Developer Experience

Top strategies to improve the developer experience of your API

NORDIC APIS

Developer Experience

Top strategies to improve the developer experience of your API

Nordic APIs

Contents

Supported by Curity

Nordic APIs was founded by Curity CEO Travis Spencer and has continued to be supported by the company. Curity helps Nordic APIs organize two strategic annual events, the Austin API Summit in Texas and the Platform Summit in Stockholm.

Curity is a leading provider of API-driven identity management that simplifies complexity and secures digital services for large global enterprises. The Curity Identity Server is highly scalable, and handles the complexities of the leading identity standards, making them easier to use, customize, and deploy.

Through proven experience, IAM and API expertise, Curity builds innovative solutions that provide secure authentication across multiple digital services. Curity is trusted by large organizations in many highly regulated industries, including financial services, healthcare, telecom, retail, gaming, energy, and government services across many countries.

Check out Curity's library of learning resources on a variety of topics, like API Security, OAuth, and Financial-grade APIs.

Follow us on Twitter and LinkedIn, and find out more on curity.io.

Preface: What Is Developer Experience?

by **Bill Doerrfeld**

Developer experience, sometimes abbreviated as DevX or DX, is similar to user experience (UX) but focuses on the experience developers have while using a software tool. A tool's DX is a benchmark for how usable or intuitive the service is. And whereas DX used to be viewed as an afterthought, it's now becoming more of a necessity to stay relevant in today's digital economy.

What sets DX apart from UX is the frame of interaction. DX goes beyond the standard graphical user interface to consider the holistic developer journey across all interaction points, whether it's the reference documentation, command line, SDKs, libraries, API endpoints, or sandboxes. Generally, a service with quality developer experience is well-documented and comes with a solid getting started guide and sample code for common executions.

Naturally, software tools have varying degrees of developer experience. For example, if one method or parameter is not fully documented, an engineer might have to expend wasteful energy to guess how to interact with it. Or, incompatibilities between the reference and production behavior can cause frustration, leading to unhappy consumers.

Developer-Facing Tools

Developer experience can encompass various developer-facing tools, such as web APIs, cloud-based platforms, or open-source projects. Some of these developer-facing tools and products include:

- Software Developer Kits (SDKs) and code libraries
- Application Programming Interfaces (APIs)
- Open-source code repositories
- Code libraries, sample code, or tutorials
- Software-as-a-Service (SaaS)
- Platform-as-a-Service (PaaS)
- Infrastructure-as-a-Service (IaaS) and CloudOps
- Specifications and open standards

It's not just public SaaS that can benefit from DX — it's in a company's best interest to ensure their internal tools have quality developer experience too. Investing in DX can increase developer satisfaction and reduce burnout, improving employee retention.

Considering The Developer Journey

If we consider the entire developer journey throughout these tools, there are many areas in which the experience could be positive or negative.

To begin with, how easy is it to *find* the service? Developers may come across it through word of mouth or a simple keyword search and command-line install. Or, they may discover it through curated lists and registries. This is known as discovery.

Next, the onboarding phase could introduce friction or create a seamless experience, depending on the quality of documentation and reference materials. Are there code tutorials or sample apps for the most common use cases? How quickly can users get to their first `Hello World`? (For APIs, this is commonly known as Time To First Call (TTFC)).

Then comes the actual testing and production behavior. Does the tool behave as intended? Is it reliable? How easy is it to maintain, and what is ongoing support like? Many non-functional operational aspects could affect DX, such as rate-limiting, the developer dashboard experience, or costs.

What To Expect in our eBook, *Developer Experience*

In our eBook *Developer Experience*, we've collected our top-performing articles on developer experience from the past couple of years. The Nordic APIs writing team has shared countless insights on what it takes to create usable software that developers love and that makes them productive.

First, we'll look into improving the initial discovery and onboarding process to avoid leaky funnels that drop potential consumers. Then, we'll investigate how to enhance knowledge sharing with stellar documentation and helpful sandboxes. We'll cover techniques to measure the success of your developer journey and iterate it over time. Lastly, we've included many ways to retain consistency throughout a suite of APIs.

Popular public APIs continue to set the bar for developer experience expectations. And although the particular focus of this eBook will be on *API* user experience, these tenants could certainly be applied to improve the developer experience of many other flavors of software services.

Enjoy *Developer Experience*!

I truly hope this volume helps your journey to create more usable developer-facing tools. And if you like what you read, consider following Nordic APIs or signing up for our bi-weekly Newsletter on API design and strategy. Our blog is also open to submissions from the community if you would like to share *your* insights with a broader audience.

API Onboarding Is Broken (And How To Fix It)

by **Art Anthony**

Leaky funnels are often the result of poor developer experiences.

Even the best APIs in the world have leaky funnels. Potential users get as far as signing up, maybe even completing that action, only to abandon the process entirely. In some cases, the reasons for this are unavoidable. The product might be too expensive, for example.

In other cases, something happens between signing up and actively using an API. Examining how far people get in the sign-up process might help identify why they drop off.

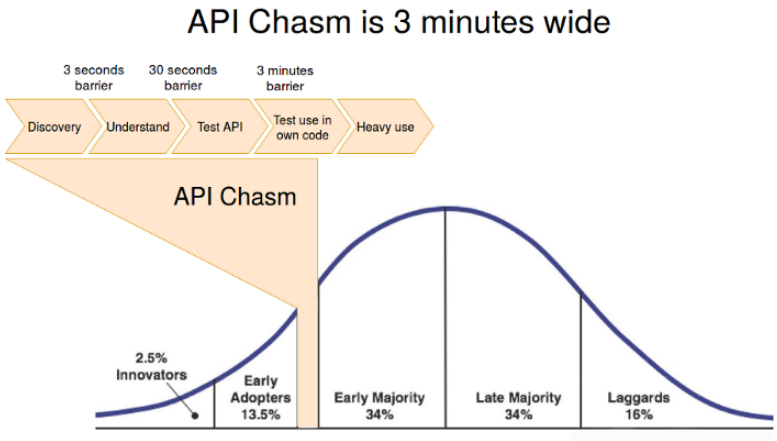
In this article, we'll be looking at some of the reasons a potential user might give up on an API before they start using it. We'll also

be looking at tried and tested ways, along with some emerging solutions, on how to reduce API abandonment.

Show, Don't Tell, (With Code Samples)

Geoffrey Moore's Crossing The Chasm is a book on tech marketing published back in 1991. You've likely seen the book's signature chart, a bell curve that visually outlines the pattern of innovators, early adopters, the majority, and laggards in tech adoption.

Applying this to APIs, writer Jarkko Moilanen suggests that the API chasm — dividing early adopters from the early majority — is **three minutes wide**.



To maximize engagement early on, he theorizes that potential developers should be able to test APIs using code examples within just 180 seconds of engaging with the product. While that may be overstating the importance of copy-paste code samples, his thesis is interesting.

The idea that code samples are as important as, although maybe

not more important than, documentation is one that it's difficult to disagree with. In an analogy, it's unlikely that you'd buy a digital camera after looking at its instruction manual without seeing some pictures taken using it or holding the camera in your own hands.

On the idea of "holding the camera", providing a sandbox is a great idea if you can provide one. If not, consider a free trial or a freemium model.

Creators of paid APIs might balk at the idea of giving away calls for free, but maybe they shouldn't — giving developers free rein to explore an API before committing to a purchase is part of a good onboarding process.

If a small number of monthly calls is enough to fulfill their requirements, they would probably never sign up for a paid account anyway. At least, in this model, they remain active and might upgrade if their needs for your service scale up.

Support vs. Peer Support

Chris Chandler, founder of sudoHERO, suggests that "unlike other groups of customers…developers actually prefer to find answers by reading through information which might be of use to solve their problem before engaging a support rep." A small-scale study from 2017 suggests the average developer performs 16 searches a day to do their job, with one major search category being "Understanding an API."

That's not (just) because developers don't like to admit defeat when they're facing a problem. Chandler continues that "to accurately diagnose a problem, countless variables need to be communicated in order to replicate the issue and provide a reasonable reply."

When submitting a support request, some of that important context is often missing. The result? "After waiting hours, or even days,

for a response, initial support tickets may be responded to with a request for clarification on some part of the question. Once provided, the message goes back into the queue and the developer waits…Again."

In other words, offering 24/7 support might not be quite the selling point you think it is.

Peer support, which feels more like brainstorming with fellow developers than submitting a support ticket, can be a compelling addition to the onboarding process because it gives potential users a sense of what those already using the product are doing.

This is something that Miro does well, with an appealing UI somewhere between Hacker News and a '90s forum. Note that Miro's Developer Relations Lead regularly jumps in to help people out; this helps the support site feel more like an active community, full of searchable Q&As rather than people shouting into the void.

This is a very different ballgame than searching for an issue with your MacBook or iPhone on Apple's Support Community and finding hundreds of other people with the same problem… but no one with a solution.

The Documentation Paradox

Brian Helmig, Zapier's CTO, said the following in conversation with Codementor: "information should dispense automatically by design, when it's needed."

We've written time and time again about the value of great documentation, but there's a type of conflict implicit in doing this successfully: the best API documentation should be digestible enough that it doesn't overwhelm, but should also include everything (or close to it) a developer could ever need to know. Those ideas stand at odds with each other.

Using automation, Chandler is trying to solve this problem with sudoHERO's Contextual Support tool. He suggests that "when working on troubleshooting an issue, the developer typically goes back to the documentation first to review the syntax and confirm their understanding of how a command or function should be used. If an answer isn't found there, they'll then look elsewhere for additional information."

Clearly, that's not ideal for API developers who would rather keep consumers in "their world". To combat this drift, the company offers a tool to embed knowledge base articles and relevant insights from other developers directly into their documentation.

We've already seen companies like Yext looking to shake up search for consumers using AI, but the idea of "going beyond search" has some really intriguing possibilities in the context of the API space.

Based on feedback over time, the product gets better at delivering the best available answer to developers. In turn, tools like these can identify which parts of their documentation generate the most queries and can use that information to clarify it.

Measure & Iterate

Cultivating an engaging developer experience is not a "one and done" process. When you first create your developer portal and documentation, there's a significant amount of guesswork involved. But, as time goes by, you may want to refine this process.

Finding a way to track the behavior of users, and potential users, in your API portal is a great way to identify bottlenecks and areas of interest. When you know what those are, you can address and accentuate them respectively.

Pronovix's Zero Gravity developer portals are an example of such a documentation experience, prioritizing iteration and an engaging editorial experience for authors/editors.

There's (much) more to creating compelling documentation than just using an OpenAPI spec and Swagger tools, some of which can be used to generate multiple documentation versions from OpenAPI specs, but it remains a good starting point.

API Onboarding: Final Thoughts

In our article on the importance of Time To First Call (TTFC) as an API metric, we talked about aiming to remove roadblocks in the process of signing up and getting started with a product. Above, we've outlined a few key ways to do that:

- Code samples to give people an idea of how to use the API
- Extensive documentation once they're ready to jump in themselves
- Going beyond traditional support by building communities and better context
- Iterating on the processes above based on feedback and analysis of user behavior

The title of this post states that API onboarding is broken. That's not always entirely true, but API onboarding often does lag behind some of the well-established funnels that exist in the SaaS space. And, with APIs now being respected more as freestanding products, there's no excuse for that.

Fortunately, it appears that tools and systems are on the way (or here already) to help API developers take their onboarding to the next level.

Everything You Need To Know About API Discovery

by **J Simpson**

Focusing on "discovery" is a first step to getting your API noticed and increasing use.

Did you know that there are over 24,000 APIs listed on ProgrammableWeb alone? There were around 19 million developers working with APIs in 2020, as well.

The API industry is *exploding*, and it will only grow more pronounced as the years go on. Business owners and developers realize there's money to be had in developing APIs. This is leading to a virtual Gold Rush as developers raise to capture the hearts, minds, and wallets of their audience.

This leads to the question — how can you stand out in such a crowded marketplace? What makes your API different from the rest in the API directory? And if you're a programmer, how are you supposed to find the right API for your specific needs?

API discovery.

The ability to be found is one thing that determines if an API soars or sinks like a stone. This means that API discovery is vital for your API's success.

We've put together a thorough guide to API discovery to help you learn how to make your API stand out, get found, adopted, and spread!

Guide To API Discovery

API discovery is a bit more complicated than simply creating a landing page for your API and letting SEO take care of everything, although that's certainly part of it. It also ensures that your development process is as efficient as possible as you're less likely to end up doing unnecessary work.

APIs tend to come in two forms — internal and external. As far as internal APIs are concerned, API discovery makes it so that your team can find existing APIs for a particular application. It can also give you an idea of new features to add to your API to make it even more useful.

For external APIs, API discovery is how others find your API. Even more importantly, it helps your users know how to *use* your API once they've found it.

API discovery is also an essential part of API lifecycle management, as it helps your development team find the right API assets for their projects and helps to make sure they're maintained and secured.

Types Of API Discovery

API discovery *also* comes in two major types — manual and automated. Naturally, automation is important if you have any hopes of your API discovery being scalable. You simply wouldn't have time to document every single feature for every one of your API assets.

Automated API discovery functions similarly to an API specification. This standardized format achieves several goals at the same time. First, it has many of the same benefits of an API specification — it's easy to automate and scale to whatever size you need. More importantly, this same specification lets others *find* your API without creating more work as yet another benefit.

The existence of standardized API discovery also means that there are useful and powerful tools for API discovery. These tools can often perform additional functions simultaneously, so it's even possible to add an API discovery tool without having a whole, separate solution.

As the API industry continues to develop and flourish, there are efforts to make API discovery more standardized. For example, APIs.json standardizes API discovery in a way similar to how OpenAPI standardizes an API's description.

This standardization allows for simple-but-essential tools, like a spec-driven API search engine. For example, APIS.io, which uses APIs.json, is an API search engine that pulls APIs from a wide array of different sources, all of which are tagged in a variety of useful ways.

API specifications end up having a similar end result, acting as API discovery tools in their own right. There are search engines for Swagger and web APIs. Postman even created their own API network formatted for Postman.

Tips For API Discovery

As we stated at the top, API discovery is integral for *anyone* working with APIs. For API developers, it's the key to getting your API found and used. This means discovery is key to realizing your API's goals, whether that's monetization or increasing your company profile.

With that in mind, here are some more tips to make API discovery as effective as possible for you and your company.

Document Properly

At its heart, the key to API discovery's success is essentially the same as successful API documentation. The two can even sometimes be handled at the same time.

You can even think of API discovery and documentation as part of the same process. API discovery will help users find your API. Proper documentation will show them how to use it once its been found.

Make Sure It's Machine Readable

One of the reasons API discovery is so powerful is due to machine readability. While it's important to have clear, understandable language for human comprehension, the ability for a computer to decipher what your API is about may be even more important to your API's success.

If you're serious about your API's success and think you'll be managing a larger suite of APIs in the near future, you'd be advised to look into one of the emerging specifications in this area.

List Your API With API Directories

Even though automated API discovery tools are becoming more common, you simply can't overlook API directories if you want your API to succeed. Sites like RapidAPI and ProgrammableWeb can help generate traffic for your service. You simply can't afford to miss out on that potential audience!

Of course, ProgrammableWeb and RapidAPI aren't the only API directories in the game. Public-APIs is a GitHub repository with thousands of free APIs organized by topic. If you've got an open-source or free API that you'd like listed on Public-APIs, or if your API has a free tier, you can read their submission guidelines here.

APIslist.fun is another API directory with a clean, easy-to-navigate interface that's worth bookmarking. While it's not as extensive as Public-APIs either, it's still worth a look. You can add APIs to their directory, too.

API directories can be helpful in other ways, as well. Consider some other listings on the directories to learn what information needs to be included in your API description.

Incorporate SEO

One of the advantages of API discovery is removing unnecessary tasks and redundancy in your API workflow. We've already talked a great deal about the usefulness of API directories and discovery tools. However, many developers will be looking for your API using a good old-fashioned Google search.

If you work SEO keywords and phrases into your API discovery documentation, this will help to capture the search engine traffic. It could even add some SEO clout to your API, as the API directories can serve as inbound links to your other web assets.

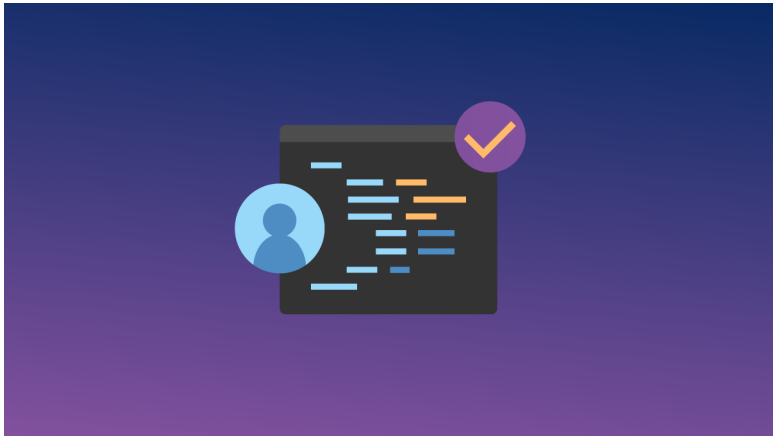
API Discovery: Final Thoughts

The API industry isn't going to become less competitive. As we've seen, 7,000 new APIs were launched in a single year. How many will the next year bring? What about the year after that?

Things like API specifications have been integral in the industry's rapid acceleration and adoption. And, API discovery will likely play a similar role. After all, it's truly good for all parties involved. Programmers can find APIs to work with, and API developers get a *much* wider audience. It's a win-win for all involved.

5 Ways to Make Your API More Self-Service

by **Art Anthony**



Making your tool more self-service can improve efficiency and reduce one-on-one support.

In 2019, TechCrunch wrote that APIs were "the next big SaaS wave." Years later, it's looking more and more like author Daniel Levine was right on the money.

Twilio and Stripe are still prominent examples of what you might call "API companies," but there are plenty of other examples of API-first companies out there: Square, Shopify, Algolia, Zapier, and others. In fact, in a recent Forbes article, Iddo Gino (founder and CEO of RapidAPI) asserts that "almost every company is now an API company."

This is exciting news for those in the space, but might also bring

trepidation for some — just because a company has been building APIs for a while doesn't mean that they know how to market them as products. One big part of designing an API as a product is to make it as self-service as possible. With that in mind, we put together some thoughts on doing just that.

1. Decrease Time To First Call (TTFC)

Time To First Call is a vital API metric to track, and there are various measures you can take to improve it. Many of these tactics overlap with other suggestions in this article, such as providing sandboxes and using external tools.

With a traditional product unrelated to APIs, generating revenue is all about moving people through the sales funnel: someone visits the website, the product piques their interest, they sign up for a free trial or make a purchase.

The specifics might differ for API-based products, but the process is similar. Remove as many barriers to entry as possible to make it easy for potential users to make their first API call. In other words, lower the TTFC.

2. Provide A Sandbox For Testing

In a nutshell, an API sandbox is a test environment that emulates a production API. They allow API consumers to test their integrations, just like a staging area in web development, reducing the risk of errors when they deploy to production for real. That also reduces strain on your actual API.

Sandboxes are extremely valuable for paid APIs because they offer potential customers the chance to try before they buy. This could dictate whether or not they take the plunge and purchase.

Sandboxes also help developers learn how to use the service — this is great for folks who prefer to learn through practice. For these reasons, you should keep things as close to the real thing as possible; a sluggish sandbox, for example, might deter some folks.

Maintaining a sandbox indeed has costs associated with it, although you can grab a free trial from a service like Sandbox. However, those costs could ultimately be an investment in your product. For example, sandbox logs might help you identify pain points or potential areas for growth.

3. Instant Account/API Key Issuance

In some cases, it isn't possible to allow users to create an account themselves. That might be due to sensitive data, or it might be that you need to figure out the volume of data involved with that customer. But having to wait for manual approval will inherently make your API less self-service.

If you can, we recommend making the whole process as open as possible: allow people to generate an API key, let them use OAuth2 for authorization, and let them dive in headfirst. In other words, try to get out of their way.

Of course, there will be things you need to think about with a paid or freemium product. For example, you'll need to look into rate limiting if you plan to offer multiple pricing tiers. In addition to warning users when they're approaching their upper limit, consider how you can transition them as they exceed that number of calls with minimal disruption.

4. Engage With Marketplaces

Although it won't be the correct route for every product, it's worth looking into marketplaces that can expose your offering(s) to a wider audience. There are, for example, developers who make a living selling apps on the Shopify App Store using APIs.

As of late 2021, RapidAPI currently plays host to more than 30,000 APIs. That's pretty close to the number of APIs, just under 25,000, in ProgrammableWeb's API directory. Their API acceptance rate isn't published, but these numbers are high enough that it certainly doesn't seem like they're trying to find reasons to turn submissions down.

It's possible to submit an API to RapidAPI manually and, if you're using a specification — more on that below — the process is fairly simple: you can specify it using UI, OpenAPI, Postman Collection, GraphQL Schema, or Kafka as appropriate.

You can add an API to ProgrammableWeb, too, although it's worth noting that ProgrammableWeb doesn't facilitate connections in the same way that RapidAPI does.

5. Comprehensive, Easily Navigable Documentation

Lastly, easy-to-understand documentation is a must for a truly self-service API. The lines between an API specification and documentation are blurry, and something we've written about previously.

However, it's generally the case that using something like an OpenAPI Specification goes hand in hand with rigorous documentation. You can even use Swagger UI to automatically generate visual documentation from a spec.

Beyond that, looking at examples of excellent API documentation is a good way to figure out what to include. Just remember, in many cases, your documentation is the first impression you make on potential customers. So make it a good one!

Self-Service APIs: Final Thoughts

The strategies listed above aren't rocket science. In fact, they're things that many API-first companies are already doing. However, it might not immediately be readily apparent to newcomers why these tactics are necessary. Hopefully, if you've read this far, that picture will be clearer now.

Marketing an API-centric SaaS has a lot in common with marketing other SaaS products, or any other product for that matter, but there are nuances associated with it. Considering these nuances, and increasing your service's usability, is the best way to maximize the chances your API has of being successful. Making your API more self-service is also a surefire way to decrease one-on-one customer support, thus reducing cost!

Best Practices For Creating Useful API Documentation

by **J Simpson**

Quality documentation requires forethought to appease developer consumers.

API documentation is the key to a useful, usable API. An API could be all-powerful, versatile, and entirely useful, yet it won't matter one bit if users can't figure out how to make it work. Creating proper API documentation is an artform in-and-of-itself. However, not all coders are good writers, just like not every author is an excellent programmer. Creating useful, informative, understandable API documentation is a skill set all its own.

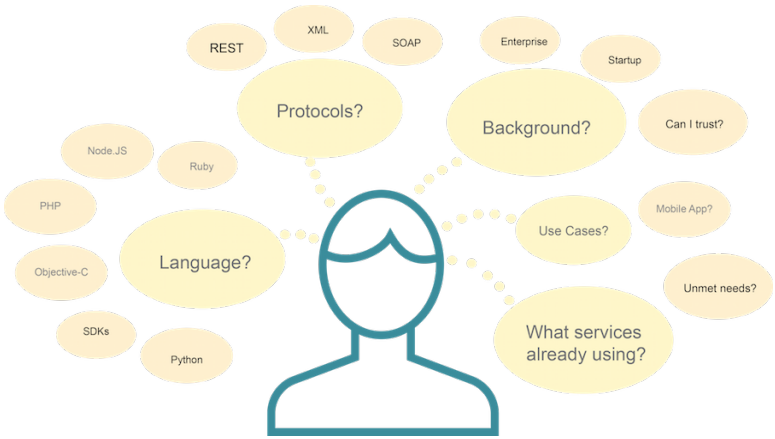
To help make your APIs more useful, we're going to share some best practices for API documentation. We'll start by examining

the essential components that every API documentation needs to contain. Then we'll take a look at some API documentation best practices.

Understanding The Audience For API Documentation

Writers are frequently told to "know your audience." Who you're writing for will influence your writing in a variety of ways, from tone to word choice. This is every bit as true for developers creating API documentation.

Very generally speaking, the consumers of API documentation fall into two main categories. There are the decision-makers who decide which APIs to use. Then there are the programmers who will be using the APIs. This means that API documentation needs to fulfill two main functions, simultaneously.



When crafting API documentation, consider: who is the target developer consumer?

You first must illustrate the usefulness of an API to convince the

decision-makers. Think about the spec sheet for a product as an example.

The developers who will be using your API are the primary audience you should keep in mind, however. They'll need detailed, thorough guidelines. This means explaining all of the functions with code samples.

Now that you understand your audience, you should be starting to get an idea of a basic template for creating useful API documentation. Keeping API documentation best practices in mind as you're developing your API will help keep you in the habit of making notes as you go, helping to compile your API documentation once you're finished coding.

The Essential Components Of API Documentation

While every API is different, most have some basic building blocks that nearly every API incorporates. Your API documentation should include documentation for these functions, as they're some of the first things developers will look for when they start to use your API.

Authentication

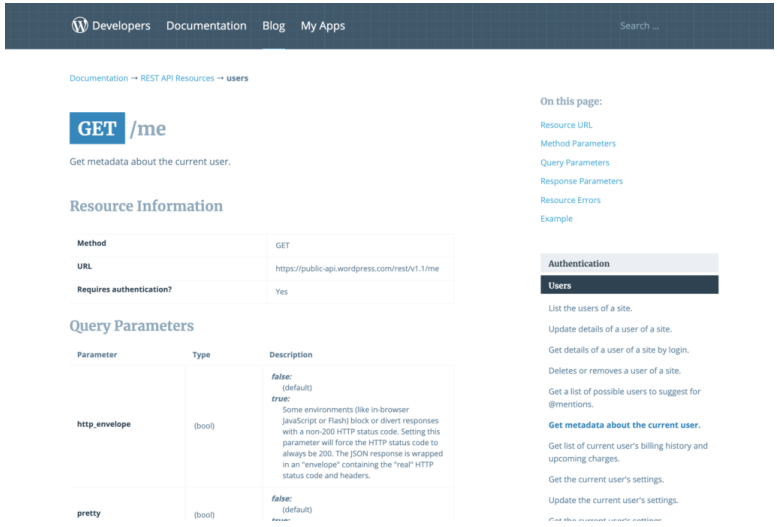
Authentication is one of the first things a user encounters when using an API. Think of authentication as a key that will unlock your API for your users. Almost every API features some sort of authentication schema, and nearly every one is different.

To start, your API documentation needs to let users know how to access your API. You can take a look at Auth0's authentication documentation for an example of what thorough, concise authentication documentation looks like.

API Resources

Users need to know what your API can *do*. List every endpoint with standard commands and responses. Listing all of the commands and responses helps you think like your end-users and create thorough, understandable documentation for each response. Think of it like diagramming your program and then creating documentation for each step.

For example, check out the API documentation for WordPress. WordPress offers a complete and thorough list of all available API endpoints. Each command has its own page, with comprehensive documentation of all of its context, queries, and error codes. Each page has example code, as well, to offer some guidance on how to get started with that particular feature.



WordPress organizes API documentation with a separate page for each function.

In all, this provides a detailed understanding of the API structure. As you can see, there are over 20 resources for retrieving or modifying user info alone. For complex API documentation like

this, organization and navigability are key.

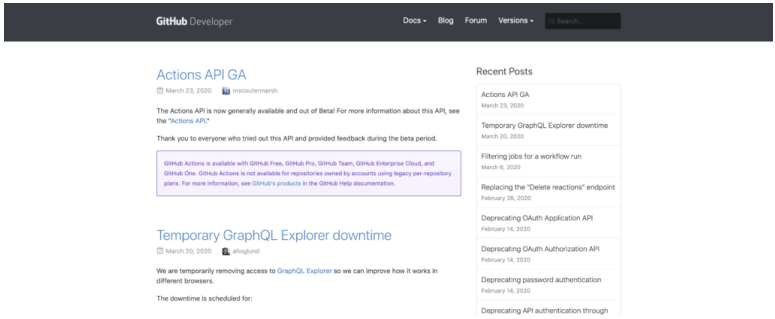
WordPress's API documentation is also an excellent example of usability. Each function features a tag letting you know what HTTP method the command requires, whether 'GET' or 'POST.' This saves you from having to read each page to have a basic understanding of how to use each API resource.

Also consider: 10+ Best Practices for Naming API Endpoints

Error Messages

Debugging is likely to be one of the main reasons people consult API documentation. You'll want to have a thorough section explaining all of the error messages your API returns. This will make your API more usable for your users, helping them to avoid frustration and have good feelings towards your API.

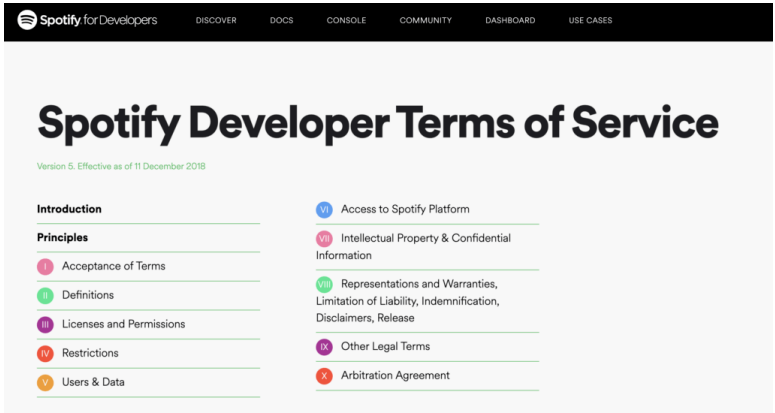
Don't limit your API documentation to listing the error messages, either. Include an example or two of how to fix common problems. Check out Mailchimp's API documentation for an example of thorough, useful error documentation.



From 400 Bad Request to 500 InternalServerError, Mailchimp's Error Glossary details global errors for the Mailchimp API.

Terms Of Use

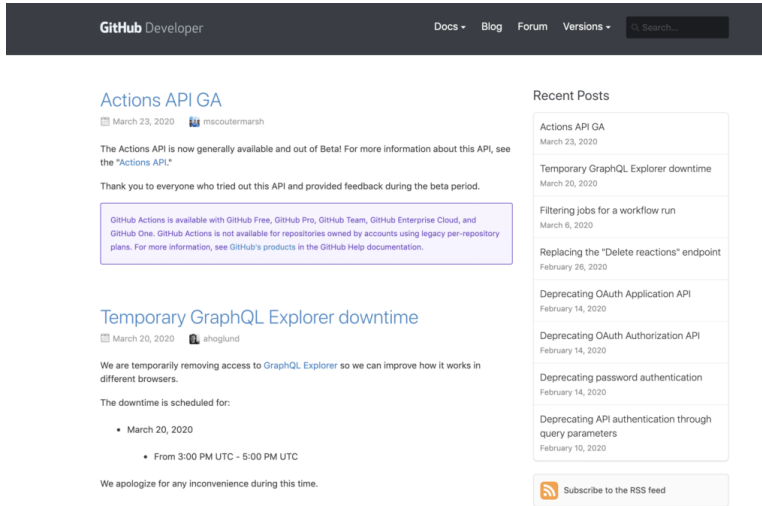
Terms of Use are the legal agreement between you and your users. In the Terms of Use section of your API documentation, you should include API limits, constraints, branding guidelines, and what usage is acceptable.



Check out Spotify's Terms of Service as a model.

Changelog

The Changelog section of your API documentation lets your users know how stable your API is. It also lets them know if anything's changed, in the instance that one of their calls stops working. You can take a look at GitHub's changelog for an example of thorough changelog documentation.



Github's Developer Changelog provides updates on general availability, deprecations, and downtime for various services.

Best Practices For API Documentation

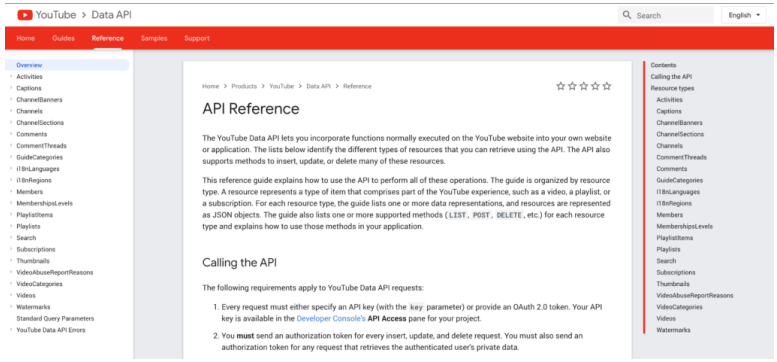
Now that we've taken a look at some of the essential components API documentation should have to be ultimately useful, let's consider some best practices to make your documentation really shine for developers and decision-makers alike.

Avoid Jargon

Remember, you have very little control over who's going to be consuming your API. There will be users of all different experience levels using your API and reading your API documentation. You want advanced and inexperienced users alike to be able to find both your API and its documentation useful and welcoming.

Excessive jargon is a pitfall that developers may fall into, and it has many drawbacks. First, financial decision-makers often aren't that technically savvy. If you're trying to convince them to invest in your API, plain language will go much further than advanced technical jargon.

Secondly and just as importantly, you want your API to be useful for as many programmers, of all skill levels, as possible. If you *have* to use technical jargon, you should include a link to a glossary, definition, or tutorial explaining that concept in your API documentation.



Sometimes being wordy helps! Take a look at YouTube's API documentation for an example of thorough, useful documentation written in plain language.

Thoroughly Document ALL Requests and Responses

There's no such thing as too much information in API documentation. Users aren't likely to read the whole thing in one sitting anyway. When a user is just starting out with your API, they'll likely need a bit of handholding until they've integrated it into their workflow.

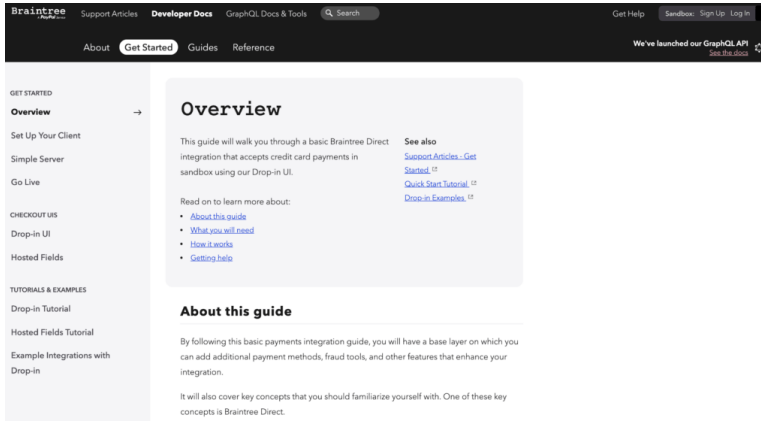
With that in mind, you should include documentation of every call your API can receive and provide some context for both the parameters and responses. Also document the responses as they'll let your users know things are working as they should. Document every potential error message, as well. This is all towards the goal of letting your users see *exactly* what will be returned from an API request. This will spare them the trouble of having to turn to Google to troubleshoot if something goes wrong.

Include Additional Resources

If something is outside of the scope of your API documentation, you should include links to the necessary information for your users. Again, you don't want your users to have to seek answers via a search engine, which can be frustrating and leave them with a negative association towards your API.

Include A Getting Started Guide

You want users to be able to get up and running with your API as quickly as possible, so they can see how useful it is. A quick guide on how to get started using your API is the easiest way to make this happen.



You can take a look at Braintree's Getting Started documentation for an example of an excellent Getting Started guide.

Include Sample Code

The fastest and easiest way to get a new user up and running with your API is to include some sample code in your documentation. The user simply needs to replace the API key in the sample code with their own key and they're off and running.

Sample code also gives developers a chance to see finished code implementing your API that they can reverse engineer and pattern their own programs after. You might consider having documentation for each individual section and then have sample code at the end showing all of the functions at work.

Best Practices For API Documentation: Final Thoughts

Good API documentation is the foundation of quality developer experience. It's what separates your API from being usable and

useful from being frustrating and inessential. What good is your API if no one knows how to use it? How should users know to invest time, energy, or resources let alone money if you're releasing a commercial API if they don't know what your API *does* or how it can benefit their business.

Having good API documentation often means the difference between a recommendation or a negative review, as well. You want users to be glowing about your API, singing your praises to their network and community. Good API documentation is one of the most essential ways you can make that happen, offering useful information, clear instructions, and easy-to-follow examples.

7 Best Practices for API Sandboxes

by **Thomas Bush**

Follow these tips to improve the developer experience of your API sandboxes.

Providing a dedicated testing environment for your API is a surefire way to improve the developer experience and encourage signups. In fact, many of the world's biggest API providers — from PayPal to Salesforce — already do so in the form of an API sandbox.

In this article, we'll look at seven best practices to get the most out of your API sandbox. But first, what exactly is an API sandbox, and what are the benefits of having one?

What Is an API Sandbox?

An API sandbox is a service that emulates the behavior of a production API. While there are no hard-and-fast definitions, we think of sandboxes as differing from mock APIs in being targeted primarily towards external developers. They enable risk- and cost-free testing of an API, making them a crucial part of a DX-oriented API strategy.

Benefits

API sandboxes have numerous benefits, both for developers and API owners. Developers benefit from being able to continuously and aggressively test new or updated integrations, without worrying about accruing a sizable bill (in the case of a paid API) or having their requests blocked. Also, sandboxes can be usually made available to any registered developers, which makes them the perfect way for prospective developer customers to test out an API before committing to a paid plan.

Both of these factors lead to an improved developer experience — in and after the onboarding process — which has a definitive upside for API owners. Also, the ability to "try before you buy" helps to grow both registrations and paid subscriptions. Last but not least, running an API sandbox will reduce strain on the all-important production API.

Examples

- PayPal API Sandbox
- Uber API Sandbox
- eBay API Sandbox

Seven Short 'n' Sweet Suggestions for Sandbox Success

If you're looking to launch or optimize your API sandbox, there are quite a few best practices that can maximize your results. In particular, we've identified seven suggestions you ought to adhere to:

1. Isolate Your Sandbox

Ensure that your API sandbox is isolated from the rest of your platform. A sandbox should allow developers to *simulate* the behavior of your API; it shouldn't, however, enable direct interaction with your platform in the same way a production API would. If you're not careful with your sandbox, it might affect production systems, expose real data, or contribute to billing confusion, so it's best to build your sandbox from the ground-up in an isolated environment.

2. Provide Free Access

Allow developers to access your sandbox free of charge. After all, the main appeal of testing against — or testing *out* - a sandbox is that it's free! This is especially true for prospective developer customers who may otherwise need to go through a long-winded approval procedure to get the corporate buy-in for even your most affordable trials.

Yes, there are costs associated with hosting an API sandbox… but chalk it up to being a crucial part of customer acquisition and DX. If you can't justify the costs of an unlimited sandbox, consider giving developers a limited number of free sandbox credits upon registration, which will significantly reduce throughput.

3. Recreate Production Behavior

Endeavor to make your sandbox as close as possible to the real thing. For developers, there's significant value in knowing that what they test with the sandbox will behave identically with the production API. Not knowing this, developers will be forced to run their integrations through the second set of tests with the real API, somewhat defeating the purpose of a sandbox.

If your production API supports `POST` requests, then support `POST` requests in the sandbox. If your production API implements pagination, then implement pagination in the sandbox. There's a good chance you'll base your sandbox on an API specification, so pay close attention to the areas your specification might fall short.

4. Remember Authorization

In particular, don't forget about the authorization or authentication methods used by your production API! This aspect of API behavior - which is also frequently excluded from API specifications - deserves special recognition, since it's such a common developer pain point. Of course, it should absolutely be accounted for in your sandbox, whether you rely on API keys or access tokens.

5. Account for Gateways or Proxies

When building your sandbox, consider the implications of any gateways or proxies that stand in front of your production API. Sure, they might not be part of the API itself, but they can still very much affect developers' integrations. Perhaps the best example of this is rate limiting: often implemented at the gateway level, it can significantly affect how integrations behave.

It's up to you to decide if and how to account for these issues. With rate limiting, some API owners limit sandboxes exactly as

they do with production APIs, while others - like Salesforce - vastly increase sandbox limits to enable more extensive testing. The author personally favors the approach taken by Evernote: limits for production and sandbox APIs kick in at the same time (after a certain number of calls on the hour), but sandbox users are only rate-limited for 15 seconds, and not for the remainder of the hour.

This way, developers can test and handle the rate limit exception without having to wait until the one-hour interval ends to resume interacting with the API.

6. Review Sandbox Usage

If resources permit, take the time to review how your API sandbox is used periodically. For one, looking at the sandbox's logs may help you to identify unexpected use patterns that you can go on to support. More importantly, looking at any frequently occurring errors can highlight common pain points, especially in the onboarding experience, which will improve retention when fixed.

7. Consider A Chaos Mock

Last but not least, mature API programs in high-stake industries should consider building a chaos mock alongside their everyday API sandbox. Coined by Microsoft architect Gareth Jones, the chaos mock is an API virtualization that purposefully embodies variability. The goal of a chaos mock is to enable developers to code against all sorts of weird and wonderful API behavior, so they can be confident their integrations will survive under all circumstances.

API Sandboxes: Final Thoughts

We've just reviewed seven ways to get the most out of your API sandbox. Follow these pointers, and you'll end up with an environment that enables accurate *and* practical testing of your API, among other benefits. Now, how else can you improve the developer experience for your API?

What Does a Bad Developer Experience Look Like?

by **J Simpson**

We asked our community what constitues a poor developer experience. Here are anti-patterns to avoid.

In many ways, being a developer is just like any other artisan — when it's time to use our tools, we need them to just work. You don't want to spend an hour troubleshooting your hammers and saws and chisels, ruining any momentum you've accumulated while losing valuable daylight. In fact, it's probably even more true for developers since we wear so many hats. Not only do we need to write code, build databases, and coordinate with collaborators, we often have to handle support, marketing, and promotion for the tech we create.

When working with APIs or Software-as-a-Service, developers want them to be *as frictionless as possible*. They need to function as they're supposed to without requiring too much maintenance effort. This allows programmers to stay in the flow and focus on remaining productive, building killer products and services, and delivering world-class customer service for their end customers and clients.

In short - if you want developers to use your software, you need to provide an exceptional developer experience (DX). So, how do we craft a developer-friendly platform? Well, to understand a complex subject, sometimes it helps to realize how NOT to do things…

8 Developer Experience Anti-Patterns to Avoid

What does a negative developer experience look like? We recently posed this question on Twitter and LinkedIn. The results were eye-opening, highlighting some common problems that may cause a developer to be wary of the service or abandon it altogether.

All developers stand to benefit by learning from others' mistakes. And these problems aren't only relevant to public APIs, either. They could arise within external, partner, or internal facing platforms. So without further ado, here are eight examples of negative developer experiences to watch out for, to help you build the best developer-centric products imaginable.

1. Lack of OpenAPI Specifications

Standardization has been one of the most important aspects in bringing web APIs into the mainstream. Much like the role of standardized parts during the Industrial Revolution, API specifi-

cations make communication and collaboration between different applications easy and instantaneous.

That's one of the main reasons that the OpenAPI Specification is so revolutionary — it creates a uniform format that is easily understood by computers and developers alike. This is why it's so frustrating when an API just *disregards* OpenAPI.

On our Twitter thread, developer Jordan Walsh talked about his frustration of having to page through API documentation when simply using the OpenAPI Specification would make this a non-issue.

2. Documentation in Non-Standardized Formats

Already, we're beginning to see a pattern emerging. Developers really want API documentation to be searchable, understandable, and easily integrated into any format they desire. Developer Sidney Maestre expressed his frustration with API documentation being published as a PDF.

At best, this requires having the documentation open as a separate tab, requiring you to tab back and forth, which is a total momentum killer. At worst, this runs the risk of the file getting lost or duplicated, taking up unnecessary room, and causing clutter on your system.

3. Lack Of Examples

Theo Gough, Chief Architect, Aidence, offered a host of excellent examples of developer frustrations, from lack of meaningful descriptions to formatting restrictions. A number of his complaints had one thing in common, though: a lack of real-world examples of an API in action.

Adding even a rudimentary example of how to use an API eliminates so much confusion. It makes things much less abstract and gives users something to model their development after.

4. Manual Integration

One of the main benefits of using APIs is automation. But manually integrating APIs defeats much of their purpose, which is a big reason why standards like OpenAPI are such a big deal. Having to figure out how to configure APIs and make them work together is a major pain and can be a stumbling block toward adopting an API, as noted by Twitter user @StefanTMD.

5. Lack Of Consistency

When you're using an API, you *really* want to know what you're going to get — surprises are only fun on birthdays and holidays. Specifically, API developers want to know that the service will work *every time* in the way they expect.

API strategist Hirok Choudhury talked about this in response to our LinkedIn post. According to Choudhury, poor developer experience often arises when there's a "lack of predictability, consistency and simplicity while trying to discover and eventually subscribe to an API." Frequent revisions are another cause for concern for Choudhury, which would also impact an API's consistency.

For API developers, the message seems clear — it's a good idea to get your API as stable as possible before making it available to the public. If you *do* need to make changes, use smart versioning practices that prevent disruptions to your existing users.

6. Questionable Deployment Options

Speaking of versioning, new iterations of an API are supposed to fix problems, not cause new ones. Developer George Jeffcock spoke of his frustration when deployment options introduce more problems than the design and code. "Developers should not also be Operation/Network…bad developers experience," he adds.

If an API is configured for a particular environment, it should *solve problems* and *make things easier*. APIs can be configured for everythingfrom AWS to Oracle to Google Cloud. The rise in content-aware network devices and the proliferation of new architectures like containers and hybrid servers have made deploying APIs more variable, with developers needing to think like network operators. Luckily, new techniques have emerged to deal with these issues, like DevOps and an increase in Integration Platform as a Solution (IPaaS).

If you're creating deployment options for your APIs, make sure they work in the environment they're intended for. To do otherwise may result in a bad developer experience.

8. Restricted Access for Documentation

API documentation typically provides a helpful rundown of methods, parameters, and examples of API requests and responses in action. It's meant to be referenced by active users, but it also provides potential consumers valuable insight into whether the API is right for them or not.

Many of these intricacies are necessary to know *before* deciding to purchase, which makes asking for permission to access the reference documentation somewhat counter-intuitive for a self-service API, to say the least, as pointed out by Ben Virdee-Chapman on LinkedIn.

Bad Developer Experience: Final Thoughts

Many thanks to those who took time from their busy schedules to share their painful developer experiences. By responding to these anti-patterns, API providers have some better insight into how to craft a more streamlined experience.

Remember, APIs are supposed to make our lives easier. If users have to dig through documentation or work excessively hard to integrate an API with their existing stack, they're likely to look elsewhere.

Developers have little patience for unusable or inefficient API products. To return to our artisan metaphor, faulty APIs are like a hammer with a shaky head — you might grab it by accident before remembering it's wonky and tossing it back in the toolbox.

But what are some examples of negative developer experiences that you've encountered? Leave us a comment and let us know! Let's all pool our expertise to make the API community flourish and thrive!

Why Time To First Call Is A Vital API Metric

by **Art Anthony**

Monitor Time To First Call to ensure users can get started quickly with your service.

We've all been there: you're sorting through your smartphone and find an app that you don't even remember downloading. You open it up, and, sure enough, you never finished creating an account. With a couple of taps, you banish the app in question from your phone forever.

You might not know it but, if you're an API developer, the same thing is probably happening to your product right now; some people are signing up to your API and either never using it or letting weeks, perhaps months, pass before they actually try it out.

We can refer to the time between the signup process and making

an initial API call as Time To First Call, or TTFC. In this post, we'll be looking at why TTFC is such an important metric to track. We'll also see if there's anything you can do to improve it. (Spoiler alert: there is.)

A Deeper Dive Into TTFC

Above we suggested that Time To First Call refers to the time between signing up and making a call. In reality, tracking this metric isn't quite that simple. Is measuring the time until a GET request is sent sufficient, for example, or should we be looking for a more complex query?

It also doesn't seem right to include error messages, as any advantages of having a low TTFC are far outweighed by API consumers encountering errors on their first attempt(s) to call. And does the clock start ticking when someone signs up for an account, or would arriving at the documentation page of the developer portal be a better starting point?

We don't have hard and fast answers to these questions, but the following seems like a reasonable template for measuring Time To First Call to us:

Time to First Call (TTFC): The time taken between a developer accessing documentation, and/or signing up for an API key and making their first successful API call (of any complexity).

The good thing about this metric is that, indirectly or otherwise, it's probably one you already measure to some extent or another. Armed with the information above, you can ensure that you're doing so consistently across all your offerings.

From Time To First Call to Active User

So what is a healthy target TTFC? Unfortunately, there isn't a standard benchmark to shoot for because TTFCs can vary considerably. Measuring TTFC can depend on the complexity of the offering, the price (if any) of the service, whether or not the API or its developers are a known quantity, and how many competing products are on the market.

In the past, Twilio has referenced their efforts to enable API developers using their services to "get up and running in 5 minutes or less." In their comparison reviews, Ably uses the following benchmarks for the comparable metric of 'Time to "hello world"':

- 5/5 – < 30 mins
- 4/5 – 30 mins to an hour
- 3/5 – 1-2 hours
- 2/5 – 2-4 hours
- 1/5 – 4+ hours

You probably already have a rough idea in your head of how long a typical customer takes to place their first call. And it's always interesting to see how that number lines up with reality.

Bear in mind that making a first call is very different from someone becoming an active user. Reducing the former won't necessarily increase the latter, BUT it will likely increase your funnel's number of potential users. Or, at the very least, an optimized TTFC will speed up the rate at which they make their way into it.

Exploring "First Call Motivators"

There are all sorts of reasons why someone might sign up to use your API and make their first call. In an article for TechCrunch, Joyce Lin covers a few of these:

- Actively searching for a solution to a particular problem
- Heard about your product and are curious about it
- A project or role requires that they use your service

Their enthusiasm, willingness to forgive flaws, and motivation to make things work will vary massively depending on which of those camps people who find your API fall into.

Depending on their first impressions, your API might become something developers can hardly resist trying on their lunch break. Or, it could be something that ends up at the very bottom of their to-do list. Regardless of how they find you, the secret sauce is to do the following as quickly as possible:

Demonstrate what your product does, how it does it, and why it's the best at doing it.

Of course, all that's easier said than done…

Improving Your Time To First Call

Reducing your average TTFC isn't like, say, improving the load time of a web page. There's no silver bullet for improving the metric, but there are different things you can do that will likely help to improve the Time To First Call of a particular product:

Extensive Documentation

At the risk of becoming a broken record, we're back at it again, talking about the importance of great API documentation. The

key to this is understanding your audience well enough to tell them what they need to know straight away, and highlighting that information appropriately, while covering what they might want to know at a later date elsewhere.

Demonstrate Use Cases and Examples

Maybe this could fit in the documentation section above, but we think it's important enough to deserve its own paragraph: providing in-depth examples of what your API can do will help to capture some of those who are on the fence about whether or not it's the right solution to their problem.

Assessing Roadblocks in the Process

The best example of a roadblock to TTFC is requiring manual approval before an API key is issued. It may not be a dealbreaker if you really need something like that in place, as long as approval is timely. If it's not, you risk both losing out on potential users who were "on the fence" and upping your TTFC.

Encourage Making Calls Early

This (probably) isn't as simple as directly telling people, "hey, time to start calling!" However, it might help to provide sandbox environments and highlight how the API reacts to what different users will do with it. Then, they'll know exactly what to expect when they call it.

Consider the Role of External Tools

Postman public workspaces, first introduced in 2020, have been described as the first "massively multiplayer API experience." The

Joyce Lin article linked above mentions two case studies that cover Vonage and Symbl.ai's efforts to let users explore their APIs before making calls.

Writing for Nordic APIs, Derric Gilling previously called Time To First Hello World a north star metric for measuring developer relations. Acting on the points above *should* reduce TTFC, though it may not do so immediately, and that's extremely valuable.

A lower TTFC is generally associated with a better developer experience, which usually equates to better user retention, more customers (and revenue), and a more robust API offering as a whole.

Time To First Call: Final Thoughts

In the past, we've written about the rise of low-code/no-code tools and the impact that increased API control might have on future adoption in the space. We've also touched on the part external tools can play in lowering TTFC above.

The process of people hooking up with your API via a third-party service, rather than going directly through your developer portal, might look different than you're used to. Still, it could be valuable when it comes to lowering barriers associated with your API.

Although the future of APIs likely isn't (just) making calls to services found in extensive directories via third-party services like IFTTT or Zapier, it will almost certainly be a part of it. It's worth thinking about that eventuality sooner rather than later.

In the meantime, however, there's more than enough to get to work on when it comes to lowering your TTFC and, ultimately, improving the extent to which people engage with your API.

Developer Marketing for API Companies

by **Adam DuVander**

Marketing your API to be developer-focused will naturally affect its developer experience.

Nordic APIs has long tracked the API-as-a-product trend, where companies expose their products primarily through a developer interface. The companies that treat their APIs as an external product need to reach potential customers, but technical audiences can be difficult to attract. They're often averse to traditional promotion techniques. So, how do you market to developers?

If you want developers to discover, use, and eventually pay for your API product, you need to change your approach. In my book, Developer Marketing Does Not Exist, I share the philosophy of "education not promotion" to reach a developer audience. In this post, I'll give you the highlights. But first, let's make sure your API

product is developer-focused.

Are You Developer-Focused or Developer-Enabled?

How you market your API will depend on its intended audience. Many companies trip up on this step because they assume their users are exclusively developers. While that's a natural expectation, it's more likely you'll be courting multiple audiences, with developers the less immediate target.

For example, let's say you offer customer relationship management (CRM) software with an API. Your customers require a developer to build integrations and automations into their own systems by writing code using your API. Most advanced use cases will need code to make API calls or respond to webhooks. These facts lead you, the CRM company, to aim your marketing at developers. The problem is that most developers are not researching the right CRM API for their use case. In fact, it's most likely to be another department —such as sales — that picks out a CRM. If developers are consulted at all, it's to confirm they'll be able to implement another team's API integration use cases.

CRM APIs are likely to be developer-enabled, at the far left of this continuum:

At the opposite end are developer-focused APIs. Here you'll expect to be solving a genuine developer problem. The API may replace code a dev might otherwise create and maintain. Or, perhaps

it connects to cloud infrastructure. There is utility for a single developer in developer-focused APIs.

It's too simplistic to pin APIs as either developer-enabled or developer-focused. The reality is likely somewhere along the developer marketing continuum. Chances are good that you know if you're on one of the extremes. The risk for the CRM company — and any product with non-developer users — is to assume you need to market your API like a developer-focused company.

To enable developers to use your API, there's a minimum bar for developer experience, documentation, and support. As you move up the continuum from developer-enabled to developer-focused, API marketing activities increase. As you'll see in the coming sections, those activities look a lot different than traditional marketing.

Find the Real Competitor to Your API

Just as many companies mistake their audience, they also mistake their largest competitor. As Nordic APIs pointed out in its API-as-Product eBook, developers may attempt to replace your product's functionality with their own code.

While some products are harder than others to duplicate, this competition is actually something you can use to your advantage. Developers love building. Their job is to solve problems with code and efficient solutions can be addicting. It's no wonder that when faced with the problems your API solves, many developers will first think about how they would architect it themselves.

Try this with any developer, whether a potential customer or an internal engineer: describe a problem that could be solved with code. They may have a few initial clarifying questions. Quickly, it will be clear how their brains are wired for problem-solving. You can almost see the pieces swirling in their mind, like Sherlock Holmes as he deduces how a crime occurred.

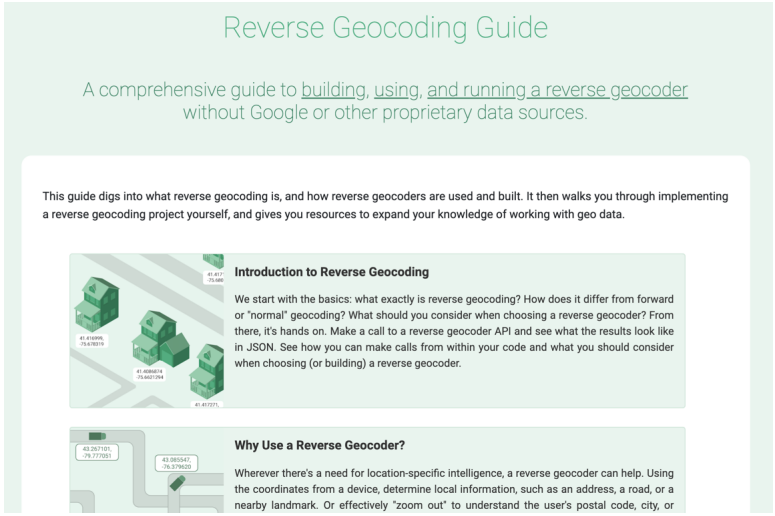
It's possible a competitor arises as a solution during this ideation process. But that's only likely if the other company has already shown that they understand the problem space. More likely, the developer's thoughts go to common platforms and frameworks, open-source tools, and the other elements they are already using to do their work.

The conundrum becomes: how do you compete with a developer's instinct to use their current tools, many of which are practically or actually free? The answer: you don't.

Help Developers Solve Their Problems

Rather than fight against this "build it myself" tendency, follow its momentum. Your marketing can speak to the ideas already swirling in developer minds. Take them down the path to build, run, and maintain a service like yours. You aren't promoting your API with this marketing, at least not directly. Instead, you're demonstrating your authority as a company that knows how to solve these problems.

I call this approach the Developer Content Mind Trick. It works in all types of content, but especially shines as a deep, foundational guide.



For example, OpenCage Data provides an API for reverse geocoding. Given a location described by latitude and longitude coordinates, their service returns a human-readable description of the place. Many developers appreciate that OpenCage is built on open data. They might also be tempted to piece together their own reverse geocoder using a combination of open-source software and community-generated data. A traditional marketing approach could have been to detail the features and benefits of OpenCage for these developers but hide behind-the-scenes details from view.

Instead, the company published a reverse geocoding guide to help developers architect and build a reverse geocoder. It includes contextual sections on use cases, why this type of geocoding is important, and how open data helps. Throughout, developers will get a sense of what it takes to create a solution (a lot!) and how much the OpenCage team knows about reverse geocoding (a whole lot!). Some developers will read this guide (disclosure: written in collaboration with the EveryDeveloper team) and go on to build a reverse geocoder — they can even follow a tutorial to get started. But, most will be convinced it's harder than it seems and will use OpenCage instead.

Keep in mind, this type of content is not documentation. Your docs are essential, and they solve their share of developer problems. However, few developers will discover you through documentation, which is by definition focused on your product.

Your API marketing should focus not on your product, but on the business and technical problems your product solves.

Developer Marketing: Final Thoughts

Not every part of your API marketing can be 10,000-word educational guides. Fortunately, you can use the same educational approach to reach developers in shorter formats like blog articles, webinars, conference talks, and social media posts. In fact, my book includes chapters on open source, events, and tools, in addition to more traditional content.

What you want to avoid in your API marketing is switching into promotional mode. Remind your content creators about the effectiveness of education over promotion when it comes to developer audiences. Aim for consistency by sharing an understanding of your developer audience across your organization.

Remember the problems you solve and the way a developer will lean toward building the solution. Follow that momentum and show developers how you've thought through the solution they want to implement. They'll appreciate the education while gaining respect for your authority in the problem space. The end result is that more developers will opt to use your product.

Why Your API Needs a Dedicated Developer Experience Team

by **James Messinger**

If possible, consider assigning team members to actively support your project's developer experience.

In the 2019 State of API report, surprisingly, only 37% of API providers viewed documentation as a top priority. When API consumers were asked to vote on the most important characteristic of an API, 60% earmarked "ease-of-use" as their primary desire when integrating, with documentation trailing in 3rd place. While documentation can contribute to overall ease of use, these numbers reveal that it is not the only element that plays into a good developer experience.

So, the question is: How can API companies improve their overall

process and deliver the high-quality experience their users want? One of the best answers to that question is: Focus on creating a dedicated developer experience team that can empower your users by making it easier to understand, easier to build, and easier to integrate (particularly if your company develops customer-facing APIs).

Understanding the Difference: DevRel and DX

What exactly is Developer Experience (DX)? DX is all about understanding developers, their needs, their abilities, their values, what they're trying to accomplish, what tools and technologies they're using, the integration points, and how they *feel *while using a product.

Developer relations (DevRel) is a vital component of a comprehensive DX strategy. Some companies are large enough to have a dedicated DevRel team, perhaps even with multiple distinct roles, including evangelists, advocates, and sometimes even tech writers and growth hackers. These roles are all aimed at inspiring positive relationships with developer users, through sharing knowledge that fills the gap between the creators and consumers of tech.

Whether your company has a dedicated DevRel team or a DX team that includes DevRel responsibilities, it would be remiss not to acknowledge the role developer relations plays under the larger umbrella of developer experience.

Why Developer Experience?

Software companies and SaaS providers that sell user interface (UI) products have recognized the importance of good user experience

(UX) for decades. A great UX can be the key differentiator that makes your product successful. It's how Apple won the cell phone market and how Nest made thermostats sexy.

DX is to APIs as UX is to UIs. APIs are products, and developers use those products. Those developers have come to expect a high level of quality, ease-of-use, onboarding, and support thanks to companies like Stripe, Nexmo, and HelloSign, who are continually raising the bar.

> "DX is the acquisition of knowledge needed to implement an API. Make the acquisition easier; knowledge more digestible; the journey of implementing it simpler; lives of developers better" — Anthony Tran, creator of the Luna design system

Why the Shift?

So why are companies suddenly starting to realize that they need a DX team? After all, they've scraped by without DX for decades. What changed?

In the early 90s and into the 21st century, businesses typically invested millions of dollars in on-premise software packages such as CRMs, ERPs, and databases. They then relied on an army of expensive contractors to customize these products to meet their needs. Integrating in decades past was a big undertaking — in terms of skill, labor, and capital.

However, as companies have moved into the cloud, they've shifted away from monolithic software platforms and toward smaller, micropayment-based SaaS products. This propagation of SaaS products has led to the need for standardized integrations between them, which in turn has fueled the rise of the API economy. By 2011, REST was an industry standard, and in just under a decade, we've seen nearly a 1,000% increase in the number of APIs on the market.

This Cambrian explosion has virtually eliminated the need for expensive consultants who understand the intricacies of million-dollar software packages. It's now possible for any developer to integrate with these APIs to link disparate systems and automate their companies' workflows.

The API economy has created a culture of expectation for APIs. It's now assumed that an API will be readily available, and in many cases unmetered, for consumers or developers looking to "connect" to your application. In fact, for many customers, your API is more important than your UI. Whether you have an API and how easy that API is to use may be the differentiators that make the customer choose your product over your competitor's.

So, how do you ensure they get the best, most well-rounded API?

You Need a Dedicated Developer Experience Team

Developers aren't the best at incorporating UI into their designs. That's why many companies employ UI specialists who are responsible for putting in a friendly interface on top of the components a dev team has built.

The same holds for APIs. Your dev team shouldn't be solely responsible for an API's developer experience, because that's not the dev team's specialty.

During ShipEngine's early days, one of the defining moments for our development team was recognizing that to increase adoption we had to provide more focus on creating a product that developers loved* enough* to justify building out a new integration. We weren't the first shipping API on the market, and we won't be the last, so we knew we needed a way to stand out.

We started to look at how API companies in other industries navigated around their competition.

Take popular payment processing platforms like PayPal and Stripe, for example. Many may recognize PayPal as one of the leaders in the industry — and, as one of the first on the market, they deserve a seat at the table. But, historically, their API has been clunky and awkward to use. When Stripe was first introduced, they knew they were offering a product that developers would love, but also knew gaining a loyal following would require a lot of legwork.

How were they able to do it?

By building an API with a good DX.

They designed a killer API with an emphasis on consistency and quality standards, wrote user-friendly documentation, provided useful code samples and powerful SDK libraries, wireframed their website to prioritize developers' needs, *and* they employed a great developer relations team that attended conferences and wrote knowledge-based articles. They hacked their way into a tight market by creating a product that developers loved, and experience they would want to share with others.

> "Happy developers are chatty developers, and when we talk to each other to recommend products, the ones with the best DX are at the top of the list." — Sam Jarman, DX speaker and writer

Stripe capitalized on their ease-of-use, knowing they could lean on developers to sell their product for them so long as they could *show* them how pleasant the integration experience *could* be. Their success was even enough to make PayPal jealous.

So, what's the takeaway?

Stripe is not the only company to quickly take over market share through improved developer experience. So how were they able to

corner the market in such a short amount of time? I believe it was by employing a diverse multi-disciplinary team dedicated to the four primary elements of developer experience.

4 Main Responsibilities of a Developer Experience Team

At ShipEngine, our Developer Experience team exists to ensure an exceptional experience for the developers and customers using our API products. By focusing on these four primary areas of responsibility, we've been able to design better features, champion the interests of developers, translate feedback, and advise other internal departments on how to better empathize with and design for developers.

The four primary areas are:

1. API Design

While product development is largely left up to engineers and product teams, a Developer Experience team should maintain responsibility for providing the guidance and standards engineers need to create a product that is received well by others. This includes every part of its interface, including the protocol, style, naming, models, operations, authentication, status codes, headers, errors, paging, sorting, querying, and more. It may also include some aspects of the behavior and implementation details of the API as well. Types of Deliverables:

- Design guidance
- Design review
- Style guides
- Specifications / definitions

2. Quality Assurance

With APIs, you always want to aim for quality through consistency! To ensure an API product and developer tooling meet a high standard of quality, the DX team must become responsible for employing automated tests, linters, and processes that verify compliance with designs, schemas, and style guides. Quality assurance also involves the propagation of a culture of quality throughout the design, product, and engineering teams. Types of Deliverables:

- Contract testing
- Specification testing
- Style guide compliance testing
- Verifying accuracy and clarity of docs and tooling

3. Developer Tooling

You want to give developers a chance to test out your API before investing in full integration. So, providing a robust library of developer tools is a great way to *show* not tell them how great you are. Developers want and need schemas, code samples, reference implementations, SDKs, and a variety of other resources to help guide them through the build-out. Types of Deliverables:

- Code samples
- Demos / reference implementations
- SDKs and libraries
- Specifications, definitions, and schemas
- Internal tooling and automation
- Integrations with developer tools and services

4. Developer Relations

And finally, the role we (and users) are most familiar with. All communications and interactions with developer customers, such

as documentation, training, release notes, community events, and user feedback studies fall underneath Developer Relations. Deliverables:

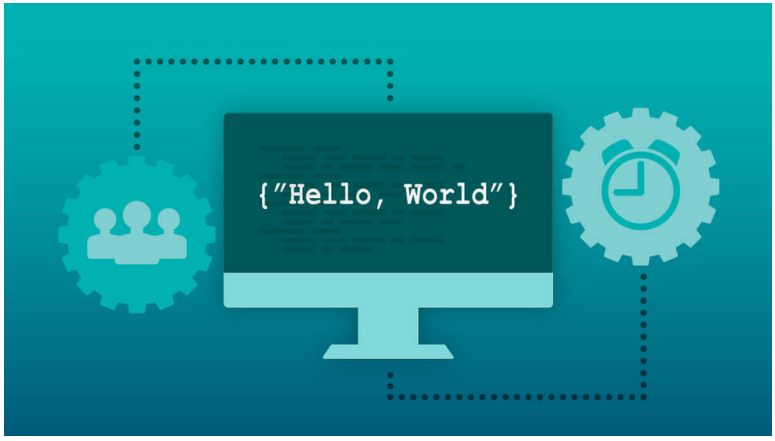
- Documentation / Tutorials / FAQs
- Release notes / changelogs
- System status info (downtime, bugs, performance)
- Media content (blogs, videos, etc.)
- Training and materials
- User research studies
- Events/community engagement (meetups, hackathons, conferences, etc.)

DX Team: Final Thoughts

Just as UI/UX has been a key differentiator for Graphical User Interface products, Developer Experience is a key differentiator for API products. And, a good DX strategy extends beyond the roles and responsibilities of DevRel.

Tips on Creating Outstanding Developer Experiences

by **Bill Doerrfeld**



Experts from our community share their top tips for improving developer experience.

Developer Experience (DX) for developers is akin to User Experience (UX) for end-users. The API products that streamline DX tend to increase interest and retain a following. But it's not only public-facing services; internal and partner APIs benefit from a focus on DX as well.

Developer portals can achieve quality DX by adopting discoverability, quicker onboarding, and accurate documentation. But good DX is ingrained into a well-designed API that follows best style practices too. Developer advocates are necessary to address issues

and collect feedback to refine feature sets and create tutorials. This is only a primer to the many facets of DX.

In this article, we dig into how to create outstanding developer experiences for our APIs. To that end, we interviewed three experts in the field: James Messinger, Developer Experience Director, ShipEngine, Derric Gilling, CEO, Moseif, and Srushtika Neelakantam, Developer Advocate, Ably Realtime, to discover what they are doing at their respective groups to enhance services for developer users.

DX For Onboarding

The hallmark of DX is meeting your user's needs, at whatever stage they are at. During initial use, these needs fluctuate based on the developer's knowledge. But a general rule of thumb is to get developers to "Hello World" as soon as possible.

According to James, a developer portal should adapt to the consumer at any stage of their journey, but should especially consider the initial onboarding experience.

"For brand-new users, the portal should provide a guided onboarding experience, with a ready-to-use sandbox environment, API keys, getting started guides, and step-by-step tutorials," said James.

Derric echoes this statement, encouraging API owners to "create an awesome onboarding flow" that makes it as easy as possible to start pinging the API. He recommends that API owners focus on the minimal number of steps required to deploy your solution. He encourages code snippets, copy/paste abilities, and "successfully received" messages for asynchronous processing.

Small DX enhancements can keep developers active and engaged. But it's also essential to consider the rate of information disclosure. If your full configuration is complex, involving multiple integration

points, these can be surfaced later using what Derric calls gradual onboarding.

Srushtika adds that well-structured documentation and video tutorials can highlight use cases. She emphasizes self-service capabilities like a sandbox for debugging can improve the onboarding process tremendously.

"Self-service developer tooling and onboarding guides are incredibly appreciated by the developer customers," says Srushtika.

DX For Developer Dashboards

But DX does not end with onboarding. As developers mature their applications, they require documentation checkups as well as a center for account information. This is where a developer account dashboard comes in.

James notes that customers who have already been converted require a portal showing metric usage, billing information, release notes, and upgrade guides. A dashboard for easy access to account information is simply necessary to sustain an API-as-a-product business.

To see a minimum viable developer dashboard setup, create an account with Dark Sky to view their API console. They have a simple developer dashboard with API consumption rates, billing information, account balance, a reset API key ability, and other ways to maintain your account.

"Seriously, though, sometimes you have to move mountains to get developers to fully adopt and use your developer platform," says Derric. He recommends the following for maintaining great developer portals that can support developers well after initial onboarding:

- Well organized, up-to-date documentation and working samples
- Framework specific SDKs
- One-click deployments
- Awesome support

DX For Developer Advocates

We discuss machine-machine connection so often; it can be easy to forget there are actual humans on both sides! Support engineers and developer advocates must have empathy for their users, a crucial human element to sustain relations and encourage use.

Derric recommends API owners put more effort into maintaining personal relationships with their user base:

"Create personal relationships with your developers. Learn from them and understand their use cases. Developers love talking about the projects they work on and can turn out to be your largest evangelists," says Derric.

Srushtika agrees that personal connections with developers can significantly benefit your program. She stresses that gathering developer feedback is paramount for early programs:

"Ask your users in a friendly way what you could do better with your docs, tutorials, events, community, etc., and what else they'd find useful," says Srushtika. "You'll definitely get a lot of pointers and a direction to go in."

Srushtika also recommends staying ahead of the curve when it comes to new frameworks and hot programming languages.

Derric encourages building a team of informed dedicated developer advocates that are more than liaisons to engineering.

"Developer support is one of the best ways to learn where there are obstacles getting started with your platform. For small companies, even the CEO should have their hand in support," says Derric.

DX For Developer Communities

Each developer community is unique in makeup. Developer advocates should seek to meet their user support needs *and* insert themselves where developers are active.

"API advocates should be wherever their customers are," says James. "Determining how to prioritize your advocacy efforts is a matter of observing your community's behavior and measuring different engagement methods for effectiveness."

Our panelists recommend utilizing community discussion forums like Stack Overflow, Discord, GitHub, Reddit, or Slack to respond to questions from developers. James also notes advocates should consider hackathons, meetups, or exhibiting and speaking at conferences.

However, for early-stage API programs, Derric notes that some community activities may put the cart before the horse. He recommends focusing on a "single user experience." This means building self-service tools, blog content, and prioritizing support for existing adopters.

"Don't launch a community forum if you don't already have an existing community," says Derric. "As your developer program grows, more effort should be invested in nurturing and empowering the growing community, especially finding key influencers and evangelists."

Nonetheless, the sheer amount of tasks involved in DX can be daunting. Srushtika notes that learning how to prioritize tasks and time management for such tasks can be difficult. She thus

stresses API providers to consider what activities are most value-generating.

DX Time and Task Management

"Dev Advocacy teams have a huge range of responsibilities which all seem\
equally important," says Srushtika. "In such cases, it really comes down to prioritizing the various activities based on the value they offer vs. the time and resources you'll need to spend on those activities."

For example, Srushtika notes how presenting a talk or organizing an event may require considerable effort and time, and not directly guarantee value. Whereas investment in technical support, which could take minimal effort and negligible time, could yield greater results, including insight into first-hand feedback from users.

In addition to considering the time and output value for developer relations activities, Srushtika adds that DevRel investment should match overall company goals:

> "Often prioritization of various DevRel activities is done based on the broader company goals for the quarter or year, but it is often a sweet spot between spreading awareness about the product, educating people on how to use it, and retaining existing customers by continually improving their DX."

KPIs to Improve DX

What sort of metrics should developer programs be considering? If you are taking all these points to heart, you will hopefully notice

more onboards, sign-ups, and more API key registrations – all the hallmarks of success. Or are they?

Derric notices that some KPIs used to measure developer experience are purely vanity metrics. Pageviews and sign-ups don't necessarily correlate to an active platform or end developer success. In his words, "these metrics only measure acquisition, and do not account for what happens after sign up."

Instead, Derric recommends API providers align all teams on two north star metrics which are more intimately tied to actual production use:

- **Weekly Active Tokens**: This number measures the distinct tokens accessing the API in a given week. This better reflects actual product usage, as opposed to sign-ups, which may not be real sustained users.
- **Time to First Hello World**: Time to First Hello World (TTFHW) is the time it takes for a developer to sign up, create a simple app or test, and make their first hello world (i.e., their first API).

James agrees with Derric that TTFHW is a reliable metric to consider. But still, you have a choice on what constitutes your API's Hello World. James recommends tracking an action that is more meaningful than merely a first successful API request. For example, At James's company ShipEngine, TTFHW is measured by creating your first shipping label.

James also encourages tracking trial conversion rates and recommends the tool Haxor for quantifying developer experience and measuring KPIs.

Srushtika also agrees that measuring DevRel is more complicated than sign-ups; correlating marketing efforts to revenue generation is tricky to achieve. "Developer programs impact the revenue

and sign-ups in very indirect ways which could be very hard or sometimes impossible to track or measure," she says.

Regardless, Srushtika recommends also searching for insights in the following areas:

- **Content**: Views of your developer-centric content, including tutorials or videos,
- **Forums**: Look for trends in engagement on support questions.
- **Talks**: Consider feedback from events and lectures, spikes in attendance, or views.
- **Hackathons**: Monitor the usability and hiccups during developer onboarding at live events.

"All of the DevRel activities feedback into the product, engineering, sales, and of course marketing, so DevRel has a powerful impact on various functions of the business," says Srushtika.

Building Outstanding DX: Final Thoughts

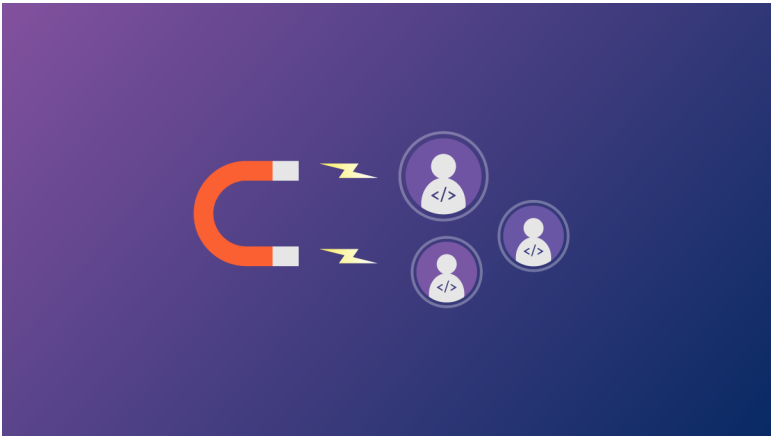
There we have it. In summary, here are some quick lessons we've learned on how to build quality developer experiences:

- **Create a quick and easy onboarding process**: As Derric says, "Build something that developers love and is easy to get started."
- **Build killer self-service tools and documentation**: Do not hide docs, have self-exploratory resources at hand. The more you invest in developer resources, the less one-one support is needed.

- **Prioritize developer experience investments carefully**. Prioritize developer support – this should come before massive marketing campaigns. Consider the stage you're at and build community tools appropriately.
- **Use the right data and feedback to improve**. Monitoring DX is tricky. Source feedback from your community, and consider tracking a stable metric tied to business value, like Weekly Active Tokens or TTFW.

How To Find An Audience For Your API?

by **Art Anthony**



Here are some tips to help refine the audience for your API.

So, you've built your API. Or maybe you're not quite there yet? Whatever stage in the process you're at, it's never too early to start thinking about who will actually use your API.

Of course, that's easier said than done. But it's an essential step of the process because, if you fail to do it, you won't know how to market it. A one size fits all approach may fail to resonate with your ideal users, and your API will struggle to grow as a result.

The key to avoiding this is finding an audience that falls in love with your product. Again, easier said than done. However, finding a market that can't get enough of your API and wants to talk about it to their fellow developers can make your life much easier.

Below we'll cover a few of the questions you should be asking to refine your audience. We'll outline some of the steps you need to be taking as you build your service to find an audience for your API.

What Does Your API Do?

This might sound like a rudimentary question, but it's one that you need to answer. Distilling what an API does into a single sentence is helpful because it prompts you to think about your positioning. Here's an example of how Twilio does this from their blog:

"Twilio helps organizations and brands of all sizes create meaningful moments with users across the globe — from the simplest text messages to life-saving communications."

It's true that "meaningful moments" is a little vague, but elsewhere in that post, Twilio talks about enabling developers "to build unique, personalized experiences for their customers." Already, a picture of Twilio's customer base is starting to form:

- Organizations who want an API for B2C communications
- Need for a high degree of customizability
- Focus on text as a medium (although Twilio covers channels like voice and video too)
- Requirement for high reliability and uptime, evident from the reference to sending "life-saving communications"

Armed with the information above, it becomes easier to think about how to market Twilio to potential developer-consumers with an elevator pitch. Or a TL;DR, as Twilio frames the above on their blog. And, as we all know, this is something Twilio has done incredibly successfully.

Carry Targeted Messaging Throughout

It's not enough just to think about why your product is useful to a specific audience. You need to make sure you're conveying that message across everything from landing pages to documentation. That means demonstrating the value you can add, including code samples and use cases where possible.

If you have competitors on the market, it's worth looking at how they perform. How are they appealing to *their* audiences? If you're trying to appeal to a slightly different crowd, think about how you can tweak the type of language used to do that. IFTTT, for example, has done an excellent job of positioning automation and complex API terminology in a way that's friendly to a layperson.

If you're marketing to the same audience as another product, look at how you can convey what you do differently to them and what makes your product more appealing than them. Or, perhaps it can complement those services. After all, we're talking APIs, so things don't have to be cutthroat and can be more collaborative where there are opportunities for that.

Where to Find Your Audience?

Where you actually look for users will vary massively depending on what your target audience looks like, but, in the case of APIs with technical implementation required, it will likely come down to mastering successful developer marketing. This typically dictates an educational, non-salesy approach.

But, once again, this all starts with a deep understanding of the value your API can offer. We're back to that big question of "what

does your API actually do?" The more resources you can point interested developers to, like use cases and blog posts, the better.

Some good news is that you don't have to do all of this on your own — consider listing your API in marketplaces, like RapidAPI, or directories, like ProgrammableWeb. If you describe and tag your API(s) effectively, these represent a ready-made audience of developers who are hungry for great APIs.

Likewise, sites like ProductHunt, HackerNews, and even relevant subreddits on Reddit all represent good spots to highlight APIs that have enough wow factor to score upvotes. In other words, it helps if your API has some extraordinary use cases!

But, however important external buy-in might be, you also need internal buy-in…

Is Your Organization as Invested as You Are?

In a previous post, we talked about getting business buy-in for an API initiative. That's really important here because it can have a massive impact on the adoption of the product outside of the company. Support teams, for example, can't evangelize the API if they don't understand how it could help customers… or even that it exists.

Talking with people from Support or Operations can help shed light on the customers' problems, enabling you to figure out API solutions that can address them. Even better if you can get insight into the problems your customers' customers are having, as this enables you to build services with a community of end-users the API will reach in mind.

Before, we've discussed treating an API as a product, and one thing we highlighted there was the importance of designing a high-

quality product that serves a specific need. Getting input from all over the business is one of the best ways to build this product/needs-focused mindset.

Finding An Audience: Final Thoughts

No one knows what your customers want better than they do. That's why it's critical to collect feedback via forms or an automated email sent after they sign up for an API key. It's worth asking them questions like:

- What brought them to your API in the first place?
- Did it solve their problem(s) in the way they hoped?
- What about the API in its current state works well?
- Is there anything about it that could be improved?

Henry Ford is often misquoted as saying that "if I'd asked my customers what they wanted, they'd have said faster horses." There's some truth to the fact that being too customer-led can be a bad thing, but obtaining feedback can shed light on issues you might never even have considered.

When you're finding an audience and marketing an API to them, this (actual) quote by Ford is worth taking to heart:

"If there is any one secret of success, it lies in the ability to get the other person's point of view and see things from that person's angle as well as from your own."

Pointers for Building Developer-Friendly API Products

by **Thomas Bush**

Some general ways to create developer-friendly APIs that stand the test of time.

"Great API products don't get built by mistake," says Rahul Dighe, the API and Platform Product Leader at PayPal. Even if you take into account all the trending principles of API ownership — design-first, API-as-a-Product, governance, KPIs, and more — you still might end up with an API that's difficult to use. So, what advice would an API product strategist with over ten years of experience give?

In this post, we'll look at key pointers from Rahul Dighe's talk at the 2019 Platform Summit. These five insights should help you to

create developer-friendly APIs that will stand the test of time.

1. Know Your Developers

It should come as no surprise that building developer-friendly API products start with knowing your developers. As a thought experiment, Rahul introduces three developer archetypes who might use PayPal's APIs. They include a freelancer who is fresh out of college; a payments expert, who has been in the payment space for the last ten years; and an uninterested developer whose true interests lie elsewhere.

When you look at these archetypes, you'll see that developers all have different levels of investment and experience. Not to mention, they each have their own set of requirements, which dictates how they interact with the APIs.

As a result, Rahul suggests you consider the usage patterns of your primary developer archetypes when building new APIs. After all, APIs are a means to an end and not the end itself. Some partners might just need a single, non-API widget for their integration, while others will be looking for a full API solution to migrate to from another provider.

A particularly actionable piece of advice Rahul gives is to write a "one-pager," which contains all of the inputs and outputs and core integration patterns you hope to support with your API. This document will act as a quick reference point during the API design process, allowing you to ensure that what you're building will provide the best experience to your developer audience.

2. Be Obsessive about Naming

Rahul's next piece of advice was the inspiration for my recent article, 10+ Best Practices for Naming API Endpoints. He says you should be "obsessive" about how you name your APIs. After all, once you name an API, it stays there for a pretty long time. And while you might be able to evolve quickly, there's no guarantee your developers will too.

If you want more specifics on how to do naming right, be sure to check out the article linked above. If you're just wondering why naming matters, the answer is simple: it might not sound like much. Still, the effect of redundant, inconsistent, or non-descriptive names can quickly add up, making your APIs challenging to consume.

3. Always Stick to Your Process

By this point, many big enterprises have a tried-and-tested process for building new APIs, that incorporates plenty of time and due diligence. It starts with a discovery phase, where you lay out the problem you intend to solve, which is followed by a design phase, where you begin to think about concrete endpoints, fields, security features, tools, specifications, and user stories. Then, you enter a development phase, before finally launching your new API.

This process usually results in intuitive and all-round great APIs. However, Rahul recalls getting into a discussion with a corporate leadership team, concerning why it took an entire month to add one field to an API. It's easy to be swayed by external pressures like this, but Rahul strongly believes you should stick with your guns.

As an API designer, Rahul says you're forced to build something that's needed today, but will still be used five or six years in the future. You can whisk through the discovery, design, development, and deployment processes, but you'll end up with a suboptimal

product. Instead, stick to the approach that works, and accept that creating or updating an API will take a little longer than others might like.

4. Build a Complete Ecosystem

While getting a single API out can be a challenge in and of itself in some business environments, Rahul says that APIs are just the start, and that you can't forget about the rest of the ecosystem. In particular, he highlights the importance of SDKs, a reliable sandbox, and debug-ready support staff.

Rahul believes that these little things do matter. Sure, it takes time to put all the pieces in place, but these supporting ecosystem assets significantly contribute to the usability of your core API products. Importantly, you do genuinely have to care about and maintain these assets: if you're not careful, you'll end up with old, unhelpful documentation, and a box ticked somewhere on the API-as-a-Product checklist.

5. Think API Governance

Rahul's last piece of advice is to think about how you organize and govern your suite of APIs. He draws attention to Conway's Law, which suggests that we build systems that reflect our internal organizational or communication structures. The traditional approach to combating this — i.e. building consistent APIs, despite having different teams work on them — is to use some kind of API governance system. In theory, by subjecting all API product teams to the same standards, the APIs should feel consistent.

In practice, Rahul believes that there will always be little nuances between APIs built by different teams, which adds complexity for

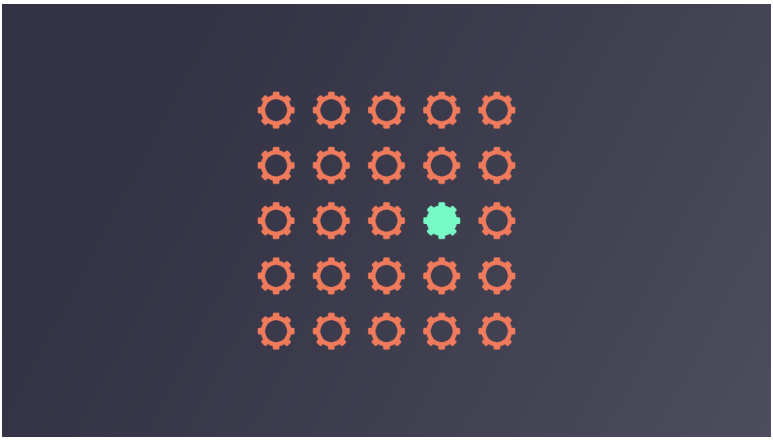
developers who are integrating two or more of them. To combat this, Rahul suggests using something along the lines of the Inverse Conway Maneuver (explained in the article above): if you're always going to have two related API products, have a single team be responsible for both developer interfaces. That way, the APIs are bound to be consistent.

Developer-Friendly API Products: Final Thoughts

The best API products are built when you do things properly. That starts with knowing who your target developers are and how they'll use your API. Later, it means sticking to unnegotiable, tried and tested processes and organization-wide governance standards, and getting names right the first time around. Finally, there's more to an API ecosystem than just the API itself, and maintaining supporting assets like docs, SDKs, and sandboxes is crucial.

9 Areas of Consistency for Great Developer Experience

by **Kristopher Sandoval**



To create a great experience across your API portfolio, be consistent!

Few things harm an API quite like inconsistency. Inconsistency within an API ecosystem can frustrate and confuse even the most experienced power users, creating a perpetuating cycle of failed calls and incorrect assumptions.

While it's seldom considered as its own problem, often being lumped in by type (such as unclear error codes), the reality is that inconsistency creates more inconsistency - if one aspect of an API suffers from this, the rest is likely to as well.

Below are some specific API design areas where inconsistency is

common. We'll consider the implications of things like inconsistent endpoint naming, variable error codes, and spotty documentation, and see how we can instead build more intuitive developer experiences.

1. Naming and Endpoint Consistency

One of the most common consistency issues in many modern API instances also happens to be the most encompassing - name consistency. Name issues cover a few domains, but inconsistency across the board can cause significant problems regardless of where that inconsistency lives for a broad range of users. Let's take a look at two possible domains where this inconsistency can be a problem.

Clarity

For a public API, one of the development goals should be that the API is easily understood. While this understanding is typically gained through ample documentation or other educational efforts, endpoint naming is often overlooked as a profound opportunity to express purpose and intent. How endpoints are named and how descriptive they are can significantly impact how the average user understands what they're interfacing with.

Imagine you are developing an API that remotely serves the weather to a user. When considering a collection of endpoints, imagine two extremes. On one side, we have an obscure version:

```
https://weatherapi.nordicapis.com/aqi10
https://weatherapi.nordicapis.com/tmplcl
https://weatherapi.nordicapis.com/usrlclctyid
```

These endpoints are pretty incomprehensible. They may represent shorthand for internal function names, yet the average external

developer would likely have absolutely no idea what they do. Perhaps they can infer a possible function, but not the whole purpose.

Instead, let's rename these endpoints to be more understandable:

```
https://weatherapi.nordicapis.com/airqualityindex
https://weatherapi.nordicapis.com/localtemperature
https://weatherapi.nordicapis.com/setusercity
```

For the average end-user, this change could be so monumental that it unlocks understanding around the entire API, increasing self-service capabilities.

Casing

Additionally, if a casing style is used, it should be consistent across all endpoints. CamelCase (in which each new word is capitalized, e.g., `AirQualityIndex` or `airQualityIndex`), Snake Case (where underscores replace spaces, e.g., `Air_Quality_Index`), and other approaches are acceptable. Still, it should be considered a best practice to keep this style consistent. For example, if one endpoint is `AirQualityIndex`, you should not also have an instance of `set_-User-City`. Small variances like this, over the breadth of a service, can ultimately result in higher complexity and less understanding.

Accuracy

Related to endpoint clarity is the relative accuracy of each endpoint. It's not enough to ensure that the endpoint is named consistently; you must also ensure that it does what it says it does.

Incompatibilities often arise after multiple revisions and variations on the core codebase, especially if it is old or depends on other libraries. For instance, let's expand the imagined weather

API example. Our users are requesting more functionality. One function has to do with integrating into calendar applications to provide a snapshot of the week's weather alongside scheduled activities. As you develop, you discover that one of the endpoints, `<https://weatherapi.nordicapis.com/weeklyweather>`, does not, in fact, serve the weather in a grouping for the week, but provides average weekly temperature and precipitation.

Such unclear naming can cause significant problems for both developers and users. For developers, especially those who did not work on the core codebase, assumptions based around named functionality can result in non-functional applications. Some endpoints can be relatively understandable, such as the example given above, while others can be less understandable. Something like `<https://weatherapi.nordicapis.com/time>` could imply so many functions behind it (local time, GMT-specific time, regional time, etc.) that it's simply too unclear to be helpful.

2. API Design Paradigm

The API design style chosen by a provider is essential, but being consistent with it is even more critical. It is not unheard of that an API adopts a paradigm based upon the solution de jure but follows that choice poorly. APIs that claim to be RESTful sometimes don't follow a core tenant of RESTful design, GraphQL APIs may not allow truly transformative requests from users, and so on.

This is not only true of paradigm misalignments, either. Sometimes the paradigm is followed, yet the internal API ecosystem is incoherent or confusing. An API may push other requests to other internal APIs, and often, these APIs could be strangely grouped (for instance, an API that handles media transformation may also manage user accounts).

In a RESTful paradigm, it's not just enough to be RESTful - you

should also be RESTful and coherent across all the APIs.

It's also important to know when not to be RESTful, when to leverage other options, and most importantly, when to separate a monolithic API into microservices (or vice versa). Ultimately, it doesn't matter what paradigm you use — if you are RESTful, be RESTful. If you use SOAP, follow SOAP, but above all else, be consistent. Utilizing an internal style guide (like the ones in API Stylebook) can help ensure and enforce this consistency. If you're going to use a guide like this, you should insist on it and use it. Again, above all else, be consistent.

3. Error Handling

One point of clarity that is often missed is in error handling. Too often, error handling is dealt with by serving general error codes with no additional information. How many times has a service returned an error that simply says "sorry, this resource is not found"? For the end-user, this experience is not only ineffective; it's frustrating.

Errors should be handled consistently, given a specific form and function. One of the best resources for standardizing error handling is using standard HTTP status codes. This is a great solution, as each type of error code is delineated and understandable simply by knowing what category these codes fall in. The standard codes are noted below:

- 1xx — these codes deliver information, typically to confirm receipt of the request.
- 2xx – 200 — this spectrum of codes notes that a request was received, understood, and accepted.
- 3xx — this band of codes notes a redirection condition and can be a tip-off that the initial request, while once accurate, is no longer the proper method.

- 4xx — these denote client errors, indicating bad syntax or some other error.
- 5xx — these denote server errors from an otherwise valid request.

Standardizing behind these code ranges and serving the correct codes each time consistently can dramatically increase the API's accuracy, efficacy, and functionality and the average user flow. The key aspect here is to ensure consistency — user errors should be client-centric, server-errors should be in the 5xx range, etc.

4. Documentation

Documentation should never be considered an afterthought — consistency is extremely important for quality documentation. So what does this mean in practice?

First, ensure that your documentation is accurate. At a basic level, documentation must reflect the actual functions, endpoints, and facets of the API in great detail. Documentation should always match production —it is this consistency that lends itself to accuracy, and thus to good documentation.

Secondly, navigation is a vital, often overlooked aspect. It's not just enough to have accurate documentation — developers must be able to navigate it. One way to increase consistency here is to think of your documentation from a scientific point of view. Break everything first down by topics, then by functions, variations, and finally examples. The specific structure doesn't need to be exact — what is most important is that if you adopt a format, stick with it.

Finally, content should be efficient. Consider the average developer consumer — do they have time to read for an hour to understand one function? Of course not – they depend on brevity, examples, and efficacy. Be brief, concise, and accurate.

5. Support and Feedback

Ultimately, the API provider is separated from the developer user. The developer consumes the API however they choose to implement it, while the provider can only assume the average use case based on analytics and usage patterns. To prevent this, support processes and feedback channels are vital.

Support can take a few forms. For some APIs, live support is very helpful. These APIs often fall within the services industry — eCommerce support systems, troubleshooting ticket systems, etc. Ongoing support not only forms the best support for the average user; it can also inform developers as to the current status of the API, consistently appearing problem areas, and unmet needs of the consumer.

Other APIs may suffer from this type of live support. Evergreen APIs that serve a single particular function, such as an API that serves local time data, may necessarily be limited to just that function. After all, if an API only serves time, it doesn't really matter whether a consumer wants a timer function, a scheduling function, etc. — that's not what the API is for. That said, there may be cases where the API is serving erroneous data, not syncing properly to certain devices, or simply not working. In such cases, asynchronous support such as error reporting contact forms, error reporting endpoints, etc., can serve a vital function.

With all of this said, no mode of support or feedback lives past the first loss of consistency. When looking at the sum total of avenues for communication, a user can quickly fall down a black hole — a ticket submitted here, an email sent there, all dependent upon the developers routinely utilizing this data and using the systems on offer. If a developer notes that a system of feedback is available, it's functionally useless if the developer then does not proceed to use that system. Too often has a provider created a mailbox with an email like "feedback@coolapi.com" only to never check the

feedback for the rest of the API lifecycle.

This is especially important when noting that there are many types of support and feedback mechanisms. Suppose an API has a public function as well as internal endpoints. In that case, it may be the correct choice to have different support and feedback apparatus choices depending on who the targeted audience is. In such situations, accuracy and consistency are even more important.

Finally, the mode of communication is only as consistent as the focus and approach. Especially at the enterprise level, communication is essentially a form of public relations. As such, all who engage in communication channels should have at least some sort of training, understanding, or skill set around providing support. A consistent user experience backed by a trained professional familiar with the core product will promote an excellent developer experience. A failure in consistency isn't going to reflect on the singular poor experience but the support platform.

Ultimately, what is important here is that some channel is provided, and that the channel is clearly demarcated, made available, and utilized in core development.

6. Change and Versioning

Perhaps the most crucial area to ensure consistency is within your API's various technical aspects — this includes how you evolve your service. Versioning is a change management method that seeks to ensure that critical deployments, interaction and usage shifts, and altered paradigms are effectively distributed and implemented to all users.

Versioning should be consistent and communicated properly. Ineffective versioning can be extremely harmful to the average API and its consumers. We've discussed this at length before, and we suggest

reading through our collected thoughts concerning API versioning best practices and change communication.

Second, having a consistent methodology behind sunsetting and deprecating APIs and API versions is incredibly important. We've also discussed this before, summarizing the core problem as thus:

"Deprecation and sunsetting terms are some of the most important policies to communicate to developers. Your communication here will not only secure your current API, but if you're migrating to something else, will likely ensure its success as well – users only trust providers who communicate. Effective, smart sunsetting and deprecation isn't just good for your code, it's good for your users, the API space, and the industry at large."

7. Security

Security is another area where consistency is incredibly important. Having consistent endpoints with consistent security standards and baseline monitoring and response methods is incredibly important. Every small failure of consistency here can compound and open more vulnerabilities that you may not even know exist.

One common security issue is allowing different access points for different user classes. While this most often happens within the context of an exposed "test server," this can also happen with test endpoints for new functions. When those endpoints are created and left open once their usefulness has expired, this only creates a risk of ingress.

Additionally, any security choice made throughout the API should apply to the API in totality. Things like rate limiting are great to prevent denial of service attacks, but if those limits don't apply to elevated credentials, then you're creating a massive flaw in your security. In such a system, the second an elevated credential

is usable by a threat actor, you've made a situation where your massive security solution does not exist for the direct threat actor.

Ultimately, security is not a single fix. "Security" in an API is not like locking a single door and calling yourselves secure — instead, it's like inspecting a two-mile-long castle wall for weaknesses, erosions, or high points that can be traversed. It's an active process of inspecting for weakness, however minor, and then addressing those weaknesses to balance the user experience with restricted access.

8. Authentication and Authorization

One major consistency issue is in the processing of user authorization and authentication. While this specific area of concern covers a handful of domains, we'll more specifically discuss a few general types here.

Firstly, the use of API keys and the storage of those keys should be consistent both in process and location. API keys can be extremely powerful, and without a proper strategy, they could be inadvertently exposed, given far too much power, or even used for entirely incorrect purposes. Keys should be provisioned based upon a consistent set of standards, and they should only be made for access to a limited set of resources for a specific purpose. There should be no "skeleton key." Most importantly, these keys should have a consistent revocation and expiration system to ensure that no key lasts forever, and that keys can be adequately cycled and distributed.

Secondly, when any account is closed or terminated, you must properly handle the associated credentials within a set expiration process. Improper expiration of certificates is a significant source of ingress into systems from unauthorized users, as they, when paired with poor API key handling, can leave valuable sources

of interaction completely open. Add on to this the fact that these credentials are often overlooked as sources of insecurity (given that they were, at one time, trusted), and you've got a perfect storm resulting in a threat actor flying "below the radar".

9. Platform Consistency

One final point of consistency is the unification of the API experience from the perspective of documentation and specification. When developing an API, unless the specification is used to create the documentation outright, there is a possibility that the output is going to differ from the input. While this is typically minor and can be fixed by an attentive developer, the reality is that these differences often propagate through multiple versions, resulting in documentation that must go through an overhaul to be even remotely usable and valuable.

It's also possible that if the specification is used to create the output API, but this process is done at the start of a series of developments, that the end output will still be mismatched. The worst part of this scenario is that the output is different without apparent cause; it is more likely that this scenario plays out without it being obvious to the developers than the former scenario.

At all times, developers should strive to ensure that their public docs and statements in specification match the actual deployed instances and that the ethos and processes are reflected adequately. This includes ensuring that the numerous diverse outputs which inform the user match the inputs, but it also includes matching the stated goal and purpose of the API are reflected in the actual functions. It's not ok for an API to state that its purpose is to do something, and once the user enters the API, it's unclear exactly how (or even if) it does this – that is the worst possible scenario for user experience.

Be Consistent: Final Thoughts

Inconsistency across an API can drive inefficiency, confusion, and failure. With all of this in mind, the solution is simple — be consistent. Even if you adopt your own standards for style, naming, error codes, etc., be uniform across your developer platform.

As long as everything is done in a consistent manner, even proprietary solutions can be quickly learned and understood within the context of the rest of the body of work. As long as any inconsistency exists, the API will forever be lacking clarification, which decreases usability and self-service capabilities.

Nordic APIs Resources

Visit our eBook page to download all the following eBooks for free! They're also available for a small fee on LeanPub and Amazon Kindle.

API-as-a-Product: In this eBook, we cover tips to help you create a working business model around a specialized public API.

How to Successfully Market an API: The bible for project managers, technical evangelists, or marketing aficionados in the process of promoting an API program.

Identity and APIs: Discover the techniques to secure platform access and delegate access throughout a mature API ecosystem.

API Strategy for Open Banking: Banking infrastructure is decomposing into reusable, API-first components. Discover the API side of open banking, with best practices and case studies from some of the worldâ€™s leading open banking initiatives.

GraphQL or Bust: Everything GraphQL! Explore the benefits of GraphQL, differences between it and REST, nuanced security concerns, extending GraphQL with additional tooling, GraphQL-specific consoles, and more.

The API Economy: APIs have given birth to a variety of unprecedented services. Learn how to get the most out of this new economy.

API Driven-DevOps: One important development in recent years has been the emergence of DevOps, a discipline at the crossroads between application development and system administration.

Create With Us

At Nordic APIs, we are striving to inspire API practitioners with thought-provoking content. By sharing compelling stories, we aim to show that everyone can benefit from using APIs.

Write: Our blog is open for submissions from the community. If you have an API story to share, please read our guidelines and pitch a topic here.

Speak: If you would like to speak at a future Nordic APIs event or LiveCast, please visit our call for speakers page.

About Nordic APIs

Nordic APIs is an independent blog and this publication has not been authorized, sponsored, or otherwise approved by any company mentioned in it. All trademarks, servicemarks, registered trademarks, and registered servicemarks are the property of their respective owners.

Facebook | Twitter | Linkedin | YouTube

Blog | Home | Newsletter