The API Lifecycle

An Agile Process For Managing the Life of an API

AUTHORS

Bill C. Doerrfeld
Bruno Pedro
Kristopher Sandoval
Andreas Krohn

NORDIC APIS
nordicapis.com

The API Lifecycle

An Agile Process For Managing the Life of an API

Nordic APIs

Contents

Preface

Nordic APIs has been in operation for about 2 years. Within that short time we've hosted hundreds of API-specific presentations featuring international experts - giving us the opportunity to take a peek into the inner workings of leading web API-driven businesses.

Throughout our work we've discovered a recurring trend: unsuccessful APIs struggle both in fostering interest as well as creating a sustainable service. In contrast, successful APIs are well-attended, treated as a product with a multi-iterative lifecycle. So we thought; what exactly are the key ingredients to their successes?

Our fourth eBook, The API Lifecycle, presents an agile process for managing the life of an API - the secret sauce to help establish quality standards for all API and microservice providers. This handbook depicts a holistic model that treats a web API as a constantly evolving business and product. From conception through deprecation, it outlines instructions on market research, business strategy, development, operations, promotion, and other vital components for successfully creating and maintaining an API.

Inspired by experienced API consultants Andreas Krohn and Travis Spencer, the API Lifecycle diagram is composed of four distinct phases: Analysis, Development, Operations, and Retirement. Our eBook expands theses concepts, grounding theories in knowledge collected during our 2015 World Tour, events which brought API industry experts together to discuss different stages of the API

Lifecycle. We also delved deeper into these ideas within our blog, interviewing leading web API figures to perform additional research, and peppered it all with a bit of our own insights to create a comprehensive overview of what to consider during the entire product lifecycle of a profitable web API.

Please enjoy The API Lifecycle, and let us know how we can improve. Be sure to join the Nordic APIs newsletter for updates, and follow us for news on upcoming events.

Thank you for reading!

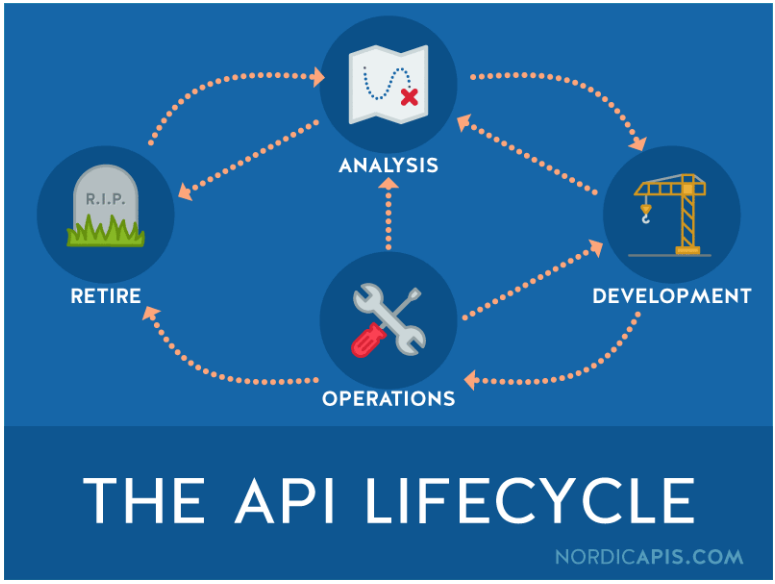
– Bill Doerrfeld, Editor in Chief, Nordic APIs

Connect with Nordic APIs:

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact

Envisioning The Entire API Lifecycle



From conception to deprecation, a Software-as-a-service (SaaS) is prone to constant evolution. Check the changelog for a high-volume platform like Dropbox for proof. In such logs, you'll find bug fixes and updates, with the latest likely made within the past couple weeks.

The APIs that open these SaaS platforms are changing just as rapidly. So much so that API Changelog was created to notify users of API documentation changes (22 updates in the last 30 days in the case of the Dropbox API at the time of this writing). This ongoing iterative process is not only

agile - it's evolution is in symbiosis with many business factors that make up the organism that is the entire API lifecycle.

As the main function of web APIs are to be consumed by third party products, the API is a unique type of business offering. It's peculiar condition as both a software and internal asset makes its lifecycle special – highly fueled by ongoing feedback and market validation. Compared with standard SaaS products that may have the power to change their UI or underlying construct, an API's granular state can make it even more elusive.

The State of Modern Software Development

Agile...Scrum...contemporary software development pushes toward quick releases and fluid responses. Response to change needs to be rapidly executed, leveraging user feedback, data, and statistics. These traits are becoming standard practice throughout the software realm. But is producing and maintaining an API the same – or is it a different animal entirely?

The API As a Living, Breathing Organism

Exactly how an API lifecycle looks depends on how the API functions as well as the business strategy at it's core. We at Nordic APIs have boiled down the common API lifecycle to four main phases: **Analysis**, **Development**, **Operations**, and **Retirement**. As seen mapped out above, these elements work together and influence each-other. Once the cycle begins, open and simultaneous exchange between each phase will help a practitioner stabilize the API against internal and external factors. New information that arises within each stage may beckon a team to move

forward or backward in the lifecycle, encouraging small revisions or large pivoting to reach end project goals. In this introductory chapter we will introduce the main phases of the cycle, each to be covered in depth in the following chapters to come.

#1: Analysis Stage

This stage involves defining overall API business objectives. Firms should critically ask themselves whether or not an API is right for their particular situation. Some industries, such as social networks, thrive off APIs as a natural extension of their digital ecosystem. On the other hand, traditionally non-digital companies must be creative in handling how APIs extend their business model.

An API should either boost pre-existing operations or work toward overall company goals in some way. In our e-book Developing the API Mindset we defined 3 types of APIs:

- **Private APIs** streamline internal operations to decrease expenses.
- **Partner APIs** work specifically with partner organizations to expand a service's reach to existing markets to generate revenue.
- **Public APIs** form new ecosystems by opening up a service to public access. If you intend to have a public release, obtain feedback while you perform

analysis to gauge interest and determine access control.

Whatever the chosen API strategy is, it's mission statement, usage projections, model for growth, marketing strategy, and estimated financial return should be clearly laid out before production begins. If you've done the necessary preparation, it's time to move to the **Development Stage**.

#2: Development Stage

With business proof and goals laid out, the next step is defining technical requirements and opportunities that are in cooperation with the API strategy. Development plans need to consider the following:

- what **operations** the API will execute
- **management** and **hosting**
- the methods and protocols used
- size and **scalability**
- **access** and security control
- **usability** and experience
- the development **team** ... among other factors

An intimate understanding of API processes will help form technical requirements. For example, knowing whether

your API will open your firms' data, or act as a content distribution channel can impact overall design.

In general, the API development stage should mimic the business objectives set forth within the Analysis phase. A methodical development phase considers design, construction and testing of API processes as well as security. Design your API with both human usability and machine standards in mind. It is a good idea at this point to also implement and evaluate API versioning strategies that will set the API in a good position for future updates.

During the Development Stage, a team may choose to return to the **Analysis Stage** to perform additional market research and general preparation. This can arise from technical roadblocks, or the urge to revision an APIs' primary objectives. If the API feels polished, tested, secure, and ready for general use, you are ready to move to the **Operations Stage**.

#3: Operations Stage

If you're team's lean as flank steak, you may be sprinting a minimum viable product straight to the masses. With loose nuts and bolts, it's natural that the Operations Stage will involve a lot of tweaking and bug-fixing in addition to marketing.

To spread awareness, API providers can host **hackathons**, improve **SEO** for home page and documentation, have a dedicated **@Dev** channel, and use popular **discoverability** methods and channels. If the resources are available,

companies should also employ a dedicated **API Evangelist** to promote the API at physical events and throughout the blogosphere.

In the end, marketing an API largely depends on an inside-out approach. Having an easy-to-consume development portal, an embedded **help desk**, along with other DX driven suggestions we've laid out in the past will help boost conversion rates.

This part of the lifecycle will likely take the most energy and time. Monitor usage statistics, and engage with users. Use any helpful data to aid in constant revising. Especially if an API program is in a closed beta, inbound tactics to collect feedback will help tremendously to aid in this iterative development process. Make sure that all API changes are well documented on your release notes.

This exciting stage will bestow on an API practitioner real market requirements, and could potentially expose a business to entirely new opportunities. Statistics and data collected here will impact tech requirements that could result in revisiting the **Development Stage** to make incremental changes or new API versions, or consulting the **Analysis Stage** to make larger pivots. If forecasts look grim, a team may consider moving to the **Retirement Stage** at this point.

#4: Retirement Stage

Old age comes, and retirement is a part of life. It's no different in the API world. It's good to know what factors contribute to small revisions as well as large deprecations. In this stage, a provider may decide to retire an API due to limited use, outdated plugins, a lack of 3rd party innovations, opposing financial incentives, and more. A

business may risk losing an advantage if an API decreases traffic from native distribution channels.

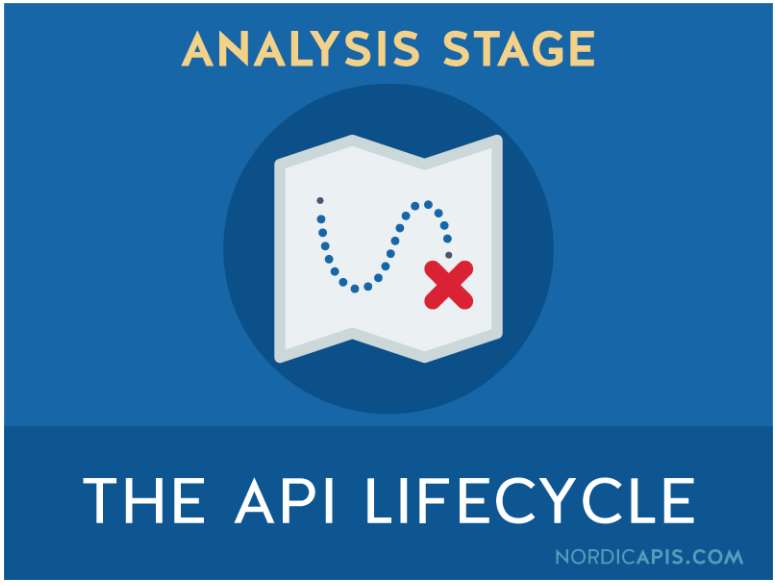
Versioning, deprecation, or lassoing in an API into a more internal state are options to consider if your key performance indicators are showing a flatline. At this point, the **Analysis Stage** can be reattempted, and the lifecycle repeated if viable. A Public API may still prove profitable as a Partner offering, for example. **Retiring** an API or API version is the best option when the API no longer is the best solution to reach the goals of the business. Whatever the solution is, we recommend a transparent public announcement that offers a reasonable timeline.

Choose Your Adventure:

In this chapter we've given a 30,000 foot view of the somewhat untraditional API lifecycle. Further chapters will offer fine-grained perspectives on each stage in the lifecycle, bolstered by specific industry use cases that demonstrate effective strategies for each. Here's where we're going:

1. Analysis Stage
2. Development Stage
3. Operations Stage
4. Retirement Stage

I Analysis Stage



An API comes with an active product lifecycle that's tethered between internal evolution and third party fluctuation. Within this part we'll begin to take a deeper look into our first step of the Lifecycle process to consider what preparation should be done within the **Analysis Stage** - before developing your API.

1. Preparing Your API Business Strategy

Overlooked by some, the **Analysis Stage** should be the first stop on any API's lifecycle. It involves research and critical decision making that will impact all future designs of your API. The purpose is to validate *why* your organization needs an API and *who* is going to use it.

This crucial stage forms a **business plan** for an API in it's entirety. Within this pre-development API phase, you should gauge interest from your target audience, perform market research, forecast trends within your sector, and make usage projections. To form a strategy, you'll want to decide on your API's business objective and what core API

functionalities can help achieve this result, and then pair this knowledge with appropriate revenue models.

Defining the above points will help an organization allocate resources for development, operations, and marketing, and should help in estimating your API's ROI. This preparation will also supply a roadmap for decision-making throughout future API lifecycle stages.

1.1 Basic Things to Consider

Developing and releasing an API should be given the same amount of preparation as a new product launch. Especially if the API is a main offering, or is being built as the underlying construct for an entire platform, the end business objective should be in the forefront throughout the entire lifecycle.

This early in the game, you'll want to formulate answers to the following questions:

- Why does my organization need an API?
- What functions will the API accomplish?
- What is the API value proposition?
- Who is my intended audience?
- Does my audience *want* to consume my API?
- If for public use, what sorts of third-party products do you imagine being built with your API?
- Will the API be a core business offering?
- Does my organization possess the resources required to successfully develop and maintain an API?
- What sort of metrics should I consider during analysis?

1.2 Determine If An API Is Right For Your Specific Situation

For many companies, providing an API is seen as an IT matter exclusive to internet giants like Twitter, Facebook and Google, startups like Algolia, Wit.ai and Context.io, or for government agencies to open data to the public. But are APIs really limited in that way?

It's true that even traditionally non-digital companies, such as Bechtel, a construction and engineering firm, Dun & Bradstreet, a business information company, and Marvel comics have succeeded in creating awesome platforms. At the end of the day, every company should consider how they can use APIs to excel their business.

Many organizations develop an API to extend their current functionalities. On the other hand, some businesses move toward API platformitization, positioning the API as a **core value.** Many facial recognition, natural language processing, and machine learning services do just that.

1.3 Perform Market Research

As with any business endeavor, it's vital to take a look at what competitors are doing in your sector to gauge the competition. Market research is a critical component of your pre-launch strategy. Use existing API discoverability tools to bolster your research.

Depending on the success of other APIs in the field, it may help promote alternative methods or protocols within your API. Create an account and perform test calls with a competitor's API, and imagine how the experience could be improved. This will help refine where there is room for your product in the market.

1.4 Align Your API With Business Objectives

An API should either boost pre-existing operations or work toward overall company goals in some way. In our eBook Developing the API Mindset we defined a taxonomy of three types of APIs: Private, Partner, and Public. Each have their own benefits and potential drawbacks. API providers may consider using multiple strategies to leverage their API and justify their Return on Investment (ROI).

The Private API Strategy

Types of industries that flourish using an internal API are ones that need their data secured, or ones that wouldn't profit from distributing their data. These businesses greatly benefit from a method of streamlining internal operations. For example, banks commonly use internal APIs to transfer funds and achieve internal efficiency.

The **benefits** of a private API may be:

- Streamlines internal operations
- Consolidates distribution channels
- Data is not exposed publicly, boosting security

Potential **drawbacks** may be:

- Potential revenue streams are not accessible
- Not as many developers are aware that your API exists

The Partner API Strategy

Many enterprise applications choose to partner their API with trusted partners in large data exchange dealings. Take the ESPN sports API, for example. Their public API was reigned in as it was decreasing the value of their primary distribution channel. Now ESPN API integrations are being done partner organizations.

Benefits of opening up an API for B2B exchange:

- Data is not entirely exposed publicly, retaining security and data control
- B2B dealings could potentially produce large revenue and mutually beneficial data exchanges
- Spreads assets to reputable, established services with pre-existing markets

Potential **drawbacks** of a sole partner strategy may be:

- Some revenue streams are not accessible via open means
- No open developer ecosystem is created, awareness is still limited

The Public API Strategy

The Public API is what we often mean when we talk about web APIs. This is a service opened to third-party developers to integrate into their apps. Anything from social networks, rideshares, project management apps, data processing tools, and more are common areas where a service is opened for third party

developers to consume in a mutually beneficial exchange. With this strategy, the provider may charge usage fees, or create a free-to-consume platform to spread brand awareness.

Benefits:

- Extending a service's brand to additional apps aids marketing and awareness
- Could produce a profitable ecosystem if developers are willing to pay for your service
- Potentially improves experience for more end users

Drawbacks:

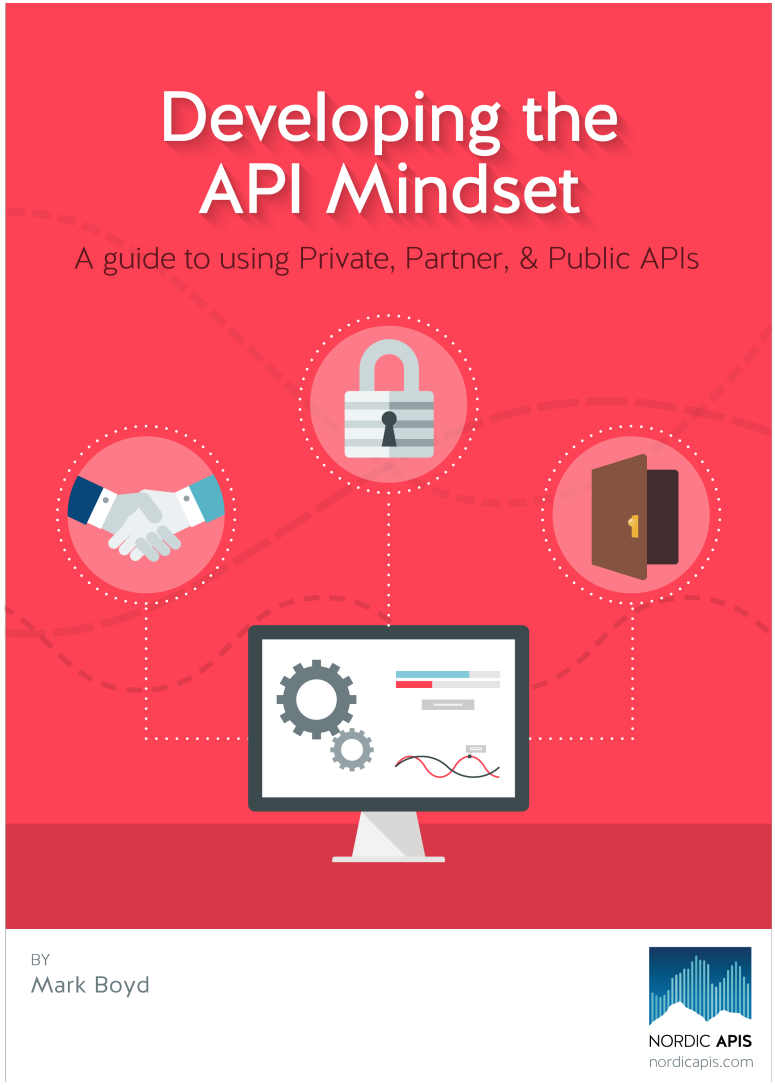
- Select assets are exposed for public, losing data privatization
- With greater awareness comes greater risk of security threat
- Third-party apps may not be as reputable or established
- Timeline for ROI is increased

1.5 Select Your API Business Plan

Estimating financial returns from an API can be difficult to perceive. With Public APIs, for example, revenue generation from third-party apps can take twice the normal time, as you must wait for a second product's lifecycle to reach sustainable operations and acquire end users. In free-to-consume scenarios, the offering may not directly generate revenue, but may boost brand awareness that could ultimately lead to growth or even help leverage deals within an acquisition.

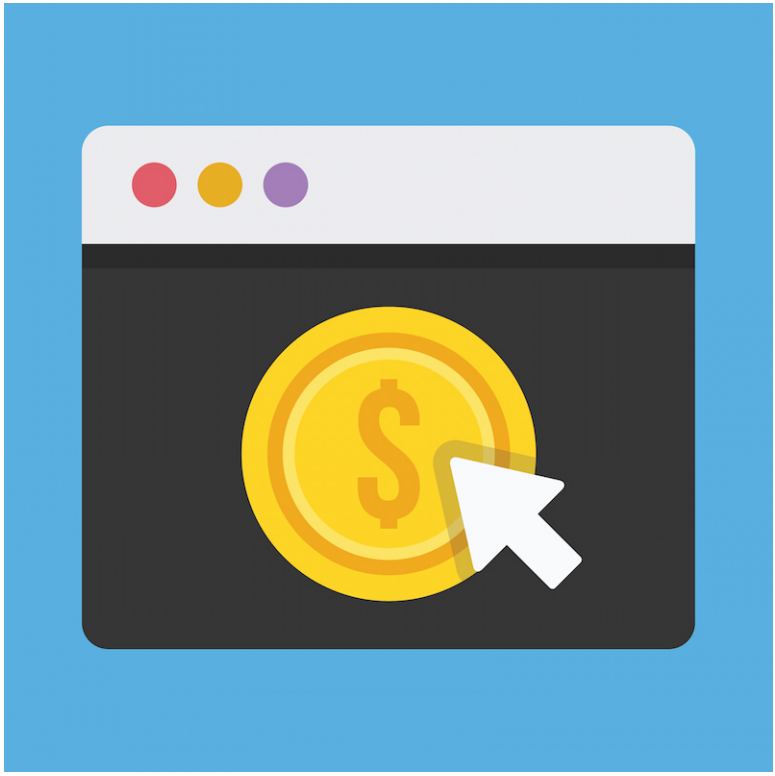
John Musser, founder of API Science, believes that “API business models are not size fits all,” having identified at least 20 variations in API business models throughout his work in the space.

Tiered subscription, pay as you go, freemium, unit based, revenue share - there are many monetization possibilities, and in the end, most APIs have a diversified ROI system that fits their service. In the next chapter, we’ll discuss ways to monetize your API.



For more on API business strategy check out Developing The API Mindset

2. Monetization Models



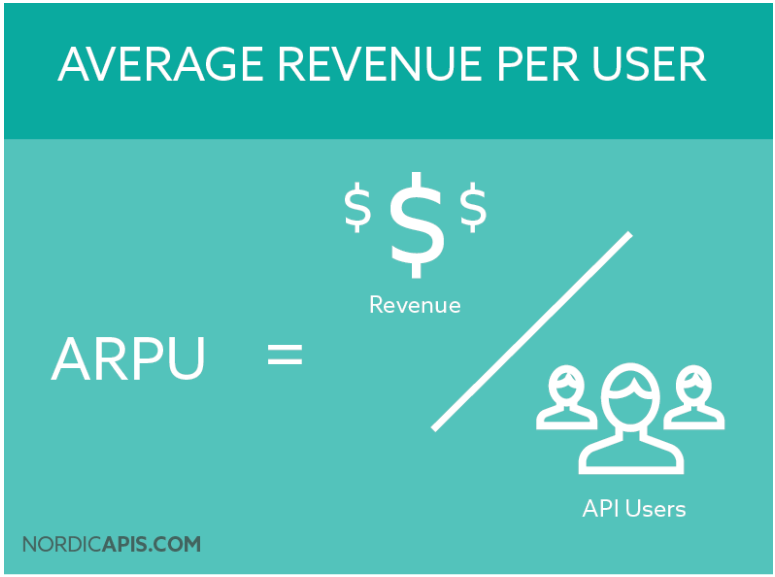
Before development, it's crucial to consider how your API will boost overall revenue. This last Analysis Stage chapter guides you to doing just that. We offer you the top **5 API monetization models** with the hope of helping you, as an API provider, bring value to developers and end users, and financially gain from hosting your API.

2.1 How Can You Measure If Your API Is Profitable?

Rob Zazueta is the director of platform strategy at Intel. His team focuses on helping clients manage, market, and monetize APIs by identifying strategies to align with business objectives. At his talk last October at the 2014 Nordic Platform Summit in Stockholm, Zazueta focused on the third tier of Intel's approach: **how to monetize your API.**

According to Zazueta, "The success of an API program is measured by how well it moves a business toward its goals." With everything your business does, you must keep an eye on your return on investment. To avoid an unsuccessful venture, before API development you need to consider common steps toward API monetization and if they apply to your situation.

In order to determine if something is going to be profitable you must measure it. Zazueta recommends using a simple Average Revenue Per User Model (ARPU) to help decide how to increase revenue, whether by increasing price or attracting more users. With any API monetization model, you want to compare the ARPU of your API-using customers against the ARPU of those who don't.



2.2 API Monetization Trick #1: Charge Directly for an API, by Call or Subscription

This is the most obvious method of monetizing anything - directly charging for it. But just because it's simple to measure doesn't mean it'll be successful. The strategy works "if your data has the kind of value that people want to pay for," Zazueta says. Before jumping right in, you should talk to your customers to see if they'd be willing to pay for these services and for how much.

Direct sales is perhaps the easiest way to sell your API, but Zazueta says it could also be the most challenging as it may be difficult for your developer audience to see the value of it. He recommends to start with a Freemium

Model to offer a taste of the data that developers would have access to, and hope it leaves them wanting more.

2.3 API Monetization Trick #2: Using API Access As A Premium Upsell Opportunity

This is a trick used in the SaaS world a lot, like with the POS giant Salesforce. Adding API access to a Premium subscription offers a strong motivator to upgrade to a higher package, as it allows end users to customize their experience and workflow more easily. API access is usually one of multiple added benefits that come along with subscribing to a higher package, making this a tricker monetization model to measure. Zazueta offers these attempts to measure this model:

1. Compare sale of packages with API included versus those without.
2. Have your sales team mark “Level of Interest in API” in sales notes.
3. Measure the number of API calls from users on that Premium package.
4. Measure ARPU or revenue generated by active API users versus non-active ones.

In addition, you can also measure the subscription renewal rate of API users versus those that don’t use this package feature.

2.4 API Monetization Trick #3: Drive Revenue-Generating Activities Through Your API

This model takes the most effort to track because it often supports indirect revenue. Depending on the situation, it could be the API monetization method most applicable to your overall business goals.

For example, email marketing software is often monetized by the email sent, but APIs are used to filter contacts automatically from a customer relationship management software (CRM). Without the contacts, you can't send the emails, which is why tracking the number of contacts that come in through the API is an excellent way to prove its value. Plus you can compare if users with the API send more emails— which affects revenue— than those users without the API.

Another method is to track if your application is actually driving leads. If you sign a pay-per-lead (PPL) agreement with a customer, you can track that lead via your CRM. For both of these, tracking can be complex if more than one third-party application is integrated, meaning increasingly granular measurements need to be made.

2.5 API Monetization Trick #4: Increase Distribution Through Strategic Partners

In the SaaS world, integrating your API with a strategic partner has the potential to open your API to entire pre-established markets. This enables you to market your brand to a broader audience to attract more potential

customers. For example, PayPal and Stripe allow services and sellers to use their API because it offers an easy way to get paid, and, of course, the more transactions, the more money these popular payment services earn.

This strategy can also aid in user retention. If your customers also have a subscription with your strategic integration partner, they are less likely to disrupt their workflow by leaving either of you. There is no questioning the value of these strategic partnerships for the consistently demanding end user, who can now build a customized workflow out of the integrated cloud tools now available. Plus, these type of integrations are on the rise:

"We see the rise of best-in-breed solutions as the next natural progression," said senior vice president of product at Zendesk help desk software, Adrian McDermott. "If the Internet is the platform, then the need to be integrated by a single provider becomes obsolete - multiple best-in-breed solutions can run seamlessly via the Internet, without needing to be overseen by one organization."

It's important with strategic partnerships to create a mutually beneficial shared data agreement. You want to know how much your API is helping their business, and similarly, they want to know the effect of their product on yours.

2.6 API Monetization Trick #5: Improve Operational Efficiency and Decrease Time to Market

"Well-built, well-designed APIs help you build applications faster. It helps you innovate faster. More importantly, it helps you fail faster," Zazueta says. "If you build a well-

built API that's secure and properly managed... and you treat that internally focused audience as your customers, you can actually get all kinds of great benefits." Zazueta says that he's seen internal processes in companies speed up from a couple months to a couple hours, solely because of the efficiency gained from an API.

Some additional benefits from using an internal API include:

- Faster customer onboarding
- Deeper access into data
- Tighter control over development
- Faster application building
- Faster testing of an application's revenue potential

Internal APIs also suit the lean start-up model, where you produce and release to market as quickly as possible. As this is the case, if the API isn't so valuable, your organization hasn't wasted a lot of time and resources implementing it - you're able to pivot rapidly.

Since this is an internal benefit, you don't use the ARPU to measure it. You start by measuring the time it took to build and introduce a certain function before the API and after. You can also turn that into dollars by way of the manpower needed. Of course, before making this move, you need to make sure that it won't take excessive effort and resources to design and start using the API within your internal systems. Review the existing processes before considering implementing this one.

2.7 Mix, Match, Measure Models of API Monetization

The reality is that your API monetization technique may involve an assortment of these models. Perhaps none are perfect for your particular situation, but it's very likely that a couple of these API monetization tricks will work to help generate revenue. As always, build what will work for you.

"The goal here is to build a successful API program that moves the business toward its goals," Zazueta says. "And a program that's not gonna do that won't be successful for long." You might be using one or more models already, but it won't matter until you measure its success.

2.8 Your API Monetization Checklist

We created this checklist to make it easy for you to figure out if you are able to monetize your API quickly. A *Yes* or a *Maybe* is certainly worth considering:

1. Are you able to profit from charging for your API directly?
2. Can giving access to your API be an upsell opportunity?
3. Are there ways to drive revenue indirectly through your API?
4. Do you have opportunities to make strategic API partnership?
5. Can an API improve operational efficiency and decrease time to market?

Then, as you implement an API monetization strategy there are two questions to constantly keep on your mind:

- Are you measuring Average Revenue Per User (APRU) throughout each possible API-related revenue source?
- Are you comparing this APRU with the APRU of customers who don't use your API?

As with anything, make sure that your API monetization strategy is directly in line with your business objectives. If it's clearly not, don't waste your time.

3. Understandig Your Target API Consumer

Consumer profiling that is a necessary first step for any business venture. For this chapter we cover strategies that help understand your target user within the specific context of APIs.

3.1 A Response to Increased Consumer Diversity

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?**

At Nordic APIs, we've tracked the explosion of this innovation, watching as new players step up to play ball — either connecting or striking out in deprecation. 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.

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."

3.2 Why Create a Developer "Persona"?

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."

3.3 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. The developer brain is naturally **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 depend-

ing 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.

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

Andreas Krohn of Dopter urges us to consider a wider scope with API marketing and 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 to consider when marketing an API.

Krohn believes we naturally target developers for the following reasons:

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

Though many people that are 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 average smartphone user 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.

3.5 Expanding our Portal: 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.

As the API space increases in diversity, more and more attributes make up the developers using them

3.6 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. 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 accell their product.

According to Jason Hilton of Catchy Agency, 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."

3.7 API Use Cases

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."

3.8 Lessen The Corporate Branding

Marketing, relations, outreach — typical B2B sales motives, labels, and tactics may conjure up negative connotations. At times, enterprise clientele may need to lessen the corporate branding to appeal to the tech community. Within the tech community the Evangelist job function is now more commonplace than ever. Alternative roles like "evangelist" or "developer advocate" help 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."

3.9 Developer Experience

The presentation of developer facing material is paramount to success in the space. 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."

3.10 Build it And They Will _____

Simply opening a platform up without having a 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.

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 before development should begin.

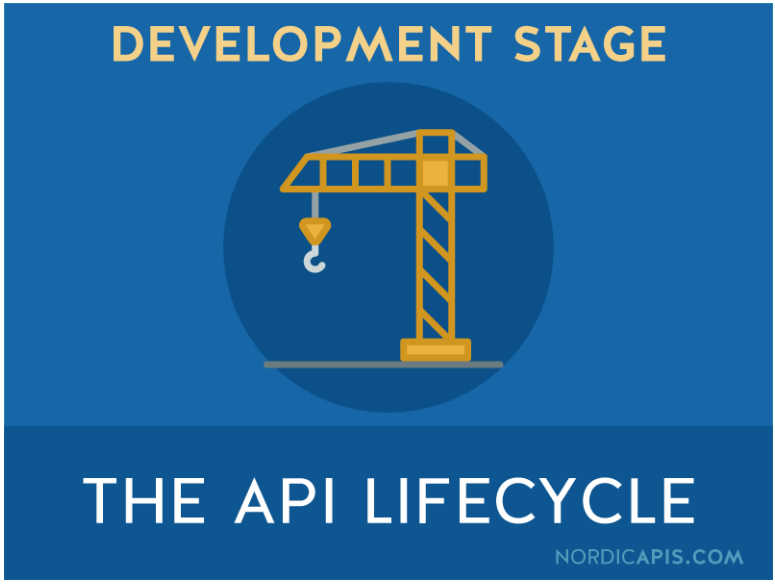
Next Steps:

The excitement surrounding APIs should not discourage critical examination into a smart and applicable strategy for your organization. Whatever the chosen API strategy is, it's mission statement, usage projections, model for growth, marketing strategy, and estimated financial return should be clearly laid out before production begins.

Large organizations may find the creation of a whole department necessary to respond to new customer concerns from an entirely new API-consumer base. Therefore, within the **Analysis Stage**, you must allocate resources for development, and anticipate creating support lines for marketing, hosting, customer support, and maintenance. If you've done the necessary preparation, it's time to move to the **Development Stage**.

1. Analysis Stage
2. Development Stage
3. Operations Stage
4. Retirement Stage

II Development Stage



Arguably the most important action in the API lifecycle phase is implementing the API and bringing it to life. Read on to learn more about this process and to understand what factors contribute to choosing the best combination of standards, designs, and programming languages for your API.

4. Constructing Your API

This chapter covers the actions to take as well as the most important challenges to address during an API's **Development** phase. We'll consider what implementation options are available and explore API Management and why it matters. We'll define API Usability and why it's important, and we'll consider what immediate steps to take after your API is deployed.

4.1 Things to Consider During Implementation

At this point you should have a well-defined API business strategy and documentation. This includes defining who your main API stakeholders are going to be, and deciding how the API will be consumed. By this point you should have answers to the following questions:

- **What value** is the API providing? Is it a means to distribute data or does it offer specific functionality that can be used to build apps?
- **Who will consume** the API? Are consumers interacting with your API to get access to data or are they consuming it on behalf of other users?
- **How many consumers** will the API have? Is it a private API consumed internally or is it going to be available to potentially millions of consumers?

Answering these questions will help you cut corners during the implementation process. Let's start with understanding what you are trying to enable with your API so that you can define which methods and protocols you're going to use.

Methods and Protocols

The best way to define methods and endpoints for your API is to follow the functionality you'd like to provide and choose the most appropriate standards and protocols. Identify what type of data manipulation your main API consumers usually have available.

If your main consumers are mobile apps, consider offering endpoints that ask for and respond with as little information as possible. This is to prevent large bandwidth consumption and speed up operations that involve calls to your API. If, by contrast, you're offering an API that will mostly be consumed by the financial market, carefully study what frameworks they're using and how they communicate with external APIs.

Whatever you do, we recommend not to follow the very latest trends presented as "the future." Trends can and will change many times before they stabilize and become widely adopted, and your API will have to keep up with this market fluctuation. Determine what 80% of your target audience is using and offer that.

SDK Programming Languages

Choosing how you'll offer access to your API is directly related to how your API will be consumed. Will it mainly be consumed by mobile app developers? Then offer ready-to-use mobile SDKs. Brainstorming your potential API use

cases is the first step in choosing how your SDK will be packaged.

If you're not 100% sure about how your API will be used you can always offer an SDK in several popular programming languages. Algolia, a search engine API, is following this approach by offering an easy way to use their service in languages such as Ruby, Python, Node, PHP or Objective-C. Be aware though that maintaining a large number of SDKs is not an easy task as your API evolves and methods change. APIMATIC and REST United are two services that offer automatic SDK generation. With these tools, whenever your API changes you can rerun the SDK generation to make fresh code versions available to your consumers.

4.2 API Management

After you implement your API endpoints and decide how consumers will interact with your API, you must consider ongoing operations. The most important factors are controlling your API access conditions and determining how to behave in usage peaks. You can follow a DIY approach or use an API management service such as 3scale or Apigee.

Whatever solution you choose, remember that you'll become tied to the way it works. Carefully define long term objectives as well as common scenarios you want to address. If you choose a solution that doesn't offer what most of your API consumers use, then you'll likely be forced to pivot in the long term.

Access Control

One of the most important aspects of API Management is defining and automating the process of controlling who can access your API and what measures are in place to enable access limitation, if needed.

Most API Management platforms provide some sort of access control features, including API call limit rating and consumer differentiation with the help of different paid plans. The very first access control feature to look for is how your API will authenticate and authorize consumers.

There are several options depending on what kind of consumers you'll have and what kind of usage will occur:

- **API keys**: The most simple way to control access to your API. Usually you'll have to issue one API key for each consumer and identification will be provided through an `HTTP` header on each and every call.
- **OAuth 2**: Perhaps the most popular authorization standard used by Web APIs. OAuth 2 lets you combine token-based authentication with fine-grained access control based on user scopes. This is the best solution whenever API consumers are making calls on behalf of other users.
- **Origin IP address**: This can be combined with other access control methods or used on its own. Using it alone is done in cases where you're providing an API to a very limited audience. To achieve better results you should also add origin information to your logs and usage analytics.

Whatever access control strategy you choose, always remember that it must accommodate your API's most popular use cases. As an example, it doesn't make sense to

use a simple API key based authentication if your main API usage is through a mobile app. In this case it would make sense to use OAuth and let the mobile app act on behalf of its users.

4.3 Maintenance

While launching your API can be an exciting challenge, keeping it up and running is often seen as something boring. API maintenance is often delegated to a second plan and only considered after things start to go wrong. Don't follow this path; implement a good maintenance plan from the start.

The usual activities around maintaining your API are related to documentation, communication with consumers and proactively understanding if something is not working as expected. As such, a big part of maintenance is done by periodically testing your API to make sure that everything is working as documented.

Testing

One way to proactively understand if your API is performing according to the documentation you provide and following your quality-of-service is to run periodic tests against individual endpoints. With monitoring in place, if any of the tests fail, you can hopefully fix the error before your consumers experience problems. These types of tests can be performed on your API:

- **Performance-oriented**: These are tests that make individual calls to each and every method your API provides. If a response takes longer than a specified

time limit, it means that there's a problem on that specific endpoint.

- **Functional testing**: This type of testing works by making singular calls to each API method to thoroughly test every API function, using different kinds of payloads, or even sending data that will produce errors. Responses are then compared to the expected behavior to locate errors.
- **Use case testing**: This is a more sophisticated type of test and can be achieved by combining calls to different endpoints into a single test. Each test should expect a specific response and execution time limit.

There are several tools that can help you run API tests periodically in an automated fashion. POSTMAN, for example, is a Chrome app that lets you run all the tests mentioned above. There is also the SmartBear suite of API monitoring tools. If you're looking for something more SaaSy you can try Runscope, a service that periodically runs tests and reports on their results. Other helpful API monitoring solutions include APImetrics and API Science.

Managing Changes

Even if everything is working as expected, sometimes you'll need to introduce changes to your API. When this happens, consider the impact on your consumers and internal systems by:

- **Informing consumers about changes**: Establish a clear communication channel between you and developers using your API to inform them of changes in a transparent fashion.

- **Keeping compatibility with previous versions**: Ensure that any introduced change doesn't break what was already working. Remember that your API consumers may not be able to update their systems immediately.
- **Minimizing downtime**: Don't bring systems to a halt — even momentarily. Your build and deployment processes should guarantee that API changes won't affect API calls and functionalities. API downtime might create downtime or loss of quality on the consumer side, so keep it to an absolute minimum.

Downtime can be minimized by following a Continuous Integration approach and testing your code thoroughly before deploying. Making sure that there are no breaking changes can be achieved by following a sound API versioning strategy. If you offer and maintain different versions of your API, consumers won't feel the changes on the version they're using. A helpful solution that bridges the gap between API doc changes and the consumer is the API Changelog, a tool that automatically discovers changes to your API documentation and informs subscribers.

Check out our Winter Collection eBook as well - a collection of our best advice from the 2014-2015 winter season

5. 12 Design Tips

During the Nordic APIs World Tour, expert API practitioners shared their insights with attendees at four international cities, touching on API development, security, design, operations, and more during intimate conference settings.

A recurring theme in many talks was **API Design**. So, in this chapter, we'll sum up important API design lessons

that were conveyed by our speakers and then add some of our own — both practical tips on technical implementation, as well as more more philosophical considerations on how to hide (or not hide) complexity when designing an API.

1: Know Your API Requirements

API design should not begin with technical documentation, but should rather originate from your fundamental goals. Knowing what your API needs to accomplish is of the highest importance. This is defined in the analysis phase of the API lifecycle.

As Phillipp Schöne from Axway pointed out, API designers should ask "*how APIs can support the business function and not how they can support the needs of IT*."

2: Think of APIs as Building Blocks

It's important to remember the unique characteristics of APIs. As a consumer, tying into a service's underlying data or functionality is inherently a unique process. Though comparable to SaaS products, Andreas Krohn from Dopter highlights two key differences:

The first is that **APIs are building blocks** and not finished products: "*If somebody is using your API they are basically outsourcing a part of their business to you*".

His second lesson is that **machines are not flexible**. If a SaaS product changes, a person can adapt, but if an API

changes, a machine is not as adaptable. This influences the design and the whole lifecycle of the API, and affects how we prepare for parameter changes and versioning.

3: Learn From User Feedback

The need to understand your audience is applicable throughout the entire lifecycle. If you are updating, versioning, or performing major redesigns on an existing API, you need to carefully respond to the **feedback** from your users to make the new version as good as possible.

Marjukka Niinioja, Senior Consultant and Manager at PlanMill shares her experience with leveraging user feedback during a major redesign: developing PlanMill's new RESTful API, UI, and back-end architecture.

PlanMill payed close attention to the feedback they were receiving internally, noticing three main recurring points:

1. The API was too complex for the developers to use.
2. The documentation was not sufficient.
3. The error handling needed work.

Using this feedback, and by testing and consuming the API themselves, the PlanMill team was able to hone in on important aspects to consider during redeployment. The takeaway is that if you are redesigning an API, you are poised to release killer API improvements — as you already have **real world data on how API functionality is used**, which can be repurposed to improve efficiency and overall API user experience.

4: Decrease Confusion For The User, Let Provider Handle Complexity

Who pays the price for complexity? Nordic APIs veteran Ronnie Mitra of API Academy argues that complexity is necessary — what should be avoided instead is confusion. Mitra contends "*it is a designers job to reduce confusion and embrace complexity in their business domain*."

There has been a move toward simpler products and simpler interface designs. Mitra, an expert in developer experience and API design, advocates for smartly designed software and APIs that retain simplicity but also cater to complex requirements. Quoting John Maeda's Laws of Simplicity, Mitra notes that "*Simplicity is about subtracting the obvious and adding the meaningful*."

A steering wheel in a F1 car is very complex compared to the steering wheel of a normal car, "*designed and*

optimised for its user and the situation". Ronnie makes a great point that "*every design decision is a decision of if you* [as an API provider] *or client app developers will pay the price of complexity*". As an example, OAuth is complex to implement by the service provider but easy to use by the client developer. We cannot avoid complexity, but we can architect our APIs so that the user-facing side is deceptively simple.

Ross Garrett from Axway also talks about what applying an API-first strategy means for enterprise organizations. In dealing with legacy systems, he notes that a missmash of tangled infrastructure can be masked by using proper API management tactics.

> "Your old architecture [may not have] useful APIs, or may be older SOAP services that don't perform well in the context of mobility or cloud integration. Some legacy things need to remain, but API management can extend and reuse by translating all old interfaces." What's important is having a clean business appearance for the end user.

5: Use Hypermedia For Evolvability

It is impossible to talk about API design without mentioning Hypermedia, a subject that came up in several presentations during the Nordic APIs World Tour 2015. Pedro Felix of Lisbon Polytechnic Institute, offers a deep dive into HTTP. He summarizes his presentation about API design this way:

> "If you have a problem, keep calm and look for an HTTP RFC"

Pedro considers hypermedia as the key factor for evolvability, allowing API providers to react to new business requirements quickly without breaking client applications. Related to Mitra's theory on the distribution of complexity, using hypermedia means more initial complexity for client app developers, but an overall reduced complexity considering the ease of future changes.

6: Learn From Real World Information Design

Brian Mulloy from Apigee demoes a hypermedia API for the Internet of Things. By using hypermedia it is easy to introduce new devices into the IoT since all possible states, actions and feeds are described in the API itself.

Mulloy describes API design for the Internet of Things as using information design for physical objects. He compared this to the early days of mass car production in his hometown Detroit. Suddenly the city had lots of cars, but infrastructure was not prepared for a large automobile influx. The resulting confusion lead to chaos and even a large number of traffic related deaths.

The solution was to use information design that resulted in standard designs used all over the world today. Pedestrian crossings, traffic lights, stop signs, and lane markers make it clear how to behave in traffic. Akin to the early days of cars, we are still working on how to organize and design today's APIs for the IoT.

7: API Design Needs To Convince the Architect

Having a shared vision within an organization is key for API interconnectivity to be accepted. To help convince API naysayers, whether they be architects or engineers, in his talk, Adam Duvander of Orchestrate walks through four ways API providers can frame their API product to foster confident adoption. These factors should be considered when designing an API in order to respond to common stigma associated with API integrations.

1. The architect wants **control**: Accustomed to traditional methods of infrastructure — data storage on local servers — the architect may be opposed to cloud operations, desiring to 'touch' the software. To workaround this stigma, API providers can offer the ability to download and store data sets locally, or even offer an on premise, dedicated, managed option the API.

2. The architect wants **zero downtime**: To foster reliability in the service, having transparent developer-facing logs that communicate API uptime is critical.
3. The architect wants to see **responsibility**: API systems need to be designed as secure systems using modern approaches.
4. The architect requires an integration that guarantees **longevity**

8: Develop With a Long-Term Mindset

Within a "develop for now" approach, developers are creating an API for **immediate** use, focusing entirely on integrating existing feature sets and supporting specific sets of queries. The downside of this approach, however, is that they are only robust at what they are designed for, and nothing more. Designing for the immediate needs and requirements of a service or system is all well and good, but it does little for long term support.

9: Be Consistent

Maintain **consistency**. If your API uses calls that don't run in the same environments, bug hunting becomes a chore. If your documentation is inconsistent with actual functionally, you risk confusing and segmenting your userbase. If you have a methodology to refer to outside sources, use this methodology on *every* reference. If you cover a request by another multipart request, make sure there's no needless redundancy.

10: Allow for Manipulation of Data

Data is only as valuable as it is understandable. When users receive data from your API, consider the methodology of how it's delivered. Delineate data by type, and allow sorting within certain limitations. Powerful, dynamic services that allow for far more usefulness of API data is a common requirement amongst many developers.

11: Effectively Validate and Report

Returning useless **errors** that state the obvious (something went wrong), the useless (ERROR), or the downright confusing (error code:9442394) is a losing proposition — if your users can't find out what went wrong during basic errors, they'll stop using the API entirely. Similarly, failing to return a valid JSON, and refusing to effectively document what error codes actually mean can make a potential API development partner second-guess integration with your services, further killing potential user base growth. **Make sure all error returns are understandable**

12: Support Uptime

Uptime is our fifth and final tip as it is a wonderful metric by which we can holistically judge an API. It represents everything working in concert — if a single element of an API is poorly managed, uptime will likely be influenced. At

the time of this writing, Bit.ly reports 100% uptime over the past 24 hours, with a yearly uptime rating of 99.9%. Likewise, the Twitter API reports 100% uptime with an average uptime of 99.9%. The proof, as they say, is in the pudding (or in this case, the uptime).

5.1 Summarizing API Design Tips

Considering the development of your API is a serious endeavor. The choices made now will impact your future success in a big way. While these solutions can be implemented after the fact, it is far easier to implement them at the start of the development cycle. Not only will this deliver a more completely integrated feature set, it will also give you a product with simpler end-documentation and user experience.

These 12 design and development concepts may be simple, but the implications of their applications are deceptively huge — implementing them can lead to explosive growth, high user adoption, and long-term use statistics. Design an API that allows an API provider to handle most of the complexity, and simplify processes for your client application developers. Above all, do not forget that the API design should serve your overall business requirements.

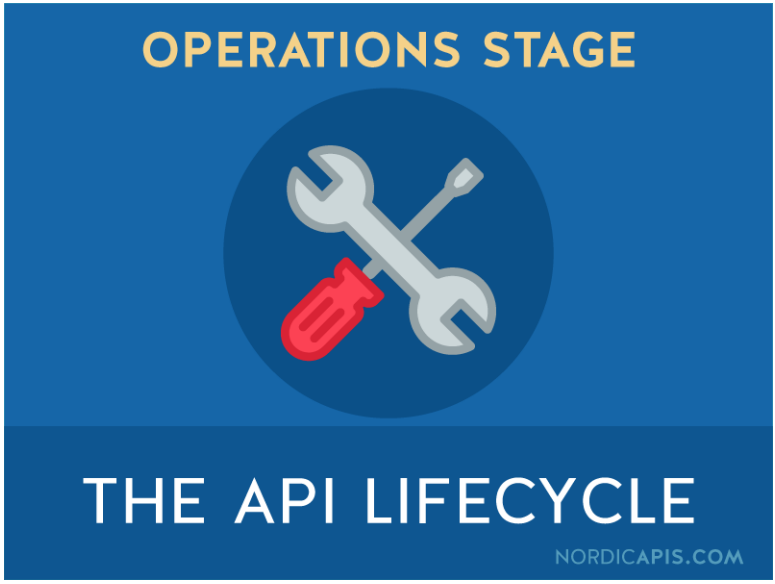
Next Steps:

As you now understand, developing an API is not limited to writing the code and making it available. Understanding your API use case plays a major role in building the foundation of your offering. Maintaining proper API health means never leaving it unattended â€” as you risk a decrease in consumer retention and a damage to your brand.

This stage of the API lifecycle is so critical that numerous services have been launched and many startups created just to help companies deal with all thatâ€™s involved. These tools will help you understand what you have to do next. If you are motivated to introduce major changes to your API we recommend you visit the Analysis Stage to treat those iterations as a brand new API launch or version. Otherwise, jump into the **Operations Stage** and run your API like a product, fostering its user base.

1. Analysis Stage
2. Development Stage
3. Operations Stage
4. Retirement Stage

III Operations Stage



In this part we discuss what factors make up the **Operations Stage**, and how they should be addressed. Transforming your API into something developers will want to consume, and learning what impact this will have on your business are things we'll consider at this stage. Other critical actions involve maintaining a positive relationship with your API audience and offering them appropriate support channels.

6. Marketing Your API

To start, let's take a look at your API from a different angle — one that presents your API as not just a developer-oriented tool but as a full product that solves a real problem. In this chapter we'll see how this angle can fine tune the promotion of your API to focus more on your audience's perspective and needs.

6.1 Running Your API As a Product

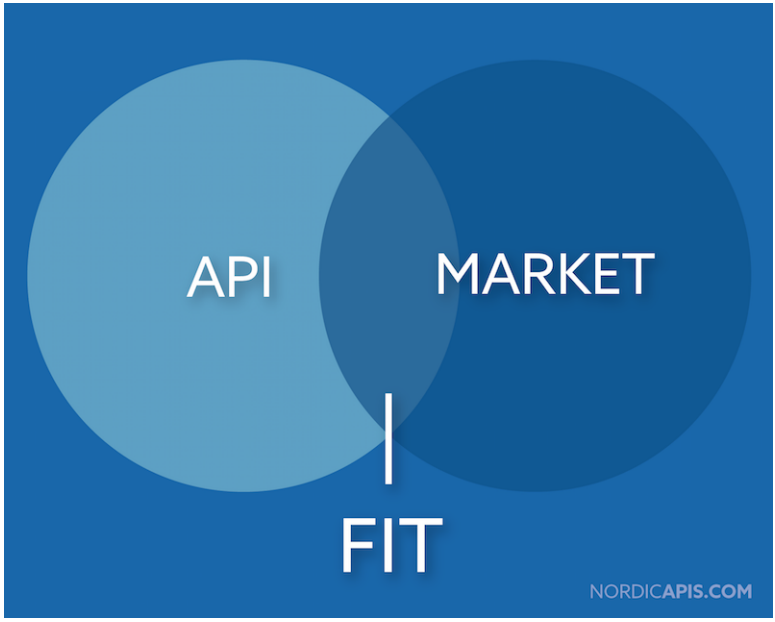
Why should you treat your API as if it were a full-fledged product? Isn't the API something that's *supporting* your existing product? Though it may sound strange, running an API as you would with any other product actually makes a lot of sense.

By definition, a product is something that exists to satisfy the needs of a group of consumers. Usually, during product development, you research what consumers need so that you can design your product accordingly. You also conduct research to understand *how* consumers will use the product that you're designing. The results of these activities will eventually lead to the creation of your new product.

By applying the same strategy to your API, you'll not only have an API that developers really **want to use**, but you'll also fully understand your target audience— a vital aspect of attracting developers to your API. The second activity is not any less important, as it will greatly inform the API design process. By learning how developers will consume

your API you'll be able to adapt the following items, if needed:

- **Access Control**: Are developers building applications that act on behalf of several users? Does it make sense to use API Keys, OAuth or something else?
- **Methods and Protocols**: Do developers prefer REST, SOAP, Thrift, or something else? What makes more sense from a consumer point of view?
- **SDK Programming Languages**: Are developers using Ruby, Python or other languages? What SDKs should you offer and support?
- **Asynchronous Operations**: Is your API mostly consumed by Web browsers, mobile devices, or backend code? Are API calls made from behind firewalls? Can your API use webhooks?



If you don't conduct product-oriented research then you'll never understand how your API should be offered, and it will be much harder to pivot or gain traction in the long term. If you follow this strategy you'll have the equivalent of product/market fit: **API/market fit**. Next, you'll want to fuel your API usage growth by employing various marketing tactics to attract more and more developers over time.

6.2 Marketing Your API

Marketing an API is very similar in concept to any other kind of marketing. You'll want to produce material that will promote your API in a variety of ways. This starts with launching your **developer portal,** which will act as the official point of contact between your API and its consumers.

As your developer portal will be central to all your API-related activities, it should offer an easy access to all the following resources:

- **API Documentation**: Probably the most important resource, documentation should always be up-to-date and provide all information needed to quickly begin consuming your API.

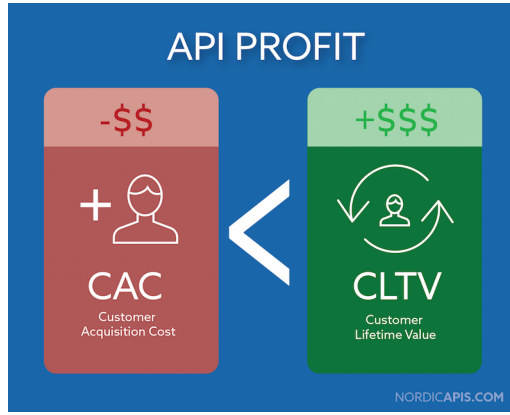
- **Application Registration**: Offer an easy-to-follow registration process that any developer can understand. The output of this process should be an API Key or a Client ID and Secret if you're using OAuth.
- **API Console**: Developers usually want to test your API without having to write any code. Either offer an API Console or provide ready-to-use code that can be copied and pasted. This first impression of API functionality should be succinct and powerful.
- **SDKs**: After testing, your developers will want to consume the API from within their existing applications. Offer the appropriate SDKs and follow the packaging and installation standards for each programming language. For example, if you're offering a Ruby SDK, offer it as a Ruby gem.
- **Announcements**: Keep your audience up-to-date with anything you do related to your API. Announce all upcoming API changes and other relevant activities.
- **Use Cases**: Provide comprehensive use cases that your audience can identify with. Complement them with sample code that uses your SDKs. It's much easier to understand how your API works by reading a story than by going through the full reference.
- **Support**: Offer a simple support channel directly from your developer portal. Whenever developers have a question they should be able to contact your support team from within the portal. Provide quick feedback on support requests and always follow up.
- **Links to External Resources**: Link to external resources that could help developers use your API. Have your documentation link to the standards that you're following and provide relevant social media links so developers can follow your announcements.

If you implement all these suggestions you'll have a superior API Developer Portal, but you still have to attract developers to it. Site activity should be combined with a permanent instrumentation of your API and Developer Portal usage so that you can correctly measure **traction**.

6.3 Using Different Traction Channels

In order to obtain the best possible results, fostering your API user base should be done across a number of different channels. Along the way you should measure activity on each channel and adjust accordingly. As you're about to see, marketing channels have different costs and you should understand how much, on average, each new developer is costing you to acquire.

In the Product world, this value is often called **Customer Acquisition Cost** or CAC, and can be used as a model to determine if any of your channels should be abandoned. To append this, you should also estimate the total future revenue obtained from each new developer using your API. The **Customer Lifetime Value** or CLTV is what defines how much revenue, on average, each developer will generate.



To make a profit, you should always aim for a CAC smaller than the CLTV. Otherwise, you'll end up paying more for developers than the potential future revenue that you'll obtain from them. With this in mind, let's look at some possible traction channels.

Evangelizing Your API

Evangelizing is one of the methods used by most of the popular APIs. Evangelizing is usually done by meeting developer groups in person to share stories about how your API can be used. So-called "API Evangelists" are used to promote APIs using online media, and in person by attending developer-oriented meetups, conferences, and other related events.

Naturally, this type of promotion can have a cost range of nearly nothing — especially if you're doing it by yourself online — but can be more costly if you employ a group of dedicated evangelists to attend every possible developer conference. Be pragmatic by starting small and measuring your results along the way.

Hackathons, Events, Conferences

Another popular traction method is promoting your API is by hosting a hackathon or developer oriented event. Hackathons may be geared toward a specific industry or theme and are often location-based — allowing you to target specific demographics and discover and promote new use cases that can be showcased on other channels.

Integrations and Partnerships

An interesting way to gain traction is to attach your API to the growth of other products and APIs. This can be done by creating integrations that will make your API easily available and consumable by developers who use other products. Integrations often address specific use cases, so carefully study which one will attract the largest number of developers. Some integrations might be as easy — such as writing the code and publishing a guide — but in some cases you might need to engage into partnership development.

Here, the cost will mostly be associated with software development activities — but don't forget that you'll also have to maintain and support the integration after it goes live.

6.4 Supporting Your API

Support always works better the closer you are to your audience. Understand where your API consumers usually

ask for help and have a saying in those places. Don't force your developers to use specific tools and processes that don't make sense from their point view or you'll start to see them abandoning your API after an unsuccessful usage.

A simple way to grow your support channel is to keep things open. Keeping API-related activities as publicly transparent as possible means your support team will spend less time replying to the same requests.

Use GitHub

GitHub is considered the most popular developer-friendly service and API providers should take it seriously. By using their free plan you can create a repository to provide an easy-to-use support channel. GitHub offers a way to manage *issues* related with code changes and openly interact with developers. An example company following this approach is Spotify, who has been very open about all their API-related issues through GitHub.

Other Developer-oriented Support Tools

Research your developer audience to understand which other web services they use the most. It's important to have a presence on them as well. Popular services include Stack Overflow, specific subreddits, Twitter or Facebook. Although social media isn't specifically a support channel, you can use it to quickly give feedback to any questions developers might have about your API. Remember to always be open and to provide helpful responses.

7. The Importance of API Metrics

Success and failure are relatively subjective terms — what defines success for one business might be considered a failure for another, and the relative of success in certain areas of performance can change from industry to industry, department to department, and even on a case to case basis. This volatility in expectations, outcomes, and impacts can make API documentation, implementation, and support a difficult undertaking.

Luckily, those who work with APIs have a secret tool that can make sense of the hectic world of performance evaluation — **metric analysis**. Proper use of metric analysis can allow businesses to understand their consumers and

aid in further development and implementation of user-friendly and effective APIs.

In this piece, we'll take a look at metric analysis to demonstrate how it can be used to amplify success within the API space. We'll suggest types of metrics to analyze, demonstrate a theoretical application of metric analysis, and discuss two real-life examples of success and failure arising from differing metric analysis methodologies. By the end of this piece, you should have a firm grasp on the definition, application, and impact of proper API metric analysis.

7.1 What Is Metric Analysis?

According to the Oxford Dictionary, metrics are "method[s] of measuring something, or the results obtained from this". Metrics are the way we measure the values of a targeted part of a system, measuring an event from inception to completion, including the effects caused by its implementation.

So now that we know what a metric is, why is it so important to APIs? Metrics, and by extension **API-centric Metric Analysis**, are invaluable tools for the modern business. Metrics can be used to develop new processes, create a fundamental understanding of the product and targeted consumer, drive a holistic understanding of your manufacturing process and methodology of delivery, and create opportunity to monetize and optimize your API. Using metrics can bring you to a bird's-eye view of an entire API process, following the age old saying — "you can't see the forest from the trees."

7.2 Metric Analysis Tools and Services

There are many types of metrics that can be captured, analysed, and used to develop more effective API systems. Metrics can largely be grouped into two categories: **internal metrics** and **external metrics**.

Internal Metrics

Internal Metrics are those that are derived from data captured by internal web servers, user feedback forms, and trends observable through internal systems. These include:

- **User type**: Is the consumer a repeat user or new user?
- **Days since last session**: How long ago was the API last used by repeat users?
- **Traffic sources**: Are functions within the API being called through your own web application or a third party application?
- **Function grouping**: How often a user calls a certain function along with other functions.
- **Types of data requested**: Is your server serving media requests, plain text requests, or other types of requests more often than others?
- **Access speeds**: How quickly is your system responding to requests? Where is the bottleneck?
- **Service requests**: How often are some services being requested? Are there any services that are never requested?
- **Error reporting**: How often does a user report an error with the system, and what is the specific error?

External Metrics

These metrics are derived through the use of processes and applications that originate outside of the API developer. While internal metrics are concerned more with the functioning of the actual API and overall system, external metrics focus more on the community and potential user bases. These metrics may involve third-party systems:

- API and service adoption rates - WebTrends
- Redirection and publicly facing data - Google Analytics
- Market trends and behaviors - Adobe

7.3 Example — The Tokyo Traffic Problem

A way to understand the importance of metric analysis is to show a situation in which it is applied properly against a

situation where it is not, comparing the results of both approaches. Let's imagine that we are engineers for the roadways of Tokyo, Japan, one of the largest cities in the world. We are tasked with solving a congestion problem in one of the most congested intersections in the entire city.

Let's first approach this problem without using Metric Analysis. We will use an **observational approach.** By standing on an overpass straddling the most congested area of the road, we mark down our observations in our notebook, noting both the number of cars and the number of lanes provided. Using these observations, a solution is designed to expand the roadway by an extra two lanes. The time budgeted for construction is set on a 24-hour schedule. You assume the congestion will ease due to your solution, and you step away. Some months later, the congestion you observed has only spilled into the new roadways, worsening the problem. The observational approach has resulted in more problems than it fixed, and is considered an abject failure.

Now let us approach the problem in a **metric-based approach**. After being assigned the project, you first look at the city road maps. You observe that the congested road is likely facing issues arising from a stoplight intersection some miles ahead of the congested area; because the highway empties into a city thoroughfare before continuing onto another highway, this area functions as a "bottleneck", increasing congestion by constraining the throughput of the system as a whole. Using this data, you decide to add a three-lane highway that bypasses the city thoroughfare, allowing traffic to flow both into and around the intersection, dependent on the driver's needs. You also decide to implement a roundabout as opposed to a stoplight, thinking the flow of traffic should

be eased in this manner. This plan is placed on a construction schedule over a handful of weeks, avoiding work during the rush hour. After implementing this solution, you collect data over an entire week, making notes of points of failure or issues at certain times of the day. Congestion has eased considerably, and the metric-based solution is considered a success.

7.4 The Traffic Problem Solution

What was the difference between the two approaches? Both aimed to ease congestion, and while the first approach was different than the second, on paper they both should have worked. The difference is the fundamental use of data in implementation.

- In the observational approach, the *global issue* of congestion was attacked by a *specific* solution. In the metric-based approach, the *global issue* of congestion was attacked with a *global solution*.
- Issues arose within the first approach from failure to understand the *end consumer* (expanding the intercity thoroughfare, and using a 24-hour build schedule interfering with rush hour). The second approach understood the requirements of the end consumer, creating a highway bypass and avoiding construction during the most congested periods of the day.
- The observational approach eschewed data analysis, assuming a localized solution would fix the problem, while the metric-based solution observed the *measurable results* of the solution, tweaking and adjusting to any new requirements that arose during construction.

7.5 The Traffic Problem and APIs

So what does Tokyo congestion issues have to do with API development? The two approaches above are perfect analogies for the development and implementation cycle of APIs, and show the measurable need for Metric Analysis. When developing an API, one must monitor internal and external metrics to ensure their API is effective in the long-term. When an API is developed and implemented, the following vital factors must be considered:

- The needs of the *end-consumer*, that is, the user who will interact with the API and its front-facing systems
- The method and timescale of *implementation*, i.e. whether or not the API will be implemented immediately or over time
- The type of implementation, i.e. whether or not the API is *specific* (defining a single use or purpose) or *global* (completely overhauling the already existent structures and systems)
- The *measurable solution*, or the effects caused by the implementation of your API on your *end-consumer*

7.6 Metrics Throughout the Lifecycle

To better understand this concept, let's break down the API Lifecycle point by point, examining the role of metric analysis in each stage.

In the first stage of the API Lifecycle, the Analysis Stage, metric analysis is perhaps most effectively and extensively used. While determining the usefulness of an API, the demand for its implementation, and the decisions between Private APIs, Partner APIs, and Public APIs (a

subject covered more fully in our eBook Developing the API Mindset), the use of metric analysis is specifically designed to help you understand your client base, their needs, and the relative importance of ease of accessibility and extensibility. By analyzing market trends, reviewing web server data, and polling prospective clients and users, metrics can help effectively narrow and define the objective of an API system.

During the second stage of the API Lifecycle, the Development Stage, metric analysis switches from an external role to an internal role. By analyzing the systems available to your developers, tracking the management and implementation of features, and using varied methods of rigorous testing and bug-tracking, quality is supported and the product is refined into its best possible initial state. Failure to perform proper analysis on the varied metrics in this stage can result in massive hidden failures, missing feature-sets, and an overall displeased user base.

Nearing the end of the API Lifecycle, the [Operations Stage(#Operations) is nearly as heavy in metric analysis as the first, as it is concerned with the public usage, reception, and internal response through iteration and patching. Monitoring user statistics, including the methods and durations of use, general feedback concerning usability and extensibility, and the overall impression of the API system can not only help you further refine your product, build a confident, strong user base, and quickly respond to bugs and issues, it can also help you prepare for future development projects and make the first stage of your next endeavor that much smoother.

Finally, the fourth stage of the API Lifecycle is the Retirement Stage. This stage is often the direct result of effective analysis throughout the previous four stages. When

retiring and deprecating APIs, metrics concerning usage rates, operating system and browser support, financial response, and user base confidence can all inform the decision to retire, continue, or reiterate an API.

Failure to conduct API metric analysis could result in a product undergoing an expensive and lengthy development cycle only to find little demand upon release, making the API financially impractical. Effective metric analysis, however, can help create a stellar product, produced quickly and with less expense, resulting in higher demand with a longer lifespan. This is the raw power of metric analysis.

7.7 A Real-World Failure - Heartbleed

API security is a huge issue, and is becoming a more prominent concern as more companies adopt the API-centric design concept. One of the most well-known examples of security failure arising from poor metric analysis is the infamous Heartbleed bug. This bug, which affected users implementing OpenSSL, a widely used security protocol, had a vulnerability in its data input validation algorithm, which allowed for excess data in its buffer overflow (a system meant to allow data exceeding the maximum buffer size to "overflow" into a secondary buffer registry) to be read in its entirety without being validated or checked for malicious code. Due to this bug, data could be forced through the overflow system without validation, executing commands that made internal data, systems, and services vulnerable to external, malicious attacks.

This bug is directly a result of improper metric analysis. During the initial construction of the OpenSSL system and

API, “negative testing”, or the testing of failure scenarios, was not properly conducted against the buffer overflow; if it had been, the issue would likely have been found early on and patched before it was abused. The needs of a secure system for *end-consumers* was not properly identified, as the vulnerabilities of the system itself were not properly tested against common validation failure and malicious attack scenarios. By not acknowledging the rate of buffer overflow incidents and the type of data that failed validation during such scenarios in the metric analysis process, an entire class of vulnerabilities was essentially ignored until it was too late.

7.8 A Real-World Success - The FedEx ShipAPI

When FedEx set out to develop an API for their shipping and freight systems, they had one issue in mind — efficiency. By eschewing the “established is better” mantra and focusing on an agile mode of API and business development, FedEx’s API, known as ShipAPI, is about as impressive a success story as one could hope for.

At one time, FedEx was flooded with what they call “WISMO” (Where IS My Order?) calls — customers dealing with extreme variations in ship times, inaccurate delivery dates, and a lack of updates from FedEx to the customer. To rectify this problem, FedEx analyzed the frequency of these calls, the locations from which they were made, the types of packages being sent, and the shipping services used. Additionally, FedEx examined their own internal practices, the rate of adherence to electronic scanning of barcodes upon receipt at a FedEx facility, and the effectiveness of their long-distance package tracking system.

By identifying the weaknesses in their own system through the use of internal metrics (utilizing both anecdotal metrics from users and operators as well as hard metrics derived from servers, scanning systems, and internal sorting reports), FedEx was able to pinpoint their points of failure, decreasing the time it took to deliver a product, increasing customer satisfaction and confidence, and creating a channel of communication that could update the customer faster and more accurately than any system previously used by the customer. Almost overnight, the FedEx brand became synonymous with quick, efficient shipping, and that is greatly due to their effective development and implementation of an API utilizing proper metric analysis.

7.9 Security, Effectiveness, and Commerce

Fundamentally, the process of metric analysis is not just one focused on security. Commerce, effectiveness of solutions, and more can all be determined through the proper derivation, analysis, and application of solutions deriving from metrics. By determining your end-consumer, their needs, the needs of your process, and the total result of the application of an API, you can dramatically increase revenue, security, and even your consumer-base.

The lesson of it all? Understand your consumer base. Understand the API you are developing. And, most importantly, measure your success and learn from your failures.

Next Steps:

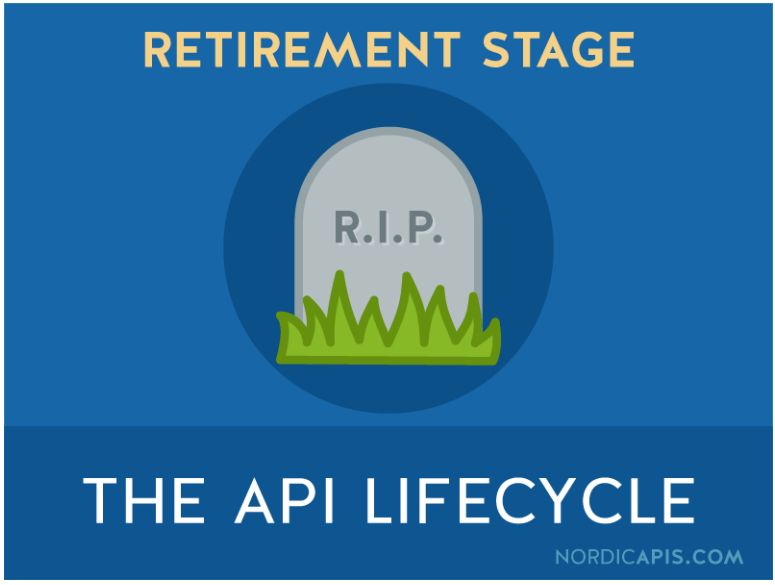
By now you should know that in order to grow your developer audience in a sustainable fashion, an API should be treated just as if it were it's own product. Your Operations activities will then consist mostly of marketing your API by implementing the best possible Developer Portal and by following a combination of different traction channels.

The Operations Stage will inform you on what exactly it takes to push your API into high-growth mode. You'll also discover marketing expenses related to developer attraction, aiming to have that value be much lower than your revenue — otherwise you won't have a business.

As you learn and your API experiences real world consumption, you'll likely have to refine your offering. If you're not obtaining the desired traction, consider revisiting the Analysis Stage to refine your business objectives. If, during Operations, you find that there's a missing feature or change needed you should revisit the Development Stage. Finally, if your acquisition costs are so high that you can't possibly maintain a flow of incoming developers you should consider jumping to the Retirement Stage of the API Lifecycle.

1. Analysis Stage
2. Development Stage
3. Operations Stage
4. Retirement Stage

IV Retirement Stage



Why would a web API be retired? what business factors contribute to this decision? In this part we examine the **Retirement Stage** of the API lifecycle, taking an in-depth look at example API deprecations to attempt to answer these tough questions and examining how to successfully retire your API while saving face with your developer community.

8. A History of Major Public API Retirements

Old age comes, and retirement is a part of life. It's no different in the API world. The retirement of the Netflix, Google Earth, LinkedIn, and ESPN APIs are examples of some recent large public API deprecations. Every major API retirement tends to spawn worry within the developer community; are public APIs doomed?

We don't think so — don't forget there's currently an estimated 13,000 web APIs in operation, and the sector has shown exponential growth and is slated to increase with the IoT (which isn't just a fad). However, all things come to an end. As conditions change, technologies must evolve to meet new market conditions. A public-facing API may prove successful for some organizations, whereas for others, open environments may not be viable and lucrative.

8.1 What Does Retirement Mean?

Retirement can mean many things. An API version may be scheduled for complete deprecation, or the API could still exist in a limited format with increased access controls. An API may simply slim down it's excess functionality to become more consolidated and lean, or be fully replaced by an internal technology or outside competitor.

In order to fully understand the conditions that beckon major API changes, we'll lay out **5 common causes that**

beckon API retirements. Using large API deprecations from recent years as examples, we'll explore the reasoning behind each one that we've been able to surmise through our research. Sound fun? We think it's pretty fascinating what can be learned from hindsight...

8.2 Retirement Reason #1: Lack of 3rd Party Developer Innovation

A common scenario that may cause an API shutdown is if the API is receiving limited or unsolicited use by developers and end users. Though this could be due to inadequate marketing, it could also be proof that the API is not offering a true value to it's consumers.

Netflix

Netflix proudly unleashed their API program in 2009, citing the possibilities for new app creations by third-party developers. Eventually they stopped issuing new API keys to developers, and by 2014 this was shut down to all users aside from select partner integrations. According to Daniel Jacobson, Director of Engineering at Netflix, their reasoning was "to better focus our efforts and to align them with the needs of our global member base."

In the end, all of Netflix's public API requests equated to about one day of private API requests. The user base for the public API was too miniscule to warrant it's existence.

The result was to eliminate their Public API program to embrace a single private/partner Java API. Nowadays, every Netflix app on any device taps into the Netflix private API, which performs data gathering, data formatting, and

data delivery for all consumers. Netflix has also eliminated versioning within their API to embrace impermanence and constant change.

We learn from this case that when a brand name and native platform is so powerful, creating an ecosystem with a public perhaps not necessary. Also, according to technologist Andy Thurai, it "becomes very expensive and time-consuming to maintain" both a Public and a Private/-Partner API.

8.3 Retirement Reason #2: Opposing Financial Incentive, Competition

A Public API may be retired if the interface interferes with primary business objectives. In exposing their data for anyone to consume, providers run the risk of losing a market advantage, especially if this is taking business away from native distribution channels.

ESPN

In the case of ESPN, their reasoning to retire their sports data platform was "to better align engineering resources with the growing demand to develop core ESPN products on our API platform." Today, if you visit the ESPN Developer Center, you'll notice that all APIs are open for "strategic partners" like IFTT, Pulse, and foursquare.

Discussion on Hacker News suggested that it was because developers were monetizing their apps. Andy Thurai agrees, positing that "the actual reason seems to be that they want better control over their unique and licensed content, and better monetization, which public APIs may not offer." We learn that by exposing internal

assets, a business may risk losing an advantage if doing so decreases traffic from native distribution channels.

Strava

Strava offers fitness software for wearables with a developer API to extend their applications. Starting with Strava API v3, they began to curate their developer consumers with a limited access program. These new requirements prohibit "applications that encourage competition" as well as ones that "[Replicate] Strava functionality." A large API retirement can occur when developer consumers are incorporating monetization techniques into third-party apps that break the original terms of agreement.

8.4 Retirement Reason #3: Changes in Technology & Consolidating Internal Services

The tech sphere is constantly evolving. As such, advancements in these technologies have the potential to make an API obsolete. This happens often during internal redesigns, acquisitions, or external industry advancements. API retirement could arise from evolving protocol trends, such as replacing SOAP with REST, or changes in end user experiences, such as Netflix shiting it's primary business model from DVD distribution to video streaming.

Skype

Skype's reason to decommission their Desktop API was, according to them, that it didn't support new mobile development trends. The Skype Public API, first introduced

in 2004, was taken over by Microsoft in the 2011 aquisition. The Microsoft-owned Skype Desktop API was shut down in order to embrace Skype URIs and the Microsoft client. The termination irritated some developers as it erased income for third-party applications consuming the API as well as lessened functionality through the Skype URIs.

Google Maps Data API

At the same rate Google releases new APIs, it also deprecates older versions. In 2010 Google announced they would discontinue the Google Maps Data API, a service made to store geospatial data. The main reason was that a superior feature was launched with v3 of the Google Maps API, offering an improved method of query and storage that made the Google Maps Data API obsolete.

Other Google APIs scheduled for deprecation due to outdated technology:

- Google Webmaster Tools
- Google SOAP Search API

Fedex

At the time of this writing, Fedex is in the process of updating its APIs to Fedex Web Services, replacing XML-based tracking applications with a large redesign.

8.5 Retirement Reason #4: Versioning

Versioning is by far most common reason to retire an API. Since APIs are a relatively new technology for most

businesses, you'll often see v1, v2, or v3, still in use today. Similar to OS upgrades on devices, it's often easier to scrape and replace an entire version when instigating large change. Versioning can arise when major edits to the API parameters or new usage controls are required.

Some alarming changes that dramatically affect developers and end users may be quietly hidden within new API version documentation. Functionalities may be erased, permissions limited, classes altered, etc. As versioning is a common component to many API's lifecycles, we don't intend to create an exhaustive list of them all. Rather, these two examples of version updates came with them a few surprises for developers and end users...

YouTube

YouTube, in the process of migrating to the YouTube Data API v3, surprised users when it was found that Data v3 would no longer support the YouTube app on expensive Smart TVs.

Twitter v1

Twitter's v1 was finally retired in 2013. Being the 3rd most popular API in use today, the announcement sparked worry throughout the developer community that simplistic integration was a thing of the past. The update added OAuth, requiring client keys and the creation of an app with Twitter. According to Mathew Mombra, these changes were likely instigated to:

- enforce call rate limiting with less liberal access
- reduce their system load
- decrease API vulnerability
- promote their embeddable tweet program

8.6 Retirement Reason #5: Security Concern

A consideration of many large-scale retirements, API providers may choose to discontinue a public API specifically because of the security concerns associated with making internal data and processes public.

Google Earth API

The API was shutdown because it relied on NPAPI Framework, an outdated technology Google mentioned had become a security concern.

SnapChat

Though SnapChat never technically has released a public API, the popular app has spawned a variety of mashups on various platforms that all tap into their "private" API. This went unchecked, until early 2015 when SnapChat partnered with the Windows store to crack down on these rogue applications.

9. Preparing For a Deprecation

Entire software ecosystems can quickly emerge that depend upon APIs to survive. And just as quickly, they can be destroyed. As these third-party developers have depended on your service to support their own products, complete shutdowns can inherently cause very negative reactions. This is why any change to your API requires the proper preparation and communication, whether it be small edits, versioning, or complete deprecation.

9.1 Rippling Effects

Though most clients are quick to adapt, some may be unaware of API changes. This means they won't respond quickly to update their system to be compatible with new versions. Stagnancy, freezed dependencies, forgotten forks, and inactive projects can be common with APIs that have been active for a while.

Mircea F. Lungu, a computer science researcher at Bern in Switzerland, found that when deprecating API methods or classes, the total **adaptation time** within an ecosystem can vary tremendously. The amount of time between the first reaction and the last reaction can be months, or even **years.**

In his study on the case of the SmallTalk ecosystem of 2,000 consumers, Lungu found that 14% of deprecated methods induced negative rippling effects within the ecosystem, and 7% of deprecated classes triggered ripple effects. Only 16% of deprecations had a systematic replacement, which meant that most edits were done manually. Thus, any changes to an API induces ripples that can have far-reaching results.

9.2 Preparing For Developer Reaction

In order to keep good face with your community, we recommended transparency using the following techniques to prepare for an API's retirement.

- **Schedule a retirement plan:** Offer at least a 6 month time frame for complete deprecation. This will hopefully give teams sufficient time to seek new integrations or partnerships. Consider extending this

time frame if certain teams are struggling. Google Maps and Youtube follow a one year announcement to deprecation policy.

- **Make changes in stages:** Tier your retirement plan in a way that aligns with past promises for support and versioning. If still running, shut down older versions that receive limited use earlier in the process. Twitter has created handy retirement timelines in the past that display when each version will be retired.
- **Public announcements:** Write and promote a blog post that explicitly outlines API changes, when they will go into effect, who will be effected, and any other vital information your community needs to know. Disseminate this information to your dedicated developer social accounts.
- **Ease the transition:** If your API is being consolidated, versioned, or merged into a separate service, ease the process with a **migration guide** and **FAQ**. For example, when Google Maps Data transferred over to Google Maps, they created a Maps Data Liberation Tool to offer a complete download of data or transfer to their new console. Awkward transitions can especially be a headache during acquisitions.
- **Use blackout testing**: Blackout testing is a planned shutdown of all API functions for a short window of time. This is helpful for developers to see how the absence of the API is going to affect their systems, and also acts as a call to action for developers to update their code. The Youtube API, Twitter API, Mendely API, and many others have used blackout testing in the past.
- **Don't violate your own terms of service:** Review your Terms of Service to ensure that you have no

binding contracts for continuing support before you pull the plug. A smart way to plan ahead is to write a deprecation policy whenever a new API version is created.

Next Steps:

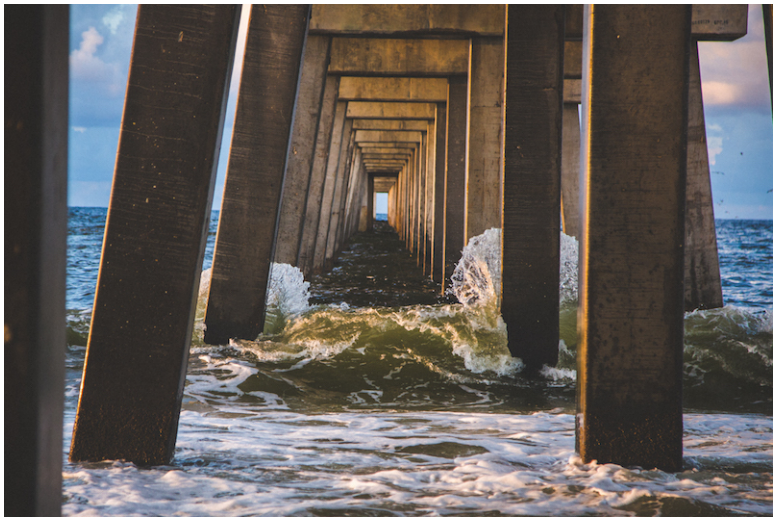
We've highlighted the major concerns that would usher a large-scale API deprecation, but many retirements occur for a combination of many these reasons. Other reasons to consider large-scale API depreciation or redesign may include:

- Overwhelming competition with other APIs in niche sector
- Inability to structure data to be received effectively
- Not generating a revenue to support operations with current offering
- and many others

Scheduling the retirement of an API version may be a great advance toward restructuring how your data will be received by your consumers. In this case, revisit the **Analysis Stage** to consider your primary business objectives.

1. Analysis Stage
2. Development Stage
3. Operations Stage
4. Retirement Stage

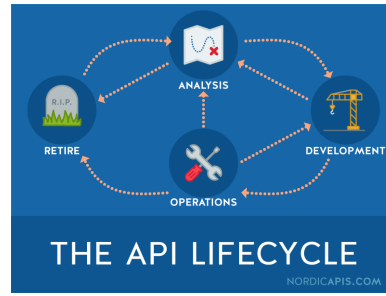
V The Agile Mindset



Each phase of the API Lifecycle should not be considered static. Progression is often non-linear, meaning information garnered in a phase should simultaneously affect other ends. Iterative communication between stages is critical for influencing smart progression and arriving at a sustainable internal ecosystem.

For example, if you receive multiple feedback complaints from customers not satisfied with a certain operation, alterations should be induced that only effect **Develop-**

ment, but inform **Analysis**, and perhaps even **Retirement.**



Now that your API has travelled through all stages of it's life, it may be entirely changed, or might have kept most of it's original form. Now that we have a good grasp on each stage, in this part we consider what adopting an 'agile' mindset means for developing and maintaining an API, and offer helpful resources to expedite the process.

10. What Makes an Agile API?

Agile. API. API. Agile. It has a nice ring to it, but is it necessarily so? Having become a buzzword, 'agile' often comes loaded with misconceptions. At a Nordic APIs conference, Christian Nilsson of Beepsend SMS messaging solutions explores the concept of what it means to be truly agile in

the context of APIs, offering 6 tips to help an API provider determine what 'agile' really means for them.

10.1 What is Agile?

Agile can be defined as an approach to problems, a software development process, and a business process. Like it's name literally suggests, agile is lean, nimble, and filled with passion, with the ability to react to change rapidly.

To Nilsson, agile has three common themes:

1. Agile is breaking things down—but not to the point that they break.
2. Agile means you can quickly shift business strategy.
3. Agile cannot be burdened by previous decisions.

Agile API Misconception #1: If You're Not Agile, You're Not Successful.

In general, the closer a service is to the end consumer, the more agile it needs to be in responding to problems. As Nilsson says, "Internet users are fickle and promiscuous. They're curious. They'll leave your service in a heartbeat."

However, some businesses are naturally inflexible. When these businesses increase their distance from the end consumer, they become less agile and more stable. This gives the business a stronger reputation for reliability, which is a highly marketable trait to less flexible industries like banking.

Services should ask themselves: **Why do we need this API?** What does 'agile' mean to us and our business model? Is it right for us? Is it right for our customers? In

the end, according to Nilsson, "you're not automatically successful just because you're agile, you're successful because you're passionate about your core model."

Agile API Misconception #2: There is Only One Perfect Software Development Process.

So often, project management tools are thrown at software developers in order to make them work faster in a more transparent environment. But software development processes vary by needs, and corresponding tools should support – not enforce – your software development process.

Project managers for APIs should start by assessing the size and makeup of their development team, along with critically analyzing the problem the team is trying to solve together. Considering these factors, the right strategies for development will naturally arise, often straight from group discussion on the topic.

Agile API Misconception #3: Your System Is Secure.

Security doesn't start and end with outsmarting hackers. Nowadays, no matter how secure your system is, you are constantly battling against an infinite number of scripts, human errors and stealthy passwords. In this constantly unstable and insecure world, one shouldn't relax and assume that all will remain stable.

Bugs such as Heartbleed roam the web, and system dependencies can be altered at any time— such as recent MySQL changes enforced by AWS. A way to overcome these obstacles is through centralized configurations, which make sure that keeping your systems up to

date is not labor intensive. You must be vigilant for any holes in the technology and to make sure that all developers are taking active steps to avoid cross-site scripting (XSS flaws).

For an in-depth study on implementing API security check out: API Security: Deep Dive into OAuth and OpenID Connect

Agile API Misconception #4: Your Extensive SLA and EULA Will Save You.

Paypal includes 97 pages in their SLA. The reality of a Service Level Agreement (SLA) or End-User License Agreement (EULA) is that it is a best-case scenario. Some users will inevitably use a service improperly. How often will your team or the customer have the money and patience to bring a side to court on a trivial matter? You may like that your SLA or a EULA creates a sense of fear, but Nilsson argues that API providers should be more inviting to consumers instead of scaring them away with lengthy user agreements.

Agile API Misconception #5: Your API is 'Special.' Keep That Secret Sauce a Secret.

According to Nilsson, there are no secrets in coding anymore. The "secret sauce" of a business truly lies within execution. The secret sauce is how a you can adapt and transform to fluctuating markets. It's how you practice agility. And it's also how you provide fantastic customer service.

> "If your secret is that no one has figured out what technology you use, you're screwed, you're doomed, you're beyond saving. Your secret

> must be your staff, your DNA, your business model, your ability to change…"

Technology is just one more conduit with which you can support your employees and help them strive for greatness. Plus, if you keep your tech a secret, when things go wrong, who will you be able to lean on? Communities established around open code and collaboration can help APIs dramatically increase in value; that's what smart cities all over the world are realizing by opening up their data silos and API source code.

Agile API Misconception #6: NoSQL is the Future That Will Power All APIs.

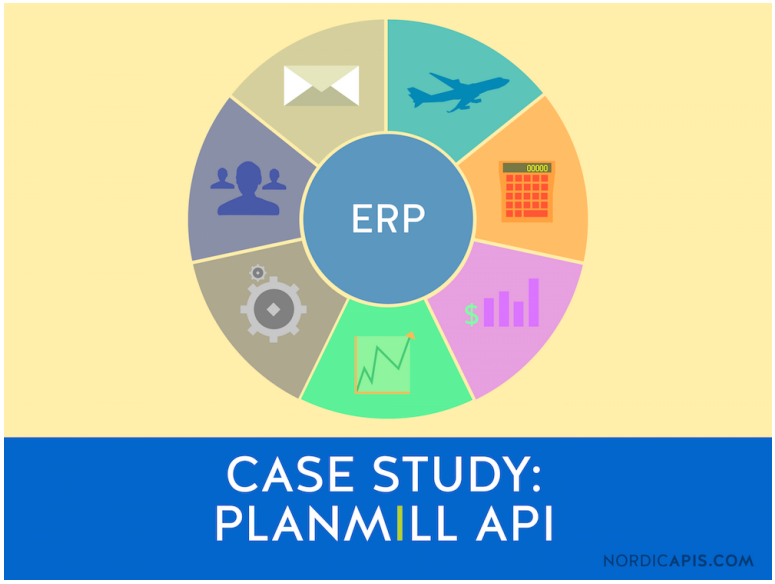
Nilsson argues that the NoSQL database for storing and retrieving data just isn't all it's hyped up to be. He concedes that NoSQL has the ability to solve many problems, but argues that it cannot solve not all of them. Like with all things in the programming and development world, solutions need to be agile. Problems need to be critically assessed and paired with the best solutions. Often times this means pairing NoSQL in combination with other solutions like a relational database.

10.2 Conclusion

API providers must incorporate creative combined solutions for development that are tailored to their unique position. When building an agile API, you must keep all these facets in mind without falling victim to misconceptions of what it means to be agile. Following these considerations, and the four stages outlined in previous chapters, you should be prepared to react properly to

your team's needs, your business's needs, and the end consumer's needs.

11. Case Study: PlanMill API



It's true that APIs are accelerating normal B2B communications and ERP technologies, but masking a 10+ year old system with a clean interface is no easy task.

Tuning into design lessons learned throughout research, development, and implementation stages, in this post we study a case that epitomizes the mullet in the back philosophy. Read on to learn what processes created a fully functional and partnership critical API as we study the history of the PlanMill API. From simultaneous UI development, Eating your own dog food, to reasoning behind specific architecture decisions, Senior Consultant & Manager Marjukka Niinioja describes to Nordic APIs the

nitty gritty details of developing an enterprise level API from scratch.

11.1 What is an ERP?

PlanMill is an **Enterprise Resource Planning** (ERP) and **Customer Relationship Management** (CRM) application — a real time front and back end project management platform for professional services. The PlanMill API exposes the Planmill ERP platform, allowing for the commercial exchange between various business applications to expedite processes like accounting and CMS. Maybe due to it's sheer weight and business-oriented subject matter, ERPs are often seen as pretty boring. According to Niinioja:

APIs, on the other hand, are the wave of the future — they are fast and nimble. So how can you combine the boring and the agile?

11.2 The Slow Initial Release

The PlanMill API went through a painful birthing process — a 12 month period of stress and test. Pretty early to the game, the PlanMill API 1.0 was born in 2009. A very **quiet launch** worked well, as it gave the team time to create usable developer **documentation**, envision a **pricing model**, speak with consultants, garner a potential customer **user base**.

Eventually, some of the first real use cases occurred internally, with larger projects and time reporting integrations. This was followed by a first big API use case within a larger application, including massive improvements to the API.

The first partnership was eventually made possible (Atlassian Confluence with Ambientia) using the PlanMill API. According to Niinioja, the presence of an API became an immediate selling point compared to competitors in the ERP space.

11.3 API Usability Problems

So, what did the PlanMill team learn along the path to version 1.0? Developer experience is a critical component to your API strategy, and designing documentation and API functionality in a way that mimics the user's desires is pertinent to success in the API space. Though certain standards and API design guides do exist, actual implementation is almost always unique. What PlanMill did correctly was listen to their customer's responses to improve the overall experience to increase usability.

As the system originally intermingled much functionality with a backend system that was 10+ years old, the following usability concerns arose in the initial use of the API:

- **Calls**: "Why should this take 5 requests, can't I consolidate into a single call?"
- **Rate limits**: Some users would send in 10 requests per ms, crashing the system.
- **Backend exposure**: There was too much initial reliance on old JavaScript. Lesson learned is that not all ideas are compatible from front and back end perspectives.
- **Documentation**: Not every granular function was initially documented.
- **Error Messages**: The users were receiving stack traces as error messages in the API responses.

- **Response ID**: The ID wasn't in the response of whatever had just been created with the API.

11.4 Improved Process for API 2.0

For the API v 2.0, a new, improved process for development was necessary. Gearing up for a new product lifecycle, the PlanMill team performed impressive customer research to dial into what their customers desired. The PlanMill team stressed **iterative** changes bolstered by continuous stages of modeling the product both internally and externally to receive **feedback from the developer community** by demoing and attending API related events.

1. **Research Project**: The team created a thesis-driven formal research project to investigate the needs of their customers, asking partners to share their own experiences using the API and thoughts on how to improve the API. They found, in general, that customers were eager to share if it meant helping to improve the system.
2. **Pilot technologies and first service**: Using the research, the team brainstormed what could they do differently if they could start from scratch. They redesigned their technology stack, and masked old things that weren't a well designed to offer a clean interface to their clients. Sleek front, mullet in back.
3. **Demos**: What followed was internal discussion, knowledge sharing, attending seminars, and demoing the API.
4. **Further Research**: The team read Nordic APIs, researched the community, tried new approaches, decided on architecture and technologies, attended

additional seminars, and researched monetization strategies.

5. **Internal Beta** : Operational version developed for internal testing.
6. **Ate own Dogfood w/ New UI**: A new user interface was designed simultaneously with backend development. The team found that insights from a UX perspective were critical for creating an intuitively designed API.
7. **Public Beta 1.5**: This staged involved developer community testing, and opening the API for third party developers to use. They held a usage seminar, reached a version 1.5, and made incremental updates.
8. **Feedback from Developer Community**: Currently the team is at this stage.
9. **Publish 2.0**: Plans to publish full platform soon. The team aims to use their research, feedback, new UI, architecture decisions, monetization strategy, and more to culminate in a well-informed and well-prepared version 2.0 release.

11.5 Specific Architecture Decisions Made Along the Way

Let's take a peek under the hood. In response to their research phase, the PlanMill team decided to go with the following architecture choices:

Authentication: HMAC and API keys

PlanMill went with HMAC as an authentication scheme as it allows Business-to-Business communication (B2B) via system level integration capabilities. For PlanMill services,

server-to-server file sharing is preferred for the transfer of payrolls, for example. Though PlanMill acknowledges they still need to improve their unified identity management process with additional technologies, HMAC is adequate for their current situation involving one user and one set of credentials to talk with another single user and single set of credentials.

Data Format : JSON (and more) over XML

The PlanMill team chose slick JSON, a format that's simpler to understand and requires less configuration overhead than XML. Some might scoff when they hear that in addition to returning JSON, the PlanMill API produces formats like CSV and... PDF. In their case, this return type actually makes sense as certain business intelligence software receives PDF or CSV formats for their operations.

HTTP Verbs Properly Used

The PlanMill API utilizes standard HTTP verbs (GET, POST, PUT, DELETE). Though not implemented in PlanMill API v1.5, Niinioja advocates the use of PATCH, for the reason being that it can elleviate headaches and bloat that can occur when trying to update delta records. Minute changes in large datasets, such as a minor user info edit, can be made heavyweight with redundant GET and POST requests. The team plans to embrace proper PATCH standards with the 2.0 release.

RAML Documentation

PlanMill decided to go with RAML over Swagger, Blueprint API, or other specification formats for REST APIs.

PlanMill also leverages APImatic to generate SDKs in various languages.

REST or SOAP?

Is REST really better than SOAP? Is SOAP better for critical business transactions? There's an argument for both sides. For PlanMill, SOAP and WSDL have a place in their platform for invoice, account, and payroll. However, as Niinioja describes:

11.6 Sharing Data Carefully and Securely

It's important to note that in the B2B environment some critical level data or company secrets cannot escape the system, but still must be shared via API across partner channels.

Things like bank account numbers, sick leaves, payroll data, etc, need to be shared with systems, but cannot escape out of those boundaries. You also have government and trade treaty control, such as in Finland, PlanMill's base, where **personal data** laws place strict confines on what can legally be shared. Also a company may have an internal policy controlling shared data. All this can inhibit open integration with platforms like Zapier or IFTT.

In the end, Niinioja recognizes that a business needs to support both opennness, for basic contact information or other such data that can share freely, and point-to-point routes, for sharing API data between partner systems through secure connections.

11.7 Understanding the Lifecycle of an API

PlanMill understood the lifecycle of their API, bringing an agile mindset to enterprise API platformitization. Each stage in this process plays an important role in advancing the project as a whole. From analysis, development, operations, and throughout versioning.

Though PlanMill still has their work cut out for them, hopefully this case study can give insight into what mistakes were made early on in the process, and how they were rectified, with the hopes that new API practitioners can have a successful release armed with this knowledge. We wish PlanMill a successful 2.0 release, and hope their production saga can help as a model for others to have similar success in the API space.

11.8 Additional Resources:

- PlanMill API v1.5 Documentation
- Accidental API Developer: Slideshow

[PlanMill has participated in Nordic APIs past events but did not sponsor this post]

12. Third Party Tools

So, you're confident and ready to hit the ground running, but still could use guidance in managing your API. Well, we've assembled the following list of popular third-party tools built to assist API providers - from documentation, monitoring, to security solutions - that we've mentioned throughout this e-book or otherwise recommend for use. This list is not meant to be exhaustive at all, but rather is a curated list of what services we are attracted to.

Documentation

Frameworks that ease the process of generating API documentation.

- Apiary
- API Blueprint
- Swagger

SDK Generation

- APIMATIC
- REST United

API Management

- 3Scale
- Apigee
- Axway
- Mashery
- Mulesoft
- SmartBear

Performance Monitoring and Testing

Services that allow a provider to monitor the activity and status of their API or API dependencies.

- SmartBear
- API Changelog
- APIMetrics
- APIScience
- POSTMAN
- Runscope

Security

Services and tools to assist in embedding proper identity control, access management, and more with authorization, authentication, delegation, etc.

- OAuth 2.0
- OpenID Connect
- Ping Identity
- Stormpath
- Twobo Technologies

Discoverability

API directory services, hubs, marketplaces, and more that assist in making your API discoverable by third party developers:

- APIs.io
- Mashape
- ProgrammableWeb

Developer Support

In addition to using social media channels and internal blog for updates, these developer-oriented channels can be used to grow your audience:

- Certain sub-Reddits: API, Programming
- GitHub
- Stack Overflow

Nordic APIs Resources

API Lifecycle Talks

In May 2015 Nordic APIs went on a World Tour in the theme of the API lifecycle, visiting London, Copenhagen, Munich, and Seattle. If you missed out, you can still watch sessions here:

- Introducing The API Lifecycle, Andreas Krohn
- You Need An API For That Gadget, Brian Mulloy
- Pass On Access: User to User Data Sharing With OAuth, Jacob Ideskog
- APIfying an ERP, Marjukka Niinioja
- Integrating API Security Into A Comprehensive Identity Platform
- A Simpler Time: Balancing Simplicity and Complexity, Ronnie Mitra
- The Nuts and Bolts of API Security: Protecting Your Data at All Times

More eBooks by Nordic APIs:

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 Winter Collection: Our best 11 blog posts published in the 2014 - 2015 winter season.

Nordic APIs Summer Collection: A handful of Nordic APIs content offering best practice tips with a focus on becoming an API platform.

Endnotes

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

- Select icons made by Freepik and are licensed by CC BY 3.0
- Select images are copyright Twobo Technologies and used by permission.

Nordic APIs AB Box 133 447 24 Vargarda, Sweden

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact