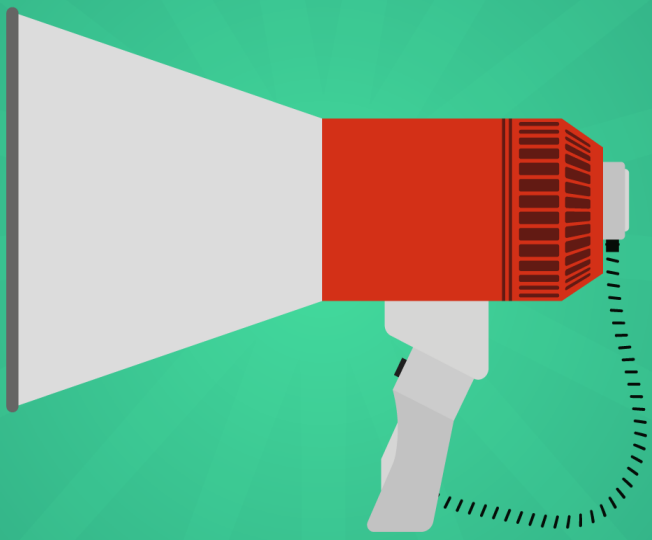
How to Successfully Market an API

Fine-tuning Developer Relations and Platform Advocacy



AUTHORS

Bill Doerrfeld
Chris Wood
Kristopher Sandoval
Vassili van der Mersch
Jarkko Moilanen
Oscar Santolalla

NORDIC APIS
nordicapis.com

How to Successfully Market an API

Fine-Tuning Developer Relations & Platform Advocacy

Nordic APIs

Contents

Part Four: Advocacy 117

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

From Hello World to Hello Developers

In the widening API sphere, marketing an API business involves knowing your community intimately, and fine-tuning your developer support channels to help users excel. With APIs continuing to surge in importance across all industries, spreading API knowledge becomes increasingly important. More APIs equals more competition, meaning that API evangelism, the job of promoting a developer-centric program, now needs its own strategy.

This eBook is the first of it's kind. A compilation of advice and research geared specifically to DIY marketing for a public web Application Programming Interface or similar developer-oriented program. These aren't tips for amassing hundreds of developer emails - no. This is about creating a usable frontend for your API platform that employs healthy developer outreach to naturally increase the prestige of your Software-as-a-Service.

API Marketing is one of the Nordic APIs 6 Insights to API Practice; a core facet of providing an API. Thus, we've collated our best advice within this book to explore the Marketing Insight from four important angles:

- **Part 1 - Planning**: Tips on starting your program, understanding your target consumer segmentation, market research, and more using the API canvas model to position an agile API business.
- **Part 2 - Developer Relations**: Discussions on onboarding and quality developer experience when it comes to overall API design as well as documentation and developer portal resources.

- **Part 3 - Promotion**: Outlines emerging API discovery techniques, API directories, and relevant press resources that will help spread awareness of your product, as well as ideas for creating informative content that informs and engages potential API users.
- **Part 4 - Advocacy**: Lastly, we define community building best practices, the roles of program advocates, tips on holding your own developer conferences, API demoing, as well as case studies into developer relations failures and successes that companies have had in their public API programs in recent years.

As you can see, we've taken a holistic approach to marketing in this volume. The end goal is help readers extend reach and onboard more developers to their API. So please enjoy *How to Successfully Market an API*, and let us know how we can improve. Be sure to join the Nordic APIs Digest for biweekly blog updates, and follow us for news on upcoming events.

Thank you for reading!

– Bill Doerrfeld, Editor in Chief, Nordic APIs

Connect with Nordic APIs:

Facebook | Twitter | Linkedin | Google+ | YouTube

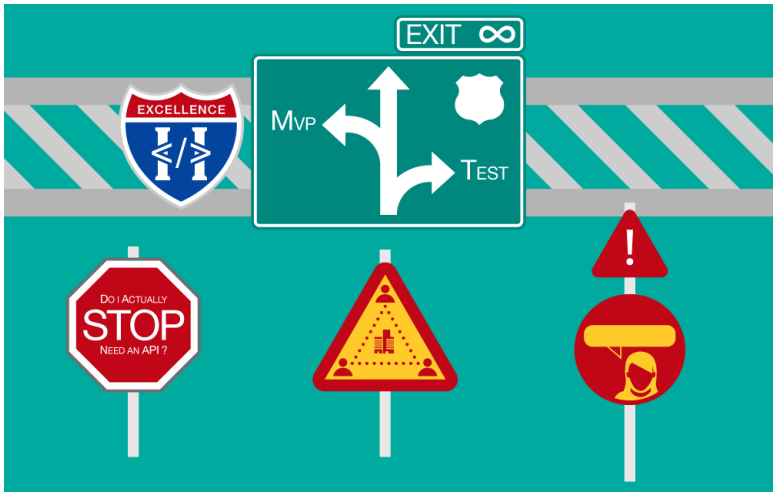
Blog | Home | Newsletter | Contact

Part One: Planning

Prepping an API for consumption

Strategically marketing an API platform relies on understanding the **core business model** and API product lifecycle. Thus, this section on **planning** covers our tips for building from the ground up, finding your target developer for market segmentation, and transforming your business and platform design into one that embraces developer experience as a top priority.

Building from the Ground Up: Tips for Starting Your API Program



New public APIs hit the market all the time and behind these APIs sit a myriad of different companies and organizations; individual developers, startups and established businesses who have concluded they need a public API program in order to better serve their audience. Doubtless these new API providers will all have different ideas on starting a public API program, the implications for their business, and what their future plans for expanding or enhancing their API entail. In this chapter we've drawn together **5 tips for starting a public API program**, with some advice on what to consider when you go from zero to a production API deployment.

Starting a public API program is a **massive** topic. For the sake of

brevity we're only scratching the surface in this chapter, hitting the main points as relevant to **API Marketing**. For more on Platformitization, Development, and Design see our other eBooks.

Clarify Your Needs

First and foremost, before doing a scrap of work your first task is to clarify that your organization *actually needs* a public API program. This is hugely subjective given the spectrum of organizations interested in APIs, from one-man-bands to huge multinational corporations, and is compounded by the varying motivations behind constructing an API. As soon as your API program is considered necessary stop and ask yourself these questions:

- **Would my product/platform benefit from having an API?**: There are many compelling reasons to start an API program: enabling choice for customers in how they consume your products, extending the reach of your product to your customer's customers, and enabling new business channels with partners, to name but a few. Regardless of the hype surrounding APIs, if you can objectively say yes to any of these reasons then starting an API program is worthy of investigation;
- **Are my customers asking me to deliver my product/platform to them differently (especially via an API)?**: Much of the clamour for delivering an API will come from the userbase. Again there will be a huge amount of subjectivity and self-interest in their reasons for asking for one: Some will ask purely because they've heard the term and noted that some of your competitors offer them; while others will have a genuine need, such as the desire to integrate your product

with other applications they use. Integrating your product stack into their workflows is disconnected and manually intensive, so integrating via an API is hugely advantageous. Both of these are compelling reasons to act: If you don't have an API but your competitors do, what's to stop your customer from doing business with them?

- **Do I understand the effect that introducing an API will have on my business (both good and bad)?**: It's possible that your organization could be profoundly affected by introducing an API. For example, an API *may* change the operating model of your business from daytime hours only to a 24x7 operation, or may be so transformative as to turn your business into an 'Infrastructure-as-a-Service' model. One needs only to look at Amazon to see how disruptive the effect of creating a public API program could be on your business.

If you can answer these three questions positively then starting an API program is definitely in the interests of your organization. Your task now is to get wholesale buy in from stakeholders in the business.

Get Buy In (From Everyone)

As we started to discuss in tip #1, you should be clear that unless your API is the main channel for selling the product you are building, starting an API program could **fundamentally** change the way your business operates. For example:

- Your API *could* disintermediate parts of your own business (for example, if you rely heavily on direct sales). Is the leadership team ready for the implications this could have on the workforce?

- If providing an API drives volume, can your business cope with that extra volume without change?
- Do you have the team and resources to build adaptive code and scale your developer relations so that the API is treated as a full-scale product?
- Do your backend systems that will be exposed by the API have the means to support an abstracted security model or identity provider, and are you aware of what doing so will mean for the threat of cyber attacks and risk management?

In order to be successful in starting your API program you need every member of the team to understand that you are effectively **extending your internal systems outwards**. The impact will have both a technical and business focus, so it's important to communicate in a way each stakeholder understands. Fostering a culture of security, and adopting an API mindset throughout an entire organization is vital for success.

Aim for a Public MVP

With the need established and the team on board, you should have a clear picture of *why* you are building a public API and *who* needs to be involved. You can now create a vision of what your API will deliver in the form of a minimum viable product (MVP), delivered to the market as an alpha or beta. This **MVP** will allow you to get your API in front of potential API consumers as quickly as possible and allow you to start collecting feedback. The MVP are steps 1 and 2 of a 3-step process:

1. Initial build out of the MVP with internal **private** testing only;
2. Offering the MVP to a select audience of existing customers or **partners** who have the potential to become consumers of your API;

3. Migrating the MVP to a production-ready version of the API, ready for **general availability**.

In order to get the most value in your goal of offering a public API the MVP should be comprised of a number of things:

- A publicly available endpoint or simple sandbox that can be used by potential API consumers to trial your products, with enrollment either by invitation or with a set of test credentials known only to your target audience;
- An API description in a specification format of your choice (OpenAPI, API Blueprint, RAML, etc.);
- Documentation, possibly in the form of a cookbook that describes how to implement an application that consumes your API;
- A well-formed **security model** so that potential API consumers will understand the implications of integrating with your API;
- Establish a means to support API consumers who are trialing the API through a medium of your choice: Twitter, Slack, email or whatever makes the most sense for your organization;
- Analyze how those trialling your API use it in an effort to understand if there are any potential design flaws;
- A high-level terms of use so API consumers are aware of the expectations surrounding the usage of your API: If there are stipulations that may discourage certain consumers it's probably better that they aren't part of your audience when you introduce the MVP.

With the MVP available to potential API consumers you are at liberty to take advantage of the next tip: Acting on the feedback.

Act on Feedback

With the MVP in the market and channels of communication back to the API team in place, you have a unique opportunity to elicit and act on feedback before a public release. Your API will of course continue to evolve but with this early feedback you are at liberty to perform elements of wholesale reengineering. Such actions might include one or more of the following:

- Revisit the design of the API and ensure it's updated to reflect any feedback on usability;
- Ensure you listen to reasonable customer demands for different encoding types: The majority of new APIs use JSON as their encoding of choice, but if it makes sense to provide the data as XML because the majority of your potential API consumers prefer XML for legacy reasons you should consider it;
- Ensure the rate limits and throttling you introduced with your MVP can realistically be sustained when higher usage is applied;
- Tweak your conditions of use to ensure you address your target audience correctly, giving your API consumers terms they are happy to commit to, but also allowing business to sustain as your API grows.

By taking these actions you will be in a much better place to release version 1.0 of your API. However, you've got version 1.1, 1.2 or even version 2.0 to think of; another point in our arsenal of tips for building a public API program is to **future proof** your organization by building an API practice that can support future releases.

Build your Practice

Building an API practice is similar to creating a center of excellence or community of practice as in any other technological or architectural discipline. However, the API practice should be geared towards a method for developing APIs that ensures you can go from zero to production with a known series of standards and methodologies and the minimum amount of fuss. Some tenets of this practice are:

- Expressing a clear vision of the API and the roadmap of features, releases and enhancements that will help your business meet that vision;
- Create an architecture and development framework that support productivity and will allow you to iterate rapidly on the items in your roadmap: Using practices like API-first design with a API description format that best suits your needs and possibly a tool like Stoplight to help model your APIs will give you a better chance of delivering new APIs and API versions with the greatest efficiency;
- Create an **internal style guide** for your API developers that can support your productivity efforts by making API designs consistent;
- Make the most of your infrastructure and DevOps tools: The options for choosing an approach to API development are vast and include different models; SaaS-based API lifecycle management tools, web application frameworks and PaaS, containers, and so on. However, whichever option is right for your organization will ensure that you leverage the value of continuous integration and continuous delivery to really help accelerate delivery;
- Keep your internal stakeholders informed and involved: There may be aspects of your roadmap that draw new areas of the

business under the API banner and therefore affect different operational or support teams. Having clear lines of communication and engagement with these teams is vital for the continued success of your API;

- Communicate externally: Your MVP has been a success and you've started to engage with your target audience, but there's a momentum that needs to maintained. Ensure you are communicating your goals with this audience in a way that that casts the net wider; publish articles and blogs extolling the value of your API in the context of the industry you operate in and ensure you address discoverability techniques like Search Engine Optimization and directory bookmarking for your documentation pages so that potential customers can easily find your API.

Final Thoughts

The tips we've drawn together in this chapter should help shape your thinking if you are considering starting a public API program. However, every business or organization's circumstances and motivations are unique, so most importantly **trust your gut instincts**. APIs are a great method for taking products to market, so trust your own judgement and use what advice you can from this post to help make your public API program a reality.

Define Your Target Developer Audience

Throughout webby plains of interconnectivity, over fifteen thousand APIs now expose data and systems — adding awesome functionalities to mobile apps, and allowing entirely new data-driven businesses and new user experiences to blossom. Within any economy showing such exponential growth, the **diversity** of its players will naturally increase as the market evolves. It's a fact that the API space is becoming increasingly diverse — it's not the old days where a few independent developers created mashups for fun. APIs have entered large business dealings, gained the attention of enterprise-level product designers, led to the creation of multi-billion dollar startups, influenced creative marketing campaigns, and more.

So, with all this new interest from varying audiences, API providers may be asking: **who are we selling to?** With more and more consumers entering the API space, it's now more important than ever to consider the large breadth of **specializations** amongst third party developers. These factors take the form of:

- varying technical understanding
- different industry backgrounds
- geographical location
- online activity
- API use cases
- protocol preferences
- programming language preferences

- ...and more.

In order to **segment** marketing correctly, and to do so in a way that is appealing and in good taste, it comes down to intimately knowing the needs of your audience. In this chapter we interview developer program strategists from Catchy, Dopter, and Stateless to explore why **knowing your target API consumer** is now more important than ever.

Why Create a Developer "Persona"?

Mike Kelly runs Stateless.co, an API strategy consulting firm based in London. He describes the wide breadth of developers currently in the space:

> "There are of course broad categories such as system integrators, mobile, web, embedded systems. In reality there are innumerable forms of developers each with their own set of constraints and requirements, which is why developing a set of realistic developer personas is important."

Traditional business models necessitate a process of consumer profiling. The same can be done in a way that makes sense for this niche API sector. Intimately understanding your target consumer is crucial as it will influence the following:

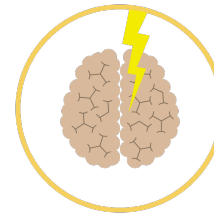
- **Market Fit**: Knowing your consumer can help you discover unmet needs in the market your API could potentially satisfy.
- **Functionality**: Understanding use cases can help design API functions around that need.
- **Segmentation**: Knowing your user can help segment marketing efforts and decrease customer acquisition cost.

- **Creative Marketing**: Knowing what your audience is interested in will help tailor your marketing to this audience, influencing your message, tone, appearance, design, style, and more to attract your target audience.

Kelly goes on to mention the importance of establishing developer personas is that they establish an important **frame of reference**:

> "If you're unable to clearly describe your target customers and their use cases for your API, that usually indicates that the underlying proposition is not focused enough. Likewise, if you're unable to clearly determine how a proposed feature provides immediate value to one of your personas, that is a strong indicator that it doesn't belong on the roadmap yet…I'm a huge fan of any methodology that encourages approaching the strategy and design of an API by focusing on the client side, rather than the server side."

The Developer Brain



'Know your demographic.' 'Understand the **psychology** of your consumer.' We hear phrases like this frequently in general business discussion. Is it possible to apply the same philosophy to marketing APIs?

Developers are not your average consumer. In a conversation with Nordic APIs, General Manager Jason Hilton of Catcy Agency, an international developer program management outfit, pinpointed the following attributes. Developers are **analytical**. They appreciate **authenticity**. And in a sea of competing tools with rampant overzealous

marketing, they have the right to be **skeptical**. A web developer especially wants to pick up and play with a product, expecting an **instant proof** that it behaves as advertised.

Take these generalizations as you will, but they're worth to consider when developing a marketing message and story applicable to this certain audience. Both content and aesthetics can either incorporate or alienate depending on their execution. Creating a developer portal, for example, should, in response to our brief psychological examination, be intuitively designed, with **transparent** information, an **attractive layout**, and **interactive** modules that allow one to test API calls.

But Plenty of Other People Are Interested in APIs, Too!

Andreas Krohn of Dopter urges us to consider a wider scope with API marketing. In his talk at the APIStrat conference, Krohn encourages us to rethink developer evangelism. Instead of focusing so heavily on developers, API providers should create more inclusionary *customer* personas.

> "I strongly dislike how developers are worshipped in marketing"

In his work at Dopter, an API strategy and consulting firm, Krohn routinely encounters providers that want to create hackathons to reach out to *developers* to promote their APIs. Though that can be an effective strategy, *are developers really the sole audience that may be interested in APIs?* The truth is that designers, entrepreneurs, marketers, and other business leads are just as important constituents.

Krohn believes we naturally target developers for the following reasons:

1. APIs are technical.
2. Developers relate to other developers.
3. Silicon valley influence.
4. Myth of the single developer.

Though many people aware of APIs are developers, not *everyone* is a developer. We must remember that the world is bigger than the Silicon Valley. Just as the laymen smartphone consumer can brainstorm innovative app ideas without needing to know how to code, there is a similar low bar that allows anyone to understand the possibilities APIs offer to then envision creative applications. In a company, products are developed by large teams — entrepreneurs, project managers, designers, etc. Developers are crucial to any software process, but in reality, a diverse array of experiences contribute to innovation in the space.

Expanding our Portal: Developer End User Evangelism

If we incorporate a wider audience into our understanding of their target API consumer, how does that change the way an API is marketed?

Krohn encourages us to remember that *real* value is only created when an **end user** uses the app that's created with the API. And who creates that experience? Entrepreneurs may create the product, managers oversee production, designers envision the user experience, developers connect the backend, and other domain experts may contribute. All these stakeholders intimately understand the value of API integrations, and thus all their perspectives and **needs** should be considered.

Think of standard web API portals as they are now. Can entrepreneurs immediately discern the **end user value** when they

view API documentation? More often than not, their needs are excluded. There is a lack of high-level summary and sample use cases to inspire non-technical minds — this experience is often non-existent.

Krohn believes that Developer evangelism should rather be End User Evangelism. As Krohn says, "be aware of who you are including or excluding, and make it a conscious decision." Think you know your target audience? that may soon change.

Varying Industry Backgrounds

With the recent rise of B2B and enterprise interest, should API providers be selling to individual developers? Or is it a better idea to seek out business partnerships directly? The way APIs are marketed and consumed varies tremendously on the **industry**. API providers may range from a two person startup to an enterprise development team spanning hundreds of employees. The same diversity is present on the consumer end, affecting the way APIs are acquired. The core value message also changes based on whether the API is directed toward startups, engineers in a large organization, or toward convincing upper leadership.

Jason Hilton of Catcy says we can begin by separating the consumer side of the API space into two distinct groups:

- **Enterprise Developers**: Developers working within a large organization. The challenge faced here is that they may not be the decision maker. Working on the inside, means they must make the case upwards, involving multiple parties and potentially creating a longer decision making process.
- **Freelance Dev Shop**: These are small startup teams looking for helpful API integrations to accel their product.

According to Hilton, industry variance mean it's "It's worth applying specific techniques and strategies. Just as much energy and consideration needs to be put into marketing a developer program as with any other product."

Mike Kelly shares a similar experience, encountering differences in user acquisition based on who's consuming:

> "It varies a lot between different businesses and markets. If a relatively lightweight, phased on-boarding process for partners and clients is realistic and commercially viable; then absolutely APIs can, and should, be used to generate leads from developers. We refer to this as **bottom-up user acquisition** … If the business is tied to a traditional **top-down user acquisition process**, with procurement decisions happening in upper management, then developers and the API will play much less of a role in sales. Having said that, if deep technical evaluation and due diligence is a key component of their decision making process, then the API will still play a key role."

Location & Demographics

With so much personal data online — GPS, demographics, bio, history, taste, "likes"— the amount of data opens up a pandora's box for hypertargeting. However, simple **geographical location** is still considered vital when creating a profile of you target API developer:

> According to Catchy, "location is a crucial factor. AT&T, for example, has APIs that are useful to developers who are producing apps for the US market. Indeed, many of their APIs are only available to developers with a

> US billing address because of the monetization aspects. Trying to attract non-US based developers to adopt these APIs is completely pointless."

Other than geographical pinpointing, additional **physical** locations like developer centric programs such as hackathons, meetups, and conferences can be used to discover your consumer. If we consider **online** activity, our target developer location varies depending on their needs. Gauge how active your target consumer is on Stackoverflow, Github, Twitter, Reddit, among other communities.

API Use Cases

Next, API providers must imagine the API's **use case** in the wild. Why would someone want to use your API? Providers should consider the possibilities the data offers and brainstorm **applications** that could be created using the API. Kelly believes this process will help identify consumer needs:

> "The key to a good persona is in establishing concrete user scenarios and stories…You need to understand the needs of your customer before deciding what to offer them."

This process also involves considering the **business model** of the product that will be created using the API. Hilton notes that "the business model of the API is important — we know that serious developers will monetize either through paid apps, or carrying ads. Trying to get the developer of a paid app to implement an ad API is pointless."

Technology Preferences

Developers have varying technological specializations, focusing in different programming languages, attracted to certain protocols over others, and coming with a unique history using specific tools. So, how critical is it to consider these concerns when marketing an API?

> Hilton: "As well as understanding *why* someone would want to use your API, it is vital to having an understanding of *who* could do so. Likely there will likely be some criteria that would directly influence the sorts of developers who could become potential users of the API. Such criteria might include prerequisite technical knowledge, such as coding languages, platforms, or protocols, or else might entail non-technical factors such as location. If an API marketing campaign is to be successful, it should target all and only those developers for whom the message is relevant. Understanding your API's technical requirements and restrictions, and then being able to identify and reach relevant developers, is a key facet of API marketing success."

> Kelly: "JSON over HTTP seems to be the standard preference these days; likely because they are both simple and ubiquitous with strong tooling on the vast majority of development stacks."

Lessen The Corporate Branding

Marketing, relations, outreach — typical B2B sales motives, labels, and tactics may conjure up negative connotations. Hilton acknowledges a common pain point among working with enterprise clien-

tele is knowing when to lessen the corporate branding to appeal to the tech community.

Mike Boich of Apple was arguably the first self-proclaimed software evangelist — now throughout the tech community, the title is more commonplace than ever. It's no wonder why the tech community has embraced alternative roles like "evangelist" or "developer advocate." It helps communicate a **true passion**, and having a passion, and a true conviction for what you are promoting will win the day. The lesson of 2015, according to the Catchy team, is that "If you have a developer evangelist, you have a dev program."

Successful copy can't be homogenous. It must be **direct**, imbued with enough **transparent** technical knowledge to communicate value. This becomes a concern for larger clients that are not accustomed to exposing their documentation. When asked his opinion about the pros and cons of opening up an API to the public — weighing the loss of security and authority over proprietary knowledge — Hilton advises as follows:

> "It depends on the goal of the API owner. If volume is critical, then opening up the API is a crucial first step. If, however, it's niche and / or dependent on the quality of products produced using the API, then keeping it closed is the more sensible route.
>
> As a general rule, if the API is for enterprise then being 'closed' can work. An example here would be SAP. Being closed allows the API owners to retain strict control of the ecosystem, ensuring all apps retain business look-and-feel, and also quality measures. But closed APIs, entailing log-in to a proprietary ecosystem, will always be a barrier to entry for the majority of developers.
>
> If the end product is commercial (i.e. apps in the Play store) then open APIs have significant advantages. For

> one, they are more popular with developers, who can fork the source code to invent completely new products. This allows developers to innovate, creating products that the original stakeholders would not otherwise have thought of. Open drives innovation, and keeps barriers low. Android apps being the paradigm example."

Developer Experience

The presentation of developer facing material is paramount to success in the space. According to Mike Kelly, "providers may see the best growth opportunities in customer acquisition and activation by developing tutorials, improving documentation, exposing sandbox environments, tailoring the API design to specific use cases."

API providers can use tactics to help acquire developers, increase onboarding efficiency, and maintain user retention by appealing to the tastes and needs of the audience. Stellar reasoning behind creating quality developer experience can be found in John Musser's "10 Reasons Why Developers Hate Your API." Harking back to Andreas Krohn's stance on the need for increased inclusion, API experience should also consider the entrepreneur experience, the manager experience, and the designer experience. In that spirit, embrace developer language, but also cater content, **appearance**, UI, and UX to your audience. According to Kelly, "developer experience is the most important metric of quality for an API. It's vital."

Build it And They Will ____

Simply opening a platform up without the right forethought, planning, and intimate knowledge of your consumer will not work in this sector. Product teams within technology companies often

assume the product will be used, but too often, marketing is either too slim or inefficient, leading to a low adoption.

Understand Your Audience

Think you know your target audience? that may soon change. As diversity increases in the API economy, we reconsider API user segmentation, defining specific traits that make up today's unique API consumer. Perform the necessary research to find who your consumer *really is*. Who is the key influencer that can communicate the value of your API? It comes down to knowing the needs of this audience, and this requires thorough analysis. Wielding this knowledge, you should have helpful answers to the following:

- **Technical Perspective**: Understand what type of person will be interested in your API, and make sure your marketing and API portal does not exclude interested parties.
- **Industry Background**: Consider who will be attracted to your API: is your API a public offering encouraging startup developers to check it out, or is it a partner integration used by enterprise teams?
- **Needs**: Understand the needs of your audience first, then tailor functionality and experience to address those concerns.
- **Architecture**: Build in relevant modes that your target audience find useful.

Developer Experience is a Key Ingredient of Quality APIs

According to The State of API Survey Report 2016 by Smartbear, nearly 85% of the respondents agree that **API quality** is crucial to their organization. The same survey identifies the top three reasons keeping organizations from delivering quality APIs are:

1) Increased demands for speed 2) Lack of integration between tools and systems 3) Managing expectations of different stakeholders

A remedy for the first reason can be found by fine-tuning architecture and optimizing code. The second reason requires a DevOps

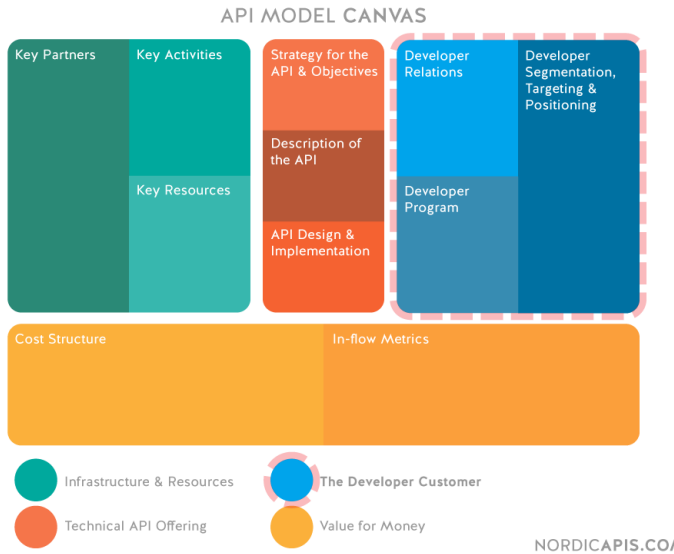
approach to operations and automated solutions, which in an API-centric point of view can be labeled as **APIOps**. The third reason is a bit more tricky and is discussed in this chapter. Part of the solution is to utilize the still rather fresh idea of the **API Model Canvas**.

API Model Canvas – offspring of Lean Canvas

The API Model Canvas should be familiar to startup people since it resembles **Lean Canvas**. The Lean Canvas proposed by Ash Maurya is an approach for entrepreneurs and startup businesses that is more problem-focused than its ancestor Business Model Canvas. Canvas models take customer needs into consideration, force business design to be iterative, and unlock fast, adaptive building methods.

At first sight these methodologies might look pretty similar. API Model Canvas is, as name suggests, more focused on the API Economy and for situations where the API is a product. What differentiates API Model Canvas from its ancestors is the focus on *developers*, the rockstars of the API Economy.

Developers are the Rockstars of the API Economy



Developer eXperience is a concept referenced in the API world very often, and for a good reason. Developer consumers are the lifeblood of the API based economy. The API Model Canvas reflects this importance with three sections dedicated toward keeping them in the spotlight: Developer Relations, Developer Program, and Developer Segmentation.

Developer Relations

As Phillipp Schöne of Axway sees it, API providers must go beyond great API design and documentation, and work toward creating superior end-to-end experiences:

> "If you are in a competitive situation it can become a differentiator to have a good and well thought out experience, even if the price is a little bit higher. Often when API providers consider Developer Experience, API Design is usually only taken into account, which is wrong."

Running an API with quality developer relations means intimately knowing your consumer, their needs, and advocating on their behalf. For active community participation, Phillipp sees hackathons as a "great way to learn what external people think about your offering… start with smaller groups and go bigger if you get more confident with your offering."

Developer Program

Maintaining DX means having a holistic product approach to APIs; part of this means forming an internal product team, with the responsibility of API program upkeep and marketing. Tuning into consumer feedback is important, but Phillipp also believes that "if providers 'eat their own dogfood' they usually experience the caveats and hurdles of their offering quite fast." ### Developer Segmentation, Targeting and Positioning We've seen how the developer consumer market has diversified in recent years. As with any product, the right segmentation needs to be considered. This involves an intimate understanding of your consumer and where they fit within the API economy. How you target specific developers depends on the market and audience:

> "For example, if you try to create something specific for a vertical or niche try to leverage industry specific forums, websites, and events to promote the idea. If you target a broader audience it's more complicated and you have to search for developers who feel the most pain and gain and would benefit the most from your service."

Schöne adds that quickly compiling customer references and testimonials is paramount for establishing a reputable image.

Addressing the Entire API Model Canvas

The API Model Canvas is easy to follow and fill in. Having succinct answers for each section can help you nail down your value propositions and specific strategies. You should not do it alone or fill everything in like writing an essay, rather, it should be designed as a team. More importantly — do not design an API business model alone in the office cubicle. You should involve IT people, business strategy people, and a few developers from candidate customers. Because your API is expected to bring value to these customers, the foremost aim should be to solve *their* problem and solve it well.

Case Study: National Library of Finland

Use of the API Model Canvas has been a proven asset across several teams. One of the latest teams to use it was the The National Library of Finland. They have millions of data objects about Finnish journals and newspapers dating back to the 17th century. All this information needed to be made accessible in a way that current newspaper publishers, researchers, and citizens could efficiently consume. In brief, they needed a quality API.

The project was a three step process, with the API Model Canvas used in the second phase:

- Define an API strategy. Obviously strategy emerges from the organization, which in this case was National Library of

Finland.
- Utilize API Model Canvas to craft a business model for APIs needed.
- Describe API with design driven language such as RAML or Swagger.

Of course after design is finished and tested with a mockup server, the API must also be implemented. For this project, we decided that we would leave implementation out at this stage and focus on business model and API design.

MVP for next step

During the first APIOps camp, we did not go through all boxes in the API Model Canvas. Instead we took a lean startup approach and started from the middle at "Strategy for the API and Objectives". After that we jumped around the boxes to gather a Minimum Viable Product (MVP) for the next step. This was possible since we had experience in Lean Startup style development and canvas models. For example, we did not touch "Cost Structure", "Developer Program", or "Developer Relations" at this point.

The idea was to collect just enough information for the next iteration of the API's Swagger description and not to plan anything unnecessary. In the process, two collections were identified from data and feedback gathered from data consumers: newspapers and journals. Based on that we decided to define read interfaces for both with similar structure. In practice we defined newspaper endpoints first and then cloned the structure in Swagger design. We did not touch write interfaces at all, since we narrowed down the scope to just API consumers and excluded data administrators.

After adding some basic endpoints to the API, we collected feedback from developers who we had engaged with earlier. Including API consumers in the API design phase gave us insights on what

developers were looking for in the documentation, and shed insight on what keywords and writing style grabbed their attention. It also convinced The Library of Finland to dig deeper into DX than just providing nice documentation: code examples for utilizing API, case descriptions, and providing customer support. In other words, the client understood the need and value of a proper developer portal.

This cycle goes back and forth: filling in and altering the **API Model Canvas**, returning to Swagger design and developer testing. After some rounds, you'll have a solid business plan, and golden API Design loved by developers. At that point you are ready to implement it.

Gain speed and make it fun

API Model Canvas is an important tool and in this particular case worked very well. For a team with a startup mindset and familiarity with Lean Canvas, getting started is easy going. At the end of the day, API Model canvas can make business model design fun — even people participating in this type of process for the first time will see API Model Canvas as a solid tool for API centric business design.

Incorporating third party developers into the modeling process right from the start eliminates the risk of losing your Developer Experience focus. Keep in mind that developers are the people through which you'll reach business agreements and convince upper tier managers. If developers hate your API, you'll lose revenue— every developer lost is revenue lost.

Final Thoughts

There are many theories on API business modeling; the Nordic APIs philosophy of API practice involves 6 core tenets, for exam-

ple. Whether following these insights, the API Model Canvas, or other toolkits, Phillipp notes that the "true importance is that the API Provider starts thinking about the different dimensions and implications of their API." For him, the main traits a lean API MVP design must consider are:

- Product ideation and preparation
- Monetization and growth projections
- Legal aspects: are there any legal obligations or regulatory mandates we need to worry about?
- Infrastructure Questions: how to connect to existing Data sources, how do we manage API consumers, how do we want to scale, how to we monitor our SLA and service quality?
- Security Questions: do we need security measures like rate limiting and other protection mechanisms and how do we enforce them?
- Backoffice Questions: how do we bill our consumers?
- Marketing Questions: how do we promote our product and how can we drive adoption?

Lists like these go on and can easily be extended. It may seem daunting, but the modeling phase is really similar to building out any software or other product. Many areas require continuous improvement, which is why the APIOps strategy is so helpful, as development and running services is based on or automated with APIs entirely.

Part Two: Developer Relations

Creating a helpful forward facing presence

As your developer portal is the center for API documentation, support, and extra developer resources, it really acts as it's own product. Since this online presence is a main bridge to developer users, this section on **developer relations** covers best practices for

developer center design, developer resource support, code tutorials that instruct API usage, and how to protect your brand identity with platform policy guidelines.

Ingredients That Make Up a Superb Developer Center

What is a consistent attribute across successful API programs? They all have awesome developer portals. Good API documentation is easy to navigate and understand, but the best, shining **developer center** pushes onboarding and actual implementation to new levels of usability, to the point where integrating the API becomes simple as cake — well, at least as simple as technically *possible*.

In the past we've described what good UI design for a developer center *looks* like, but what **information** and **guides** should you prioritize for your developers? It varies from API to API as functional requirements differ, but some tenants hold true across the industry.

Good developer centers allow one to check documentation, get API keys, view sample apps, check uptime, toy around with sample

calls, and manage their account via some sort of dashboard. Many management solutions in the API space have built-in developer portals with these sort of functions, but if you were constructing a quality presence yourself, what key factors would you make sure **not** to leave out?

So, for this chapter we compared 10 successful public API programs to see what attributes their developer centers have in common, and distilled this research into **seven main ingredients** providers should prioritize when creating a developer center. Whether or not your API is free, monetized, or strictly B2B, all these items are going to hold true for any usable kit.

Getting Started Guide

The goal of a **getting started guide** is to make the onboarding process to '`Hello World`' as quick and easy as possible. The best developer centers outline a step-by-step process, guiding a user to **Register** their app, acquire an access **Token**, and use these credentials to initiate their first **API Call**. Twilio is king of the onboarding department — their Quickstart SMS API guides purportedly get developers up and running in a matter of minutes.

At this stage, also overview what other integrations your platform offers. An API will bring the deepest integration capabilities, but perhaps SDKs, webhooks, plugins, or widgets are more easily accessible for some users. If you offer a suite of APIs, consider helpful ways to organize them — the Google Map API picker does this well.

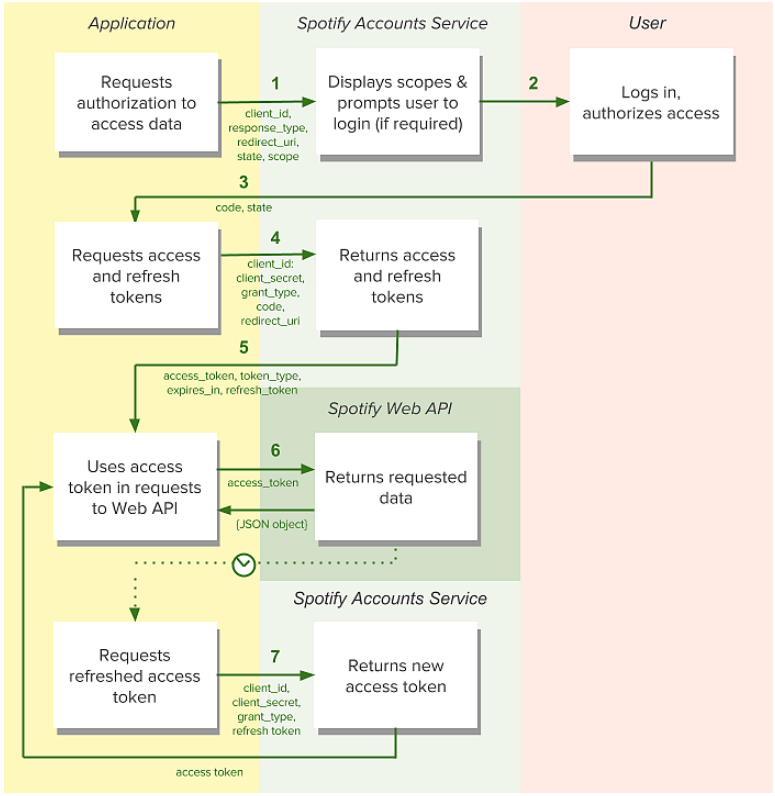
Authentication Guide

All quality platforms dedicate time to explain the authentication mechanisms required to access the API. Too often, authorization

relies solely on **API keys**, but as we've explained before, API keys shouldn't be a sole reliance when it comes to platform security.

Likely, your API will be using OAuth or a combination of OAuth and ID tokens. After the client has registered their application, this process enables an app to authenticate a user on their behalf. Most platforms create a unique guide to OAuth 2.0 for developers unfamiliar with the workflow. Overview the token exchange process in your own words or refer them to other OAuth resources for more information.

For example, Spotify has a very detailed Authorization Guide acting in tandem with their onboarding process. Some developers interacting with end users will require varying scopes of authorization to be granted, all of which flows should be document in a digestible manner.



Spotify authorization workflow for a long-running application

API Documentation

The **reference** is by far the focal point for all API developer centers. Endpoint documentation is the main tool developers will have in understanding precisely how your API behaves. A common approach for structuring readable documentation is a 3-columned arrangement: endpoint on the left, example call in middle, and sample code in various languages on the right hand side.

These specs describe each resource accessible with an `HTTP` verb

(`GET`, `POST`, `PATCH`, or `DELETE`) in technical terms but also offer human-readable description. This means outlining the following:

- Endpoint name (.../`v1/user/data`)
- Describe the endpoint's purpose: what is the data or functionality?
- Describe the parameters used in the HTTP request to query the API
- Show an example JSON formatted response
- Identify the kind of response (String, Boolean, Int, etc.).
- Type of authorization required

Aside from a few exceptions for SOAP/XML, most platforms use RESTful designed web APIs and JSON formatted responses. For rendering readable documentation, it may make sense to use a specific API specification mode. For this you have options: Swagger/OpenAPI spec, API Blueprint, or RAML are the three most used specification formats.

In addition to specifying endpoint functionality, further API behavior specific to design, like **pagination**, **rate limiting**, and **error codes** are documented and made accessible from the developer portal menu on these developer centers.

Testing Environment

The next pixel in our image of the perfect API involves having a **demo** of API functionality so that prospective users can instantly understand how the API behaves. This is often an interactive console where sample HTTP requests are made to mock endpoints. Spotify, along with many APIs in existence, offers such an interactive **API console** accessible even before registration.

As with most software projects, debugging is a time-consuming process. Therefore, many also offer a **sandbox** that simulates a product environment using mock endpoints so developers can test their integrations. The Paypal Sandbox for example, is a self-contained virtual testing environment that allows developers to create test accounts for user entities and make mock transactions between customer and app. A user's account **Dashboard** could be used to track integrated apps, sandbox accounts, and live transactions.

For another example of an API demo, take what Postmates, the programmable courier service, has done. They offer an interactive API demo through Postman, a tool for developing, testing, and sharing APIs. You can use the Postman Chrome web app to initiate calls to an API endpoint.

Twitter similarly uses an Apigee test console to demonstrate API behavior. Mailchimp takes a seperate route — their API playground is not a sandbox, in that calls made using the account holder's API key **do** tally on their account usage report. If you chose this method, communicate this stipulation openly to consumers.

Developer Resources

Developer resources are additional tools that aid the API integration experience. This includes **code tutorials**, **sample code**, or **Software Development Kits** for integrating an API in the programming language or OS of choice.

Alchemy API, for example, has specific guides for consuming their REST API in Python, PHP, Ruby, and Node.js. Often community maintained libraries emerge to consult programmers in their language of choice, but taking ownership of the unique cases where users interact with your REST API from the onset establishes trust — maintaining your own code libraries and workflows helps ensure consistency across the platform.

Support Channels

Great support is a crucial and all encompassing tenant to many successful API programs. Below we categorize the type of support offered by starling developer centers into two groups: **status channels**, and **human support**.

API Status Channels

Actively maintaining the following information is necessary for any platform support, as it helps a prospective user gauge current status, and an active user respond to updates. These ingredients are prominently displayed throughout popular, high-use API programs:

- **Uptime**: details like percentage uptime, response time, and history of past incidents.
- **Changelog**: timeline of changes made to the API.
- **Issue tracker**: feedback mechanism to track issues and suggest changes.
- **Versioning**: If you plan to version your API (v1, v2, v3….), include the historic API documentation, and plans for future

updates. Communicate a deprecation policy from the onset, and clearly denote when any new changes will go into effect.

Mailchimp updates their developer userbase with a bold community announcement

Human Support

Static pages can be useful, but when was the last time an FAQ actually catered to your unique technical dilemma? The best developer portals offer instant help through human-human support methods.

We write a lot on **developer relations**, and for a good reason; increasing the positive experience a developer has with your API is absolutely paramount. It's no wonder that active engagement on **Stack Overflow**, a **Google Group**, **Developer Twitter** handle, and of course a **Developer email** are tools used by most on our top 10 list. Surprisingly still, many big name APIs could do their developer relations a lot better if they embraced instant chat windows with provider support representatives.

Google's comprehensive developer outreach strategy

Google has this department on lockdown. They have an active Issue tracker, Stackoverflow forum, Github, dedicated developer Twitter account, and **Developer blog**. It's also important to also have a feedback mechamism in place — know what questions to ask, how to ask them, and iterate on the information you recieve.

Platform Policy

Have. A. Readable. Terms. of. Use. Period.

Yes, platform policy legalize in this setting will likely be lengthy. Let the lawyers do their bidding, but when they're finished, summarize the main **Restrictions** and **allowed API use cases** in a bulleted list that users will actually read.

Another thing well-established developer centers do well is protecting their brand identity — in fact, extending brand image may be the sole business advantage to exposing a free API. **Brand guidelines** often specify required conditions for naming, logo placement, color pallette, and more. Spotify, for example, even specifies the padding between their logo and app foreground space to ensure their identity is spotlighted.

Spotify's brand guidelines

Cater Your Home Presence to Non-Developers Too

As the API economy grows, so will the general public awareness of them increase — especially now that IPOs from CA Technologies, Apigee, and Twilio are bringing API strategy into the public domain. Therefore, web presences must evolve to meet new audiences. Twilio recognizes this, and offers a rebranded home page entitled "Not a Developer" with helpful resources for non-devs, as well as options for working with partners.

General use cases that display the end consumer experience are helpful because they inspire entrepreneurs. In the same vein, citing

sample apps that are already using your API in the live establishes credibility, acting as testimonials. Uber does both well, with hypothetical example integrations and live apps as samples:

Potential Uber API use case

Establish credibility with example API consumers

Don't assume your visitor is completely accustomed to lingo. Concepts relating to APIs — take HATEAOS — may appear foreign to even some experienced programmers. Adam Duvander recently went as far to say that APIs are mainly for "non-developers with a business problem".

Thus, making these front-facing entities accessible is an important trick to bringing APIs to the masses. Pitney Bowes showcases their suite of APIs very well to non-programmers, with digestible video descriptions for each specific API product.

Final Thoughts

These seven concepts are arguably the bread and butter to sustaining any developer user — the building blocks to creating a consumable API. Of course every SaaS business will have it's own requirements that will require a unique perspective.

What other ingredients must your public-facing developer portal have? If you are implementing these seven general concepts, than you will likely require some sort of **account dashboard** for account management like billing and usage monitoring. For all the ingredients mentioned, carefully worded page descriptions using targeted

keywords can help optimize your API home page, making it more discoverable for search engines.

In this study we reviewed what some of the developer darling API programs have already done to structure their service — but your case is unique. Prioritize what tools *your* unique consumers need, and build out intelligent developer centers that both inform and inspire **creativity**.

Crafting Excellent API Code Tutorials that Decrease On-boarding Time

The success or failure of an API program is made by any number of things: The business value of the product or platform the API exposes, the ease of use of the API, or the number of ways it can be used. However, core to getting an API program right is **delighting the developer** by ensuring they have what they need to get their job done. At the heart of this effort are **code samples**, cookbooks and **code tutorials** that provide this knowledge by example. An excellent tutorial that decreases the time to get up and running with an API adds massive benefits for all concerned: To the API consumer and their development team, who'll report on how good the API is and how easy it was to create their software integration, and also reputational value for the API provider themselves.

In this chapter we take a look at what makes a **great API tutorial** and what it takes to transfer your important API knowledge to your developer community. Like a story, which has a beginning, middle and end, an API tutorial has three distinct parts that help create a narrative on how to use an API:

- Setting the context;
- Exploring the details;

- Creating an application.

Throughout we'll explore each of these parts in turn, citing several quality tutorials that you can model when constructing your own development guides.

Setting the Context

First and foremost, an API tutorial needs to reassure the reader that they are in the right place: visitors will likely understand what they need to accomplish functionally, but they may not yet be aware of the implications of integrating, and may not realize all your API has to offer.

For example, when choosing between the several Twitter APIs, a developer will need to make a choice between RESTful and streaming versions of the API. Alternatively, different APIs may have different security models and developers will need to be aware of what the impact of implementing one API over another may be.

Setting the **context** means devoting words to explaining to the reader the significance of creating an integration with an API, the skill level and provisioning status required, and what type of behaviors should be exhibited. Though this section won't be exhaustive it still should include:

- **Reinforcement of the subject matter**: The tutorial should first reaffirm what the API does and what the developer can accomplish using the tutorial, with introduction of the core technological themes that will be introduced throughout, including programming language used, level of experience required, and security architecture. For example, this tutorial on Instagram exemplifies this point by keenly focusing the mind on the objective at hand;

- **On-boarding requirements**: The developer needs to know what is required to begin the tutorial. Do they need an API key? Is a user account involved? Is there a generic test account? Are there test card numbers, magic numbers a la the Twilio API? A sandbox with a complete data set? The questions are obviously specific to the API itself, but should be in the form of an easy-to-use checklist with links to both the resources that explain or expand on each point in the list or where the developer can act on the information provided. This Facebook API integration tutorial from Appery demonstrates well the kind of information required, with a series of screenshots describing how to register an application on Facebook (a pre-requisite to making an API call). The Braintree Getting Started guide also provides an excellent example;
- **Runtime behaviors**. Finally, the developer needs to know about any restrictions the API places on usage, such as limiting connections, consumption rates or throttling throughput. Providing this information prior to starting the tutorial will ensure developers understand how such behaviors will affect the design of their application and how to mitigate for them. Again, the tutorial does not need to cover the details in full but should provide links to more information.

Armed with this information the developer should have the necessary context to start exploring the API in detail.

Exploring the Details

With the context set it's tempting to dive straight into creating a solution and letting developers understand the API from the source code of a built solution. However, if an API tutorial is going to achieve the goal of decreasing on-boarding time it needs to be

inclusive and address the majority of audiences; not just those with skills and capabilities to decipher the important information from the code.

Rather, help users **understand the code** with a series of explanatory statements accompanied by code snippets which build up a complete picture of the API integration from nothing to a successful API call. The snippets need not amount to a working application, but if all were pulled together they should amount to something close to a working package, class or script (depending on the technology you're using in the course of the tutorial) — building towards the complete solution discussed in the next section. Developers with more advanced skills are obviously welcome to skip the exploration section and go on to the completed solution.

The snippets themselves should exhibit a number of important characteristics:

- **Explain what packages or libraries are involved**: This will give the reader a better idea of what they need to understand alongside the API or allow them to correlate them with alternatives that they may already use. For example, if you use an OAuth package or library briefly explain why you chose that particular one and the benefits it delivers to developer;
- **Dare to be verbose**: The majority of developers write code and then refactor it to make it as terse as possible: Indeed, it's one of the things that generally puts a smile on a developer's face. However, such terseness can be challenging for junior developers to understand: Moreover, you will be talking to developers in the tutorial who aren't experienced in the language it's written in, but need to use it as a reference for implementing their application in their own language of choice (Whilst some API providers like Twilio can provide tutorials in several languages, not all API providers have the resources available to do this). For example, if a Python list comprehension can be understood better by more developers

through verbose syntax consider doing so:

```
# Less Pythonistic but easier to understand for a non-Pyt\
hon developer

verbose_payments_list = list()

for payment in payments:
    If payment.amount > 0:
        payments_list.append(payment)

# More Pythonistic approach:

terse_payments_list = [payment for payment in payments if\
 payment.amount > 0]
```

- **Comments, comments, comments**: Ensure the snippets contain lots of comments or are well annotated with text that clearly labels the intentions of your code. Continuing the theme of verbosity, comment more than you normally would to guide the reader through each step. A great is example is this Rails tutorial on Stripe integration, which clearly explains the purpose of each snippet.

Creating an Application

The final step in a building a great tutorial is bringing what you've explained to life by creating an **application** that a developer can actually run. Depending on the depth and detail of the solution itself this could be a simple code snippet, gist, GitHub project, or a Docker image that can be executed with minimal effort on the part of the developer. Naturally you are not going to walk through the

entire solution in the tutorial, so make sure you summarize it with a few well chosen **screenshots**.

The bounds of your application are entirely dependent on your imagination, skills and the needs of your developer community, but you should look to do the following:

- Ensure it can be run in a minimal number of steps;
- Like the code snippets, insert a full range of comments and pointers that explain the significance of the implementation;
- Demonstrate a really **compelling use case** for your API.

Ensuring your application demonstrates these features will really help in concluding your tutorial and ensuring all the points you've raised are reinforced.

Final Thoughts

We've explored what makes a great API tutorial and have highlighted the core themes that make it easy for your developer community to comprehend and digest the key information about your API.

It goes without saying that not every API tutorial will exhibit everything discussed here; however, if you focus on the points raised you should succeed in creating a compelling narrative that not only holds the reader's interest but does meet the goal of decreasing the time it takes to on-board into your API program.

What is the Difference Between an API and an SDK?

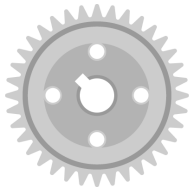
Understanding the difference between an **API** (Application Programming Interface) and an **SDK** (Software Development Kit), and knowing when to provide each, is incredibly important for fostering a developer ecosystem. In the modern development landscape, these two tools and the synchronicity between them are the driving force behind web communicatoin and the implementation of third party APIs.

Accordingly, it helps to know what exactly we mean when we talk about APIs and SDKs. In this piece, we will attempt to create an inclusive definition of both concepts. We will give an example of each, explain how they interact with one another, and find how an API provider can effectively implement one or both of these tools to improve their offering and end developer usability.

Define: API

An API is simply an interface that allows software to interact with other software. This is part of its name — API, Application Programming Interface — and is core to its functionality. Think of an API as a rosetta stone, a tablet by which two vastly different

languages, two different instruction sets, can be translated and transferred for mutual understanding.



APIs come in many shapes and sizes. The browser that a reader would likely use to peruse the Nordic APIs website uses a variety of API sets in order to convert user commands into usable functions, request data from servers, render that data into a viewable format for the user, and validate the performance of their requests.

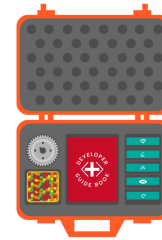
Even something as simple as copying and pasting on a computer utilizes an API. Copying text converts a keystroke into a command, data is stored into RAM on the clipboard utilizing an API, the data is then carried from one application to another using that same API, and finally, data is rendered when pasting using yet another API.

On the world wide web, the API takes on a slightly different function. **Web APIs** allow for interaction between disparate systems, often for specific use cases. For instance, when a user interacts on Twitter, they're utilizing an API to comment, to store their data, to follow a user, to delete tweets, and so forth. Ultimately, a web API is simply a set of instructions, just like the personal computer API, but based in the web space.

Perhaps most important is the fact that APIs allow for consistency. In the early years of programming, the computer was a wild west of commands and instructions, loosely coded and rarely documented. With the advent of modern computing, APIs have allowed for consistent coding in stable environments, allowing for replicable functions to be delivered the same every time the request is submitted with reliability and predictability.

Define: SDK

SDK stands for "Software Development Kit", which is a great way to think about it — a kit. Think about putting together a model car or plane. When constructing this model, a whole kit of items is needed, including the kit pieces themselves, the tools needed to put them together, assembly instructions, and so forth.



An SDK or *devkit* functions in much the same way, providing a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allow developers to create software applications on a specific platform. If an API is a set of building blocks that allows for the creation of something, an SDK is a full-fledged **workshop**, facilitating creation far outside the scopes of what an API would allow.

SDKs are the origination sources for almost every program a modern user would interact with. From the web browser you work on all the way to the video games you play at the end of the day, many were first built with an SDK, even before an API was used to communicate with other applications.

Squares and Rectangles

Part of the confusion behind the difference between APIs and SDKs is the fact that, more often than not, an SDK contains an API. In geometry, "rectangles" is inclusive of both rectangles and squares, whereas "squares" is inclusive only of squares.

The same is true with APIs and SDKs. By definition, an SDK is a kit that includes instructions that allows developers to create systems and develop applications. APIs, on the other hand, are purpose built for an express use — to allow communication between applications.

It should be no surprise then that, when an SDK is used to create an application that has to communicate to other applications, it includes an API for this functionality. Inversely, an API is used for communication, but cannot be used solely to create a brand new application.

Another way to understand this is to think in terms of houses. APIs are telephone lines, allowing for communication in and out of the house. The SDK is the house itself and all of its contents.

Examples

Luckily, we have a great example of the difference between an API and an SDK in the Facebook suite of solutions. Because this suite provides tools for both active users and developers, it includes both an API and an SDK, each with different functionalities and use cases.

Facebook APIs

Used internally and with third party application providers, the Facebook API allows for communication across the wide Facebook social platform, and utilizes the social connections and profile information data points of every Facebook user to conduct application functions.

These functions include pushing activity to "news feeds" and "profiles" on the main Facebook site, but also include third party application functions such as registering for external sites and subscribing to media outlets. Page, photo, event, friend, and group data is collected and collated and used to form meaningful and useful connections that increase the extensibility of the service.

The API also allows for the limiting of this data sharing on a per user basis, allowing for users to limit their profile content and the use thereof. This integrated security allows for extensive use of multiple data points and resources while still maintaining high privacy and security levels.

The functionality of this API extends beyond internal usage, however. One of the greatest strengths of the API is the tie-in to the Graph API Explorer. This service allows for the observation of relational data between users, photos, accounts, feeds, and more. This sort of analytic generation is incredibly powerful — as we've previously stated, metrics is one of the most powerful assets an API provider can have.

```
1  GET graph.facebook.com
2    /me?
3      fields=albums.limit(5){name, photos.limit(2){name, pi\
4  cture, tags.limit(2)}},posts.limit(5)
```

Here we see a sample API issuance. In this call, the API is used to request a user's photo, the URL the photo generates, and all the people tagged in the photo. While this is a rather simple use, consider the possibilities — a restaurant manager or even host could use this API call to generate a list of users in a photo shot at a specific engagement, generating a list of social accounts they can reach out to for further publicity or promotion. Try doing that without the API!

Graph API isn't the only API in town, either. Facebook also provides the Marketing API, designed specifically to allow brands

to craft engaging and effective social campaigns for their products.

This API not only shows how powerful the Facebook platform is, but how powerful properly structured API design can be. Because the Marketing API primarily drives advertising campaigns, the structural design reflects this purpose, and is laid out in such a way as to inspire proper campaign design as a secondary benefit.

The benefit of this structure can be seen in how the Marketing API deals with optimized effective CPM. CPM, or "cost per mille", is a concept wherein actions are given a value and interaction is given a cost. These costs can be best optimized by the advertiser to prioritize marketing goals and deliver ads in the most effective and efficient way possible.

```
use FacebookAds\Object\AdSet;
use FacebookAds\Object\Fields\AdSetFields;
use FacebookAds\Object\Values\BillingEvents;
use FacebookAds\Object\Values\OptimizationGoals;

$adset = new AdSet(null, 'act_<AD_ACCOUNT_ID>');
$adset->setData(array(
  AdSetFields::NAME => 'My Ad Set for oCPM',
  AdSetFields::BILLING_EVENT => BillingEvents::IMPRESSION\
S,
  AdSetFields::OPTIMIZATION_GOAL => OptimizationGoals::LI\
NK_CLICKS,
  AdSetFields::BID_AMOUNT => 150,
  AdSetFields::CAMPAIGN_ID => <CAMPAIGN_ID>,
  AdSetFields::DAILY_BUDGET => 1000,
  AdSetFields::TARGETING => array(
    'geo_locations' => array(
      'countries' => array(
        'US'
      ),
    ),
  ),
```

```
));

$adset->create(array(
  AdSet::STATUS_PARAM_NAME => AdSet::STATUS_PAUSED,
));
```

In this example, the Marketing API has created a campaign that can be bid upon under constraints set up by the campaign budget values. This dynamic bidding makes for a very powerfully optimized system that captures the highest-value impressions and establishes a value that makes sure the ROI ratio of input to expense isn't exceeded.

A dynamic bidding system allows for the best return on the dollar — this is the power of a properly crafted API, allowing for complex interactions and manipulations above and beyond what any portal or internal page could deliver on its own.

Facebook SDKs

We can see the main difference between SDKs and APIs in their expressed functions. While the previously mentioned APIs are clearly designed for interaction between applications and campaigns or other applications, the SDKs provided by Facebook are clearly designed for the *creation* of these applications.

Let's look at the Facebook SDK for iOS. Designed specifically to allow for the development of Facebook applications for iOS, the SDK is fully featured, allowing for a multitude of functions to be defined and called.

As a basic example, the following code snippet is from the SDK reference guide for iOS:

```
1  //  AppDelegate.m
2  #import <FBSDKCoreKit/FBSDKCoreKit.h>
3  - (void)applicationDidBecomeActive:(UIApplication *)appli\
4  cation {
5    [FBSDKAppEvents activateApp];
6  }
```

This example allows for the logging of application activations, and is thus one of the more basic possible examples to provide. Nonetheless, one can see the difference between an API and SDK in the basic structure of the calls. While the API calls existent sources and functions to perform an action already defined, the SDK is used to first define this function and to create a way to call the source and function.

The Android SDK is much the same, but translated into the language of the Android OS. Further changes can be seen in the Web SDKs, such as the JavaScript SDK, which utilizes JavaScript to perform the same basic function building as the iOS and Android SDKs.

The SDK is the building blocks of the application, whereas the API is the language of its requests. This is an apt description, "building blocks", which is made abundantly obvious when one looks at what an SDK contains. **Libraries** from which to build functionality, **code samples** for increased understanding and easier implementation, and **references** for easy linking and explanations — without any of these, an application or service might be functional, but it certainly would be severely hampered.

Of key interest is the fact that our analogy still holds — the API references existing functions and calls, while the SDK calls the API. See this following example code from the SDK reference guide:

```
FB.ui({
  method: 'share_open_graph',
  action_type: 'og.likes',
  action_properties: JSON.stringify({
    object:'https://developers.facebook.com/docs/',
  })
}, function(response){
  // Debug response (optional)
  console.log(response);
});
```

This code creates a share dialog that pops up over the application page when an action is performed, and publishes an Open Graph action which can then be tied into the greater ecosystem and used to generate complex relationships and metric data.

Apples and Oranges

Realistically, the comparison between API and SDK is often confusing only because of how far they overlap — a problem only complicated with the addition of new methodologies for organization and segmentation, such as Docker containers, which require their own specific API and SDK documentations. To simplify the concept, remember the following:

- SDKs usually APIs; no APIs contain SDKs.
- SDKs allow for the creation of applications, as a foundation allows for the creation of a house;
- APIs allow for the functioning of applications within the SDKs defined parameters, like the phone lines of a house.

With this basic understanding, and a few key code examples, the difference between SDKs and APIs should now be obvious.

Developer Resources: SDKs, Libraries, Auto Generation Tools

Shipping a great API isn't just about exposing an endpoint in a RESTful manner. Yes, plenty of developer users will be fine making HTTP requests, but for some, that is not enough. Whether community curated or vendor supplied, code libraries are often created to help extend an application programming interface — API — into **specific languages**. These libraries help onboard third party users and demonstrate that you're willing to work on *their* home turf.

In this chapter we examine the importance of helper libraries and how they affect your audience. We determine what languages you should try to support, backed by research into API library language trends across various industries, tuning into what players like Twilio and Stripe are doing right. We'll describe the difference between an API and SDK, and see how we can automatically generate code libraries using APImatic and RESTUnited.

What Are Helper Libraries?

Helper libraries are **developer resources** that allow a developer to call an API within the language they are most familiar with, whether it be C#, Ruby, Node.js, Scala, Go, Clojure, Objective-C, or

many others. Helper libraries, code libraries, wrappers, Ruby gems, Python bindings, Node.js modules, and Software Developer Kits (SDKs) all help in this regard.

Why Not Just Let Them REST?

Usually, the Uniform Resource Locator — **URL** — is a friend, allowing access to a specific resource and enabling software to leave isolated environments. The existence of URLs have enabled the Internet, along with an incredible array of products to thrive. However, accessing resources from something other than a web browser can be a complete pain. A cURL POST request to the **Twilio API** asking to send a text message, for example, may look something like:

```
$ curl -XPOST https://api.twilio.com/2015-11-09/Accounts/\
AC5ef823a324940582addg5728ec484/SMS/Messages.json \
        -d “Body=Jenny%20please%3F%21%20I%20love%20you%20<3” \
        -d “To=%2B1415924372” \
        -d “From=%2B424572869” \
        -u ‘AC5ef823a324940582addg5728ec484:{AuthToken}’
```

This type of request is machine readable, but it involves a lot of encoding and rare symbols to print a relatively simple phrase. Enter REST — a way of building web services by following a set of specific constraints between consumers and providers. The specification describes ways to interact with URLs in a handy and comprehensible method.

However, even if an API is RESTful, the Ruby code required for a POST request to initialize a URI is still clunky. Here is a POST request written in Ruby to access the Rackspace API:

```
uri = URI.parse(‘https://auth.api.rackspacecloud.com”)
http = Net::HTTP.new(uri.host, uri.port)
http.use_ssl = true
http.verifiy_mode = OpenSSL::SSL::VERIFY_NONE
request = NET::HTTP::Post.new(‘/v1.1/auth”)
request.add_field(‘Content-Type’,’application/json’)
request.body = {‘credentials’ => {‘username’ => ‘username\
’, ‘key’ => ‘key’}}
response = http.request(request)
```

All this is doing is establishing a connection and verifying credentials. However, if something goes wrong with this request, results things can easily become messy. Most people are aware of HTTP 404 error code (Not found), but there are over 80 other total HTTP error codes specified, and each API's response behavior is different according to how extensive the provider has chosen to be. Dealing with **error codes** is a pain, so software engineers have often chosen to abstract and wrap in a method, another reason for adopting helper libraries.

Data Problem

API providers also need to embrace conventions, and consider what data formats are being used. For example, when you have to send dates, what date standard are you using? Perhaps your service uses a different log than base 10. When users are browsing your documentation, they need to understand it in the scope of *their own* languages.

This is where helper libraries come into play. Just exposing an HTTP resource is fine, but we need to give developers a bit more help. For example, using a Ruby gem for the Twitter API:

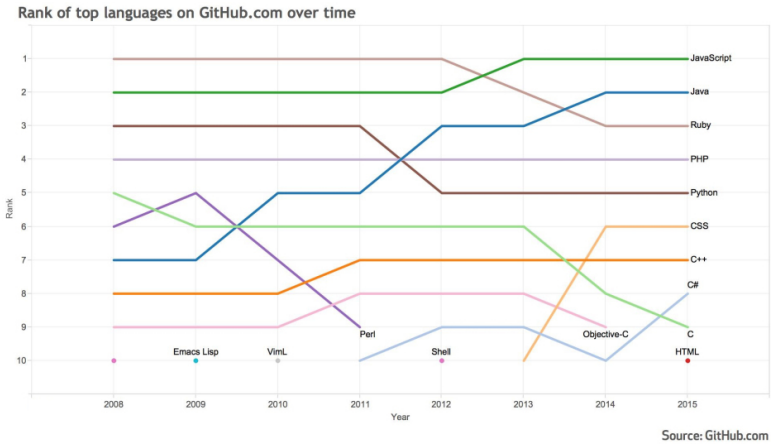
```
client = Twitter::REST::Client.new do |config|
  config.consumer_key        = "YOUR_CONSUMER_KEY"
  config.consumer_secret     = "YOUR_CONSUMER_SECRET"
  config.access_token        = "YOUR_ACCESS_TOKEN"
  config.access_token_secret = "YOUR_ACCESS_SECRET"
end

client.update("I'm tweeting with @gem!")
```

This simple code found on Github initializes the helper object, and uses it to send a tweet "I'm tweeting with @gem" to Twitter — library. It handles everything we don't want to care about.

Programming Language Trends

So you're convinced you need to construct helper libraries for your developer consumers — but now comes the moment of truth. How do we decide which languages to support? Do we address 30+ languages?

To narrow things down, let's take a look at a couple sources. VentureBeat cites the top 10 languages used on GitHub as:



Adam Duvander , using ProgrammableWeb data, has cited the top languages used for API helper libraries as:

1. PHP
2. Python
3. Ruby
4. .NET / C#
5. Java
6. Perl
7. ColdFusion
8. Node.js
9. ActionScript

Discover What Languages Your Consumers are Using

Regardless of industry trend, the most important step is to listen to your developer consumers — they have likely already told you what languages to add. Discovering this is relatively simple — perform

monitoring of the HTTP requests to your server. If you are logging those requests (hopefully this is already the case), then you can easily figure out the user agent for those requests to determine what programming language they are using. Should look like this:

```
curl/7.31.0
PHP_Request/curl-7.15.5
Java/1.6.0_31
```

Developer users are your customers. Always respect the fact that your users code, meaning they *could* always do your job and write *their own* API if need be. Especially in the case of Twilio, their developers are consuming their API as a time saving mechanism, so the developer experience needs to be as easy as possible. Log activity, determine what your consumers are using, and create developer resources accordingly.

Who Should we Model?

Here is some self-collected data on what languages significant API vendors support for libraries/SDKs across various industries. Twilio has a great outreach library program, as does Stripe. It's important to note that many helper libraries live in the **open source** and **Github** realm — meaning that many libraries are maintained by the surrounding community, perhaps not even touched by the provider themselves.

Provider	Official	Community	Helper Library Home Page
Twilio API	PHP, Ruby, Python, C#/.NET, Java, Node.js	Go, JavaScript, C++, Scala, Perl, Erlang, Adobe ColdFusion, Adobe, LiveCycle, Mule ESB	Libraries
Twitter API	Java, Android	ASP, C++, Clojure, ColdFusion, Go, Java, Node.js, Lua/Corona, Objective-C, Perl, PHP, Python, Ruby, C, .NET, Erlang, Java, many more.	Twitter Libraries
Box API	Java, Python, .NET, Ruby, Mobile (iOS, Android, Windows)	APEX	SDKs
eBay API	.NET, Java, Python		(eBay SDKs
FitBit API	Java, PHP, .Net		Dev Center
Square API		Java, Objective-C, JavaScript, Ruby	Open Source Libraries
Stripe API	Python, Ruby, PHP, Java, Node.js, Go,	Ruby, PHP, C#, ColdFusion, Perl, Clojure, Scala, and more.	Internal API Libraries

HTTP is Language Agnostic

With HTTP you can browse the web, make a REST request, hack Snapchat, or make SOAP requests. But helper libraries need to be **Language Idiomatic**. A Python developer doesn't care if your libraries behave the same way — they must behave in a way that makes sense for that specific language.

Python example using Twilio's REST client:

```
client = TwilioRestClient("accountSid", "authToken")

message = client.messages.create(to="+12316851234",
                                 from_="_15555555555",
                                 body="Hello there!")
```

On the other hand, C# needs to *look* and *behave* like C#:

```
var twilio = new TwilioRestClient("accountSid", "authToke\
n");

var msg = twilio.SendMessage("+15554446666",
                             "+15557778888",
                             "that was easy!");
```

5 Tips for Helper Library Design

- **Keep Vocabulary consistent with documentation**: Though client libraries need to be catered to the doctrine of the specific language in mind, they still must be consistent with your REST API documentation and parameter vocabulary. These terms should be coherent across all resources that integrate with your platform. Being idiomatic with languages is important, but terms and features need to be consistent.

- **Surface Area**: How much of your API should your helper library cover? The answer is **all**. Wawra recommends to marry the API with the web representation. This means that all functions your website can do, your API should handle as well, and in turn your helper library as well. If you are using an API-first building strategy, this becomes a *lot* more intuitive.
- **Simultaneous platform-wide publishing and updates**: There is no point in versioning your API or adding new features if your helper libraries are not in sync with updates. The rule of thumb is, for your officially maintained libraries, release changes to API functionality across documentation and all libraries **simultaneously**.
- **TEST**: Treat your helper library as a product, and test just as vigorously as you would your API.
- **Open Source**: It's not a strategy for every company, but API developers love Github, and providers can benefit from the community aspect of open-sourcing their library. Creating a community around your APIs and libraries encourages improvement to the code.

Last Line of API Code: Your API is Never Really Finished

At the end of the day, the helper library that you ship will live on the user's app. Since it is in the user's app — you have limited control. However, developers are always creating new apps, meaning that your API is never truly finished. Having an agile mindset with constant iteration is crucial.

A Human's Guide to Drafting API Platform Policy

There are some languages in the world that are tough to master and for your average application developer none more so than *Legaleze*, a language so inaccessible that only a learned few can speak it fluently. The diction and terminology is generally so difficult that most don't even attempt to learn: When presented with an API's terms of use or **platform policy**, most developers simply click 'I accept' and get on with coding.

OK, so this description of the language found in legal documents is meant to be tongue in cheek, but it's amazing how organizations in general expect non-lawyers to make informed decisions on whether or not they can accept lengthy terms of use. Deciphering such documents often involves collaboration between a lawyer and a software developer, which is less than ideal given there is room for error or misinterpretation as one attempts to inform the other (lawyers who are also software specialists being a rare, expensive commodity). There are even websites such as Terms of Service; Didn't Read dedicated to evaluating the terms of use of major web apps to help users combat what TOSDR calls "the biggest lie on the web." Failing to digest the small print correctly could result in the organization falling foul of a clause in the terms of use that could be passed on directly to their customers.

The blind acceptance of an API's terms and conditions presents

an interesting quandary for anyone running an API program. One could simply take a *caveat emptor* view and not lose any sleep over whether a consumer reads the small print. However, one could also view incomprehensible terms and conditions as a lost opportunity: If an existing customer disregards the small print, are prospective customers being put off? In this chapter we've put together a practical guide on creating an **API platform policy**, including what items to include and how to convey the most vital information. By the end of the article you should have an idea of how to present your policy so your customers actually read and understand it, with the goal being to ensure an application developer can move from discovery to development as quickly as possible.

Key Themes

Nearly every API provider publishes one or more legal documents that define the terms and conditions under which their API can be used. These documents can include any of the following:

- Terms of Use;
- Privacy Policy;
- Terms of Service;
- Cookie Policy (for a developer portal).

Legal documents such as these are common in all sectors of technology, and collectively they define a contract between the API provider and consumer, specifying the legal obligations of each upon each other. Whether you sign a piece of paper or simply start using the API, as a consumer you are bound by this legal contract. One could simply say "why bother with contracts", but this is not a tenable position: The need to define legal contracts for using an API covers a myriad of different subject areas including intellectual property, data protection, regulatory obligation, and of course commercial agreements where access to the API is paid for.

When drafting an API platform policy of your own it's important to understand that these documents tend to cover several key underlying themes, namely:

- Defining responsibilities;
- Setting expectations;
- Describing good behavior.

Given the realities of legal systems it's difficult to make these points bubble up to the attention of the developer (as Stripe reflects upon in their terms of use. However, it's paramount that these themes are drawn out to pique the interest of the API consumer and ensure they've either consciously accepted the terms and conditions they are signing up to, or have drawn it to the attention of their organization's legal team.

Defining Responsibilities

Almost all legal documents start with a definition of the responsibilities of all the parties involved in the contract and API platform policies are no exception. A traditional legal contract will contain several sentences with prescriptive definitions of terms like 'You' and 'Us': If possible, keep these terms in core legal documentation but summarize the key points with sufficient brevity to keep the reader's interest. The level of brevity required is demonstrated well by Instagram, who use a numbered General Terms section to clarify for the API consumer what they absolutely need to know whilst keeping the more lengthy terms of use in the background. It's important to "dumb down" the key points in this way to highlight the responsibilities that have the greatest impact on the API consumer. As we've seen before, Instagram actively pursues taking down consumers who breach their terms, therefore they phrase their agreement in plain English to ensure consumers know **exactly** what they can and cannot do with the Instagram API.

Setting Expectations

Once a clear map of responsibilities is created, the API platform policy should then describe what the API consumer should expect of the API provider. Setting expectations is synonymous with a **service-level agreement**, but again the goal should be to define it such a way as to make it straightforward for the API consumer to digest. This is almost the most important part of the policy, as the API consumer may need to pass these expectations onto their customers: If an API provider defines the availability of their API as being 99%, for example, this implicitly becomes part of the terms and conditions between the API consumer and their customers.

It would be easy to create a huge compendium of the aspects that API providers should take account of when setting expectations, but a brief list would include:

- Availability of the API;
- Types of support available and their responsiveness (in high-level terms, not email, 'phone numbers, etc.);
- Access to developer portal, sandbox environments, refresh of authorities and entitlements (API keys, OAuth client IDs, etc.);
- Requirements for certification and on-boarding (screening, compliance checks, certification of implementation, etc.);
- API functionality as referenced in documentation.

Now that we know what to expect of the API provider, the final step is to define what is expected of the API consumer. This next part describes what is acceptable and not acceptable behavior, and communicates the actions the API provider might take should these good behaviors not be observed.

Describing Good Behaviors

With a clear picture of responsibilities and expectations, the API provider should then define what their expectation of "good behaviors" are (which may be tied into a commercial agreement with paid-for APIs). Good behaviors fall into two groups: **application behavior** and **consumer behavior**. These groups tend to reflect the interest of the application developer and legal team respectively.

Application Behavior

Application behavior refers to how a software program that consumes the API is coded to ensure it does not breach what it is responsible for or attempt to exceed expectations. The behavior expected or forced on the application can be manifested in a number of ways:

- Enforcement of rate limits to ensure "bursts" do not exceed a given throughput in a defined period;
- Enforcement of quotas with an upper limit on the number of API calls over a given period;
- Dropping long running HTTP connections or blocking new connections where an upper limit has been breached.

API consumers need to ensure they design and implement their applications to be cognizant of these good behaviors. Without doing so, they run the risk of providing a poor user experience and decreased functionality to the users of their application. Moreover, if an application repeatedly breaches the good behavior policy then access to the API could be rescinded completely.

Consumer Behavior

While application behavior can be coded, ensuring good API consumer behavior is a bit more tricky: It involves a human being making a decision about how they are going to use an API that may not concur with the terms of use defined by the API provider. Consumer behavior generally (but not always) comes as a result of a business decision and should not be made in isolation by a software developer but *in conjunction* with the product's legal and commercial teams.

The majority of humans are generally subjective in their points of view, so it helps if API providers are prescriptive in what an API consumer can and can't do with their API with a summarized list in the manner of Stripe. API providers can take breaches of their terms of use very seriously, as with the 9 month take down of Politwoops by Twitter showed: It's therefore important for both the provider and the consumer to be 100% clear on both the definition of consumer behavior and the consequences of not observing it.

Final Thoughts

Making an API policy interesting, relevant and to the point is a goal that any API architect or product manager should take seriously. Whilst the lawyers will always have their day in wrapping up your policy in jargon, you should actively shape the output, ensuring the key points are easy to digest, understand and act upon. If possible reflect the following in your policy:

- Produce an accurate summary of the key points of the legal documents, covering responsibilities, expectations and behaviors;
- Ensure the summary is in easily accessible language that anyone can understand (for a great example see the Google Developer Guidelines);

- Clearly explain the mitigations or consequences that will be apparent should any terms of use be breached.

Taking this action can only make your API more accessible to the developer community: As the API economy continues to grow with increasing competition between different API providers, such an approach can only, *prima facie* (as the lawyers would say :smirk:) help develop competitive advantage for your API in the marketplace.

Creating A Brand Guide for Your API Program

What you let others do with your **image** is just as important as what you let them do with your **API**.

Hosting a free-to-consume public API is a powerful tool for expanding your functionality and branding into entirely new networks. But many programs lack comprehensive and digestible guidelines to protect their brand. Without a developer **Branding Guide** you leave your company name, product name, logo, style, and message open for reproduction in untasteful or even illegal ways.

For free APIs, data and functionality is given to third party developers to integrate into their apps in exchange for outsourced R&D and increased marketing. Attribution is commonplace, and at times *desirable* for third party applications as it helps establish *their* credibility with end users. Therefore, most platforms establish some sort of branding guideline for the API consumer to follow based on their platform policies. This is unique from a press kit or internal design style guide as it is **developer focused** — similar to a partner or affiliate marketing guide — making it an important asset to carefully prepare and maintain.

In this walkthrough, we'll lay out typical brand guidelines that providers offering a public API should use to ensure their company logo, style, and message is reproduced in the best way possible on third party channels. It's ok to be specific throughout, as specificity will help your developers in the long run.

We'll help decipher a bit of design jargon so that any provider, small or large, can offer a **branding guide** and **asset package** that controls *how* developers implement the API provider's company image within their apps.

Platform Strategy Dictates Brand Requirements

First, keep in mind that your branding guidelines are part and parcel of your greater platform policy and business model. The monetization strategy you chose will likely impact the level of attribution you require of your developers. If the API is free use, high attribution will be necessary so that your marketing return covers the opportunity cost — if the API is pay-to-play, the income generated will hopefully outweigh the need to demand much attribution.

Strict brand guidelines may be a huge turn off for some developers looking to leverage your API. Imagine if every programmatically triggered Twilio SMS was accompanied by "Sent by the Twilio API." That would be annoying, and would de-legitimize the developer's app, and confuse the end user.

Likewise, if your SDK forces an in-app user to leave the original app to a different channel dominated by the API provider's branding, again, you're doing things wrong. Especially if developer consumers are intending on monetizing *their* app, they will not likely sacrifice their image for your functionality. Thus, a happy medium needs to be forged.

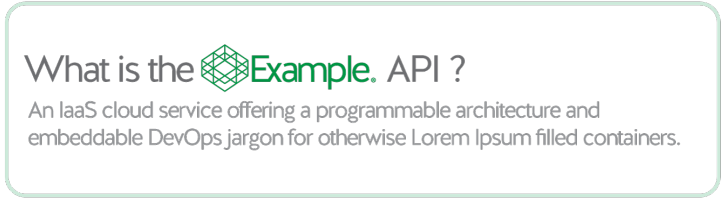
Brand Guide Components

Now we'll overview the main steps to create a brand guide that helps free-to-use APIs control how branding — your message, name, logo, color palette, placement, attribution, and more — appears on third party apps

Message

Branding first starts with creating a story. The tone, perspective, and voicing that make up your company's **copywriting** should also resonate throughout your API's home page, developer center, and descriptions of example API use cases.

An API brand guide should therefore define the platform functionality so that if developers describe the integration on their own channels, they explain it correctly. It doesn't hurt to also explain the **voicing** or **tone** the provider company adopts to communicate with its userbase.



Naming

Most closely guarded is a company's registered name or trademark. Third party apps shouldn't reproduce the provider's name or resemble it too closely, therefore API Terms of Use typically protect brand

identity by listing example names or phrases that are not allowed. Thus, your guide should:

- State your company name and proper capitalization — **Example API**,
- Not allow portmanteaus that include some portion of the provider's name,
- List other acceptable/unacceptable naming conventions for services your consumers may also reference.

In general, it's a good idea for developer consumers to give their service a unique name that is distanced from the provider's. Above all, third party apps shouldn't confuse or mislead users regarding your affiliation — the brands must stand alone.

Attribution

If the API provider's brand must consistently accompany third party app behaviors, be explicit in its implementation. Should branding link to the API provider's home page at all times? Remember when, where and what:

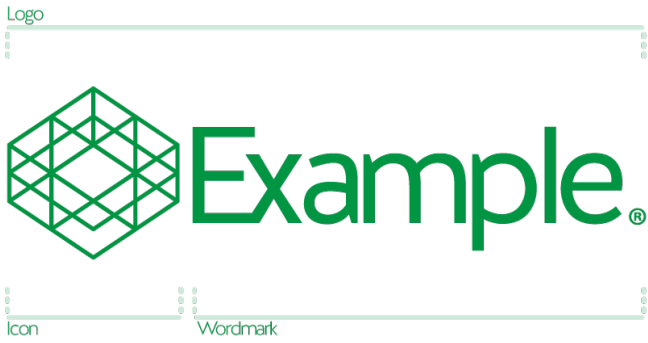
- Communicate *when* your brand should appear: Many APIs require branding at all times the API is used to access the provider's data. A more relaxed method is to require your logo at initial sign-in, and/or in the app's About page. A third party app may also chose to reference the API provider as credibility. Even in that scenario, you can control how your branding is presented.
- Communicate *where* your brand should appear: Perhaps the API is called to query a database, populating a search field. In that case your company logo should be located near the search field. For other situations, delineate where on the user interface your branding should appear — Top Left, Top Right, Center, Bottom Left, Bottom Right, etc.

- Communicate *what* exactly should appear: Clearly specify what assets and attribution copy needs to be used and under what circumstances — "Powered by Example API" badge, "works with Example API", etc.

Giving credit is different than implying an endorsement, so the attribution shouldn't suggest there is a partnership between the third party developer and the API provider. So this is clear, the provider shouldn't be the most prominent element or logo on the Web page, nor the only branding on the page. The best guides lucidly communicate what level of attribution is required, and use graphical aids whenever possible to help illustrate the point.

The Logo

A tech company's **logo** will often come in varying iterations. Many logos include an icon as well as a wordmark, and app icons are often represented in isolation. Clearly delineate what your logo is exactly —the icon, wordmark, the typography used, the relative measurements, and color palette.



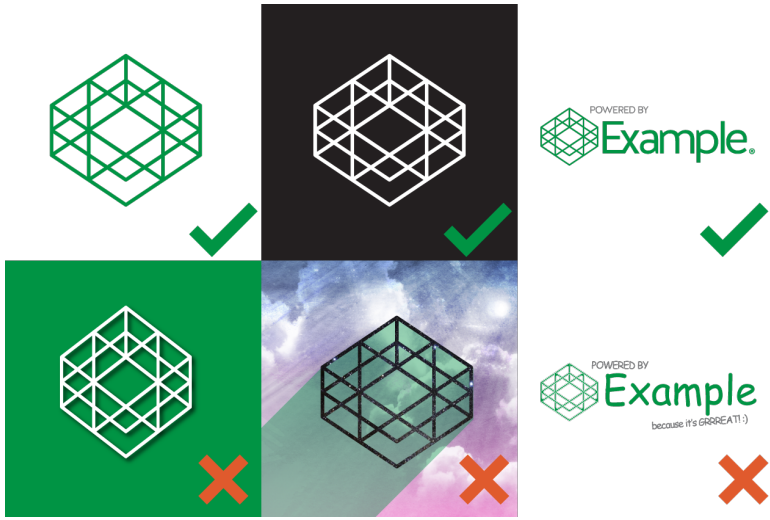
For your logo, consider adding stipulations for your branding guide like:

- Describe relative spacing between icon and wordmark
- Establish a minimum acceptable size — i.e. 80px or 25mm for print.
- The third party logo should not resemble the provider's style in a similar way
- Ensure your logo always links to a specified provider page
- Icon can exist without wordmark, but the wordmark shouldn't exist without icon
- Don't add a tagline or motto to the existing logo

If you have logo variations, note which is the best possible brand representation. Logotypes are most accurate and should be used whenever possible, but icons/glyphs are acceptable after brand has been established or if there is limited horizontal space in the UI.

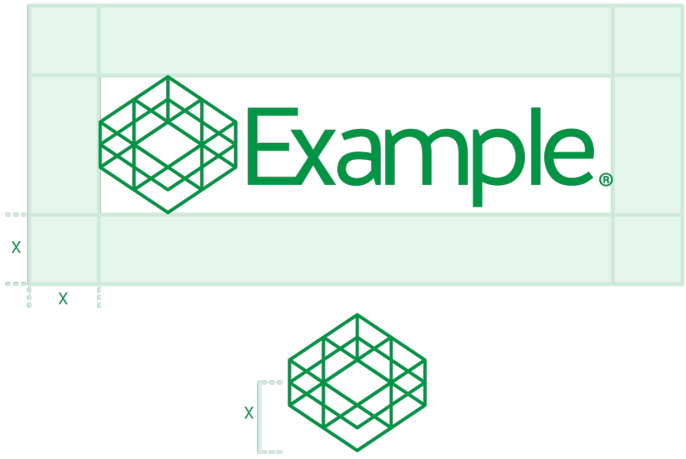
Make sure to note that it's not acceptable to alter the logo in any way. Embellishments like drop shadow, distortion, or clip masking, as well as inserting the logo in front of busy backgrounds or complex patterns should all be discouraged. So that the logo still

remains visible with bright and dark backgrounds, have both a colored, white, and grey/monochrome logo matching your platform's color scheme. It's pretty easy to dream up of some hideous **logo misuses**:



Exclusion Zone

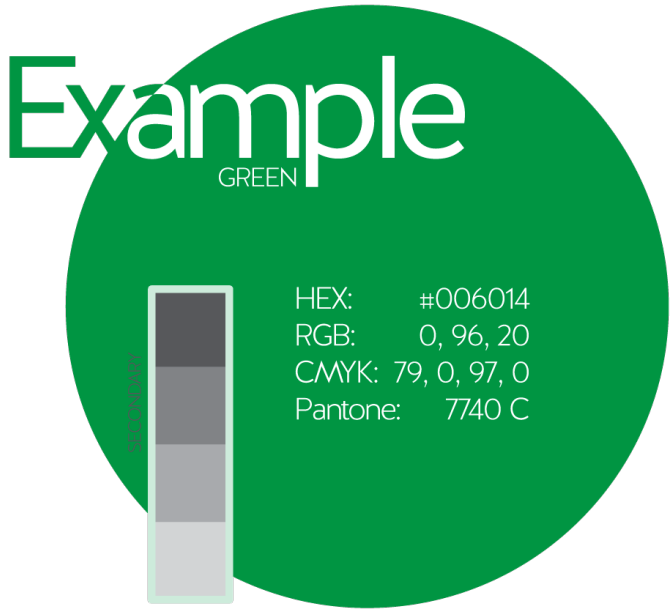
The **exclusion zone** is the minimum padding, or white space around the logo that ensures the logo is prominently displayed. As pixel distances are relative, communicate this minimum safe distance in relative terms — for example, make the exclusion zone half the size of the icon diameter.



This clear space is important to protecting the brand, as it makes sure that third party company logos, phrases, or other graphical elements aren't displayed too close to your assets.

Color Palette & Typography

What are the colors used within your logo and throughout your branding? Represent your default logo color for a light background, and preferred color for a dark background, and list the specific color hexadecimal as well as RGB, CMYK, and Pantone indicator for each. Our hypothetical API, for example, has a nice evergreen color. Similarly, note the font used on your logotype, headings, and in body text for your general communications.



Brands take time to choose the right colors that match their service image. Unfortunately though, considering the varying browser color settings and screen calibrations across thousands of different devices, it's hard to match colors exactly. Therefore, have **fall back color options** for devices or screens with less optimized color grading. As long as your identity is obvious and color not significantly distorted, you should be fine.

Formatting Your Design Guide

The **presentation** for the guidelines is JUST as important as their meaning. Organize a brand guideline PDF, and have easily accessible digital logo assets.

Brand Guide PDF

Collate all the aforementioned points into a web page presence or serve the guide in a PDF for download and immediate in-browser viewing.

The Brand Assets

Lastly, package your assets in .ZIP file and ensure you have enough color variants and transparent images that can be used against various backgrounds. Remember, keep it lightweight (each image file size should be no more than 1000 KB) and offer an array of **file formats** (EPS, PNG, PDF) and color scales (RGB, CMYK, PMS).

The Effect of Zero or Poor Branding Guidelines

Image is *nearly* everything in our web-infused age, as sleek professional design attracts the bulk of screen-watchers. Your program should thus not only seek to incorporate healthy design practice, but **associate** itself with apps that also embrace quality design.

If you are too lax in protecting your brand you face the prospect of a fragmented and inconsistent development ecosystem. Integrations appear murky, and user data and privacy will come into question.

API Evangelist adds that the lack of a branding guide equates to missed value for API programs:

> "Even with this being a major concern, I see many APIs implement very poor branding guidelines, giving developers zero direction regarding how to properly provide attribution. This is a missed opportunity to not just protect the API providers brand, but actually extend it and increase its value." -Taken from *Protecting your brand with API branding guidelines*

So, help nourish your developer ecosystem by supplying in-depth guidelines so that developers share your mission.

Final Thoughts

Maintaining consistent platform-wide branding for a company is hard enough as is. Extending that image to social channels can be tricky, and retaining identity within a third party developer's application is even more difficult.

In this chapter, we've outlined a step-by-step process to create your own **API branding guide**, modeled after some of the best programs in existence. As spreading platform brand awareness to as many eyes as possible may be a primary objective for a public API, the importance of defining these guidelines cannot be overestimated.

Use this cheat sheet as inspiration for your own program, but at the end of the day, you control *your* branding, *how* it is represented and *where* it is present in the developer's app. Learn from the developer relations missteps of Instagram and Twitter; if you plan on releasing an API with branding restrictions, do so **carefully** and in a really transparent way.

Examples of API Branding Guides in the Wild:

Most companies with developer programs have unique brand guidelines as part of their API terms of service. Here are some to model:

- Spotify
- Zillow
- Dropbox
- Orange
- Slack
- TokBox
- Walkscore
- NYTimes

Part Three: Promotion

Launching an API with finesse

Much of API marketing involves a lot of inbound tactics. Nonetheless, there are ways to **promote** an API to widen it's discoverability, using SEO, press outlets, and other methods to widen your impact. In this part, we'll focus on best practices for launching an API, out-

lining directories your API needs to be submitted to, and important channels to contact to spread the news of your release.

Perfecting Your API Release

Build it and they will come is certainly not the case in the API world. If you are managing an API program and seeing little to no adoption, then perhaps you haven't framed your service correctly.

Hopefully, the functionality is awesome, but perhaps your developer portal is lacking in usability or the API is not easily discoverable. Launching an API without having it drop dead in the water means having a strategy to get on the radars of relevant developer-attended networks.

We must reiterate that focusing on technical details is simply not enough. Treating an API as a product means giving it a marketing arsenal the same as for any product launch. So what do successful programs do? On top of having a working program, they planned, they participated, and they got the word out. This chapter walks through a general approach for **releasing** an API, many of which points will be expanded in following chapters.

What do I release?

Are you looking for alpha or beta users to help give feedback? Will you open source components in order to have a community participate in designing your API? Or will you have a more authoritarian hold on your product and wait for a full release? In our world of

continuous software iteration, it's likely best to release in stages of increasing magnitude before a v1. Public announcements are best suited to grow in magnitude as the product is refined.

Time Your Release

What is the correct time to release? Schedule dates midweek to hold major announcements - no weekends, not in the heat of Summer, during known holidays or vacations, etc. If you want a developer community to participate in an open source project, timing may be different than releasing an API that accesses a corporate dataset. Unless your tech is revolutionary or you have an awesome demo, most credible tech news sites will only run full release annoucnements, not features on closed betas still under development.

Widen Your Potential Audience

People interested in the power of your API are not limited to developers - give journalists the cues to describe aspects of your program: functionality, access levels, protocols, parameters, and use cases. This means having a succint one-sentence pitch, like "An API that turns any black and white photograph into stunning full color high resolution images"

Have the Right Monetization plan

As a startup, selling to other startups can be a bit awkward. Don't scare potential users away with high subscription fees. The best APIs demonstrate an astounding product, and help their developers grow their products before asking for money. Especially within

freemium subscription models, monetizing an API and reaching a return on investment can be a long process, as you must wait for a second development cycle before your client's app can afford to graduate to a higher tier.

Have a Demo

Realizing your business acumen off the bat can improve how you market your API. Giving visitors a taste of functionality through sample calls or API playground is absolutely neessary if you expect your product to compete for their time.

Have Awesome Branding

This really goes without saying. But having a beautifully designed front end with unique branding can seperate the winners from losers.

Ready to Promote? Read On

Preparing a product launch takes months and months of preperation. Other good practices are blogging, giving active developer support, and attending or hosting conferences. Next up, we'll discuss how to position an API service online for maximum discovery potential - to do this, we'll divulge two helpful resources; a **Cheat Sheet of 10+ API Directories**, and a list of **Important Press Networks in the API Space**, along with some other helpful advice. Let's get started!

Tips to Make Your API More Discoverable

Even if you have a functionally brilliant service, without the right positioning your conversion rates could still be very low. Assuming you've made your developer program visible to the public, how should you then **promote** your API without having it drop dead in the water?

There still may be tremendous growth potential in your service if it has the right online representation. Perhaps you didn't orchestrate your initial release launch with enough gusto, or the service isn't positioned in a way that increases its visibility across search engines, API directories, and community forums in a way that highlights a unique value proposition.

Getting the word out by attending and speaking at events, or by holding your own hackathon can make for excellent publicity. A lot of this naturally depends on word of mouth and the human-human element — nonetheless, there is still a lot you can do **online** to increase the exposure and breadth of your API program to ensure all possible avenues for growth are utilized.

In this chapter, we dig into three inbound methods that a provider can use to improve the discoverability of their API:

- **SEO**: Optimize your API homepages to make them more searchable.
- **API Directories**: Promote your API by submitting to our cheat sheet of 10+ directories.

- **Service Discovery Automation**: Increase discoverability with machine readable techniques.

SEO Approach: Optimization of API Homepages

The most obvious method a developer is going to use to find your API is through a Google search. SEO or Search Engine Optimization is therefore a top priority for any startup. Make your page more searchable by optimizing content and metadata so that big name search engines continually spotlight your site. As the API Economy is highly niched, keyword optimization is a close friend to many.

Keyword Frequency

Consider what a developer would search for when looking for a certain tech in your space. As an example use case, let's say you are entering the Natural Language Processing market with an API that accepts bodies of plain text and provides statistics such as word count, word frequency, tone analysis, and performs other NLP functions. A **focus keyword phrase** that a potential developer consumer may search for would be *Text Analysis API*.

The number and frequency of keywords found in the copy for the top five results from a Google search for "Text Analysis" is as follows:

Company	"text"	"analysis"	"API"
Aylien	19 (6%)	11 (3%)	13 (4%)
AlchemyAPI	2 (1%)	2 (1%)	3 (1%)
Text Razor	2 (1%)	2 (1%)	3 (1%)
Bitext	6 (2%)	6 (2%)	20 (6%)
Saplo	11 (3%)	7 (2%)	18 (5%)

Results from search performed on 5-4-2016. WordCounter.net used to calculate Keyword density.

Google's algorithms for ranking content are in constant fluctuation, but as you can see above, sometimes page ranking comes down to something as simple as keyword frequency. Alyien's presence stands out from the rest with this search as it scores high keyword density for all three search terms.

Allowing your API to soar naturally upwards in search rankings by focusing on a focus phrase, supported by a **diversified portfolio of supplementary focus keywords** in your site copy will be a powerful tool in your marketing arsenal. This means that having a carefully worded non-technical description of API functionality is paramount to search engine optimization.

Having good SEO and API meta data also makes you more discoverable by API-specific search engines, such as the APIfinder on APIhound.com, or engines that use the APIs.json format for discovery (more on that below).

Separate Pages that Overview Each API

Let's take a deeper look into why Aylien performs so well in this test search. The site copy isn't the only thing performing well, the URL `http://aylien.com/text-api` specifically contains two of our keywords — "text" and "analysis." Though Aylien supplies different services (two APIs, apps, and publisher tools), they have separate home pages for their two API services — Text Analysis API & News API. Separating service presences with **dedicated developer centers** that use simple, targeted URLs is a boost to SEO. This is necessary for API suites that cater to many specific services, such as Microsoft's Cognitive Services — a library of artificial intelligence APIs. The BarChart on-demand financial market data platform takes this even a step further, isolating the public documentation of each API *parameter* onto unique URLs.

Separate homes pages for your multiple APIs helps organize site architecture, and widens your content impact to take advantage of specific focus keywords, opening you to a wider net of searches. Home pages per service also allow a space to introduce technology to **non-developers** — a good example is the Pitney Bowes Developer Hub — for each individual API they have a page with detailed copy, and an accessible video that communicates the value proposition for the specific function.

Don't push the segmentation of actual documentation too far though, or you risk a decrease in usability and developer experience.

It's in the Name

This may seems obvious, but choosing the correct product name is crucial. An API-centric company promoting a single product has the benefit of tying niche functionality into a single brand image — Wit.ai, for example, uses the Artificial Intelligence acronym (AI) within their name and domain extension. However, larger tech companies must partition multiple brand identities under one roof, thus increasing the importance of targeted product naming and service identity.

Service Discovery Automation

API discovery is the sum total of processes a developer uses to search for, find, and research APIs. As described above, this typically starts with a Google search, or via the aforementioned directories. But more than often, successful programs get traction through a bit of luck and word of mouth.

After a developer has found a few potential services, there are still many factors that make comparing similar APIs a complex

affair. Potential users must consider things like licensing costs, rate limiting, data formats, usage policies, authorization methods, and documentation types, and then must experiment with actual code implementation and testing before deciding on a single service to use.

Bruno Pedro, API specialist and co-founder of Hitch hq, sees the entire process of API discovery as thus:

1. *Initial Searching*: Google, directories.
2. *Documentation*: Understanding API parameters.
3. *User Provisioning*: Must authorize/authenticate with the service by some means. SAML, OAuth, etc.
4. *Code Generation*: Consumer will want to work with the language of their choice, so SDKs, ready-to-use libraries are generated from API definition.
5. *Integration*: Once API is consumed in app, it must be monitored for uptime and changes.

At the very end of all this searching and testing, there's no guarantee they'll even find the right match. So, the question is, how can we automate certain parts of this discovery process to make finding the right service more easy?As we've outlined before, solutions exist to automate Code Generation, and there are many API monitoring solutions on the market as well.

But the other points are not as easily automated.

The hope is that by creating machine readable and consumable **meta format** for describing APIs, machines can easily understand what an API is capable of doing, and can describe the human-facing documentation, the price, signup process, authorization mechanism, endpoints, and more. That's where **APIs.json** comes in — a project that hopes to combine these items into a single format to make all APIs more *discoverable*.

APIs.json

The open APIs.json format, supported by 3scale and API Evangelist, is an emerging format for describing API operations to increase the chance of automated discoverabiltiy by software robots. The idea is novel because it is the first approach to standardizing an API's operational metadata.

> "For each API listed, you give it a name, description, and supporting properties, which may be as simple as providing a link to your documentation, or be as complex as providing a link to a machine readable API definition in the Swagger or API Blueprint format."

Dubbed as an index for API operations, APIs.json can be thought of as similar to how websites use a sitemap.xml file to describe site architecture.

Your APIs.json file should reside in the root of your domain. Tkae Fitbit's current implementation on `https://www.fitbit.com/apis.json` as an example. (Also view a detailed explanation of said Fitbit APIs.json file here). Once the APIs.json file is created, you can register it with APIs.io, an open source API search engine which currently lists over 1000 APIs.

Outside of APIs.io, APIs.json will be a driving force behind other API-specific search engines and other projects. Consumers can also create API.json files to describe the 3rd party APIs that their applications depend on. This ecosystem will even enable **API brokers** to act as aggregators that assemble collections of multiple APIs relevant to specific industries.

The project is still accepting feedback, but as more people use it, future standardization with a body like W3C seems possible. Many believe that API indexing - also check out APIs.guru -will act as a huge leap toward standardization in the API space — so better to jump in on the fun now rather than later.

Cheat Sheet of 10+ API Directories to Submit Your API to

Ok. Now we have a good take on refining brand identity and copy. Next is making sure that your API is represented in the following **directories** so that developers using resources other than Google are able to find them. We don't have any numbers on site traffic for these sites, but as submitting profiles is a relatively easy process, it seems worth it to increase the online visibility of your API.

ProgrammableWeb

ProgrammableWeb is the best site to follow for new API releases and to search for APIs. If you want to target developers who are looking for a tech like yours, this is a great first stop. What makes ProgrammableWeb great is that they are pretty open to featuring content from the community. There are a few things you can do on their site:

- SUBMIT a profile for your API to add it to the directory.
- SUBMIT a profile for your SDK. This also includes helper libraries, and/or code samples.
- If you have an announcement or Press Release, CONTACT the PW.com Editors.
- They also publish sponsored content as well as place ads.

Mashape

Mashape is well known for their developer portal and analytics services for APIs and microservices. Even if you're not using their management tools, you can still list your API on PublicAPIs.com, their extensive marketplace:

- ADD your API profile to PublicAPIs.com
- or PUBLISH your API with the Mashape network

APIs.guru

APIs.guru is becoming the open source Wikipedia for APIs. Since the directory is open sourced, APIs.guru has integrated with Any-API, SDKs.io, Cenit.io, and other open source software, meaning this could potentially expose you to a wide net of software developers. Anyone can add or change an API:

- CONTRIBUTE to this project on GitHub

Other Places Where you can List your API:

- **Exicon** : ADD your API to their directory in a "matter of minutes."
- **IBM API Harmony**: SUBMIT your API for review with IBM's curated third party collection.
- **Swedish API Directory by Mashup.se**: SUBMIT your API to the Swedish API catalogue maintained by Andreas Krohn/Dopter.se
- **API For That**: SUBMIT a profile to this small, but curated API directory.
- **SDKs.io**: Add your SDK to this collection of Software Development Kits.

- **APIs.io**: Create an APIs.json file and register your API with APIs.io. (We'll explain why below)
- **API Changelog**: Request an API with this API monitoring site so that your API consumers have automated updates when your docs change.
- **Hitch hq**: A service API providers can use to grow their community and make their platform more discoverable. They have a growing list of APIs, and providers can sign up with them to add their API to the list.
- **More?**: Please contact us to expand this list!

Now, the next two aren't exactly API directories, but smart to consider as part of a promotion strategy for **web apps** that have APIs.

IFTTT

If This Then That (IFTTT) is a platform where users can set triggers between web apps â€" if one thing happens in one app it will influence a function on another app. The cool thing is that IFTTT does accept requests to build a **new channel** on their platform if your app has a public API. If you want to get your functionality and branding in front non-developers, this could be a good outlet:

- SUBMIT a channel request with IFTTT.

Zapier

Zapier is similarly a platform where users can create home brewed concoctions to automate the sharing of data between various web apps. If you want to submit your app into the Zapier marketplace, it's actually possible.

- Add your app to Zapier if it has a REST or XML-RPC based API.

We've previously noted that there are around 11 ways to find APIs from the *developer*'s perspective. Note that however, not all those API directories allow you to *submit* profiles â€" some are search consoles for API management networks like Apigee or Mashery. Others are proprietary API collections (IBM library, Google Discovery Service, etc.). Thus, in this chapter we have listed networks that **any API provider** could submit to.

Important Press Networks and Developer Channels in the API Space

Where do developers talk? If you need to get your API in front of the right people, where do you send it? In this chapter we discuss alternate means to get the word out to **developer communities**. It doesn't hurt to do things the old fashinoed way - making a press release, and contacting relevant news parties. But more involved actions often produce the best results; offering an interview with the founder or CEO, publishing helpful original content, or resharing content you feel helpful to your audience can all be interesting methods for increasing the breadth and existence of your program. Not meant to be exhaustive whatsoever, below we mention some outlets to get you started with API promotion.

Press Release Distribution

If you have the budget, ramp up press distribution by submitting to MyNewsDesk. The service was created by journalists who where tired of getting a lot of irrelevant press releases sent to them. Via my news desk journalists can sign up and subscribe to news that's relevant to them and you, like APIs, developer economy, and startups.

API-Specific Blogs, Thought Leaders, and Digests

- ProgrammableWeb: The largest database of APIs and news source for API releases. The ideal resource to contact with your API announcement.
- API Evangelist: Kin Lane, supported by 3Scale. Kin has branded himself as the API Evangelist, a figurehead for the API community. He writes a lot, reviews APIs, and initiates discussion on a wide range of topics.
- API Handyman: Quirky, technical, helpful. Arnaud Lauret writes on API design on his blog and sometimes for Nordic APIs too.
- PUSH POST PULL: Weekly newsletter all about APIs that interviews emerging startups, maintainted by Gordon Wintrob of Linkedin.
- API Developer Weekly: Curated weekly content published by various players in the API world. Managed by James Higginbotham of LaunchAny and Keith Casey of Casey Software.
- API2Cart: They publish digests that sum up weekly API news.
- Restlet: They similarly publish thought pieces and digests in the space.

General Tech & Developer News

The following blogs often cover the intersection of business and technology and the developer economy. Depending on the scale of your business, some may be:

- TechCrunch
- TheNextWeb
- InfoQ

- Vision Mobile
- Gigaom
- Gizmodo
- Computerworld
- GeekWire
- Wired
- VentureBeat
- ZDnet
- Cnet

Nordic Tech Press/News

- A Swedish-based competitor to MyNewsDesk is Cision, which similarly enables you to reach relevant journalists and influencers to spread news.
- ComputerSweden is a leading source for tech and business news
- Arctic Startup
- Swedish Startup Space
- Nordic Startup Bits

Social Bookmarking

- Product Hunt
- Reddit

API Events

Curated by Matthew Reinbold of VoxPop.co, webapi.events catalogs all upcoming API events, meetups, or conferences around the

western world; a helpful tool for practitioners looking to participate more in their local communities.

The Everpresent Commentator

Who knows, traffic could arise from of a single link posted in a Q&A forum like StackOverflow. It's not always in good taste to market your API in blog comments or forums where developers are going to seek unbiased help. Rather, using your niche knowledge to offer advice and resolve issues is a more helpful approach. When you do feel the urge to blatantly self promote, a disclaimer never hurts.

Utilizing Product Hunt to Launch Your API

As more startups are formed, the web continues to break down into smaller independent services, increasing the amount of awesome SaaS tools available but also changing the way they are promoted. No longer are new ideas granted 15 minutes of fame — it's 5 seconds at best. Enter **Product Hunt**, the Internet's leaderboard for cool products and tech. Product Hunt is a community driven site that is becoming more and more important for discussion around new apps, devices, and **APIs**.

A Y-Combinator graduate, Product Hunt (PH) is an upvote list-structured platform that relies on the addictive nature of learning about new technologies. Users can upload or claim product pages, categorize them by category, create product collections, and follow other users. What Product Hunt offers is a **viral potential** to have hundreds or thousands of users test a burgeoning API in the matter of a day. But as profiles remain visible on the site, it can also add a nice steady boost to traffic for months, or even years following. This scale of exposure could very well be positive to establish name recognition, refine products, and acquire new developer consumers.

If you haven't profiled your API on the site yet someone else could always could, meaning there's always the potential for a front page feature and unanticipated attention spike. However, truly successful Product Hunt campaigns are given forethought, utilizing referral links, a carefully maintained sign up list, and a vigilant eye to respond to user comments and fix bugs that may arise in the

process.

There are evidently best practices for owning your Product Hunt presence and utilizing the community as a means to announce a release. Thus, in this chatper we review some success (and horror) stories from past releases on Product Hunt. We'll interview folks from Sheetsu, Batch.com, and Flutterwave about their API launches, and also tune into past campaigns from API Plug, API Castor, Deepgram, and others. We'll use these collective experiences to arrive at ideas on how to correctly prepare for your own future release. Though we'll focus on the unique qualities of **API evangelism**, the same general roadmap can be followed by other types of startups creating a Product Hunt campaign.

Alpha, Closed Beta, Open Beta, or Full Release?

VentureBeat idenfities Product Hunt as a "Lanuchpad for startups and VC deals." The projects here are new, but are well executed.

Knowing this, when is best to launch a campaign? When do you announce a release? There are benefits for structuring your campaign as a closed beta, giving Product Hunters a rare look inside your platform. But for the most part, it's safer to announce while in open beta or full release. As Michael Oblak, creator of Sheetsu, told Nordic APIs:

> "Always allow users to check your full product. So never closed Beta or Alpha. Allow all users to sign in, take a look, check your landing page, use your product … Product Hunters really like to check the product, play with it."

It's better to scale software product development holistically, with simple functional releases paired with overall user experience

Product Hunters have high standards for functionality, usability, and appearance. Therefore, submitting a pre-launch alpha is discouraged. Antoine Guénard of Batch.com echoes this sentiment, acknowledging that though a closed beta could include exclusive deals for Product Hunters, you will miss organice traffic following your anouncement; "Open beta or full release are the safest then, your product is mature enough to keep the users and benefit of huge organic effects."

Preparing for a Release

Now that we know *what* to release, **how** do we do so using PH? As your PH release will direct visitors to your developer center, keeping updated documentation, tutorials, and other developer resources is critical.

Michael Oblak shared some insights on **preparing your product** that he picked up on his success demoing Sheetsu:

- **Check for bugs, be sure that everything functions appropriately**. "Be sure that everything is working like a charm. You can't afford 404 pages or your server returning 500 errors. Everything needs to work. Treat PH submission as a 'big opening day'"
- **Simplify the use of your product**: "The second very important thing is the simplicity of using your product. If you are providing an API, you need to provide docs, which are easily accessible, with examples of usage, which are also very clear for the reader. Have in mind, that many of the people

visiting your website might not have an idea how to use the API you are providing. It's always a good idea to prepare an easy **tutorial** for them, with screens if needed, to help them configure and use your API."

- **Prepare easy signup process and retain emails**: Prepare for the surge of interest by creating an easy way to collect emails. The most clean way to do this is without those pesky email confirmations.
- **Have a ready-to-use demo**: "PH submission is a chance for you to demo your product to many people. Make your demo short, easy and preferably with your user's data, so it's more appealing to them. With that, they can see value instantly."

In the case of Sheetsu, the Product Hunt profile brings visitors to a demo where there is a quick login using Google. All visitors have to do is paste a link to the Google Spreadsheet, click a button, and the tool generates a ready to consume API using data pulled from their spreadsheet. Under the hood APIs are complex, so mask this with an easy-to-consume demo that impresses.

- **Present examples and use cases**: Demonstrate the example use cases for your product. These could range from apps already in use, to potential uses or end user experiences.
- **Increase server capacity**: Though Product Hunt itself won't "hug you to death", it could lead to it. In preparation, make sure that your servers will handle the traffic. Oblak advises to prepare to handle at least 1k concurrent sessions. This amount is petty for larger organizations, but a startup with limited space may need to increase capacity or put a limit on traffic. API Castor's unexpcted Product Hunt experience found that implementing a wait list feature would have been helpful for his product:

"Mental note 1: Consider limiting signups to a set # and having a waitlist feature in place prior to launch to

ensure traffic is predictable and potential signups aren't lost."

- **Enter the community as soon as possible**: Aside from the product side, it's good to start establishing yourself within the community early on. This obviously means creating an account, playing around and commenting on the site, and perhaps seeing if a friend has an **invite code** [explained later].
- **Reach out to the makers**: People like Bram Kanstein have great insights on the PH process. Kanstein is the creator of Startup Stash, the most-upvoted product so far. For example, as the Board of Innovation began to prepare for their hunt, he offered invaluable advice on optimizing their landing page and PH post.
- **Reach out to moderators for help**: They seem like a friendly bunch. Here's their About page. This is essential if you intend to make your hunt an exclusive…

Offering Exclusive Deals: The Gold Star

Another thing to consider in the preparation stage — some developer programs go full out with a **dedicated landing page** to welcome Product Hunt visitors, offering discount codes and sweet deals. This could include a cost savings, free use for a certain period, free server space, or other creative gifts. Being a PH exclusive hunt also comes with a **gold star**, and a bit more attention and love from the community.

You can can sign up your exclusive with the PH team here, and the PH FAQ also explains what qualifies as an exclusive program — bascially offering a **meaningful discount** and to clearly introducing the **deal**. Algolia, for example, offers 2 months free to Product Hunt visitors:

Most landing pages welcome the visitor with a cat wearing Google Glass

Or, API Plug's Product Hunt referral page offers a 20% discount and $25 Credit from Digital Ocean Digital Ocean:

See what we mean about cats? It's sort of an expected thing.

In Batch's case, the team opted for the Exclusive access using a dedicated landing page, which gave the community early access for 24 hours. This preparation took a bit of planning for the team; and required "managing individual sub-access accounts, handling redirects, and ad-hoc design integrations)", but they found that it helped grant more control on launch day, and enabled them to work more closely with the Product Hunt team to schedule their launch.

Actually Submitting a Profile on Product Hunt

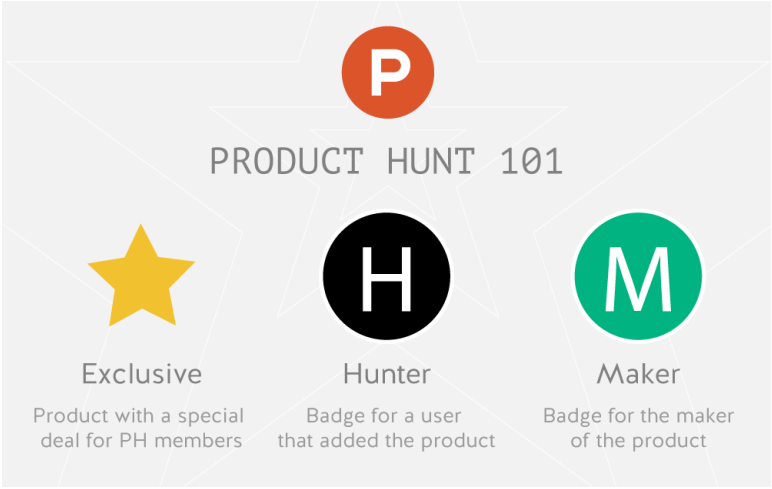
Definitely read through the Product Hunt FAQ. Part of the allure of Product Hunt is that it is still partially an invite-only network — anyone can register and upvote, but only members can comment and post new products. To get around this, you'll have to be nominated by a member in order to submit. With a little finagling, you should be able to find a connection to the platform. If not, moderators will usually help credible makers enter their realm.

Once you have permissions and are ready to submit, the "+" button in the top right will create a profile submit page, with only a few categories to fill out including:

- Category - e.g. Tech, Games, Podcasts, Books
- Name
- URL - Direct link to the product page (avoid links to press or blogs)
- Tagline - Very short description of the product (make it catchy!)
- Images

As you're trying to create the best first impression possible, succinct eye-grabbing copy triumphs. As PH describes:

> "Taglines matter. The best taglines are succinct, describe the product, and avoid cliche descriptions like "email on steroids" or superfluous buzzwords like "a beautiful SoLoMo livestreaming experience."



Lastly, to maximize the amount of upvotes, **timing** is huge. Kanstein recommends that 10.30AM CET is the best time to post to PH, while Oblak recommends aiming for the Europe lunch time and USA morning.

The Launch: Introduce Yourself, Play Nice, Get the Word Out

So, you've hit submit. Now what? Well, don't expect to sit back — posting is only the beginning. It's a great idea to kick off conversation with a comment that introduces yourself, and details the functionality of your API, and what you hope to achieve.

> "Pro tip: those that answer questions and actively engage in the discussion (examples here, here, and here) tend to get more attention and appreciation from the community." - Product Hunt FAQ

Lastly, don't rely on random upvotes or soliciting likes, but try to increase traffic using natural means, like sending a **newsletter**, sharing your PH profile across your **social networks**, and contacting **podcasters** like who may be interested in featuring you.

Startup Stash creator Kanstein initiates conversation. As he said in his blog, "I decided this would be my main focus while launching"

For many, the gauge of success for a PH launch is becoming **featured**. Though PH won't disclose the exact algorithm that powers their ranking system, it is a combination of upvotes, time, and traffic that decides what appears on the **home page**. Even after all this prep, traction will largely hinge on word of mouth, ultimately decided by the awesomeness of your API demo. If you mimic other services or don't have a niche unique value position (if you are one of the hundreds of SMS APIs, for example), or if your presence is too technical for the non-developer to understand your API parameters, then you likely won't be successful on this platform.

The Unanticipated Launch

Ready? Maybe you're not. In late October 2015 API Castor Nicholas Hilem, founder of API Castor, awoke to a slew of notifications related to his startup. Oh god, he thought. Someone had added me to Product Hunt. He was only half prepared. Cancelling all plans he had, he tried to fix as many bugs as possible before traffic exploded.

> "Apparently, one of the PH mods had seen our HN [Hacker News] posting and submitted us. A sense of dread washed over me as I anticipated the flood of traffic that might follow. I quickly made a short list of easy polish items that I could still tackle that night. Cracked out 80% of them, accepted our fate, and called it a night."

The attention spike from a Product Hunt launch

Since any member can post to the site, this sort of potential is always there. Horror stories aside, the moderators are pretty understanding, and have allowed teams to submit again in the past to get the traction they deserved. If this happens when you're not already a member, tweet at @ProductHunt and they'll likely give you access.

Use this badge to prevent the unanticipated hunt

The Return on Investment

So is it all worth it? From what we've heard from others, the whole experience can be really helpful in getting leads and retaining users, allowing others to validate early stage products.

> "Product Hunt produced a solid amount of traffic (4696 visits in 24 hours) with a decent sign-up rate (6.63%). More importantly, PH set themselves apart in their ability to continue to drive traffic to our site over the week, amounting to an additional 150% of what we had on launch day, resulting in a grand total of 13,866 visits. They have a tech-loving audience with a product/design angle, who posed very interesting questions from the community." - Batch.com

Though Product Hunters have a smaller average registration, PH had higher initial and recurring traffic. Taken from Batch.com blog (now archived).

For Sheetsu, Oblak is still "noticing visitors from PH today (10 months after submission). Many of my first customers and users were from the PH. Most of them are still using Sheetsu now."

The Internet's Watercooler is Product Hunt

You may be astounded, how were we able to write an entire blog post about Product Hunt with only two cat pictures? Though PH certainly acts as an anchor for meme culture, founder Ryan Hoover and the PH team has created a seriously useful platform, which has

become a staple addition to the startup marketing arsenal. What makes PH alluring is not a multi-million person userbase, it is that the community is made out of **influencers**; product creators, managers, and entrepreneurs.

Just because it's free to submit here doesn't mean you shouldn't devote attention into preparing your launch using these tips. Granted, the platform is still in flux, so these processes may change in due time, but hopefully the best practices we've learned from past campaigns will aid the tenacious and earnest API advocate.

It's also important to note that many strategies listed here are basic developer experience knowledge, and can help prepare for listing your API on other sites. PH and **Reddit** can be helpful, but adding your stack to **Stackshare.io**, or posting your API to **Hackernews** are even more relevant to APIs as they appeal to a more technical crowd.

After you've had a successful campaign, it's tradition to write about what you learned, share statistics, and help others succeed. Good luck hunting, and please leave feedback on how to improve this guide!

Resources

From the Product Hunt team:

- Product Hunt Pro Tips
- Product Hunt FAQ
- APIs with exclusive offers to the PH community can sign up with the PH team here
- Here is the entire PH team and their Twitter handles

PH Campaigns mentioned:

- Sheetsu | Michael Oblak, Founder

- Batch Insights API | Antoine Guénard
- Startup Stash
- API Castor
- Flutterwave

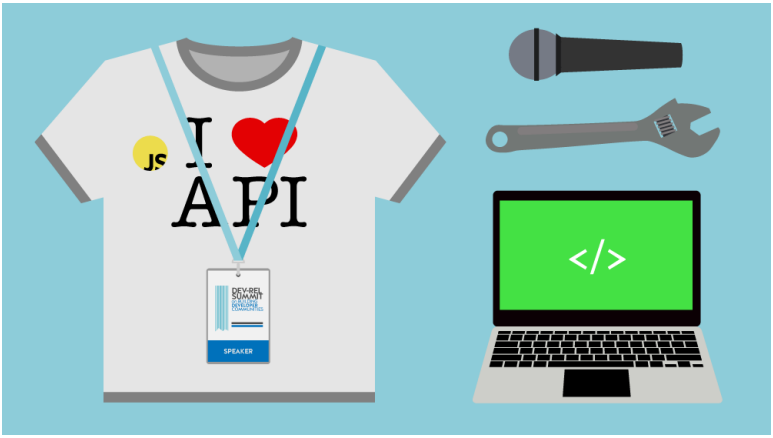
Part Four: Advocacy

Building a community around an API

Creating a discoverable API with developer center materials is only the beginning. The number of developer evangelists promoting API programs have skyrocketed in recent years, and for a good reason. Having community leaders that represent developer needs is essential. In this section we discuss how to **advocate** your API using evangelist best practices, developer outreach, feedback

and surveys, holding conferences, and creating stunning technical demos. The API programs that succeed in doing so will naturally form a community that can scale itself simply through word of mouth.

Day in the Life of an API Developer Evangelist



Evangelist, advocate, community builder, whatever you want to call it, since Guy Kawasaki of Apple popularized the concept of "technology evangelism," the role has become a staple addition for software outreach. New startups and corporations alike now often hire **evangelists** — a rather nebulous breed of employee between sales, marketing, engineering, and support. Evangelists tread many different camps, know their product in and out, receive feedback from the community, and work to better the product and consumer experience as a whole.

Nowhere is the title more relevant than in the **API space**, where Application Programming Interface (API) program advocates attend or host hackathons, create tutorials, give assistance, and more, spreading the good word of API on and offline. From advocacy at Google to Twilio's evangelism, the precise makeup of developer

relations is still being defined. With all this interest in a relatively new title, it made us question, what exactly does an evangelist do on a daily basis? And, most importantly, what advice do they have for creating a successful, thriving **developer community**?

So, we've reached out to the masters for their insights. For anyone seeking to become a tech evangelist, or who has recently been ushered into the role of promoting an API program, this is the chapter for you. For this Q&A session we interviewed five API developer evangelists from throughout the industry.

8 Important Job Roles of a Software Evangelist

1: Be sure who your users are

Most agree that the number one job of a software evangelist is to intimately understand your consumer. According to Daniel Rudmark, a senior researcher at Viktoria Swedish ICT, attracting developers to an API program must be grounded in empathy:

> "I often find that organizations publishing APIs do not sufficiently recognize that developers are not paid up front for their work — they come to you for some other reason which you need to understand and support. Remember, they spend their valuable time on your API"

As we mentioned in a previous chapter, there is an increasing amount of diversity amongst third party developers — varying technical understandings, different industry backgrounds, geographical location, and more contribute to your user persona, who may turn out to be different than you think. According to Guillaume Laforge, product lead and APIs dev advocate at Restlet:

> "There are tons of different developers, using different languages, different tech stacks, focusing on different devices… think about which ones to target first"

Evangelists must intimately understand this audience, but must also foresee shifts in audience that could occur as a product grows. Evangelists placed into a **legacy community** have just as an important role in sustaining and extending a user base. As Keran McKenzie, Developer Evangelist for the MYOB API program told Nordic APIs, it all starts with knowing your audience:

> "In some ways I was lucky in that I inherited a legacy community so I had an initial base to start with, in other ways because of that legacy base, it dictated the path we took. Once you know your audience (and "All developers" is not a valid answer to who is your audience) you can begin to work out targets for building new community, plans for content & resources and event activity."

2: Communicate the value(s) of the product

The next most important role of an evangelist is to efficiently communicate the **value** of the product. As developers are the lifeblood of APIs, they are the consumers that must be sold on the **benefit** of your API if they are ever going to use it. This role stems from a technical understanding of the product, knowing how the product stands out from its competition, and an easy, open relatability with others.

An evangelist must always be prepared to communicate value in many situations — whether it is a quick description to a colleague, writing tutorials in a developer center, a long-form blog post, or pitching a demo at an event,

> "the primary role of a developer evangelist in forming a developer community is to help customers and potential users see the value and benefit in your product or API to such an extent that they themselves become evangelists for your company." - Liz Rush, Developer Evangelist at Algorithmia

A successful API program turns *users* into evangelists, thus enabling the most effective, time-tested marketing tool possible — **word of mouth**.

Awesome functionality is easy to communicate — for example, the Star API gives you real-time lumosity, color, and spacial data for over 100,000 astronomical positionings. Though there is certainly value in the data or functionality wrapped in your API, McKenzie adds that **reputation** is just as interconnected with value:

> "In many cases however the value of the API is actually the value of your brand, your market and partnerships beyond the initial data/content or service"

3: Ensure the program is attractive and usable

It makes sense that developer experience (DX) has become a focal point for treating API services as usable products. As an API evangelist is the bridge from product to community, they must evangelize DX as well. Evangelists often build and upkeep developer-facing resources, such as the developer portal, SDKs, and language libraries.

> "…you need to provide technical tools that enable swift problem-solving to avoid developers having to figure out the many quirks of your product" - Daniel Rudmark

We've covered the many ways to improve API usability with things like content negotiation, hypermedia, API gateways, Mullet-in-the-back architecture philosophy, and much more. To be agile, API programs must balance simplicity and complexity, have ongoing testing, and always be compiling feedback from users on how to improve.

Proper design is unique to the service and thus complex to define — but you know bad design when you see it. Evangelists need to evangelize the best version of their product not only externally, but internally as well.

As with any technical product, support channels must be in place to aid **onboarding**. A good dev center and documentation should answer most of this, but the best evangelists will jump on a call to quickly aid a developer user to success.

4. Always be observing, talking, and gathering feedback

Evangelists are *vocal*. Online they blog, prowl Stack Overflow, maintain GitHub repos, manage developer-dedicated social channels, and respond to all comments or questions. Evangelists often take on a customer support role, and must offer rapid and effective customer service.

To Rush, the most important aspect of being a successful evangelist is a willingness to talk and answer people's questions…no matter the setting…even the dance floor—

> "…you have to be prepared to be "on" at almost any time. Yes, the majority of it is done at meet ups, online, conferences and the like, but honestly it happens at unexpected times too, ranging from someone who overhears you talking about algorithms in line at the

> coffeeshop to chatting with someone you just bumped into on the dance floor at a warehouse party."

McKenzie sees transparency as "hugely important in the role of an evangelist. We are often the face/voice of the brand, so we need to be as open as we can." As developers rely on the uptime and consistency of your product, this means communicating negative changes with care. When an API is updated or retired completely, the way a company delivers this message is often "more important than the message itself."

5. Host, attend, or speak at events

The sea of API-related events has expanded tremendously in recent years. A traditional event model has been to host hackathon competitions that encourage developers to build things with your API, often for a prize incentive. Proposing talks at larger conferences is also important to get the word out and learn from others. However, evangelists must be selective, and tailor their travel time and events

to be in-sync with their program objectives. Often, this means supporting smaller events or partnering with other hack events rather than hosting your own:

> "Too often people think building a community means running a hack event, we found that running our own hack events didn't fit our API, however attending & participating (as a team vs API vendor) in hack events was invaluable for building out a community. The best thing you can do is roll up your sleeves and get involved." - Keran McKenzie

Digital events like screencasts or webinars can be helpful too. But physical conferences are places to share projects, discuss strategies, and easily network with people who have the same interests. How could we not append this role? :) Contact us if you'd like to speak at a future Nordic APIs event, and come to our API Stack Conference in April!

6. Build and maintain a knowledge center

Blogging is important, but some argue that *building* is even more so. A developer advocate creates content that speaks louder than they could alone. This includes instructive tutorials, integration walk-throughs, and shared stories from the developer community, all organized with beautiful site architecture and design. All this does a lot to support your API, but at the end of the day, **documentation** is king.

> "In the API world, you have to treat the documentation and content you provide as your **number one priority** as an evangelist. If you have poor or no documentation for your API, most potential customers won't be able to use it and will give up almost instantly." - Liz Rush

LaForge adds that the onboarding process and documentation should be constructed in a way that appeals to the specific **use case** for the API:

> "Spend a lot of time on the on-boarding process, how developers get started, with great API documentation. But not just "reference" documentation, but thinking hard about concrete use cases, in terms of scenarios: developer X wants to do Y, here is how to do it with the API."

7. Translate technology trends

Evangelists help revitalize stagnation. Following fluctuations in technology can help an API program be agile and responsive to industry momentum. Adam Duvander, a developer advocate at CenturyLink Cloud, began his career writing with Wired and ProgrammableWeb, and continually shares what he observes and researches in the field:

> "I translate back-and-forth between the technical and non-technical to put trends into context"

Framing the technical in a way that entrepreneurs, marketers, or designers can comprehend is helpful for all involved. A learned, vocal, opinion on innovative tech strides, whether it be microservices, Golang, or Docker, etc., can be greatly appreciated by a community that depends on your specific tech to survive.

8: Build a community of heroes

At the helm of the API ecosystem, the evangelist fosters a sense of **community** around an API. If your marketing team executes all

of the activities stated above (on top of having a stellar product) you should see overall usage increase, and a community emerge around the service. That being said, there's a fine line here — you can't really force "community" to happen. Evangelists can only encourage its evolution. Be helpful, but don't be a drag:

> I like to talk about "being in their face" but "out of their way". We need to be visible, the developer community need to know we are here to make them heroes in their own right, at the same time we need to put everything in place so we can be invisible and out of their way." - Keran McKenzie

For McKenzie, community building means constructing all the tools for support — resources, documentation, articles, tutorials — everything that enables a community to be as "self sufficient and effective as possible."

> "A community could emerge out of the blue, by simple virtue of having a useful and popular API…but more often, you'll have to encourage developers to use your API, by providing on-boarding resources, easy demos, developer portals with top-notch documentation, a forum, a super reactive support team… you'll have to play on several fronts to get a community to form, especially on the communication channels you'll open up with that community." - Guillaume Laforge

What does an Evangelist do each day?

The daily work of developer evangelism and advocacy comes down to responding to the user's needs. But, certain tasks come up often

enough to get a general idea of the role. Through interviewing evangelists in the field, we found that some of the tasks they perform on a daily basis are as follows:

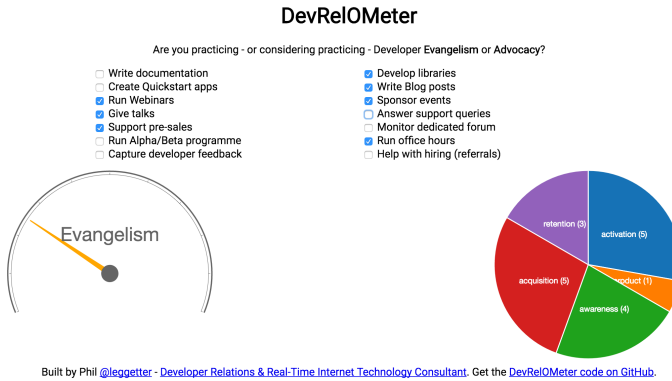
Customer relations	Events	Support
Social media activity	Travel	Respond on Stack Overflow
Authoring blog content	Host webinars	Dev center maintenance
Weekly newsletter	Speak at events	Test the API
Recognize and award hero developers	Research, gather feedback	Work on Github helper libraries

Evangelism vs Advocacy

There is arguably enough of a difference between the job roles of an 'evangelist' and 'advocate' to make certain distinctions.

In the proposed 5 level maturity model for developer relations, *advocacy* is at the top, defined as a two-way dialogue in which developer support and gathering feedback is paramount. In the same hierarchy, *evangelism* is a rung below, and consists of attending conferences, explaining, and the like.

Writing on his personal blog, ex Pusher advocate Phil Leggetter analyzes the dev relations mission statements for Google and Twilio. Using these cues, Leggetter built the DevRelOMeter, a cool tool that will tell you if your team is actually performing evangelism (awareness & aquisition) or advocacy (product support & retention).



Use the DevRelOMeter to see if you're practicing 'evangelism' or 'advocacy'

What developer relations — or devrel — consists of will inevitably be unique to that company, but critically considering the ROI for certain activities surely helps.

Q&A Section

What lends to the best interaction with customers? How do you help developer users the most?

McKenzie: "I think it's the unexpected events that tend to lead to best interaction. Let me give you an example. Last year Jack Skinner & I were in Auckland, NZ for an event when a support ticket from a developer came through. We were just a block away, so Jack headed over to their office just at 5pm as they were packing up. He spent an hour or so with them going through the issue, making sure things were solved and delighting this customer. They were not expecting Jack to turn up, let alone go that extra mile to ensure everything they

needed was solved. Of course we can't always go out and see customers like that. Sometimes it's as simple as picking up the phone vs emailing a response as a great way to drive a delightful experience the developer community."

Rudmark: "From my research where I have both performed a great number of in-depth interviews and analyzed video recordings of developers having used APIs I would say that the number one way to help developers is understanding what developers may be struggling with and make sure to resolve these issues. The tricky part is that such information seldom reaches the API provider and that the developer often just quits working with the API. This means that you need to be very mindful about how you deal with and act on the (little) feedback you actually get - it is quite often just the tip of an iceberg."

Laforge: "Be kind, be friendly, be helpful, try to provide the best level of support, and on-boarding experience. Then when your API evolves, be sure to keep users updated, try to avoid breaking compatibility if it can be avoided, try to improve the API (improve usability, response time, provide more features)"

Nash: "The best interactions come either when a developer has integrated the API and are delighted at how simple it was and just want to say "thank you" or when a developer is having trouble with a certain part of the API and I'm able to help them out and get them back to building their application."

How do you retain users?

McKenzie: "In our space our developer program is much more than just an API. It's about a partnership…retaining developers for us speaks to building a true partnership. I personally spend a lot of time with this group of developers talking about *their* customers, *their* solutions and how to bring *those* to market."

Rush: "In the API world, the question of customer retention is primarily taken care of with one simple rule: don't break things. Of course, this is dependent on what you've done to bring users in: you have to prove the added value of your service, show how easy it is to integrate & use, and more broadly speaking, solve more problems than you create for your customers."

Nash: "User retention is the job of the entire company. If the product works, is priced well and has good support, account management and much more then users should be retained! I don't concentrate on user retention exactly, as I see evangelism as more outreach and finding and supporting developers…though I do pitch in with support on social channels, such as StackOverflow and GitHub"

In your opinion, is dev evangelism support, sales, marketing, or a new breed of employees?

McKenzie: "It's definitely a merger of all three. You can't be a developer evangelist if you can't cut code, likewise if you can't have a solid sales conversation, discuss go-to-market plans and of course actually get

out and market your API, then you aren't going to go far…. Finding a Developer Evangelist is (I think) one of the hardest roles in business to fill. When you find someone, do everything you can to nurture and hold onto them."

Rush: "This is a contentious question among evangelists — in fact, many don't even use the term "evangelism" because of the fact that it has become nebulous and often misapplied to sales engineers or support engineers. I think of evangelism as a new but growing role different from support, sales, and marketing. Since evangelism is often most popular among companies that provide APIs, the terms don't fit quite as well since these traditional divisions are often blurred, especially in the startup world."

LaForge: "It's really at the crossroad of all those activities. That makes the job even more challenging and interesting because you can tackle different classic areas: a developer-oriented evangelist might be more keen on crafting a demo for supporting a customer, another one with a creative personality might love authoring great articles and documentation resources, etc.…It's also the kind of job that you can tailor yourself, depending on your own aspirations, as well as according to the needs of your company."

Nash: "At Twilio we are part of the marketing organisation and that works well for me. We may perform support tasks, but getting out there in developer communities and talking to developers face to face and online is the priority."

Conclusion

The actual tasks an API evangelist performs will often fluctuate, especially as roles are blurred in the startup world, as Rush points out above. Arguably, openness and transparency triumph in the position. If you are going to have an active voice in the developer community, it definitely helps to integrate these core ideals into your presence.

At the end of the day, API programs want apps to flourish — meaning that evangelists are only successful when the developers they support are successful. Phil Nash, developer evangelist with Twilio, framed it well when he told us:

> "I am successful when the developers I serve are successful, whether that is writing a Twilio app or not."

Interviewees:

Thank you to our interviewees for participating in this Q&A! We didn't intend to be exclusive with our panel, and hope we get the chance to talk with other API evangelists and advocates on the topic in the future.

- Daniel Rudmark | Senior researcher, lecturer| Viktoria Swedish ICT, University of Borås |
- Guillaume Laforge | Product Manager, API Dev Advocate | Restlet
- Keran Mckenzie | Developer Evangelist | MYOB API
- Liz Rush | Developer Evangelist & Software Engineer | Algorithmia
- Phil Nash | Developer Evangelist | Twilio

How to Offer Unparalleled Developer Support

APIs are necessarily a communal endeavour — the **community** fostered between the users, the providers, and those who depend on the API for the functions of their own services drives the development environment of the API space.

Accordingly, understanding what makes **developer outreach** so essential to cultivating your own network of users and agents is incredibly important. Beyond understanding this importance, figuring out the proper channels, frequency, and methodologies of effective developer outreach can turn a powerful product into an absolute powerhouse.

The Importance of Developer Outreach

First of all, let's consider specifically why developer outreach is so important. A system is only useful when it's used — without a group of developers who use a system, that system is essentially isolated, and loses a lot of its potential.

API providers face three unique obstacles to API adoption — Lack of Awareness, Lack of Understanding, and Lack of Vision. Let's separate each of these, and consider their impact on an API provider.

Lack of Awareness is, simply put, the lack of awareness that a platform or service even exists. How many times has a product been

displayed, a solution first encountered, where the average person says "of course, why didn't I think of that?!" The fact is that most people are unaware of the tool in our hypothetical because they didn't know they had the problem, and they were thus unaware the solution ever existed.

The same issue comes into play with API provisioning. If a developer needs a solution for a complex problem — and they often do — not marketing a product or making others aware of the product's existence is as bad as having no product whatsoever.

Lack of Understanding, however, is a more fickle, and possibly damaging, beast. While a developer might be aware of the product, the daunting requirements for its use, whether actual or perceived, can end a budding relationship extremely fast. Poor error returns, poor documentation, even lack of forum presence can doom an API to the "rejected" pile for the simple reason of "hard to use, no support".

Finally, **Lack of Vision** plagues many developers. This is not to say developers are not creative, or are unable to envision a product —- quite the opposite in fact. This *is* to say, however, that developers are often unable to envision your product and its usefulness unless clearly identified.

Questions like "why do I need this" and "how does this help me" are often issues of lacking vision, but again, not because the developer lacks the skills to see the possibilities — the API provider has simply provided no example use cases, or functional descriptions in plain english upon which to base this vision of successful implementation.

Thankfully, each of these can be easily rectified using a few basic techniques.

Email and Social Media

A joke for the internet age — a company consults with a firm on how to improve their marketing and sales. The firm asks if they have any significant social media presence. The company scoffs, and says "of course we don't have an social media presence — we're the internet provider!"

It might not be a particularly good joke, but it is certainly insightful — when it comes to business on the internet, many companies still think in the age old "paper and direct sales" mentality. Be professional, directly market, and you're good to go! That's not the reality of the modern age, however.

API communities, as with other online spaces, are primarily ones of consuming online resources and communicating in that medium. Accordingly, API developers are some of the most active and prolific users of both email and social media, and not just for "CatFacts" and live tweets about Game of Thrones. API developers use these channels for distributing new services, testing functions and calls, discovering new solutions, and even authenticating with tools.

Failing to have a proper presence online is one of the best surefire ways to doom an API provider to obscurity. Leveraging proper online presence, however, can leverage your initial successes to dizzying heights.

There are basically two approaches to online presence generation and long-term maintenance. These approaches are forked in two general directions — direct and indirect.

Direct outreach is simply direct communication with interested developers or developers within a stated demographic. Engaging users on email or social media when they have demonstrated an interest in or desire for a given solution can inform that the product already exists, and has a stellar team behind it.

This does run a rather significant risk, however — when's the last time a consumer received a casual cold-call and said "oh, thank you, that's great, we will absolutely use your service"? The fact is that poorly formed initial discussions and pushy sales methods can do more harm than good, so these relationships should be somewhat casual in nature, framed within the concept of one developer helping another. A great example of this is the "cold response" when signing up for an API. Seeing a "personal message from the co-founder" that is obviously automated is one of the worst symptoms of this situation.

While this might seem ineffective, you will always find more success and goodwill in the concept of "one dev helping another" than you will in the concept of "I have a great product to sell you".

The other approach is one of indirect marketing, also referred to as "word of mouth". Creating a stellar online presence publishing blogs pertaining to the industry, showing new, experimental projects, or even just sharing partners that you think are amazing can do a lot to link the brand of the API provider with the success of the referred products.

This is not to say that online portals should be an advertisement — reputable testimonials are definitely helpful, but something like "I'm Kristopher Sandoval, and when I generate API documentation, I use DocuGen" simply comes across disingenuous, and could do more harm than good.

Instead, offer solutions — post chunks of code you find interesting, or platforms that you think others could use, and tie them into your API, framing it as a "portal to great solutions". If you open these channels of communication, when developers come looking for solutions, they will find you.

A quick note on email and social media responses — this is not a one-sided conversation. When you are tweeted at, commented on in YouTube instructional videos, contacted via email, etc., you must respond in a timely, professional manner. Networks live and

die by the strength of their individual components — even one poor experience made public can color the way all other developers interact with you.

Event Hosting and Attendance

The API space might be a digital one, but the developers are entirely human. We too often forget that behind every server, behind every terminal, integral to every chunk of API code, there is a living, breathing, unique human, with a range of experiences informing their skill and opinion.

Events, therefore, present a unique opportunity to engage the community in ways that might not otherwise be possible. When we discuss events, what we are really discussing is any event hosted by an API provider where like-minded developers, providers, users, and other interest parties can congregate to discuss the API space, either in detail or generality.

There are a huge range of potential benefits of hosting an event. First, there is the obvious value-adding experience of the event itself - after all, how can a developer who comes into the city for a three day bonanza of free coffee, great donuts, and an amazing service leave with a poor view of the provider?

More importantly, however, is the opportunity for the developer to engage in a more direct, personal way with the userbase that social media simply cannot match. Identifying big players in the API space, what they might need, and what their users have communicated to them not only helps you coalesce knowledge that would otherwise be unattainable, it also helps establish the API provider as a helpful entity seeking to make the industry a better place.

There is a third benefit, and its importance simply can't be overstated — hosting events is a great way to create an independent,

thriving community and **knowledge base**. Inviting developers to a get together and walking them through common use-cases and solutions tests your product, informs you as to its shortcomings and strengths, and creates in users a direct knowledge of the product from the creators themselves.

This helps additionally create an inventive and forward-thinking space. Giving a vision of the way things could be with your service or product is one of the hardest things an API provider can do — and one that is done rather effectively in such an event.

Keep in mind, however, that these events cannot be just about the API provider — it needs to be a unified community effort. Don't be the kid that demands everyone come to their birthday party with presents, and then is mysteriously never around for anyone else's.

Documentation and Knowledge Bases

Documentation will set you free. That simple phrase has guided more than its fair share of developers, and it rings truer as the systems that drive this crazy and beautiful interconnected information superhighway become ever more complex.

The quickest way, therefore, for an API provider to fail in attracting developers is a lack of documentation and knowledge provision for their services. Failing to document the core functionality is a nail in the coffin for the grave dug by the API provider — but even failing to document the small things can gradually add the dirt over your head.

There are two trains of thought on documentation, however, which run in stark contrast to one another. One is the idea that documentation must be done directly and through the efforts of the developer creating the system itself, and that this documentation should be the end-all-be-all.

While this certainly adds a great deal of control over the resultant documentation and the community the uses it, it has several drawbacks. The first and most prominent is the added stress upon the API provider. While basic documentation is a core requirement, documenting every solution to every problem that might or might not arise in an infinite number of situations is daunting, and borderline impossible.

Just as important is the fact that documentation from the developer will often come from those who wrote the code, and at time, can be too technical in nature. Writing documentation for machines is an unfortunately common mistake — especially when human readability is the ultimate goal for most of these projects.

While having good error reporting, syntax, style constraints, and documentation codes can do a lot to help in this regard, the undue stress is simply too massive for most API providers.

The second train of thought is the concept of the user knowledge base. The concept is simple — creating a database where users can tag issues, perhaps in ticket form or even a wiki-like interface, allows users to document radiant and new issues, as well as their solutions, for future users.

The main benefit here, of course, is that the documentation will be by its very nature human-readable. Because the documentation was generated directly from the userbase, it will be easily read by the userbase, and if it in fact is found to be too difficult to read or use, the number of users willing to fix it will be more often than not larger than what could be provided by the API provider.

There is of course, as with all of these topics, a huge caveat. Repeat the following — knowledge bases do not replace human contact. Too many corporate databases exist documenting issues found on APIs, operating systems, and applications that terminate with a generic answer — this is a failure in communication.

When a user finds a knowledgebase article, the best hope is that it answers their question and gives an easy solution. If it does not,

however, terminating at the article and replacing your social media presence with this simple article or ticket can doom an API faster than anything else.

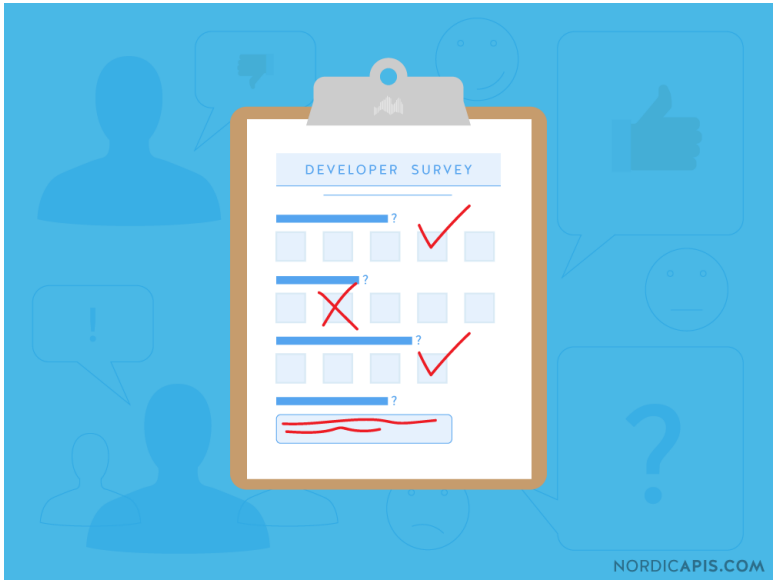
Remember — knowledge bases should augment and support email, forums, and social media, not replace them.

Conclusion

These basic techniques and approaches should comprise 99% of most API provider's strategies when it comes to fostering unparalleled developer outreach. While some of it seems common sense, many would be surprised at just how common and epic these failures in basic outreach are, and how damaging they are to the API space.

Adapting these solutions will provide the leverage that API providers need to succeed, catapulting initial success and goodwill to a new, higher plane.

Accumulating Feedback: 4 Questions API Providers Need to Ask Their Users



American activist Bryant H. McGill once said, "one of the most sincere forms of respect is actually listening to what another has to say." For API providers, listening to the average user, accepting **feedback**, ingesting these experiences, and iterating on this information is a powerful exercise.

That being said, what questions should developers should ask their

users in the first place? Knowing what to ask, when to ask it, and *how* to ask it, is just as important as ingesting the eventual response — perhaps even more important, when considering how it sets the developer/user dichotomy and tone of discourse for the future.

In this piece, we're concerned with answering a singular question — *what* questions should API developers ask their users and *how* should they ask them?

Why Feedback is Important

We're not paying lip service, here — **consumer feedback** is possibly the most important element of any development ecosystem. Building an API is essentially a design puzzle, matching the specific needs the user has communicated with the specific talents, skills, and limitations of the development environment.

When developers don't match the needs of their customers to their development strategy and long-term goals, a rift is created between the user and the developer. This rift is the prime cause of most hardships when it comes to customer acquisition, retention, and user experience.

So how does a developer prevent this rift? That's where communication comes into play. **Developer relations** takes many forms, ranging from basic social interactions on Twitter and official forums all the way to email campaigns and direct conversations. The utilization of effective and complete API metrics is also a key factor leading to the success of an API.

Now that we understand the importance of this feedback, the question arises — what questions should be asked?

What Do You Expect From This API?

As we noted in our article on drafting readable platform policy, expectations drive all business and social interactions. The matching of user and provider expectations will drive the overall user experience of the API and relevant services, and can go a long way towards informing both your public image for other potential users and the internal experience of utilizing the service.

Let's frame this as a common-sense manner. It's your birthday! As part of your celebrations, you've decided to go out for dinner at a restaurant that just opened down the street. Looking at the prices, you decide it is rather expensive, but worth the extra expense.

When you go to this restaurant, your expectations of service, food quality, and interactions with the wait staff defines your perception of that restaurant and that meal, for better or for worse. If you have low expectations of the restaurant, but instead receive a five-star meal with wonderful service, that communicates a fault in your expectations. Conversely, if you expect the best and receive low quality food and slow service, that communicates faulty expectations from the service provider.

In this example, you expect quality for the high cost. The same is true with API users — discovering and utilizing your service over others requires an investment of time and effort, and failing to provide value to that resource investment can lead to a negative perception of the entire experience.

By asking directly **what your users expect**, you can not only gear your performance and service to a high expectation, you can help to level out their long-term expectations of development and implementation.

A great example of this is the development cycle adopted by Mojang, developers of the hit PC game Minecraft. During their Early Alpha development cycle, they made it clear that, though

they had big plans, implementation would be slow. When they began selling Minecraft and entered "beta", they noted that the user experience may be fraught with bugs and issues, but that users should expect periodic updates to fix these issues.

Mojang asked the users what *they* wanted. They asked them what they expected the game to look like, what items should be included, and what mechanics they would like to see. They then communicated the realities of the development platform, expressing what was feasible and what wasn't. They tempered expectations while gathering these expectations as a platform from which to guide future development.

Everything from inventory slots to combat mechanics have been tweaked and manipulated given user feedback. Every build of the game is released in a beta channel for user testing, and the Minecraft forums are often flooded with data points for user experience that the dev team often calls from during their second testing phase.

While this example isn't necessarily in the API space, it does demonstrate specifically how powerful an open channel of communication is. Mojang is known amongst its community of followers as a company that cares, a company that communicates, and one that can be depended on to implement things when it says they will implement them, with few exceptions. Most importantly, users are aware of the realities of their expectations, and whether or not they can be implemented at all.

By making beta builds open, users can test the code — a benefit previously discussed in our piece on GitHub. Likewise, the open channel of communication allows for common security vulnerabilities and "happenchance" discoveries to be communicated and quickly fixed, preventing zero-day exploits and vulnerabilities.

API providers need to follow suit. **Ask your users what they expect the functionality to look like**. Ask users what they want the API to do, and how they want it to do these things. By under-

standing what your userbase expects, you can guide development in such a way as to minimize backlash and maximize satisfaction with the end product.

What Is Your Greatest Frustration with the API?

Often, issues with an API aren't communicated directly — not out of a lack of channels for communication or out of fear — but out of simple embarrassment or perceived "bother" for the developer. Users can think "well, this is a beta API, so I won't bother them with a request; hopefully it is resolved in a later revision". Still other users can say "maybe this isn't an API issue, but *my own* issue… I'm not a very good coder, after all".

Much of this thinking can be harmful to the API ecosystem. By assuming the fault lies with the user, and not the provider, legitimate issues often go unchecked or unmanaged, only to be found out at a much later date as part of a bug audit or a feature-breaking update.

The best way to work around these issues is to engage the user in a conversation about what they perceive as "frustrating". Instead of asking a leading question, such as "where does your usage often fail" or "what do you feel you can't do", **ask about their frustrations**. This will lead to some greater insight about the functionality of your API, and can potentially highlight issues that may otherwise be unaddressed.

Asking about common frustrations helps to inform where your user experience fails. When a user runs into something frustrating, sometimes it's a result of confusing navigation, poor documentation, or faulty functionality notations. Highlighting these failures and addressing them improves the user experience, and thereby improves the quality of your API.

Secondly, allowing your users to vent frustrations helps guide development by showing weaknesses in functionality. When frustrations arise that aren't related to documentation or other similar issues, they're largely because of poor functionality. A user might find a call frustrating when it doesn't perform as expected, or returns incomplete data.

While this is often corrected on the user side or processed through error-correction, finding these issues early on and correcting them helps put development on the right track. Identifying common issues and rectifying them can turn a middling API into a truly useful and functional service for the API's users.

Finally, and most importantly, providers need to create a **communication channel** with developer users. Whether this means having official API forums, dev Twitter handle, a public email address, or even just a custom Google form, ensuring there's a path to vent and discuss is just as important as accepting this feedback.

Why Did You Choose Our API?

A "potential user" is of limited metric use, as they're a complete unknown. Potential users are wildcards, and attracting them to your API in the first place may be a complex discovery process. "Current users", however, represent high value and important metrics because they all share **common interests** that drew them to your API. Metrics are an incredibly important and powerful tool for API developers, and failing to tap into these types of metrics could doom an API to obscurity and low user integration.

Asking **why users chose your particular API** over the bevy of other choices on offer does a lot to inform the developer about not only the specific user's wants and behaviors, but helps define your unique value proposition, and aids your marketing attempts in targeting others in their set demographic.

The API economy has evolved, and their are now many types of consumers using APIs. You likely have a wealth of demographic data on your specific user; age groups, hardware and browser profiles, geographic location, etc. Pairing this profile with what specifically enticed different consumers to your API could be powerful knowledge for segmenting lean marketing campaigns.

If you know why people flock to your API, future development can be geared toward specialization, emphasizing the qualities and functionalities considered "unique" and "attractive" to these users, while mitigating the negative aspects that might otherwise have turned them away.

Keep in mind however that this sort of data can be overreacted to, leading to "mob rule". While adjusting to the wants and desires of the majority user group is important, it is equally important that developers attempt to stave off feature creep and bloat. This philosophy is most commonly referred to as the 80/20 rule in agile software development.

If You Could Change Our API, How Would You?

Sometimes the best way to get useful, actionable information is to just flat out ask for it. Asking your users **what they'd change about your API** is akin to that age old "if you were President/King" question, and opens up an avenue of direct change that would otherwise be obscured.

The key to implementing this question effectively is to ask for specific, actionable responses. A response like "better integration with media services" is not an actionable response, as it doesn't list the services which the user would like to better integrate with, nor how those services tie into the API functionality.

A better response would be something like "increased tools to integrate with media extensions to the APIs data handling suite". When responses are given, extra details should be requested as part and parcel.

Keep in mind that it must be clearly communicated to users that not all changes are possible. Changes to core functionality outside of planned expansion, changes to how services interact when it would cause feature breaking, and so forth should all be noted as caveats.

Methods to Use for Accumulating Feedback

This is all well and good, but all the questions in the world won't matter without having an effective means by which the question can be asked and gathered. API providers can quickly find themselves in a sort of catch-22 — getting this feedback is important, but the source of this feedback can determine its liability, and the method of procurement could even drive consumers away.

Likewise, understanding when to ask these questions is just as important as figuring out **how** to ask them. There is no magic bullet type answer to this question, but if we look at two theoretical applications of these concepts, we can see some things to avoid, as well as some methods that excel.

Example One - Ineffective Questioning

An API provider by the name of KAL Laboratories, or KAL for short, is performing a survey of their userbase to improve profitability and identify new markets. The lead programmer, Shawn, decides to start the questionnaire with a bevvy of technical questions.

These first few questions hinge around the languages used by the API, and contains questions such as "do you like the Twitter API integration that we did with the data handling package?" and "what do you think about the use of Go?".

The questionnaire gets passed on to the public resources specialist, Sandra, who adds her own questions. Things like "did you know about our work with non-profit charities?" and "what are your favorite websites?" abound.

Finally, the questionnaire is lightly edited, put into a stock form, and is blast-emailed to all the users who have registered their email as part of the API registration process. The questionnaire gets very few answers, and the userbase declines.

What Went Wrong?

Straight off the bat, there are some huge issues with the methodology by which this questionnaire was constructed. First and foremost, the scope is extremely broad — identifying new markets and increasing monetization revenue is a huge topic and one that respondents could easily feel put-off by.

Secondly, the questions aren't very useful. Because the tone of the questions vary wildly depending on who is asking them, and go from high-technical to "fluff", the questionnaire would be hard to take serious at best, and annoying at worst. Even if the questions were well-formed, providing them in a stock form without branding or explanation makes it easier to disregard the survey.

Finally, the questionnaire was distributed in an annoying way. Respondents likely did not know their emails would be used for analytics like this, and as such, when they are inundated with what is essentially spam mail, their opinions of both the API provider and its product will decline rapidly.

Example Two - Effective Questioning

Seeing the poor response to their first questionnaire, KAL decides to re-evaluate their approach. First of all, they look at their motive and questions. Their original stated goal was to investigate how to "improve profitability" and "identify new markets". While these are understandable goals, they are not properly framed. Improving profitability comes with understanding why profits are artificially depressed, and identifying new markets comes with understanding what your API does well (and equally what it does poorly).

With this in mind, a team meets to discuss a new questionnaire. Instead of relying on only two people to construct the survey, questions are submitted to the PR team for vetting, a handful of 10 are selected, and the survey is restructured to ease feedback with easy questions up front. The entire experience is reviewed and refined internally.

A limited trial begins. A set of "power users" are asked if they would like to take a questionnaire during a routine support conversation. They agree to do so, and take the survey. They give their answers not only to the questions on the survey, but to pointed questions about the quality of the survey itself.

With this information, the team again reviews the questions and vets them before issuing a general email call. This time, instead of cold calling their users, they simply notify the userbase that they will begin issuing periodic surveys to improve functionality — and that users may simply opt out to not receive them.

After a short period, the first survey is sent out, metrics are analyzed, and the activity is deemed a resounding success.

What Went Right

The most important thing here is the fact that everything was done via a process. Before, everything from the construction of the

survey to its application was haphazard. In this variation, the team not only pinpointed what specifically they wanted to know, but discussed *how* to ask.

This **vetting** is key to the process. While an engineer might legitimately want to know what a user thinks of their complex code and resultant front end, the question should not be "What is your opinion of our front end live page?", but rather, "Is our portal easy to use?".

Next, the team reached out to a select number of trustworthy users to test these questions. Being able to test questions in a **microcosm** gives you the benefit of extrapolating general responses to your survey without actually incurring the cost of performing the survey.

Finally, once these questions were tested, the team reached out to users to inform them of their intents — and to give them an **opt out** option. Forcing questions upon your userbase is ineffective and counterproductive. Rather, tell your users why your survey exists, and make it completely optional. This gives the user a sense of control, and increases the value of their response as well as the frequency by which the average user will respond.

Think As a User

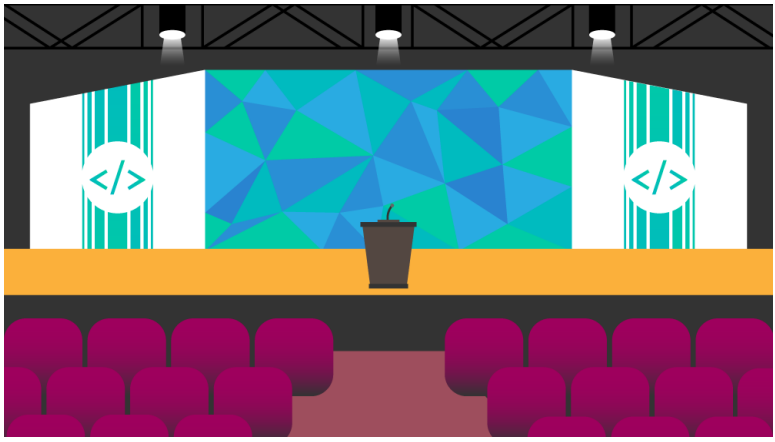
At the end of the day, compiling these sorts of questions is incredibly **useful** for an API provider. It helps validate your product's direction, as well as the viability, necessity, and desire for additional services with new monetization possibilities.

The takeaway from all of this is simple — **think like a user**. Imagine logging into your email one day to find a huge survey foisted upon you by a company that you at one time thought as non-intrusive and privacy respecting. Imagine this survey is filled

with grammatical errors, non-sensical questions, and confusing terminology. How would you respond?

Keep this in mind while forming your questions and conducting your surveys — it could mean the difference between informative analytics and a ruined reputation. If done right though, a survey process will demonstrate you truly share your developer user's journey and path to success.

How to Hold a Killer First Hackathon or Developer Conference



Part of what makes modern API development so much more powerful than ever before is the fact that, at the tip of one's fingers, an entire world of lessons, examples, and fellow developers are within reach. **Community** and the resultant feedback has made API development and the APIs that are created from this process more extensive, more powerful, and more effective.

Fostering this community is of prime importance. As an API provider, hosting a **hackathon** or **developer conference** can not only help increase the visibility of your API, your techniques, and your company, but can also create connections between developers and a sense of fellowship around your service.

In this chapter we discuss exactly what hackathons and developer

conferences entail and how to host them, including what works, what doesn't, and what pitfalls may arise. We'll consider how to market these events, and how to make them a truly powerful tool for advocating or evangelizing your third party developer program.

Types of Get-Togethers

For people new to the concepts of hackathons, meetups, workshops, and developer conferences, let's quickly define exactly what these events are.

What is a Hackathon?

Hackathons are events in which programmers, providers, and team members related to development collaborate in the usage, creation, or manipulation of software or hardware projects. Hackathons might be free form, where developers are given code or API access and are allowed to do what they wish, or might be guided, wherein a cause or goal is given and used to guide programmer efforts.

In the API space, what this functionally means is that a hackathon is a "get together" focused on skill implementation. Providers and developers unify their efforts to create something unique that consumes the provider's API in some way. This collaboration and implementation of divergent ideas and processes can reveal new development methods, highlight core strengths, and can even help highlight deficiencies in the current API space or system being demoed.

What is a Developer Conference?

A developer conference, on the other hand, is typically a "show and tell" style conference in which developers, programmers, users, and sometimes even other API providers are brought together to see what an API provider has to offer. This might take the form of live demonstrations, code workshops, or private API servers for testing.

Conferences often have a roster of speakers, keynote speeches, and parallel tracks, adding huge value to the attendees. Essentially, a conference is a direct pipeline to the minds of the best and brightest in the given topic, which is incredibly powerful and a great sell to most users in the API space.

More often than not, developer conferences are sponsored directly by API providers or API management providers for the purpose of increasing the visibility of the service in addition to establishing a knowledge-base of the API services and functionalities within the developer community.

There are exceptions, however, such as conferences where API documentation companies or groups representing developer consortiums gather to instruct one another on the functionality of various APIs and on business practices in general related to the API space.

What's the Difference?

What's the difference between the two? Hackathons specifically focus on the interactive nature of collaboration to spread knowledge and marketing about an API by allowing users to extensively use the source system. In essence, this is a form of "viral" marketing. When hackathons are supported, funded, and built by and around an API provider and their API, the word of mouth conversation about the hackathon will naturally disseminate information.

For example, if AlchemyAPI sponsored an API hackathon to develop experimental and extensible applications and systems utilizing the API, the word of mouth conversation would necessarily tie the successes of the API hacks into the AlchemyAPI itself. Likewise, developers who utilize the AlchemyAPI would be more aware of its power and functionality, and would be more likely to use and recommend it to fellow developers who are developing applications that might make use of the API.

Developer conferences and meetups, on the other hand, are much more instructive, and rely on direct dissemination. Whereas a hackathon might impart knowledge based on the interaction with a system in a loose and free nature, the developer conference specifically sets out to answer specific questions, and provide specific solutions. A simple analogy would be that a hackathon is an internship, while a developer conference is a classroom.

How to Host an Event

That being said, how does one actually host such an event? Many few factors must be considered before inviting your guests — if the following is considered, followed closely, and implemented correctly, your first hackathon will be a killer powerhouse, and your developer conference a smashing success.

1: Identify Your Goal, Theme, and Audience

Key to the success of these meetups is understanding exactly why the API provider is holding one. Is this for marketing? Is it for building knowledge? Is it for getting users excited about new systems? Identifying a **mission statement** early on is absolutely necessary, as it will guide your event theme, speakers, sponsors, marketing, presentation materials, and audience.

Knowing who your attendees will be is critical. Are you trying to imbue knowledge to students and foster the next wave of API programmers? Hosting your event at a school will attract this raw talent and establish yourself in the minds of young programmers. Are you rather focusing on business? Utilizing a hall or conference center would be a great choice, as it instills a sense of professionalism, and often provides a greater range of business services to clients and attendees.

Themes that focus on a certain technology or sector can go a long way towards making a meetup unique, and will also inform you as to the answers for the following questions and hypotheticals in this piece.

2: Secure the Financing and Venue

In terms of business interests, securing a **venue** and the **financing** for the event is of prime importance.

You must consider what sort of venue is appropriate given your circumstances. While an international API provider handling millions of contacts a day might opt for a convention center, this is far beyond the means and needs of other startups or SMB providers.

For smaller providers, a simple clean space will do just fine. Ensure that you have a venue that has decent connectivity for distributing API packages if you are connecting to an external server. Otherwise, ensure that you have production-ready local servers that can be used to handle the amount of people you are considering inviting to the meetup.

Once this is done, consider the system requirements of your conference. Is this a "bring your own device" conference, wherein users must bring their own laptops and systems? If so, consider storage spaces for equipment containers and extraneous system paraphernalia.

Expanding upon this, ensure that your conference system requirements match to your expectations. Ensure you have the following:

- Server hardware capable of handling your predicted attendance pool plus 25% to ensure load balancing is not an issue;
- Projectors, screens, HDMI cables, adapters, or other presentation platforms capable of displaying high resolution to the furthest attendees given the size of your attendance group;
- Wireless access points and repeaters near "traps", such as corners, thick doors, etc., that might kill wireless coverage;
- Load balancing devices and security systems - dedicating a security appliance to a subnetted network can go a long way to securing your network.

If you intend on giving users host machines of their own, consider whether this is in the budget or feasible given your goal. For developer conferences, it might be simpler to rent out a co-working space usually reserved for freelance programmers. For both hackathons and conferences, it might even be feasible to find a University or College which will allow you to use conference rooms or computing spaces.

In terms of financing, your main concern should be whether or not the meetup is free. **Free** conferences and hackathons are necessarily more expensive than paid, "entry fee" ones, but might result in greater attendance.

On the other hand, paid conferences can afford to provide more services and better equipment, but might attract fewer people based on cost. Much of the expense of paid conferences can be negated with **sponsors**, however, and this is becoming more and more the case in recent years.

3: Get People Excited

Let's face it — API development can, at times, be a dry affair for even the most passionate developers. Now imagine you are in marketing or business development, and you have been invited to a developer conference, or your team has been invited to a hackathon and you are to be their point of contact.

Many people's first response would be "no, this is boring". Assuaging that feeling and making people excited for these meetups is no small task, but with a little planning, you can turn that "boring" comment quickly into "wow, that sounds amazing!"

Consider adding a **challenge** to the conference, one that has an amazing award. Hosting a hackathon? Consider having first, second, and third place prizes for the result, each with a monetary or high-value prize. Developers might be more willing to attend a conference for the promise of three months of free server rental or a cash prize. Not only does this investment get people motivated, it also shows that the provider is interested in giving back to the community.

Prizes add subconscious value to an event by matching the effort required with a possible reward. If someone asked you to help them move, what would your first response be? Now imagine they ask you to help them move — and told you there was a coin flip at the end of the day, and the winner would take home $1,000 USD. Now what is your response?

Additionally, consider location and **travel** expenses. A team might be wary of visiting a hackathon in a small town far away from major tech centers. If able, hosting an event in a major city or tech hub could do wonders for attendance, and can dramatically increase the success of your meetup.

Finally, consider this age old truth — people love to eat. Providing snacks or even catering meals during the event would dramatically boost morale and interest in the occasion. Strapped for cash?

Consider inviting food trucks or stands to your event parking lot. While you might incur additional cost for the parking area, giving your patrons options for healthy onsite food does a lot to get people excited.

4: Prepare for Pitfalls

No event is perfect, and no matter how much you plan, something will inevitably go wrong. Understanding potential pitfalls to your first hackathon or developer conference is paramount to your success.

First, understand that **networks are unpredictable**. Many a conference has gone awry when providers simply can't get the WiFi to work, or have limited network access. To prevent this, try not to use built-in networks whenever possible.

Running Cat 5e cables and covering them with floor strips might be aesthetically displeasing compared to using the almighty wireless access point, but attendees won't remember wall warts and cable runs if you do everything right — they will, however, remember being unable to use your system at all, and this experience will directly reflect in what they have to say about the conference after the fact.

Use production ready servers with the understanding that nine times out of ten, nothing goes wrong — but that one time, everything goes wrong. Allowing servers to update could fundamentally break huge sections of your system, server failures or RAID cluster failovers can break database demonstrations, and even a misconfigured network card attached to the main network can break functionality depending on the network design.

Consider this when building your network, and install redundancy at every step — the only person who knows something went wrong should be the event host. The audience should not even notice a failure if you're doing it right.

Food poisoning is something nobody ever forgets — and it will always be tied to you and your conference if you're not careful. Catering is typically safe, depending on reviews, but hiring food trucks or operating in a venue that provides a food court can negate this threat and shift responsibility somewhat.

Environmental issues are also extremely important to consider, both in terms of impact and surroundings. In terms of **environment**, providing proper methodologies and locations for disposing of technological waste and general waste is incredibly important, and can certainly influence the perception of any given conference.

Make sure your venue is clean, that it has air conditioning, and that the design of the presentation booths provide for optimal presentation spaces, safety, and flow of the crowd. A great example of failure in this realm is the 2016 Google I/O conference, which is infamous for its long lines, sweltering heat, and generally poorly managed facilities and surroundings.

Also consider event **security**. Malicious attacks are not beyond the range of possibilities, and you need to plan accordingly. Implement cyber security systems, secure local routers and hubs, consider using baseline security filters, and consider providing custom systems with security implementations that won't harm the network. Secure the production servers from other servers containing personal information or company documentation, and create a virtual private network for any essential equipment that must share this information at the conference.

Finally, a huge element of a great conference is having capable **presenters**. Having an "MC" who can direct the flow of the conference, announce events, and coordinate all of the solutions presented above can go a long way to not only preventing these issues from cropping up, but negating the issues that arise when you fail to do so.

5: Don't Sweat It

Hosting these meetups improves the community and integrates the API provider as a source of knowledge. Bringing people together to learn about or use your system builds community knowledge of your product, develops skills that can then be brought into the fold either through volunteer efforts or direct hiring, and brings the community closer together.

In conclusion, the best piece of advice that can be given to an API provider is simply this — "don't sweat it". The API space moves so fast and is filled with so many entrepreneurs that you can always call upon others for a little bit of help.

Has your system failed? If you're in the right location, networking supplies and servers should only be a phone call away. Ran out of food? If you've planned ahead, ordering in a few more trays of food should be simple. Network facing internal attacks? Cut off the virtual private network behind a DMZ and find the source of the problem.

Once you've planned it out, let the conference or hackathon be what it's going to be — forcing things will only cause issues, and if you've taken this advice to heart, it should "run itself".

What Makes an API Demo Unforgettable?

This is the Age of the **API** (Application Programming Interface). No matter what type of software you work with, you will likely have to deal with APIs on some level. When a developer visits an API website, you will see plenty of documentation, sample code, use cases, and endorsements that make you feel one click away to start consuming. However, there is something that can convince you more than a website can: watching an amazing live API demo.

If you are on the other side of the story and have your own API, you **must** prepare an API demo. Your API must compete for attention and then convince others that — among all the existing alternatives — it is the best solution and is worth trying. Usually the Developer Evangelist is the person jumping to the floor with a laptop to rock the stage with their API demo at a hackathon, meetup, or conference. If you are a developer building APIs, it's likely that sooner or later you will have to demo your API at these sort of events as well. Whether you've done it before or not, knowing the best practices is invaluable.

After observing a large number of great API demos — from amazing to dull — the best, most **unforgettable API demos** have the following six points in common:

1: Describe the API, in a few words.

Finding a clear and concise way to describe your API is an obvious first step. Great demos introduce the API in a very concrete way, starting with real situations. Speaking in abstract terms is out of question, and in practice it's very easy to get lost in acronyms (HTML, REST, SOAP, JSON, etc) and developer jargon. You must answer the question "Why does your API exist in the first place?" Using metaphors, stories, or examples of people with real names can be helpful aids. The optimal explanation isn't either too long or too short, like in this example where SendGrid's Swift explains SendGrid API in a nutshell:

> "SendGrid is an API company that makes it super simple for you to send and receive emails from your applications. A turnkey solution for any time you need to deal with anybody's email on your app: password resets, alerts, etc."

2: Convince we all share the same values of the API

This is an easily underestimated point. Doing it will make sure you have identified a connection between you or your company and your audience: we have the same struggles and **we all are developers creating amazing software and applications to solve people's problems**. Developers attend hackathons and technical conferences to learn and try new things, and often to discover solutions to challenges that they already face.

Convincing the audience of a **shared value** will create a stronger emotional bond than just ending the demo saying "you're welcome to try our API". In a presentation with Nordic APIs, Ronnie Mitra

does a fantastic job of establishing shared values by reinforcing the importance of user experience design with a worst case presentation scenario — great acting!

3: Impress with how great and easy your API is

There are many aspects that make an API outstanding, such as easy onboarding, usability, scalability, maintainability, security, stability, support, and compatibility with other major solutions. However, even if you API excels in all these areas, some are too abstract for a brief demo.

An unforgettable API demo is created as a **performing show** that in addition to educating will entertain people. One of the requirements is that the code looks really easy and short. A good way to measure the length is asking yourself: do I need to scroll down the screen? A great example is Twilio's John Britton demo. Pay attention to the simplicity of the code. That really impresses. This demo created a conference call that could be joined by anybody ringing a randomly selected local number. This short demo ends capturing and displaying the full list of phone numbers of the participants (Surprise!).

4: Interact with the audience

Even though these types of demos are usually short (less that 10 minutes, even as short as 3 minutes), just watching a text-based code editor can be boring. Great demos involve the audience. There are many ways to do it: ask participants to take their own gadgets and open a webpage, write and send either an email or SMS, ask to cast a vote, look at your camera, shout, etc. One of the

most accomplished demos is again by Twilio's John Britton, who interacted with dozens of participants.

A strong reason to interact with the audience is that API demos often force the speaker to stand behind a lectern. Have you noticed this? In contrast, think of how many times you have seen a TED talk speaker behind a lectern. That's indeed very rare. Interaction breaks the barrier that is almost inevitable in every API demo.

5: Live coding mastery

> "Talk is cheap, show me the code" —Linus Torvalds

Certainly the main element of an API demo is **live coding**. Live coding mastery involves a number of good practices that when put together produce amazing results. Some elements are:

1. *Making your code short.* The live code itself must be very short (1 minute in total, which can imply 2 or even more sections) and leave the rest of the time for the audience to see the results, and even interact with the outcome.
2. *Make it easy to display code on the screen.* Rehearse in advance to see what is the best way to display your code. This will involve the choice of your **code editor**, the color of text and background (usually black characters in white background are the best), and even the fonts. Take into account that the quality of the projector, screen, and lighting of the room will affect how your audience sees your code.
3. *Explain what you are writing in sync.* Say in a clear way what every line you write is going to do. It will help tremendously if the names of functions are self-explanatory. Needless to say, if you already follow good coding habits, this is easier.

6: A theater-like script

If you ever performed in theater, you must have learned and memorized a **script** by heart, maybe adding your own interpretation to the lines. For an API demo, similarly prepare a step-by-step script for yourself. Why this is so important? You must know exactly which parameters you will use for every example — the order of steps makes a huge difference. The successful guys who have had memorable API demos are all said to have rehearsed the same sequences dozens of times. Some claimed to have even dreamt about it.

Another huge advantage of using a written script is that it makes your demo repeatable. If your next meetup isn't for a few months, you don't have to rely on your memory to start preparing yourself again. **An API demo is a play, not improv**.

An extra piece of advice is: reduce the number of steps to the minimum, and minimize the number of parameters. All this will reduce the probability of failure.

Preparation for potential technical flaws

Things we don't see also matter. It's highly recommended to have a list of technicalities in advance. Some of them can be checked well in advance and others when you are at the venue. Normally we don't see it, but a well-prepared developer will do this before the API demo. It can save you from a disaster and help you to recover quickly.

Here's a list of items that would have a big impact if they fail:

Item	How to be prepared
Laptop	If using your colleague's laptop, make sure it's configured with the same settings ready for the demo
Internet connection	As Internet connections might have unknown restrictions, have your own mobile connection ready
Web browser	Have a second browser installed and checked that the whole demo works flawlessly
User account	Create an extra user account with same privileges
Cables and accessories	Bring extra cables, adapters and other accessories. Pay special attention to this if you visit another country

A final piece of advice is: time yourself and stick to the allotted time slot. The most successful API demos are short and to the point. So, when it's your time to rehearse, set an **optimal time length** and repeat the demo until you and the chronometer are in sync.

Conclusion

Following these six best practices and preparing for potential flaws will make your API demo unforgettable, and will make you or your company's end goal of stimulating developer adoption more realizable. Great demos have the power to transform a technical event into an enjoyable, memorable, and inspiring experience. Ultimately, it will inspire developers to build great things with your API, which will bring value to end users. Feel free to share other amazing API demos in the comments section and tell us why they were unique.

We wish you a successful next API demo. It's API time. It's showtime!

Case Study: Twitter's 10 Year Struggle with Developer Relations

In September 2006, only a few months into its existence, Twitter came out with the first version of its public API. This was surprisingly early in an age when social APIs were not yet prevalent, especially since Twitter had yet to become the success story that seems so obvious in hindsight. In fact it was mostly a reaction to third party developers scraping their website Twitter.com and creating unofficial APIs.

It marked the start of a long and at times contentious **love-hate relationship** between Twitter and third party developers. In this chapter we delve into some of the events of the last 10 years and explain the motivations behind some of Twitter's most controversial decisions. We'll also compare Twitter's record on **developer relations** with those of two other social networks - Facebook and Instagram.

2006 - 2010: The early days

The first version of Twitter's API was a free-for-all and instant hit. It opened up a lot of the social network's inner workings to developers, enabling full access to tweets and content published by users on Twitter. Any developer could use the API — they

simply needed to authenticate using the username/password combination of their regular Twitter account. No further limitations were imposed. The sheer amount of data offered generated plenty of interest among developers, and many companies started building products on top of the Twitter API — apps included Favstar.fm, DailyBooth, TweetDeck, Tweetbot, Echofon and Twitterrific.

Just as Twitter's user community was credited for inventing some of its defining features, like hashtags and retweets, the **API developer ecosystem** also paved the way forward. Among the early third party apps, TinyURL and bit.ly enabled URL shortening within tweets, Summize offered up the first full-text search engine on top of Twitter, TwitPic let Twitter users share pictures, and Tweetie — not Twitter — built the first Twitter iPhone client.

Some of the other apps (like Echofon and Tweetbot) were Twitter clients that offered an alternative to Twitter.com for users who wanted to consume the same content through a different user interface (Twitter would later call them 'traditional Twitter clients').

2010 - 2012: OAuthcalypse, competing with third party apps and other perceived betrayals

The honeymoon between Twitter and third party developers ended in 2010 when Twitter's management enacted a series of decisions that were greeted with anger and disappointment.

Twitter announced that all **third party applications** requesting data on behalf of its users needed to authenticate to the Twitter API using the OAuth protocol. This change was inevitable for security reasons — OAuth was quickly becoming the standard for secure delegated authentication and authorization, and afforded a much better alternative to the Basic Authentication scheme that was in

place before its adoption by Twitter. It also afforded the company better visibility over which applications were connecting to the API.

However this change caused distress among the ranks of the developer community, as many apps had depended on their API for years and weren't ready for such a change. OAuth was not yet widely adopted at the time, and **many developers struggled** with its implementation, not to mention the changes in user experience it brought along. Bloggers were quick to call it the 'OAuthcalypse', and '#OAuthcalypse' became a trending topic on Twitter itself.

While imposing OAuth is easy to defend in hindsight, Twitter also started building features within its own website that competed with or replaced existing third party apps. Starting a trend that it would repeat several times in the future, it purchased Tweetie (renaming it Twitter for iPhone) and Summize (renaming it Twitter Search) and announced that they would become part of the core platform, striking fear in the hearts of developers building competing services.

Twitter developed their own URL shortener and threatened to deny all others. During Twitter's Chirp developer conference that year, then-CEO Evan Williams pronounced: "We are probably not going to give people a choice. If they want to use a different shortener, they can use a different app". Once again this was badly received, although Twitter had solid reasons for introducing a native URL shortener at the expense of competing specialists — namely security and analytics.

Finally, Twitter signed partnership deals with Google, Yahoo, and Microsoft to include tweets in the results of their respective search products. Furthermore, it gave exclusive data reseller access to Gnip, a data reseller it would later acquire. Both of these decisions, motivated by Twitter's thirst for additional traffic and their need to generate revenue, caused consternation when they were announced.

2012 - 2013: Token limits and open war on traditional clients

Beginning in 2012, Twitter further tightened their terms of service. It imposed a 100,000 token limit on connected users for apps that "mimic or reproduce the Twitter consumer experience", crippling many of them, and set per-endpoint rate limits across the board. OAuth authentication became mandatory for all endpoints, and Twitter introduced Embedded Tweets and Timelines as an alternative to applications that wanted to replicate part of the core Twitter experience in their own apps.

Two main reasons explain these changes. Having raised huge amounts of venture capital, and in preparation for a late 2013 IPO, Twitter needed to increase its ad revenue. As the conflict between its willingness to attract developers and its need for monetization increased, Twitter started to openly discourage the use of the API by those building tools that would **compete for eyeballs** with Twitter.com. In addition, Twitter had to some extent become victim of its own success and had faced many technical problems and outages. Curtailing the use of its API would help control the amount of traffic that was allowed from third party apps.

In the wake of these changes several Twitter competitors such as App.net were started, but such was Twitter's dominance of this space that none have come close to dethroning it.

2013 - Present: Post-IPO controversies

Any hopes that developer relations would significantly improve following Twitter's IPO would soon be dashed.

In 2015, shortly after live video-streaming startup Meerkat became a hit, Twitter acquired its competitor Periscope and temporarily revoked Meerkat's access to the Twitter API. It also acquired Gnip and shut down agreements for resale of data with its other partners (DataSift and NTT Data), which was qualified as an "evil move" and "innovation destroying".

Later the same year the Open State Foundation's API's access was suspended. The Foundation had leveraged Twitter's API to archive deleted tweets by politicians and make them searchable on its website Politwoops. Twitter later restored Politwoops' API access, but this was another blow to its reputation as an open platform and an enabler of innovation.

Most recently, in November 2015, Twitter removed the Tweet count JSON endpoint to public outcry. The REST API still has the same features, but it is less precise and exposes the data in an aggregated or limited manner. The only way to get the same analytics data is through Gnip's expensive Enterprise Search API.

Wooing back developers

Despite all of these controversial decisions, Twitter's developer ecosystem remains healthy due to its unique status as a major social network. Many companies built on top of Twitter's APIs are alive and doing well today, like Buffer, Hootsuite, SparkCentral (formerly Twitspark) and Storify.

Nevertheless Twitter is **facing challenging times**, with layoffs, a change at the helm of the company, executive departures and a declining share price. Twitter's management understands that they need to invest in a strong developer ecosystem to remain relevant going forward.

In particular, Twitter needs to find new monetization channels, and the chief battlefield is mobile, where Facebook is currently

outperforming Twitter, in part due to lack of developer interest in the Twitter platform.

In the last few years **Twitter has turned its focus towards mobile developers**. It has recently developed or acquired a set of new tools to help developers build better mobile applications while using the Twitter API, and indirectly generating more revenue via their MoPub mobile ad service.

Developers are therefore critical to the next phase in Twitter's evolution, but its reputation is holding it back. At the Flight developer conference in October 2015, newly reappointed CEO Jack Dorsey offered an apology to developers for past mistakes, echoing the words of his predecessor Evan Williams five years before, and promising to usher in a new era of collaboration between Twitter and its developer community.

New releases and optimism going forward

Beyond these soothing words, Twitter showed how determined it was to win back developers by announcing its new capabilities. Many of these revolved around Fabric, its mobile development platform, which now includes the popular **mobile crash monitoring** and reporting tool Crashlytics, a recent acquisition, along with two key new features: Beta, an app distribution and tracking tool for beta testers, and Answers, a mobile analytics dashboard.

In addition to Crashlytics, Fabric features the Twitter Kit, the rebranded Twitter SDK for mobile development, and Digits, a simplified authentication mechanism based on phone numbers rather than usernames and passwords (an enabler of SMS-based services). Another recent acquisition that made its way into Fabric is Fastlane, a tool that eases the deployment workflow on apps for iOS and Android.

Time will tell whether these efforts will convince developers to give Twitter another shot, and if the tension between the company and its developer community will subside. One reason to be optimistic is that Twitter is consciously steering developers away from building features on top of its core product and is positioning its API as an enabler instead, in the same vein as developer darlings Stripe and Parse.

Twitter is often singled out for its problems with the developer community, but its advocates argue that this is an unfair assessment, pointing to Twitter's numerous contributions to **open source software** (such as Bootstrap, Bower, FlockDB, Gizzard and Finagle).

Certainly, other social networks have had similar woes. In the next section we'll briefly look at two other giant social networks and how they handled relations with their respective developer communities.

Other social networks

Developer relations at Facebook

Facebook had the relative luxury of having a stable vision (and a stable website) before public social APIs became fashionable. Launched in 2006 — shortly before Twitter's API — Facebook's Developer API gave access to its users' friends, photos, events and profile information. This ushered in a golden age of social gaming, with game developers like Zynga, the makers of Farmville and Mafia Wars, and PlayFish, the makers of Pet Society and Who Has The Biggest Brain, benefiting from viral effects that only Facebook could enable. In addition to games, applications that leveraged Facebook's social graph, like Zoosk, got a huge boost from the Facebook platform.

A few years later though, as users complained about spammy apps and games spoiling the user experience, Facebook gradually curtailed the viral marketing tactics that powered these applications, thereby also limiting its own appeal as a development platform.

Facebook had its share of failures with their developer API, and was the recipient of a great deal of criticism from developers. It unsuccessfully tried to launch a virtual currency during the early years, and more recently shut down Parse, their mobile BaaS acquisition beloved by mobile developers, although they allowed the code to be open sourced and gave developers a one year notice period to migrate away.

One area where Facebook has made a mark is in **delegated authentication** — their OAuth API is highly popular due to the sheer number of people with facebook accounts and the social proof that users can benefit from when they connect to an application with their Facebook account.

Developer relations at Instagram

Instagram seems to have benefited from being a late addition to the ranks of super-social networks, with the advantage of hindsight helping them avoid mistakes, but at the same time suffered from lack of resources in the early years.

Instagram was launched in October 2010 and already had a million users after only three months. By that time, it was unthinkable for a major social network not to release its own public API. Because of its astonishing growth, Instagram was caught unawares as developers started clamoring for a fully-featured API.

The company wanted to focus on the core user experience, avoiding distractions, so they didn't immediately accede to the demands of their would-be API consumers. But because Instagram was a **mobile-first** product, it was possible in December of the same year for a developer called [Mislav Marohni] (https://github.com/mislav) to

reverse engineer its private API's initially unencrypted calls to and from the mobile UI and create an unofficial API. At least one fully-featured app (Followgram, a self-styled Instagram directory) was built on top of this unofficial API. In January 2011, Instagram shut it down though, and released an official API in February.

In time its main features included the ability to search for pictures by hashtag or location, the ability to incorporate pictures into a website or a mobile app and to print photos. Within days, a host of photo sharing apps and mashups were released by third party developers. The more popular ones surviving to this day include Websta, Flipboard, Casetify, Gramfeed, Collecto (formerly Followgram and Social Print Studio.

Following the Facebook acquisition in 2012 (and given the need to monetize its hundreds of millions of users), Instagram has been able to branch out in new directions, notably a very successful ad API.

Like Twitter, Instagram has recently clamped down on its public API consumers, and has announced that it would now review apps before letting them use the API in production. This follows from the InstaAgent scandal in which an application was used to steal user credentials.

Review

Throughout the course of this volume we've covered much ground. As with much in life, the keys to success lie in **planning**; knowing your developer consumer, and positioning your technology to meet your platform goals. With great API power comes great responsibility to your developers, so in this volume we also covered guidelines for crafting impressive developer portals, libraries, and other helpful resources like code tutorials that will help spur adoption and healthy **developer relations**. Usability is essential, but static Q&A pages are sometimes not enough. That's why quick developer support paired with personable **advocacy** programs, events, and demos can aid onboarding, and ignite a community of learned practitioners. But don't *undersell* the need to *sell*; take advantage of our discovery techniques, press networks, and API directories to promote your kit to the right channels. As stated in the Preface, API Marketing needs it's own strategy - now you have it.

TL;DR Checklist

In short, the DIY API Marketing process can be summed up in the following steps:

1. Perform research to hone in on your target developer; this will guide branding and help segment marketing.
2. Make your API discoverable through API directory profiling and machine readable automation.
3. Write press announcements that describe feature releases and disseminate them to our list of relevant developer networks.
4. Your developer center is both a knowledge center and important marketing facade; open functional understanding to entrepreneurs as well as developers.
5. Create developer resources (SDKs, code libraries) to bring your API into the hands of the many.
6. Consistently publish use cases, walkthroughs, code tutorials, and thought pieces to gain traction online.
7. If possible employ evangelists, participate in events, spread the word in person.
8. Use an active, but helpful forum presence to help users excel.
9. Build a community of API developer heroes.

More eBooks by Nordic APIs:

The API Lifecycle: An agile process for managing the life of an API - the secret sauce to help establish quality standards for all API and microservice providers.

API-Driven DevOps: Learn about the API-driven approach to uniting development and operations. This eBook combines all our writing on DevOps, the firestorm that empowers and extends capability for developers

The API Economy: Tune into case studies as we explore how agile businesses are using APIs to disrupt industries and outperform competitors.

Securing the API Stronghold: The most comprehensive freely available deep dive into the core tenants of modern web API security, identity control, and access management.

Developing The API Mindset: Distinguishes Public, Private, and Partner API business strategies with use cases from Nordic APIs events.

Nordic APIs Conference Talks

We travel throughout Scandinavia and beyond to host talks to help businessess become more programmable. Be sure to track our upcoming events if you are ever interested in attending or speaking, or visit our YouTube channel to watch sessions from previous events.

Endnotes

Nordic APIs is an independent organization 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.

Nordic APIs AB Box 133 447 24 Vargarda, Sweden

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact