their webinar on Developer segmentation for your API, Sam Ramji and Brian Mulloy use the term **coder** to describe an individual human being who is coding. Check out the video and slides!

There are many ways to think about developer segmentation, and they are evolving. In general, we think about it in terms of vertical, horizontal, modal, and tribal.

Horizontal segmentation

It is important to understand where a coder is employed because coders in a given industry have particular attributes, goals, needs, and approaches.

- Large ecosystem players (examples are Electronic Arts, Gameloft, CNN, Fox).
- Medium & large enterprise (writing first-party applications, including internal business apps)
- Partners (writing integration code and first-party applications)
- SaaS companies (Salesforce.com etc.)
- Software start-ups
- Small-midsize business & small enterprise
- Hobbyist/opportunist (note that this is different from software start-up but could turn into one)

A good way for businesses to think about the coders they work with (or potentially work with) is to think about whether they're working with known companies or unknown coders. In many cases, the advantage of designing and building for the unknown coder is that it helps you with your business efforts with known companies. (Unknown coders often work at known companies!)

Vertical Segmentation

There are numerous vertical segments and there are both industry-sized verticals and vertical projects or functions within larger organizations. The US government classifies more than 4000 standard industry codes and one could argue that they define as many verticals. Coders have different functions and concerns depending on the vertical – they need to think about different standards, have different cadences, and so on. Your business probably fits into one or more of these verticals.

- Social media
- Mobile
- Retail
- Digital media (audio/video)
- Publishing (FT.com, etc.)
- Financial Services (especially brokerages)
- Hospitality
- Logistics
- Airlines
- Telecommunications
- Healthcare

Modal Segmentation

You can think about segmentation around the **mode** of the API: whether the coders have access to open or closed technologies (open systems like HTTP, PHP, vs. closed, propriety systems like set-top boxes). Public or private access modes are another aspect of this.

Public vs. private API approaches can cut across segments depending on the business model.

- **Public** - The business model is usually to attract more usage of an existing paid service to improve utilization, or reduce churn, or both. You are targeting any and all coders, but especially unknown coders.
- **Private** - The business model is usually to accelerate business development and innovation with existing partners.

Open vs. Closed (or proprietary) API models are orthogonal to public vs. private.

- **Open** - Open APIs allow anyone to access after registering for an account.
- **Proprietary** - In a proprietary or closed model, only approved entities can access the API.

An Open model is smart because it accelerates innovation, but a Public model can make it easy for developers to "taste" the value, creating demand from coders, who will subsequently be willing to jump through hoops to use the super-tasty Private functionality. Think about it as analogous of the *freemium* model for websites and apps – it is designed to give the users value over and above what they are paying for at every level of the pricing curve.

Whatever your business and however locked down it needs to be, think about whether its possible to find some methods of your API that you can make available for free - no registration, no authentication, no concerns about authorization. It allows coders to get to know your API and prepares your business for a situation in which you need to be more agile than you originally expected. It can create a zero-barrier-to-entry proof of concept.

Tribal Segmentation

HTML5, Ruby, JavaScript, Android, iOS, PHP, Java, Rails, Scala, C#.net, jQuery, Heroku, Scala, AWS, Sencha, and on and on . . .

Coders identify around and become passionate about technologies. There are "tribal" affiliations to technologies and groups of related technologies. Having an understanding of the tribal segmentation and how it evolves over time helps you target the coders that are important channels for your business.

Reaching coders and plugging in to developer communities

Think about reaching coders through incentives that match the segments you are targeting.

In the horizontal segment, incentives can run the gamut from paying the coders that work for large enterprises to showing customer demand to system integrators to offering free access for hobbyists.

The vertical segment is complex. But there are technical conferences and other meet-ups that people already go to. Be there. For example, if your business is Healthcare, you can attend a HIPPA conference – learn about the needs and challenges of coders in this segment – about how to make your API HIPPA-compliant, and so on. Bottom line; participate in the communities where your coders already participate.

Depending on whether your API strategy is public or private and open or closed, you have different ways to reach and connect with coders to get your API adopted.

While building your own community can be important, there are lots of existing developer communities, large and small, formal and informal, online and offline that you should plug into right away. They depend on, and are set up to distribute and promote your great content (your API and samples).

These might include:

- Vendor communities like Microsoft's MSDN or Google's COMMUNITY works. These guys might be your partners, but if they are not it might not matter – plug in anyway. Even if you compete or do not build on those platforms – there will be developers there that work with those platforms and use your APIs.
- Platform communities, such as RubyOnRails, or iPhone.
- Independent communities like stackoverflow.com. These existing communities have audience and depend on great content from API providers and developers.
- API Directories and communities like ProgrammableWeb. These are where developers looking for great APIs and information about them come.
- "Vertical" communities and industry events: Be at meet ups and conferences that coders in the vertical already go. Have a relevant offer.

13. Don’t try to market to developers. Instead, solve their problems

Not long ago you could count the number of 'developer marketing' programs on one hand. Now there are hundreds of programs as Web companies and enterprises open APIs. These companies know that developer adoption will make their API strategy succeed or fail.

But Developer Marketing is an oxymoron. Developers hate marketing.

You cannot drive adoption by 'marketing to developers.' Sure, you can send offers to your developers but your mileage may vary.

What’s a better formula? Understand what's important to developers and give them what they need to reach these goals.

Developers want to:

- **Build new skills** that lead to the best projects and jobs. This is why new or proprietary tools and programming models are tough to get off the ground - it's a small market of new projects for the developer.
- **Increase their productivity** with great tools and by connecting developers with decent resources and each other for help. This is why sites like stackoverflow take off.
- **Be recognized for good work** and see their products used. Focus on showcasing their work, not your product. It's not about you.
- **Get paid**. Think App Store model, or affiliate marketing networks.

Talk to the folks that made the big developer networks successful and you'll hear these points over and over. Some others:

- **Developers are not buyers** but are very strong influencers. There are superstars in the developer world - make them fans and that is the best marketing you'll ever get.
- **You can't 'own' or 'use' developers** because they have an account on your service. Developers have lots of options and switching costs might be low from your API.
- **Act on their feedback**. Developers are smart and listening and acting on their complaints and ideas is critical to your credibility.
- **Developer communities are fragmented**. For example, there is no such thing as an "API developer,” but instead there are Twitter or Facebook or Salesforce developers.

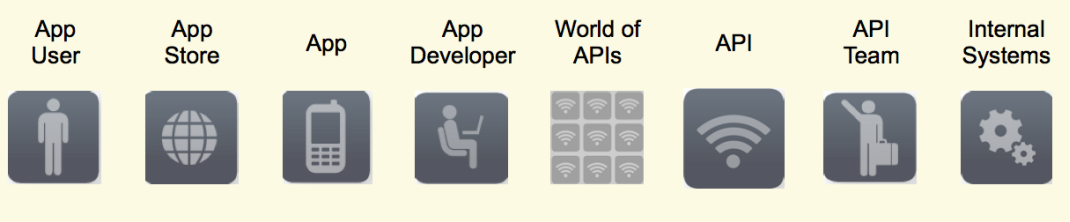
Once you have attracted a developer to use your service - they are like gold. So treat them with respect - don't try to 'use' developers or you might lose them!

14. Moving the needle: What API metrics to measure

It's an old cliché, but it's been said that you can't move the needle if you can't **see** the needle. So frequently we're asked, "**What are good metrics to measure an API program?**"

With today's rapid adoption, mobile apps economy, and open APIs, understanding what to measure is less predictable than it once was, and less predictable than when you are working internally or with partners.

We've found it's good to look at this question from the perspective of who is going to act on the metrics. Successful API teams understand the value of people and technology and two sets of people make APIs successful – app developers and the API team. App developers are innovating on your API, building cool and creative apps. The API team is tapping into internal systems to create interesting APIs.



What does each of these groups of people need to see? What actions can they take on metrics to either prepare for the future of improve the current situation?

What does the app developer need to measure?

Some important questions that metrics can answer for app developers are:

- Who is their app user?
- How are apps performing?
- How is the API is working? (From the perspective of the customer of the API – needs visibility to different things than the API team who develops the API.)

Drilling deeper. App developers will have several questions that are about the API in general. Think about whether you can provide trending charts so that app developers can access the aggregate data for your API:

- Is the API error prone?
- How does the API usually perform?

- How often is the API unavailable?

Then there are questions that are more specific to the app developer or app's usage of the API or to the point in time:

- Which API errors is my app seeing?
- Is the API slow right now?
- Which API methods are slow (generally and right now)?

  What about specific methods and parameters - is a given method typically slow or slow right now for some reason?
- Does your API have a quota?

  As the app developer, how am I doing against the quota? Am I violating the quota? The last thing you want your app developers to do is waste time debugging software that doesn't have a problem – so knowing that there is a quota and how they're apps are doing against that quota is important.
- Is the API down right now?
- When will the API be back up?
- Why was the API down?

What's in your developer portal?

You should provide data in your dashboard or developer portal that helps developers answer these questions and act to improve their apps or plan for the future or both? These are different data than those needed by the API team (see below).

Here are the key indicators and formats for metrics for app developers.

Key indicators

- Errors - rate, code, and descriptions
- Performance – trend graphs for the API (all data) as well as for the developer's specific apps using the API
- Availability – is the API up or down?
- Quota – limits (if enforced) and how an app is doing against quota

Data formats

- Everyone's aggregate data
- Developer-specific data - app by app (a developer might have multiple apps)
- Trend charts
- Categorical tables

Broadcast formats

- Status page
- Twitter
- Blog

Sometimes, the app developer needs information other than quantitative – numbers and charts. They sometimes need a blog post or a tweet that gives them the status of an API problem being worked, anticipated up time, why the API was down, whether the API provider has diagnosed and understands a problem and how not to allow it to happen again, and so on. Some great examples of API teams building trust and partnerships with app developers include trust.salesforce.com and GitHub's blog and Twitter following. It's key to think of app developers as part of the extended team – they are after all carrying your brand out to the world.

What does the API Team need to measure?

Almost everything! There are probably two modes in which API teams operate – sunshine and overcast (or stormy).

In sunshine mode, there are no problems, no app developers complaining, no new releases, and so on. This is the time to work on tuning your API metrics – your visibility into your API operations and business – figuring out if you are measuring the right things. Some of the things that help you really understand your users so you know where to invest for the future include:

- Which are our top apps?

  You need to know who/which apps are using your API.
- Who are our top app users?
- Who are our best app developers?

  Your best users and developers might be operating with you across multiple apps and devices - Web site, mobile apps, video game consoles, tablets etc.
- Which API methods are most popular?

  Your API is a product - you want to understand its usage to help pivot your business and make future investment decisions.
- How much API capacity will be needed next year?

  Capacity planning is difficult. What if your API goes viral and you have millions of users? You'll need trending usage data to help make these decisions. In this context, you'll want to see indicators in the trend charts like the number of requests (the load hitting the back-end systems), volume of data (especially for API responses), and so on.

In overcast mode, when things are not going so well, the API team will need to know:

- Why is the API down?

  There's probably not one thing that'll tell you why an API is down but the shape of your graphs can help - do your charts show catastrophic failure or a slow lead-in to failure?
- Why is the API slow?

  Does the API team have some mechanism for doing synthetic transactions? Can the team look across API methods and also across apps? (Could be that apps are doing something that's slowing your API.)
- Why is the API throwing errors?

  Test scripts can help here. Can the API team look at main apps - are they all or some of them throwing errors? Be verbose in error messages; look for key words in logs.
- Why is API traffic spiking?

  Is something new going on? Can you nail down the source of traffic – has a new app come online?
- Why did the API traffic disappear?

  The opposite of spiking traffic. Do you have a business issue - did everyone uninstall the app overnight? Or is it a back-end system problem not necessarily to do with the software - forgot to renew certs, DNS?

API Team's Key Indicators

Remembering that a mix of technology and people make your API business successful, the key indicators your API will look for are:

- App users
- Apps
- Developers
- API quality
- Internal systems data (database query times, message bus response times, external callout response times, etc.)

Data formats

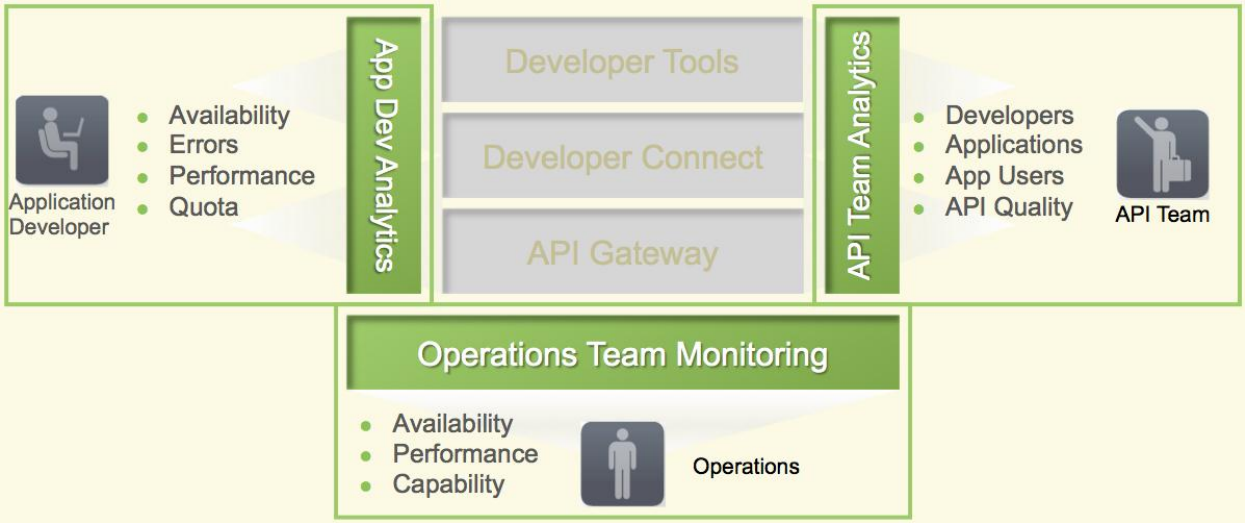
- Alerts

- Aggregate data
- Drill-down data
- Trend charts
- Categorical tables

In addition to knowing how app developers, apps, and their end users are using the API, the API team needs to measure what's going on behind the API – in the internal (back-end) systems.

It is a good idea to put internal system data in response headers. Apigee can log this data in the database with custom policies and provide metrics to the API team/provider and even to the app developer if appropriate.

In summary, decide what to measure based on what will be actionable by the three types of people who make your API business successful: app developers, the API team, and the operations team.



Check out the blog for more on metrics. *(Thanks to **seenoevil** for the photo.)*

15. Show me the money: Monetization

At some point most API product managers are on the hook to demonstrate how the API drives revenue.

And even if you never intend to charge for your API, you may be pleasantly surprised by unexpected opportunities.

"I used to be responsible for a number of 'free, open, take-it-or-leave-it APIs' at a large Web portal. After a few months, the team was surprised to get a ton of calls from big companies that wanted to pay (a lot) for use of the API within their commercial products."

The catch – they expected that the API could support some of the basic business terms of any commercial deal. For example – could the API provider:

- Meter service and give them a bill every month?
- Enforce a special rate limit and give them an "SLA" (service level agreement)?
- Insert extra data fields if they were willing to pay for more licensed content?

We also needed some other basics:

- SOX compliant audit logs for each deal.
- Reporting on the cost of the data served by the API (because we were licensing it from another company).
- The means to enforce similar terms on a 'package' of multiple APIs together in one deal.

Unfortunately, all of these capabilities needed to be coded as they came up and we couldn't lock down most of these deals in time.

So even if your API business model is already covered and baked in your API, you may want at least the option to support a slightly tailored or customized relationship. You may need to quickly tailor the API's syntax, protocol, priority, or content to a specific opportunity.

Once you have partners using an API, your business managers and partners will put your team under a lot of pressure - each day a policy change goes unmade, dollars are lost.

So consider if you'll ever need to abstract these policies – the difference between the API content and features of the commercial business and the operational terms of each deal – in a separate policy layer that you can manage quickly and independently of development cycles. Or will you need to make a single change across multiple APIs?

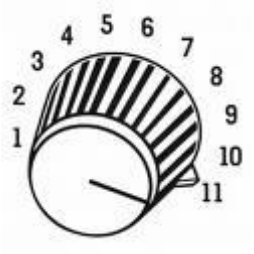
Your API roadmap: Monetization

General business and partner model questions

- How is your business and revenue model supported by your API? Are you looking for distribution and awareness only? Does the API drive a monetized transaction? Will you ever charge for 'pay by the API drink'?
- How are your costs reflected in your API? Do you pay for any of the data you are serving up? If so, how do you keep track of this for the business and partner?
- Will large partners or deals surface where your team will need to change the API content, SLA, protocol or content? Is there a partner that might want to pay enough and who is large enough to drive your team to move away from "one size fits all?"
- Will you need to create 'service tiers"?

Enforcing unique business and operational terms

- How will you meter service like a utility, so that you can bill partners and report data costs?
- What regulatory compliance will you need to support? Do you need SOX-compliant audit trails by partner? HIPPA? PCI?
- How would you create and enforce a partner-specific SLA, rate limit, or offer 'priority access' to a priority partner?
- If the partner wants any change in the API protocol or content, do you need to support a different API code-base? Is there a way to create a transformation layer to handle and scale this?
- Do you need to buffer up or queue incoming requests?



16. Turn up the volume: API Scalability

Our discussion of APIs so far has focused on aspects like security, visibility, and data protection. We've been leaving out a very important concept, though - how do you make your API scale?

The term "scale" means different things to different people; so let's narrow it down. The question is what are you going to do as your traffic increases? Do you have a plan to handle 10, 100, or 10,000 times more traffic than your API is receiving today?

The truth is that solving this problem at the truly high end can require fundamental changes to your architecture and code. The kind of engineering required to run an API that accepts a few requests per second is very different from what's required to scale to the size of Facebook or Twitter. However, the chances of any particular API needing to scale to that level, of course, are pretty slim.

Writing about all the dimensions of scalability and how to achieve them is a subject for a pretty long book. But here are a few specific things to think about:

- Caching
- Rate limiting and threat protection
- Offloading expensive processing

Caching

Caching is a huge part of any scaling strategy. Whole products and web sites are built around caching as a fundamental architectural concept. Caching works because even in an API, usually much of the data that's returned is read. A good caching strategy can decrease latency by huge percentages, and improves throughput by taking load off expensive back-end servers and databases.

Typically, caching today is done in a few different places. Caching between the application server and database using a product like Coherence or memcached helps by reducing the number of database queries needed to serve an API request. Additional caching inside the application server code can further decrease throughput by making it possible to reassemble parts of an API response from pre-cached parts. For instance, the response to a typical Twitter API call consists of an arrangement of many individual snippets of XML, each of which may have been cached to reduce the overhead needed to fetch data from the database and convert it into XML.

HTTP and CDN caching

Caching at the HTTP level is also very common. The HTTP protocol supports several headers that enable caching by proxy servers and of course by the web browser itself. This type of caching works fine for API calls. However, since it's based only on the

HTTP request, it doesn't work for everything. Imagine a SOAP API, for instance, which includes an HTTP POST body describing the request - HTTP-only proxies cannot cache the response because they can't look inside the request and see what needs to be cached.

Similarly, CDNs like Akamai work by caching pieces of content all over the Internet so that users will receive it from a server near them. Caching strategies like this are great for big files that don't change often, but by design they are poor at data that changes more than every day or so.

Caching API responses

APIs lend themselves nicely to caching responses because it is often easy to identify the cache key. If you follow the REST pattern each URL maps uniquely to a resource that has a well-defined lifecycle.

For instance, imagine a "getAccount" method on an API. Software like Apigee Enterprise can look at the parameters to the request and use them as a "key" to the cache. If a response already exists in the cache, ServiceNet simply returns the exact response from the back-end server from its in-memory cache. This cuts out all the latency to make an HTTP request to the application server, process the request, query the database, assemble the response, and so on - and unlike a cache that works only at the HTTP level, it can be configured to understand the semantics of the API so that it caches effectively and correctly. There are also ways to purge cache items programmatically, so that the "updateAccount" API method ensures that the next call to "getAccount" won't return stale data.

In other words, adding a intelligent caching layer between the client of an API and the back-end application server adds more caching options that can increase scaling and decrease latency even further than some alternatives because this cache is closer to the API client. Plus, due to the magic of HTTP, XML and/or JSON, it's possible for this caching to be performed without any changes to the API client or server.

This means that a highly scalable web site could use caching in different tiers, each with its own contribution to overall API performance:

- A cache like Coherence or memcached may be used between the application server and database to reduce database load.
- A similar cache may also be used from within the application server to reduce the overhead of assembling API responses.
- A proxy like Apigee Enterprise can cache complete API responses, reducing load on the application server tier.
- A CDN like Akamai can provide an extra caching layer for large files that do not change often.

Rate Limiting and Threat Protection

Another aspect of scaling is just keeping unnecessary traffic away from your application servers and databases. Some of the techniques that we've discussed previously, such as rate limits and threat protection, apply here as well.

For instance, an API's performance can drop precipitously if a client, on purpose or by accident, sends too much traffic. A rate limit helps a lot here!

Bad requests can kill API performance too. XML threats, which we discussed in the last episode, are one example of a way that a bad request from a client can cause performance problems or even a crash on the server side. It's a lot easier to maintain scalability if you can stop these kinds of problems before they can hurt your servers.

Offloading

Finally, consider the things that you can offload from your application server tier. The more you can offload to more efficient platforms, the smaller load your application servers have to handle. Plus, the more things you can offload, the simpler those application servers and their applications become, which means they're easier to manage and easier to scale.

For example:

- **SSL**. Load balancers and ADCs like F5 and NetScaler products, not to mention web service proxies like Apigee Enterprise, can process SSL more efficiently than most application servers.
- **HTTP Connections**. Those same products are highly optimized to handle tens of thousands of simultaneous connections from HTTP clients, and operate a smaller pool of connections to the back-end application servers. Offloading HTTP connection handling to another tier can free up a lot of server resources.
- **Authentication**. If you perform authentication, a proxy like Apigee Enterprise can handle authentication for you, freeing your application servers to worry only about properly authenticated requests. And if you're using SOAP, a product like ServiceNet can process many of the SOAP headers, such as WS-Security headers for authentication, then remove them so that the application server doesn't even need to see them.
- **Validation**. If your API depends on XML input, it may run more efficiently if it only accepts valid XML requests. Turning on XML schema validation can hurt performance of most application servers - products like ServiceNet can do it more efficiently.

Your API Roadmap and Scalability

Key Questions to ask for your roadmap:

- What kind of volume are you expecting?
- Are you prepared if you get 10, 100, or 10,000 times that amount of volume with little warning?
- Do you have a way to shut a user off if they consume too much volume?
- Do you have a way to control API traffic in case you are unable to handle the volume (see Traffic Management)?
- Are your back-end servers capable of handling tens of thousands of concurrent connections?
- Are your back-end services cacheable? Do you have a cache that you can use to reduce response times?
- Are you monitoring response times and tracking them to gauge customer satisfaction?

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://www.apigee.com.

Accelerate your API Strategy

Scale Control and Secure your Enterprise

Developers – Consoles for the APIs you