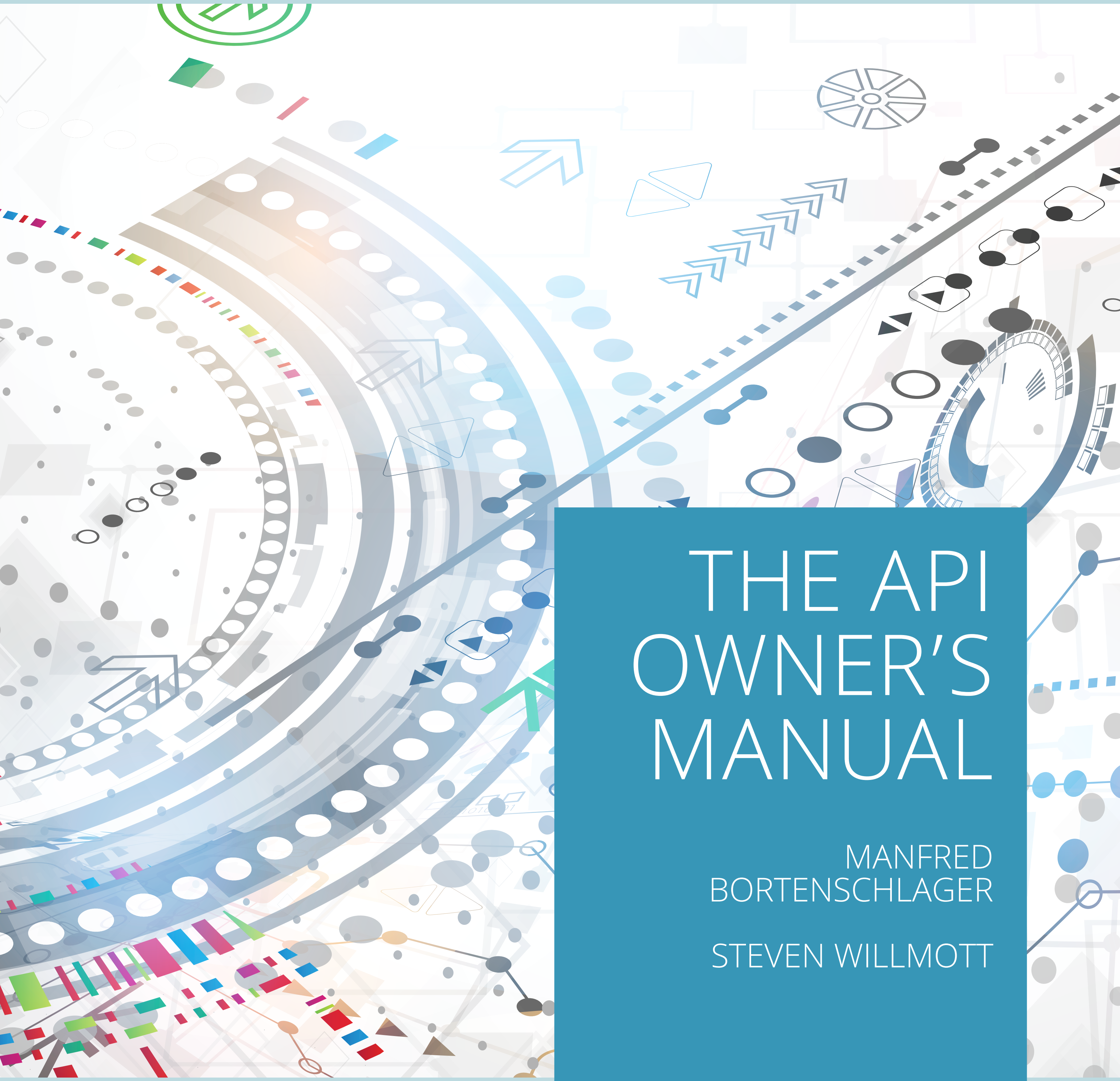


THE API OWNER'S MANUAL

MANFRED BORTENSCHLAGER

STEVEN WILLMOTT

Best practices of successful API teams

EXECUTIVE SUMMARY

APIs are becoming the digital connective tissue of modern organizations, adding new capabilities to everything from their operations and products to their partnership strategies. In 2015 it's no longer a stretch to say that most organizations don't ask whether to engage in API programs, but *how* to do so. This ebook aims to answer this question by drawing on best practices from leading practitioners in seven areas key to the success of effective API programs:

1. Focus relentlessly on the value of the API
2. Make the business model clear from the beginning
3. Design and implement with the user in mind
4. Place API operations at the top of the list
5. Obsess about developer experience
6. Go beyond marketing 101
7. Remember API retirement and change management

By the end of this book you will have gained an overview about how a successful API program runs and holds together, along with best practices about how to make that happen. We've included a number of examples of successful API programs, including Amazon, APIdaze, Context.IO, eBay, Flickr, Lingo24, Netflix, Pingar, SendGrid, Senzari MusicGraph, Slice, Stripe, and Twilio.

Most importantly, after going through this book, we hope you'll have asked yourself key questions about your own plans, and will have the right focus needed to create a valuable API.

We wish you success with your APIs!

TABLE OF CONTENTS

INTRODUCTION

APIs have quickly become mission critical for any number of businesses, and their implementation is increasing. APIs are used for many purposes—internal agility between teams, underpinning mobile or Internet of Things initiatives, enabling customer integration, or powering a partner program. No matter what the use case, one thing is clear: API success has become intrinsically linked to business success. As this correlation becomes stronger, the question is now rarely "Why implement APIs" but "How can we implement effective APIs?"

While every API program is different, there are a few common practices that teams can use to evaluate their own approach and ensure success.

This book brings together the 7 most common factors we've seen the best API teams adopt relating to success. These practices may appear simple at first—and indeed, they are, once adopted—but we often see organizations skip these steps or focus on the wrong targets or solutions. These failures can easily occur since APIs inherently mix technical and business concerns, and with many stakeholders involved it can often be difficult to pin down exactly what the key issues at stake are.

The team running an API might range from a single member tasked with implementation and operations to multiple teams that range across engineering, operations, product, support, community, and management. At times an API may be a supporting element for a specific company product line, or it may be the foundation for an entire company. Whatever the case, many of the key success factors will remain the same.

In truth, these best practices are not so secret—they're hidden in plain sight if you know where to look. In this book we'll uncover what's been used by successful API teams, with plenty of examples for reference.

In our experience there are 7 best practices that are found in almost every successful API program. While there are other elements of APIs to get right, thinking these through and following up in execution will move you a long way towards success. The 7 best practices are:

1. Focus relentlessly on the value of the API
2. Make the business model clear from the beginning
3. Design and implement with the user in mind
4. Place API operations at the top of the list
5. Obsess about developer experience
6. Go beyond marketing 101
7. Remember API retirement and change management

While these may seem relatively simple on the surface, there are subtle nuances to each of them and this is what we'll be covering in this book. Together, these best practices give a robust framework to a successful API program.

In each section you'll see an overview of what the best teams do, then real-life examples, and finally the five key questions you should ask yourself, your team, and the company to make sure you're on track.

The observations and examples come from 3scale's extensive experience working with hundreds of production APIs, as well as industry-wide best practice from a range of companies.

Before diving in, a few words on the structure of the book. We first discuss the important aspects to consider relating to the strategy for your API. After this we described the typical composition and tasks of an API team, and we cover the 7 best practices one-by-one. Finally, we wrap the book up with a summary of the key points.

STRATEGY FOR THE API

This book focuses very much on the "how" of APIs rather than the "why," and is not to be considered an API strategy guide, although it can still help you formulate some of the questions that need to be answered. However, a clear overall corporate strategy for your API is an essential starting point—if you don't have this you'll need to define your overall strategy, such as key goals and metrics before beginning implementation.

Before diving in, it's worth searching out (or indeed writing down) the key objectives of your API program for reference. Some of the questions asked later may help flesh it out, validate, or change it.

An effective API program has to build on an organization's overarching corporate strategy and contribute to its objectives. You'll know you have the makings of a great strategy when you can answer the following three questions in a clear way:

1. **Why** do we want to implement APIs?
2. **What** concrete outcomes do we want to achieve with these APIs?
3. **How** do we plan to execute the API program to achieve that?

If you don't have answers to these questions, then they have been lost in execution and inertia. Much of what follows depends on establishing this true north for your program.

A clear overall corporate strategy for your API is an essential starting point—if you don't have this you'll need to define your overall strategy, such as key goals and metrics before beginning implementation.

The Why

People often misinterpret this question in different ways. Firstly, rather than focus on the **value of the API** per se, it's helpful to think of the **value of the effect of the API**. Remember, it's the organization's core business that's valuable, not necessarily the API. An API is valuable when it becomes a channel that provides new types of access to the existing value an organization delivers.

Another common error is the belief that for an API to be valuable in itself, API users must be prepared to pay for it. This is true only if the API itself *is* the product. In most models, this is not the case (this is true for less than 20% of 3scale's current customers for example). APIs are usually driving some other metric—pizza sales, affiliate referrals, brand awareness, etc. The value of the API to users is the result of an API call (service request and response), rather than the call itself.

In our Winning in the API Economy ebook, we sketched out typical uses cases for API providers, which can be a helpful guideline:

1. To enable mobile as an additional channel
2. To grow ecosystems: customer (B2C) or partner ecosystems (B2B)
3. To develop massive reach, for transaction or content distribution
4. To power new business models
5. To drive internal innovation

The most common business drivers for establishing an API program, according to a recent survey of 152 organizations conducted by the Cutter Consortium and Wipro, are: to develop new partnerships, to increase revenue, to exploit new business models, to improve time to market, and to develop new distribution channels. The top technology drivers are: to improve application integration, to improve mobile integration, and to support the connection to more devices.

This *why* clearly needs to be strong enough so that the decision to make an investment in the APIs is an obvious choice for the organization.

EXAMPLE: FLICKR[1]

by Kin Lane

Originally created as an online game, Flickr quickly evolved into a social photo sharing sensation. The launch of the API helped Flickr to quickly become the image platform of choice for the early blogging and social media movement by allowing users to easily embed their Flickr photos into their blogs and social network streams.

One of Flickr's key drivers for exposing APIs—in other words the *why*—was to more effectively grow their partner ecosystem. The Flickr API is the driving inspiration behind the concept of BizDev 2.0, a term coined by co-founder Caterina Fake to describe their policy of requiring companies to use the API to develop applications.

The company would only contact companies who had successfully attracted users, which increased efficiencies in how Flickr engaged with its partners and allowed it to rapidly build a network of trusted, high value partners.

The What

The second question should be "*What concrete outcomes do we want to achieve with these APIs?*" In other words "what do the APIs actually do and how do they impact on the wider business strategy?" In strategy theory, both the concepts of the internal view and the external view of an organisation can help to define the *what* of the API.

The internal view refers to specific and valuable assets an organization possesses. The more valuable, rare, inimitable, and non-substitutable—also often referred to as VRIN assets—the more suitable it's for the *what* of an API. An organization that has unique data could leverage this by allowing access to the data via API. Facebook is one of the world's most popular media owner, their content gets a lot of "likes," and Facebook is the only provider who can open up access to the number of likes of a piece of content. That makes this content and the API that allows access to this content extremely valuable.

What concrete outcomes do we want to achieve with these APIs?

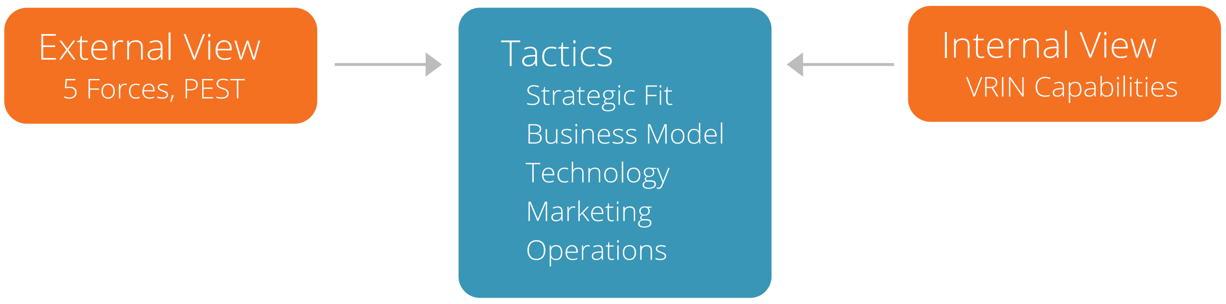
1 This example is taken from the 3scale white paper The Platform Vision of API Giants written by the API Evangelist Kin Lane.

The external view is related to everything outside of an organization, such as market dynamics, trends, competitors, customer behaviors, etc., which are macro-environmental drivers and industry forces. Macro-environmental drivers are political, economic, social, and technological, as well as industry forces such as competition, buyers, suppliers, substitutes, or new entrants (cf. Michael E. Porter's Five Forces). This external view affects every business strategy—including thinking about *what* an API should do.

An example about how external events can influence strategy is the success of the Google Maps API. Another supplier of geographic information was Navteq, and arguably a Google Maps competitor. Google API is open, but Navteq did not have an API. Navteq was struggling, then bought by Nokia, and then rebranded to HERE. HERE now does provide an open API and a fully-fledged developer program. Waze is a startup that from the beginning provided a public API for their traffic and navigation data and used this as the backbone of their growth strategy.

Another example is the fitness domain Fitbit, which offered their activity tracker and fitness products together with a public API to foster innovation and grow their ecosystem. They disrupted giants in this market such as Nike and Adidas, and both soon decided to leverage APIs to support their strategy.

When deciding about *what* an API should do for a business, both internal and external views need to be examined. The decision about the *what* is then usually a combination of the two. The illustration below shows how internal and external views can help finding the right *what* for an API program.



In concrete terms, while the *why* is unlikely to change often, the *what* may vary significantly based on external factors such markets, technical considerations, or economic conditions. Also, internal directions about the value of an asset may change, which could also affect what should be achieved with an API.

The How

The final question, *"How do we have to design the API program to achieve what we want?"* is all about implementation and execution. **The *how* is really what this book is all about.** The *how* covers many further questions such as:

- What technology is used to build the APIs?
- How are they designed?
- How are they maintained?
- How are they promoted inside the organization or marketed to the outside world?
- What resources are available?
- Who should be on the team?
- How do we track success against the business goals that have been set?

While we will not be able to answer all these questions (indeed we cannot since they are different for every organization), hopefully the 7 best practices will help provide a framework for making these decisions.

Lastly, before jumping into content, a word on the notion of an API team and who should be on it.

THE API TEAM

An API team is really most akin to a "product" team—whether your customers are internal or external, you are in charge of building, deploying, operating, and optimizing the infrastructure others depend on.

Just like product teams, API teams can also be very diverse, but typically they should include a product-centric person who acts as the keeper of strategy and goals, design-focused team members who ensure best practice in API design, engineers who put in place API technology, and operations folks who will ultimately run the API. Over time you may also have others involved including support and community team members, API evangelists, security representatives, and others.

While this could be a large number of people, in smaller organizations some individuals might wear many hats. The important thing is to try to ensure that all the ultimate stakeholders' opinions are represented—even just by one of the team members checking in on their concerns.

In many cases API teams are formed temporarily and may belong to different organizational units with different line managers. This can make it particularly challenging to define a common vision for the API. For very large API programs different API teams may have to collaborate together.

No matter how large or small your organization, the 7 best practices that are described in this book will help establish a successful API team—and it may end up including more people than you think!

In many cases API teams are formed temporarily and may belong to different organizational units with different line managers. This can make it particularly challenging to define a common vision for the API.

BEST PRACTICE #1

FOCUS RELENTLESSLY ON THE VALUE OF THE API

*"Q: Frank, what does our API do?
A: JSON, I think."*

API programs often take on an inertia of their own, and with so many moving parts it's easy to get wrapped up in details and miss the most fundamental building block of a great API Program: the value being delivered.

In John Musser's five "keys" to a great API, providing a valuable service is the starting point for everything:

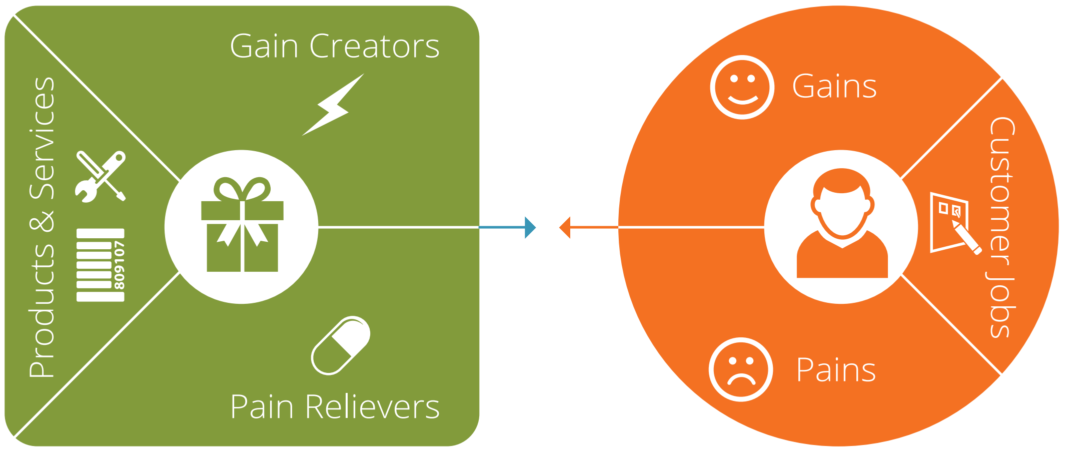
1. Provide a valuable service
2. Have a plan and a business model
3. Make it simple, flexible and easily adopted
4. It should be managed and measured
5. Provide great developer support

The first key, *provide a valuable service*, is especially important when thinking about the *"why."* In Building Great APIs: The Gold Standard (I) we discuss this in detail, the main takeaway being that it can be really challenging to find the right value proposition.

The value proposition is the main driver for success of the API. If an API has the wrong value proposition (or none at all) it will be very difficult or impossible to find users. The best marketing tactics won't work because there is no user group out there that finds the API valuable. Conversely you may find users with needs that could be met with a certain proposition, but if it's not one aligned with corporate objectives it will be difficult to sustain. If this is the case, in the long run the API program may lack the decision maker's buy-in, and financial commitment.

However, almost any company with an existing product, digital or physical, can generate value through an API, if that API links to existing offerings and enhances them. As long as the API is structured in such a way that it covers meaningful use cases for developers, it will deliver value.

One way to formalize your API value proposition is by using Alex Osterwalder's Value Proposition Canvas, which describes the benefits users can expect from your API. The right-hand side represents the user profile, which clarifies how you understand your users. The left-hand side is the value map, which describes how you intend to define your API to create value for the users. When the right and left sides meet, you achieve "fit," and the canvas can be used to define and refine this.



Source: Osterwalder et al. (2014)

In other words, fit is when users get excited about the value of your API, which happens when you solve important jobs, alleviate extreme pains, and create important gains that users care about.

What does this mean in API terms?

Finding and describing the value of your API is an iterative process. The first step is describing the jobs your users are trying to get done, e.g.:

- Automatically sending urgent communications to team members in emergency
- Backup critical files to ensure they are never lost
- Sample data to detect certain events

Next, identify particular pain points that affect users before, during, or after trying to get a job done:

- Ensuring reliability of sending with multiple tries, detecting failure, worrying about many messages being sent rather than just one, and integrating with different message delivery systems depending on the location of the user
- Ensuring safe delivery of the files but also wanting to minimize the amount of transfer bandwidth
- Dealing with massive amounts of data and attempting to correlate that in real time

The third step on the user profile side is to summarize the potential gains a user could achieve:

- Sending other types of notifications, which create opportunity rather than warn of threat
- Get rid of other storage equipment if reliability is good enough
- Automatically trigger actions based on the events

Switching to the value map side, the first step is to lay out the main functionality of your API in terms of features. Also, add non-functional aspects and additional services. Think rather broadly and list things like support, documentation, or developer portals, everything that a user could consume. Next, outline how you intend to eliminate or reduce some of the things that may be annoying to API users before, during, or after trying to complete a job, or issues that prevent them from doing so. Osterwalder in his model refers to these as pain relievers. Then describe how you intend to create gains of any sort for your API users.

Through engaging in this process, our three examples above might result in:

- A multi-channel messaging API with a single call to deliver messages and the ability to retry automatically until arrival is guaranteed (e.g., Twilio, PagerDuty)
- A storage synchronization API with optimized calls to efficiently check if new versions should be synchronized (e.g., Bitcasa, Box)
- An API aggregating several data sources into a configurable stream, which could be filtered, sampled, and easily manipulated (e.g., GNIP, DataSift)

Finally, a useful clarification exercise is to compose several statements that make the fit between the API and the user profile clear. If you find it hard to identify such fit statements, then the API model needs to be reconsidered. Maybe there are API features which need to be added, revised, refined, or eliminated. It could also be that your API does offer great value, but you are trying to address the wrong type of users.

When you then condense and abstract your fit statements into one overarching statement it becomes your value proposition for your APIs. In the case of the messaging API above this might be something like:

Our messaging API provides enterprise developers a reliable, guaranteed, no-latency text messaging functionality for highly-critical business applications. The API is also supported by SDKs covering the most popular programming languages for quick integration.

In some cases you may be thinking "this seems complete overkill—ours is just an internal API". This may be a natural reaction, but such a focus on value is key even in internal use cases. A poorly determined value proposition will lead to a lot of brown bag lunches to try to pitch the API to other teams. A well-defined one makes the API program a key contributor to the business.

When you then condense and abstract your fit statements into one overarching statement it becomes your value proposition for your APIs.

EXAMPLE: LINGO24[2]

Tech-savvy translation agency Lingo24 is an interesting case of APIs and strategy in action. Their translation APIs enable direct access to the Lingo24 translation platform, connecting two streams of translation services and providing a range of flexible solutions for high-quality "translation on tap" via two APIs:

- **The Business Document API** provides access to Lingo24's professional human translation services. These are a range of service levels from post-edited machine translation to creative copywriting in a foreign language, across all major document formats.
- **The Premium Machine Translation API** provides access to Lingo24's premium machine translation engines. Free for up to 100,000 words in pairs of English, French, and Spanish, with further paid plans that offer more words and access to more languages.

Why did Lingo24 want to open APIs via an API program?

Traditional translation solutions often couldn't address the range of customer content or rapid deadlines required, particularly for larger customers with diverse needs across business functions. Lingo24 wanted to open up their translation platform via an API program in order to provide easier, deeper integration with existing and new customers who value simplified and automated workflows. They also wanted to engage with channel partners, to grow a partner eco-system, and allow those partners to embed translation within their solutions as a value-add. This would enable the development community to build on their service.

2 This Lingo24 API example was contributed by David Meikle and Steve Griffin.

What did Lingo24 want to achieve with the API program?

In a market typically characterized by low-cost, variable quality commodity offerings, translation quality and client experience are core to Lingo24's business strategy. They wanted to be able to deliver highly-customizable solutions based on a customer's type of content or business priority. Their API program would enable them to integrate directly with customers to automate translation workflows, and work closely with them as partners—which is what their enterprise customers were really looking for.

Their API program served as an extension of their strategy by building on existing translation assets and technologies—such as their Premium Machine Translation engines that focus on specific business domains, and their Coach Computer Assisted-Translation tool—enabling easy access for customers to use these services.

How did Lingo24 design the API program to achieve this?

To design their API program, Lingo24 first looked at their key resources and services. They asked how those resources and services matched with the market by exploring various customer profiles, what those customer were looking for in a translation service, and how an API would fit with them. They used this information to develop a product vision and roadmap for their API offering. The roadmap was used, not only to frame and prioritize their development effort, but also to enable early conversations with prospective customers and partners.

While developing the roadmap, it became clear that they needed the two distinct offerings: a machine translation-based offering that would provide direct access to raw machine translation, and a professional human translation. This was necessary since the two services had different pricing structure needs, sales and marketing challenges, and scalability requirements. Although they split their offering across two APIs, they were conscious to develop a shared developer portal using the 3scale platform. This created one clear point of access to Lingo24 development resources, and a consistent way of interacting with users. The shared developer portal also provides a single integration point for both their sales and global support functions.

EXAMPLE: AMAZON WEB SERVICES[3]

by Kin Lane

The AWS API story is well known, where Amazon CEO Jeff Bezos issued his famous mandate in 2002. This mandate basically stated that all teams internally have to expose data and functionality via interfaces. Also, all these services must be externalizable. The rest is history. It worked so well, that Amazon at some point noticed that some of the services can also provide immense value to external developers and that these services could also be monetized. The external APIs to Amazon S3 and EC2 were born.

Developers using the Amazon S3 API were charged $0.15 per gigabyte per month for storing files in the cloud. Amazon EC2 users were charged varying rates for each small, large, or extra-large server they launched, paying only for every hour that the server was actually running. Amazon S3 combined with Amazon EC2 has provided a blueprint for the next generation of software engineering and business, where APIs are at the core. With this new type of API-driven business model Amazon is one of the key responsible players in the field of IaaS (Infrastructure-as-a-Service), which delivered huge value to developers and, in fact, changed software and business models.

Critical questions for consideration

To help define your own API Program's value consider these five questions:

1. Who is the user?

- This question should be answered in terms of their relationship to you (are they existing customers, partners, external developers), their role (are they data scientists, mobile developers, operations people) and their requirements or preferences.

3 Most of this example is taken from the 3scale white paper The Platform Vision of API Giants written by the API Evangelist Kin Lane.

2. What user pain points are we solving or what gains are we creating for the user?

    - This question should be answered in relationship to the customer's business, pains and gains from the value proposition canvas, and whether or not a critical need is being fulfilled (is it a pain point, is it a revenue opportunity), and what metric is being improved for the user (speed, revenue, cost saving, being able to do something new).

3. Which use cases are supported with your API?

    - Identify (e.g., with the help of the value proposition canvas) those pain relievers or gain creators that are most effective. Plan your API to address these use cases.

4. How can the value for the user be expanded over time?

    - Plan your value proposition with future changes in mind. What are important upcoming milestones relating to internal or external changes (such as trends or technological innovations).

5. What value is being created for your organization internally?

    - Consider internal benefits and how the API can be of value within the business, e.g., to other teams.

BEST PRACTICE #2

MAKE THE BUSINESS MODEL CLEAR FROM THE BEGINNING

"Don't invent a business model for your API. Create an API that supports your business model."

Being able to articulate the value of an API is already a great start on the API journey. However, APIs also generate cost, and this consideration should be balanced by value. While the value may not be measured in monetary terms, it must be real.

In fact—even for an API that delivers great value—if the business model does not add up, the end result may be a rapidly accelerating source of cost which, in a worst-case scenario, may ultimately have to be shut down or refocused.

In his crowd-created book Business Model Generation Alex Osterwalder defines the business model of an organization as "how the organization proposes, creates, delivers, and captures value." As such, a business model is much more than just "who pays." It involves a range of different aspects of an organization, such as what resources, activities, and partnerships are necessary for production and operation, and what is the required cost structure. For a business model to work, an adequate and addressable market needs to be available. The business model also describes served customer segments, the distribution channels to reach the customers, and the revenue model. The revenue model is part of the business model and describes how an organization monetizes its products or services.

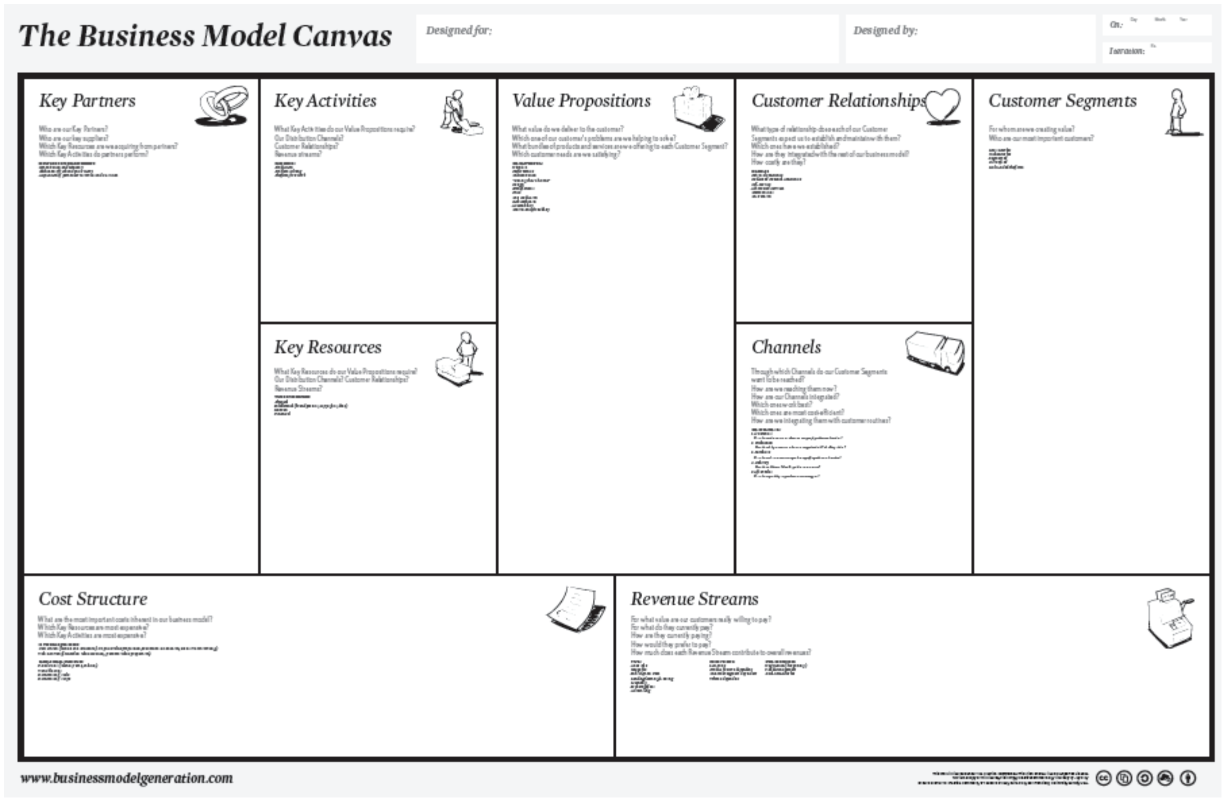
The question starts with what is the existing core business of the organization, and then extends to how an API can be used to accelerate or augment it. A great way to understand an organization's business model is to map it out via the Business Model Canvas.

The Business Model Canvas

Using the Business Model Canvas will help to analyze the core elements of the business model and their relationships in a structured way. The canvas takes into account:

1. Value proposition
2. Revenue streams
3. Cost structure
4. Customer segments
5. Customer relationships
6. Channels
7. Key partners
8. Key activities
9. Key resources

Being able to articulate the value of an API is already a great start on the API journey. However, APIs also generate cost, and this consideration should be balanced by value. While the value may not be measured in monetary terms, it must be real. In fact—even for an API that delivers great value—if the business model does not add up, the end result may be a rapidly accelerating source of cost which, in a worst-case scenario, may ultimately have to be shut down or refocused.



Source: Osterwalder et al. (2010)

In some cases APIs can lead to entirely new business opportunities outside of the existing business model of an organization—but even in that case, they generally leverage existing assets or expertise to do so in new ways.

In summary, these are the reasons why determining the right business model is important for effective APIs:

1. It brings the value of the API to the organization into focus, which drives the decision regarding long-term commitments to the API program. Without that commitment there are rarely the resources in place to complete many of the tasks required for establishing and running an effective API program.

2. It helps to define the functionality of the product, which is needed to satisfy third parties and actually drive the business model.

3. It ensures consideration regarding roles and responsibilities within an organization, and about who retains which parts of the value generated by the API. This also implies defining what users of the API gain and how that balances against what the API provider gains.

Without a clear business model (which should be considered internally along with ideally being communicated to customers and partners) the API risks being an appendage that doesn't deliver value and is unlikely to be sustained.

EXAMPLE: NETFLIX

Netflix is without a doubt one of the pioneers when it comes to APIs and API strategy. What is interesting is how Netflix used APIs to support their business model. When they first launched their API program, they decided to open the API to the public. They seemed to do everything right: provided a solid developer portal with sample code, documentation, forums, or showcases. They had people covering developer relations and engaged in events such as hackathons.

However, Netflix noticed that a public API program did not contribute to their business model in a significantly valuable way. Early in 2013, they decided to shut down the public API program. Their decision was to use the power of APIs primarily internally, and only open it to very few selected trusted partners. Netflix created a modern API and one of the most cited microservices architecture. This allowed the company to scale into the cloud, grow internationally, and now is able to serve over 1,000 different devices and operating systems.

The decision to close a public API program down and use APIs only internally is surely a difficult one. But for Netflix this was a necessary change to create better value for their business. Many may label Netflix's public API program a failure, however it's not: Netflix started as a David against Goliath. Having a public API program initially helped them to increase brand awareness, which clearly is one necessary driver for their current success. Also, other companies competing with Netflix started to adopt APIs, which is why APIs are now a very common practice in the media and entertainment sector.

What is interesting is how Netflix used APIs to support their business model. When they first launched their API program, they decided to open the API to the public.

EXAMPLE: SENZARI MUSICGRAPH[4]

The MusicGraph API is an adaptive recommendation and personalization engine delivered via a simple graph API that allows users to search through more than 7 billion music facts and connections.

Senzari offers significantly more cost-effective plans for companies of any size and budget to start building creative products and services powered by an intelligent, scalable, and context-aware music recommendation engine. They use a graph model to semantically connect the data (user and music data) and offer unparalleled flexibility and personalization capabilities for their customers.

The MusicGraph API offers:

- Context: MusicGraph integrates with a wide range of sources: social media, peer-to-peer, Wikipedia, Spotify, MusicBrainz, taking into account what is popular locally, etc. This provides rich contextual data for a significantly improved personalization and recommendation experience.

- Cost: Senzari offers the most cost-effective music recommendation and intelligence solution in the market today. Starting at $1,000 per month, the barrier to entry is much lower than the competition.

- Music Intelligence: Senzari offers the only music-specific analytics and intelligence (machine learning) platform, MusicGraph.AI, allowing customers to derive critical insight from the massive volumes of user and musical data stored in their semantic graph.

An API is at the core of Senzari's business vision and development efforts, using the company's core assets to offer key services such as Graph Search, Playlisting, Musical Data, and Social Signals.

4 This Senzari API example was contributed by Koen van Erp.

Critical questions for consideration

To align the business model for your organization's use of APIs, consider the following questions:

1. What value does the API create for the organization?

    - The value of an API may not only include monetary value. This question can be answered by thinking about how the API helps the organization, and could include analyzing how to increase reach, innovation, or leveraging network effects for content distribution.

2. How do we capture that value?

    - Once you understand what the value is, think about the best mechanism to capture the value and how to lower the barriers for doing so as much as possible.

3. What costs are occurring and how do we cover them?

    - Think about which costs are related to the API. These will most likely also lie outside of the API team, such as engineering or marketing efforts.

4. What resources need to be committed on a long-term basis?

    - You need to keep in mind that an API project is not a one-off investment. The API needs to be operated and maintained.

5. Which strategic partnerships are necessary?

    - When you develop your API and go to market you are surrounded by partners and suppliers. Try to find and leverage complementary offerings.

BEST PRACTICE #3

DESIGN AND IMPLEMENT WITH THE USER IN MIND

OH: Apologies—the calls aren't completely RESTful.

Good API design has some core principles, which may differ in implementation. The authors of APIs: A Strategy Guide have a great analogy:

Every car has a steering wheel, brake pedals, and an accelerator. You might find that hazard lights, the trunk release, or radio are slightly different, but it's rare that an experienced driver can't figure out how to drive a rental car.

This level of "ready to drive" design is what great API teams strive for—APIs which require little or no explanation to the experience practitioner when they encounter them.

Our design treatment here will be high level, but for more resources describing the functional elements and their technical implementation, see API Codex, RESTful Web APIs, or APIs: A Strategy Guide. The principles of good API design are closely aligned with John Musser's third key: "make it **simple**, **flexible** and **easily adopted**". We discuss this topic in depth in the API Gold Standard (II) post.

Simplicity

Simplicity of API design depends on the context. A particular design may be simple for one use case but very complex for another, so the granularity of API methods must be balanced. It can be useful to think about simplicity on several levels, including:

1. **Data Format**: Support of XML, JSON, proprietary formats, or a combination.

2. **Method Structure**: Methods can be very generic, returning a broad set of data, or very specific to allow for targeted requests. Methods are also usually called in a certain sequence to achieve certain use cases.

3. **Data Model**: The underlying data model can be very similar or very different to what is actually exposed via the API. This has an impact on usability, as well as maintainability.

4. **Authentication**: Different authentication mechanisms have different strengths and weaknesses. The most suitable one depends on the context.

5. **Usage Policies:** Rights and quotas for developers should be easy to understand and work with.

Flexibility

Making an API simple may conflict with making it flexible. An API created with only simplicity in mind runs the risk of becoming overly tailored, serving only very specific use cases, and not leaving enough space for others.

To establish flexibility, first find out what the potential space of operations is based on, including the underlying systems and data models, and defining what subset of these operations is feasible and valuable. In order to find the right balance between simplicity and flexibility:

1. Try to expose atomic operations. By combining atomic operations, the full space can be covered.

2. Identify the most common and valuable use cases. Then design a second layer of meta operations that combine several atomic operations to serve these use cases.

Arguably, the concept of HATEOAS can further improve flexibility because it allows runtime changes in the API and in client operations. HATEOAS does increase flexibility by making versioning and documentation easier, however, in API design, essential questions about the space of potential operations and combinations need to be answered just the same.

EXAMPLE: THE APIDAZE[5]

by Philippe Sultan

The APIdaze API exposes a real-time communications infrastructure to web developers. An audio/video/text conference bridge able to support many network users is accessible through a simple JavaScript API. Low level telecom functions like phone number management, phone provisioning, and account management are also manageable from a HTTP/REST API. And finally, a third programming interface lets developers control how phone and video calls get processed, by triggering webhooks built by developers using a simple HTTP/XML language.

The main idea behind APIdaze API is to make a set of hardware and software components achieving low level networking and telecom functions programmable for web developers. This implies picking the right interfaces that web developers will use, and therefore there is a need to make it as simple as possible to drive the underlying communications platform. Currently JavaScript, REST, and webhooks are familiar to anybody who gets involved in developing a web application.

JavaScript is obviously a must, as we can see it now on both the client (web browser) and the server side (Node.js, and its extensions like Meteor) of a web application. Having a client-side JavaScript API to control audio/video/text sessions from multiple browsers is a powerful and easy way to get developers involved, just like they would use jQuery to manage the DOM elements of a web page.

The HTTP/REST interface has a different goal, since it provides a set of synchronous and atomic actions to APIdaze's underlying infrastructure. Should a developer need to place a phone call between two people, manage SIP (Session Initiation Protocol) devices, or get phone numbers, this is the interface that they would use, which should be familiar to any web developer who knows HTTP/REST web services. This HTTP/REST interface is not HATEOAS based, and APIdaze has its documentation fixed in that regard. This is mostly because, for the sake of simplicity, it has not been considered in the original development of this part of the API.

5 This APIdaze API example was contributed by Philippe Sultan.

APIdaze also provides a way to relay events that happen within the infrastructure. If a phone call is coming in to a phone number owned by the developer, a program has to become aware of this event and take the appropriate action, like forwarding the call to a voicemail box. XML based HTTP webhooks come into play here. For an incoming phone call a URL is immediately fetched by APIdaze, which returns a set of instructions written by the developer to run on the platform in real time. Even though XML tends to be less popular than JSON (mostly because of the advent of JavaScript), any programming language can be used to yield XML text, or even no programming language at all.

APIdaze offers free access to its API, so developers can play, test it, and eventually get engaged to run their applications. Working on having great, clear documentation is also a day-to-day task for the technical team.

Critical questions for consideration

In order to think through your API design, consider the following five questions:

1. Have we designed the API to support our use cases?

- The next step after identifying the main use cases as we described in the earlier chapter is to design the API so that it supports these use cases. Flexibility is important so as not to exclude any use cases that may be less frequent, but should still be supported to allow for innovation.

2. Are we being RESTful for the sake of it?

- RESTful APIs are quite fashionable at the moment. However, you should not follow this trend just for the sake of that. There are use cases which are very well suited for it, but there are others which favor other architectural styles.

3. Did we just expose your data model without thinking about use cases?

- An API should be supported by a layer that abstracts from your actual data model. As a general rule, don't have an API that goes directly to your database—although there may be cases which require that.

4. Which geographic regions are most important and have we planned our data centers accordingly?

- API design must also cover non-functional elements such as latency and availability. Make sure to choose data centers that are geographically close to where you have most of your users.

5. Are we synchronizing the API design with our other products?

- If the API is not the sole product of your business make sure that the API design is coordinated with the design of the other products. It may well be that you decide to completely decouple API design from other products. However, even in that case this need to be made clear and communicated accordingly internally as well as externally.

BEST PRACTICE #4

PLACE API OPERATIONS AT THE TOP OF THE LIST

Q: Frank, How many developers are calling our APIs per day?
A: Not sure, but the last time we had more than 100 concurrent users it took the servers down...

Textbooks define operations management as "the activities, decisions and responsibilities of managing the production and delivery of products and services." In line with that, API operations is all about managing APIs once they are live to make sure that APIs are accessible and deliver according to developers' expectations. This boils down to two main functions:

1. Streamlining internal processes to be **efficient** to reduce cost.
2. Making operations **effective** in order to meet the expectations of developers' external to the program.

This notion ties in to John Musser's fourth key to a great API: It should be **managed** and **measured**.

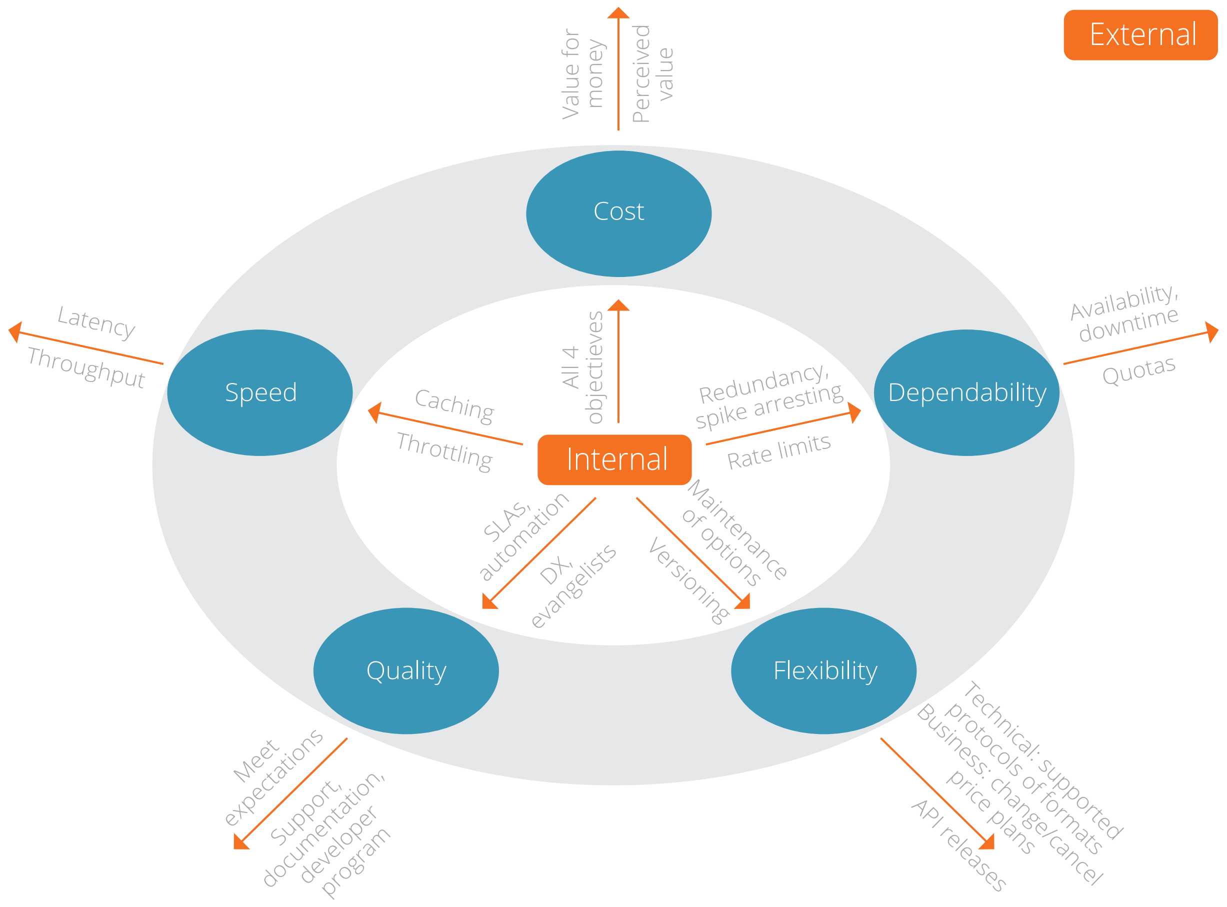
In the API Gold Standard III article we analyzed in detail what and how that can be achieved. We now cover an additional tool that should help getting API operations right: the API Operations Donut.

Streamlining internal processes to be efficient to reduce cost.

API Operations Donut

Operations management theory suggests five **key performance objectives**:

1. Dependability
2. Flexibility
3. Quality
4. Speed
5. Cost



Source: adapted from Slack: Operations Management 7th edition

The donut can be used to define operations tactics to achieve an organization's API strategy. The inner circle of the donut represents an organization's internal activities and effects; everything outside of the ring are external effects.

Dependability

Dependability is the actual availability of the API to developers. A useful metric is the downtime, which an organization can achieve through redundancy or spike arresting. Another metric is a quota (with rate limits as an internal control), which defines how many API calls can be made by a developer within a certain time frame. A quota protects an API and makes its management more predictable. Also, some API providers' business models (and price plans) are based on quotas.

Flexibility

Flexibility relates to the options developers have in adopting APIs. This could be manifested in technical options (see "Design and implement with the user in mind") or business options, e.g., the possibility or simplicity of changing between price plans or cancellation. The internal means is version control and versioning. It should be clear that, in general, the more flexibility provided the more effort (and cost) the organization needs to bear internally.

Quality

Quality is the consistent conformance to developers' expectation and influences their satisfaction. As such, quality is an overarching performance objective, which is related to the four other objectives. Conforming to expectations can be achieved by defining and meeting service-level agreements. Streamlined and purposeful automated processes can improve internal efficiency and contribute positively to quality.

Speed

Important aspects related to the speed objective in API operations are access latency and throughput. Both can internally be influenced by such techniques as throttling or caching. Throttling in particular (like quotas) can also be used for defining an API providers' business models.

Cost

The cost objective is to provide the best value-for-money relation to developers. Internally that means to optimize costs wherever possible without hampering the experience (i.e., perceived value and quality) of customers. Depending on context and implementation, all the other four performance objectives contribute to the cost objective either directly or indirectly.

At a minimum configuration we suggest to having at least the following means in place for your API management:

1. **Access Control**: authentication and authorization systems to identify the originator of incoming traffic and ensure only permitted access
2. **Rate Limits and Usage Policies**: usage quotas and restrictions on incoming traffic volumes or other metrics to keep traffic loads predictable
3. **Analytics**: data capture and analysis of traffic patterns to track how the API is being used

It's important that the API operations strategy fits into the overall API and business strategy. API management solution efforts and resources should be in line with the importance and scale of the API itself.

It should be noted that there are several vendors that provide technical infrastructure for many of these operations challenges—we at 3scale included. In many cases, using a vendor is a great and cost-effective way to address these problems, but this does not mean the strategy should not be thorough.

EXAMPLE: SLICE[6]

by Victor Osimitz

Slice has built a powerful data-extraction engine that connects to any email inbox, identifies the ecommerce receipts contained in that inbox, and extracts item-level purchase information from those receipts. This data-extraction engine has powered the Slice consumer apps (available from www.slice.com) for five years. In 2014, Slice officially launched the Slice API (at developer.slice.com), opening the same engine up to third-party developers building new experiences around their users' purchase data. In addition to supporting several large financial institutions, the Slice API has powered such diverse use cases as Gone!, a service that helps consumers sell their old stuff; IFTTT, a service that connects APIs together; TheFind, an aggregated ecommerce search engine; and many more.

6 This Slice API example was contributed by Victor Osimitz.

Slice's key performance objective in building the API was flexibility. Since the applications of this technology are so diverse, it was important to be able to support everybody from large banks, which have substantial development resources and long time horizons, to tiny startups and hackathon projects, which are quick and nimble but strapped for time and resources.

Slice found that their development partners were divided into two camps: some that wanted complete control over their user experience and were willing to invest the time to do a full white-label of the Slice platform, and others that wanted a quick integration and were comfortable using OAuth to "link an existing Slice account." Initially, Slice expected to have to pick one integration method to support at the expense of the other, but they realized that the two were almost the same except for the authorization method that they would use. In fact, the API requirements for both groups of developers were almost exactly the same: both simply needed a way to retrieve orders, purchased items, and shipment information for specified users that had authorized Slice to share their data.

Ultimately, Slice decided to support two types of authorization: vanilla OAuth 2.0, and a signature-based method for power white-label integrations. Since this decision added significant complexity to the API, Slice implemented its developer portal in such a way that most developers would only be aware of the OAuth integration method. Furthermore, Slice's API team made an extra push on simplicity elsewhere, described by a product manager as "scalpel-driven design," because this first step was to delete 75% of the fields in the original API spec. This ensured that—for the majority of smaller developers who were interested in an OAuth integration—the API would be simple and straightforward, while also maintaining flexibility to support larger partners who were willing to make the investment for a white-label integration.

Critical questions for consideration

To work through your API operations plans, consider the following questions:

1. How do we control access?

- Access control is one of the most important elements of API operations, including knowing who can access your API, who does what and when, along with ultimately being able to enforce limits around this.

2. How do we capture metrics and handle alerts?

- Next to access control, getting as much visibility about what's happening with your API during operations is key for API success. This is where analytics come into play. Based on your objectives and use cases you will have different metrics and it's important to measure them accordingly, and to have an alert system in place.

3. How should spikes be managed?

- Access control and usage policies will help you to plan your infrastructure. Spikes will happen for different kinds of reasons, and we recommend having fallback mechanisms in place like spike arresting or automatic throttling.

4. Who is responsible for API uptime?

- API uptime is one of the most important metrics. An available API is what generates value according to your value proposition. No API, no value generated and captured. It needs to be clear who is responsible and what needs to be done in case of a failure.

5. How do you deal with undesired API usage?

- In general, there are two types of undesired API usage: expected and unexpected. Expected is what you can plan for via a good API operations approach in place. The unexpected is a lot more difficult and can mostly be handled by Terms and Conditions or similar regulations.

BEST PRACTICE #5

OBSESS ABOUT DEVELOPER EXPERIENCE

"Getting information off the Internet is like taking a drink from a fire hydrant." (*Mitchell Kapor*)

While developer experience (DX) may sound like it's about API Design, it goes much further—think of it as the packaging and delivery of the API, rather than the API itself. You can have a wonderfully designed REST API but if it's hard to sign-up for and test, you've created an awful developer experience.

Having a great API that's designed with simplicity and flexibility is wasted if developers do not engage with the API and eventually adopt it. At the same time, a well thought out API design has a considerable impact on developer experience and adoption. Adoption is an essential part of the developer experience (DX).

John Musser provides a great take on what it means to get developer engagement in his OSCON 2012 talk:

- Making it very clear **what the API does**
- Providing **instant signup**
- Providing **free** access
- Being **transparent about pricing**
- Having **great documentation**

A key metric to improve API design for easy adoption is the **Time To First Hello World (TTFHW)**. This is great way to put yourself in the shoes of a developer who wants to use your API to see what it takes to get something working.

When you define the start and end of the TTFHW metric, we recommend covering as many aspects of the developer engagement process as possible. Then optimize it to be as quick and convenient as possible. Being able to go through the process quickly also builds developer confidence that the API is well organized, and things are likely to work as expected. Delaying the "success moment" too long risks losing developers.

For poster child examples of good developer experience and quick TTFHW, check out the Twilio, SendGrid, or Context.IO developer portals. Twilio's Founder and CEO Jeff Lawson has a great take on this from a wider business perspective.

In addition to TTFHW, we recommend another metric: TTFPA – *Time To First Profitable App*. This is trickier, because "profitable" is a matter of definition, depending on your API and business strategy. Considering this is helpful because it forces you think about aspects related to API operations as part of the API program.

TTFHW should be the main driver when building DX into your API product. There are several means to achieve that, all of which are summarized in a developer program. This element of an API program correlates to John Musser's fifth and last key for a great API: "provide great **developer support.**"

Crafting a Developer Program

The aim of an effective developer program is to provide outstanding **developer experience** (DX). Pamela Fox puts it this way:

Developer experience (DX) is the sum of all interactions and events, both positive and negative, between a developer and a library, tool, or API.

The two underlying principles of developer experience are:

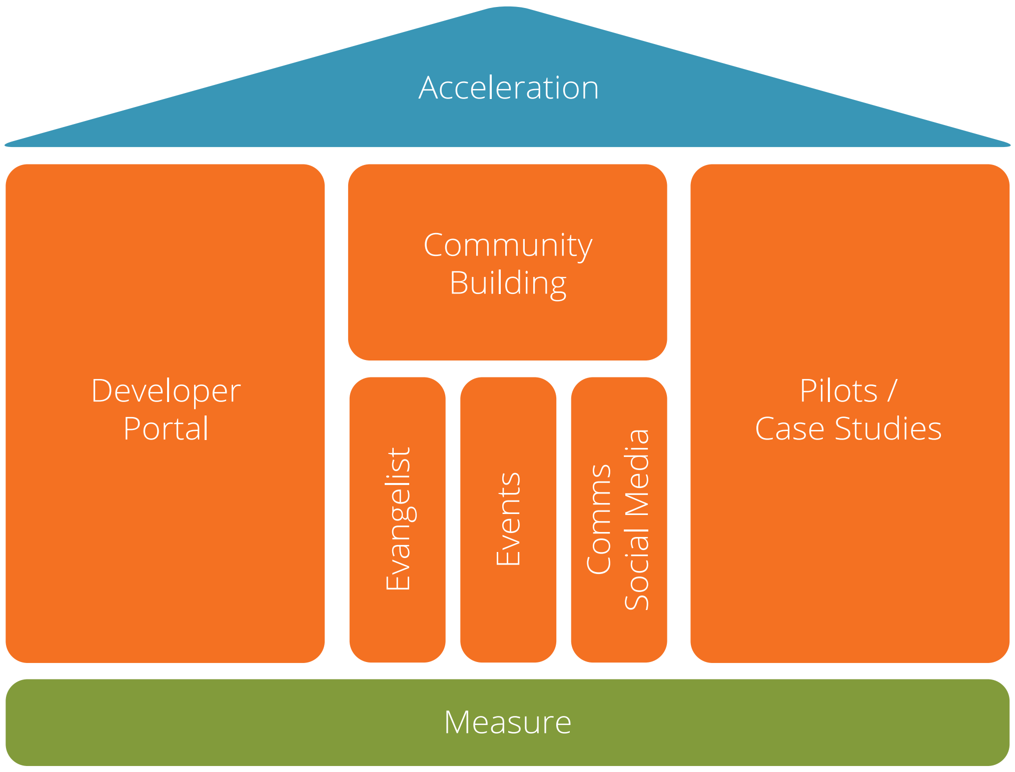
1. Design a product or service that provides a **clear value** to developers and addresses a clear pain or gain. This can be monetary value along with other value, such as a way to increase reach, brand awareness, customer base, indirect sales, reputation for the developer, or the pure joy of using great technology that works.
2. The product needs to be **easily accessible**. This can include having a very lightweight registration mechanism (or none at all), access to testing features, great documentation, and a lot of free and tidy source code.

Hopefully the first point flows easily from some of the thinking in best practices #1 and #2.

Building up on the core principles of developer experience, we suggest that most API programs should have a developer program—regardless if you expose your APIs publicly, to partners only, or internally only. A developer program should in general cover the following elements in a certain shape or form:

1. Developer portal
2. Community building
3. Evangelists
4. Events
5. Communications and social media
6. Pilot partners and case studies
7. Acceleration via ecosystem partners
8. Measuring

The provisions may be more or less elaborate depending on the audience.



Developer Portal

The developer portal is the key element of a developer program; this is the core entry point for developers to signup, access, and use your APIs. Getting access to your API should be dead-simple and frictionless for developers, who should be able to get started quickly. TTFHW is the best metric to measure this. You should also consider streamlining the sign-up process—the simpler and quicker, the better. A recommended best practice is that developers should be able to invoke your APIs to examine their behavior (request and response) without any sign-up at all. Interactive API documentation based on industry standards like Swagger are a great way to achieve that. Also, supplementary content such as getting started guides, API reference documentation, or source code are great to lessen the learning curve.

> The developer portal is the key element of a developer program; this is the core entry point for developers to signup, access, and use your APIs.

Community building

Community building broadly has two aspects: a physical and a virtual presence. Where and how you appear in both aspects depends on which developer personas you want to address. Physical presence refers to events, which are a great opportunity to get the word out about your API program in order to promote its benefits, and to increase adoption. It's also valuable to get in touch with your community and meet the developers face-to-face, which often leads to very useful insights and can influence future design and implementation of the API or the API program. These types of events include large developer conferences, developer days, barcamps, hackathons, workshops, or trainings.

Developer evangelists

The role of a developer evangelist is fairly new. There are several resources on the Web which describe this role in more detail. Next to the developer portal, an evangelist is another key element of a successful developer program. The Onion model of developer evangelists summarizes some of the most important types of activities of an evangelist. This model is also scalable. It can be applied to a one-man-band evangelist, as well as to a team of evangelists.

Pilot partners and case studies

Engaging with pilot partners can be very effective. These are usually early adopters who you work closely with you to adopt your APIs. Getting engaged with pilot partners has two main advantages. First, you get early and useful feedback about your technology being deployed in a real-life setting, and it can be improved accordingly. Second, if the pilot works out nicely, it can be used as a successful case study example for general promotion and awareness activities. It also shows that your technology works and what's possible with it.

Acceleration via ecosystem partners

As an API provider you are operating in an ecosystem of partner and vendors. These partners often have their own content distribution and communication networks and means.

We recommend identifying alliances, which can be effective in helping to increase the adoption of your API. Often such alliances can be found when APIs are complementary and provide value to developers when combined.

Measuring

Only what's measured can be managed. Measuring is an important element of the developer program in terms of understanding its effectiveness and to find out which aspects should be improved. The developer program contributes to the objectives of the API program and business strategy, and should be the starting point for deriving metrics. Some examples for typical metrics for the developer portal are page visits, signups, API traffic, or support requests. Events can be measured by the number of attendees, API adoption at hackathons, or leads. It's advisable to create correlations, such as "did a talk at an event trigger more API signups?" Swift has an insightful presentation about the Nuts and Bolts of Developer events.

EXAMPLE: CONTEXT.IO[7]

by Tony Blank

At Context.IO you often hear the term "developer experience" in conversations on a daily basis, which is part of every decision made. Context.IO is constantly thinking of ways to improve the experience for customers—and it can always be better.

The biggest hurdle for the company has been at the very beginning; getting developers to understand the value and wide range of use cases for Context's API. Context.IO takes the complexity out of connecting to email servers and enables developers to easily build apps on top of their user's email data. If a developer has ever tried to work with IMAP or Exchange, they understand the value almost immediately. It may take a bit longer for others, in which case Context.IO usually runs through a handful of app examples until the "lightbulb moment" appears.

Another early step is prompting the developer to connect an email account they control via Context.IO's interactive console, which is how "TTFHW" is reduced to seconds. The client can make requests against their own email account and see responses without setting up a dev environment or writing any code. The console on the Context.IO website is a great way for developers to rapidly consider options, and learn what Context.IO can do. Client libraries are easily available on Github when it's time to dive into writing code.

7 This Context.io API example was contributed by Tony Blank.

After a developer has had a chance to try out the API, someone from the Context.IO Evangelism team follows up with each and every one of the new users to make sure they don't have any outstanding questions or problems. While this "high touch" approach has scaling issues and may not be practical depending on many companies' new customer volume, Context.IO has gotten an incredible amount of value out of these efforts. New API users have the best feedback on how to improve onboarding. Not only do those early conversations help improve resources, but a conversation can smooth over any issues early users may experience.

As developers become more familiar with the API, an often-seen occurrence is for them to think up new and unique ways to use Context.IO. Email as a data source has limitless use cases, so it's useful when something new is revealed. Documentation is usually complete enough for developers to get started, but if something is missing or Context.IO doesn't have the answer, the company works collaboratively with the developer to figure it out.

Overall, it's fair to say Context.IO has a pretty "high touch" approach to developer experience. It's absolutely more work, time, and money than other approaches but, at the end of the day, the great results and the information collected is invaluable.

EXAMPLE: SENDGRID

SendGrid—the company that sends and manages emails via APIs—is another great example of DX done right. The SendGrid developer portal incorporates most of the best practices we summarize in this book and we recommend checking it out. Right from the start SendGrid understood the power of communities and one of the key objectives was to build a thriving one around the SendGrid APIs. This article about SendGrid describes their four layers of focus: developer education, startup outreach, DX, and events. It also describes the seven metrics they use to determine how effective their activities are. SendGrid evangelist Martyn Davies presented another interesting metric, "beer driven adoption," at the API Strategy and Practice conference. Finally, developer evangelists are crucial for the SendGrid API success. Here is an experience report about how SendGrid hires.

Critical questions for consideration

Questions to consider to assess your developer experience:

1. How do we explain the value of the API in the first five minutes?

    - Develop an "elevator pitch" about the value proposition of your API that best speaks to developers.

2. What is our TTFHW and TTFPA and how do we reduce it?

    - This is a powerful way to improve the developer friendliness of your API by thinking about the end-to-end TTFHW. We recommend keeping TTFHW/TTFPA in mind with all elements that are added to the DX (like portals), and every aspect that changes.

3. What is the onboarding process for developers, and is it as painless as possible?

    - This needs to be in line with the use cases of your API, and needs to be appropriate. The level of security naturally needs to be higher for more sensitive APIs or data access, which probably needs more formal agreements. For everything else it should very simple and straightforward to allow for early developer success (TTFHW).

4. Are we leaving enough on the table to make the API attractive for developers?

    - It's great if you've found the right value proposition, and developers sign up for your API. Keep in mind that helping them to be successful will retain and grow their numbers.

5. How do we support developers if they face problems?

    - In general we believe in the self-service approach, which will help you to scale. Many developer questions can be covered by good documentation, FAQs, or forums. But self-service has its limits, and for more in-depth questions or, e.g., invoice problems, there should be some type of support mechanism in place.

BONUS: What support is there for developers who go rogue outside of the normal use cases to do something new—how good is our documentation?

BEST PRACTICE #6

GO BEYOND MARKETING 101

"Throw a couple of hackathons, add beer and pizza, we're done."

Not quite. The marketing of APIs is often seen as a little unsavory (developers often reject the idea they should be marketed to) or simplified to a notion that hackathons are the key way to attract users to the API. While there are some techniques specific to APIs (like hackathons), in truth APIs needs to be marketed just as any other product does.

Marketing is about bringing the right product to the right customer in the right way. The same is true for marketing the API program. Marketing depends on the product. After the value of the API is defined, it's the responsibility of the marketing program to establish it in the market, and get it to developers. A great framework to achieve this is Segmentation, Targeting, and Positioning (STP). We use this framework and explain how each stage can be applied to the marketing of an API program.

Segmentation

Customers of an API provider are people or companies that develop software, regardless of the scope of an API. There are various audiences that may have to be taken into account as different segments for API marketing:

1. Completely internal usage
2. Exposing the API to close partners or suppliers
3. End users of the apps or services resulting from API integrations
4. External companies or developers

Marketing an API program is mostly about developer marketing—also often referred to as business-to-developers (B2D). Estimations by the WIP Factory claim that there are some 43 to 50 million developers globally, which is the total addressable market. Clearly that is too large and diverse to appeal to and engage with a compelling product offer. The total market needs to be divided into smaller developer segments, with particular characteristics and behavior in mind.

A great way to establish developer segments can be done via the jobs-to-be-done method. VisionMobile applied this approach to mobile developers and came up with a classification model of 8 groups. This could be used as part of the segmentation exercise for your API marketing strategy.

To achieve the segments specific to your organization the WIP Factory developer segmentation methodology could be used. This method applies filters to the total market, which reveals the varying segments. These filters are broken into four imperatives:

1. Technical: this relates to platforms, operating systems, programming languages, or tools.
2. Individual: this relates to skills and the experience or persona of developers (which is similar to the psychographic dimension as mentioned above).
3. Business: this relates to the types of companies or organizations, their market position, supply chain position, or financial strength.
4. Market: this relates to secondary markets. That means these are markets dependent on the prime market, such as suppliers, buyers, or other depending verticals.

After applying each iteration of a filter check whether the resulting segments are relevant, large enough, or valuable to your API program, and whether there're the necessary resources to address these segments.

At the end of this exercise you will have identified various segments. The next step is to choose the most relevant and valuable one(s), which is referred to as targeting.

Targeting

In marketing, targeting is the process of evaluating each market segment's attractiveness and selecting one or more segments to enter. To select the most important developer segments, take into account the following characteristics:

- The selected segment is accessible. The segments is only of value if it can be realistically reached. If you, for instance, don't support a particular programming language or architecture stack or if you cannot reach a certain developer community geographically then addressing this developer segment will be very difficult.

- The selected segment is substantial. There should be a critical mass of developers in your segments. Also check if the community is active and growing. You may not want to choose a segment which is disappearing.

- The selected segment is differentiable. The various segments you choose should be sufficiently different. In that case a differentiated set of tactics that addresses them will make sense and become effective.

If your selected segments pass these tests then proceed to the next step, which is to define tactics to address these target groups of developers, in other words, positioning.

Positioning

Marketing textbooks define positioning as "arranging for a product to occupy a clear, distinctive, and desirable place relative to competing products in the minds of the customer." By now you will have a good understanding of the profile of your selected developer segments. Positioning actually means applying your marketing tactics and capturing the developer segments you decided to address. The best marketing is targeted and presents a product—an API in our case—that solves important jobs, alleviates extreme pains, and creates important gains that users care about, as we described in the Value Proposition Canvas above.

If you target several developer segments, then your positioning will differ. Make sure that you understand the pains and gains of the selected target segments, and specify your positioning tactics accordingly.

Developers are people too and they have a very high bullshit detector. That's why it's very important to use tactics that are specific to developers. Getting the developer experience (DX) right as described in the previous section is a crucial element. Some of the most effective tactics are:

- Working with developer evangelists
- Providing outstanding developer portals
- Participating in and supporting developer events
- Providing support and lightweight processes (e.g., registration)

VARIOUS EXAMPLES: TWILIO, BRAINTREE, AND PINGAR

twilio
Braintree
pingar™

An often-cited example related to successful API marketing is Twilio, which provides an API to enable communication via voice and messaging, like SMS. They've placed offices physically in markets that are important to them: US (San Francisco, Mountain View, New York), Europe (London, Munich, Tallinn), and Bogotá in Colombia. Their API documentation, tutorials, and processes are very clear, simple to use, and quick. Although Twilio provides also a plain Web API, they know that their most active developer communities are around programming languages such as PHP, Ruby, Python, C#, and Node.js. That's why Twilio provides helper libraries for exactly these languages. Finally Twilio is also known for their active event engagement with developer evangelists. Ricky Robinett is a developer evangelist at Twilio and gave a great talk about "Being Alfred: Serving developer communities and making heroes." Other programs Twilio ran to attract developers includes the Twilio Heroes and the Hacker Olympics.

Developers are people too and they have a very high bullshit detector. That's why it's very important to use tactics that are specific to developers.

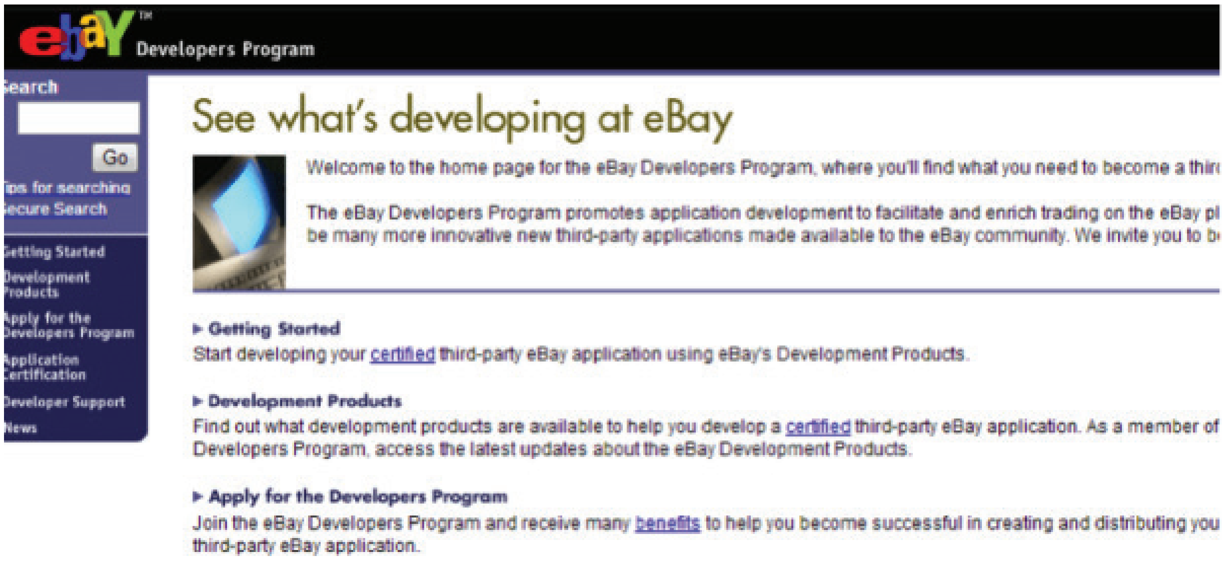
Another interesting example of a company which targets different segments with different API programs is PayPal. PayPal provides and operates two payment APIs as products: the PayPal API and the Braintree API. Braintree has gotten a lot of positive developer mindshare especially among "long-tail" developers via a public API. The PayPal API remains the first choice for close enterprise customers and is accessible via an API partner programs. Tactics for targeting developers in these two different segments are completely different. For the PayPal API it's in many cases 1:1 marketing and sales engagements, for the Braintree API it's a lot more 1:n, and similar to what Twilio is doing, like working with developer evangelists. It seems that in the particular case of Braintree most developers use iOS, JavaScript, and Android on the client side, and on the server side Ruby, Python, PHP, Node.js, Java, and .NET, which is why SDKs are provided accordingly.

An example of a company that exposes APIs only to a closed set of partners is Pingar. Pingar offers an enterprise search solution which, based on document analysis algorithms, provides new value from masses of unstructured data. The partner program is closed. Pingar only executes very targeted marketing addressing key verticals such as legal, financial services, HR, pharmaceuticals, and government. With their API program, Pingar grew its business development and developer channel to 80 mostly enterprise partners and customers in three months. Using the Pingar API, Fuji Xerox has solved the issue of producing metadata for storing documents while scanning them from paper, greatly reducing the time it takes to digitize an office.

EXAMPLE: EBAY

Without a doubt the eBay API is one of the longest-standing API programs, and often cited as an example of an API success story. And for eBay, exposing APIs to their platform to allow developers executing eCommerce functions was definitely was a huge success. 60% of eBay revenue is generated via 3rd party applications using the eBay API.

eBay was also one of the first API providers who planned and executed a dedicated developer marketing program. eBay's developer segments and targets after the API launch in 2000 evolved and changed over time. It first started aimed at only at few selected partners, who had to obtain a special license. eBay soon noticed that in order to achieve the desired reach they needed to open up access, which also required a stronger self-service model of their developer program. The result was the eBay developer portal.



See what's developing at eBay

Welcome to the home page for the eBay Developers Program, where you'll find what you need to become a thir

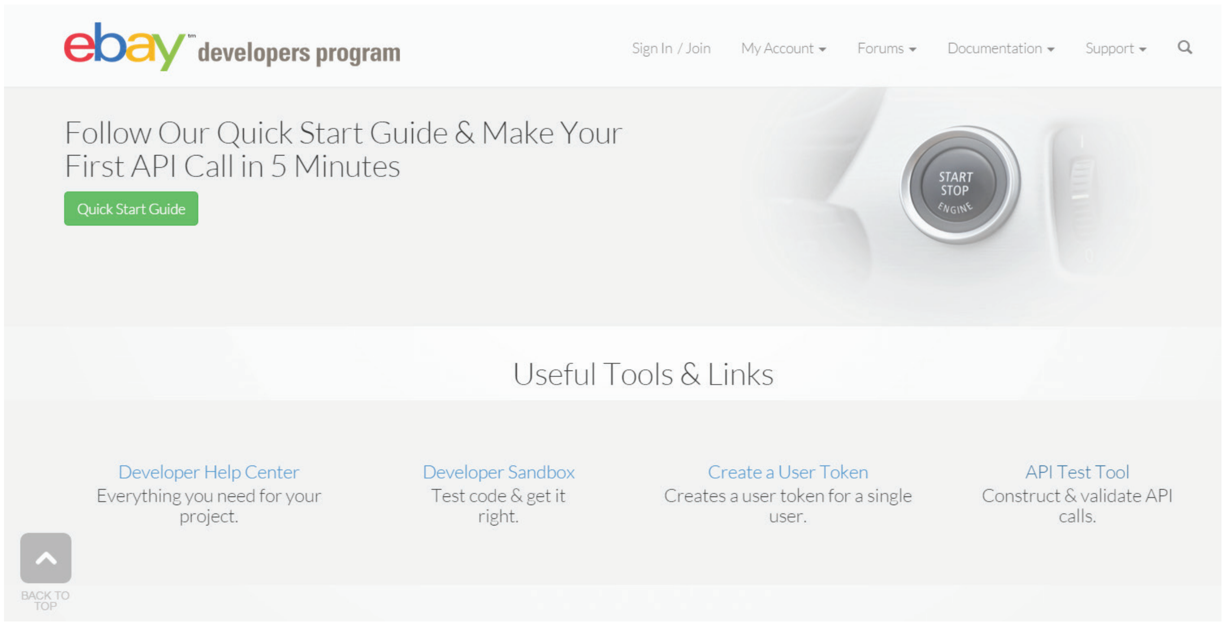
The eBay Developers Program promotes application development to facilitate and enrich trading on the eBay pl
be many more innovative new third-party applications made available to the eBay community. We invite you to b

▸ **Getting Started**
Start developing your certified third-party eBay application using eBay's Development Products.

▸ **Development Products**
Find out what development products are available to help you develop a certified third-party eBay application. As a member of
Developers Program, access the latest updates about the eBay Development Products.

▸ **Apply for the Developers Program**
Join the eBay Developers Program and receive many benefits to help you become successful in creating and distributing you
third-party eBay application.

eBay Developer Portal, around 2001 (Source)



Follow Our Quick Start Guide & Make Your First API Call in 5 Minutes

Useful Tools & Links

Developer Help Center	Developer Sandbox	Create a User Token	API Test Tool
Everything you need for your project.	Test code & get it right.	Creates a user token for a single user.	Construct & validate API calls.

eBay Developer Portal, now

Critical questions for consideration

Consider these questions when planning your API marketing activities:

1. What type of audience are we trying to reach with the API: Internal Users? Close Partners? Existing customers? The outside world?

   - The answer to this question is critical and may not be obvious at first—your API may go through several stages of evolution—ensure you're focused on the current key user set first.

2. If we decide to work with evangelists, what type of evangelists are most appropriate to support the value proposition of your API?

   - Experts who can promote your API internally or externally using the right language, i.e., evangelists, are always recommended. The role of an evangelist is very diverse and expertise in various areas may be needed (engineering, support, sales, product management). It's important to identify what expertise is the most important for your API.

3. Which events are most appropriate for communicating our message?

   - There are a wide variety of events that could be relevant for API providers. Horizontal events vs. vertical/industry events, global vs. local, conference vs. hackathon, formal vs. informal. Depending on your API and what you want to achieve will determine why and how you want to get involved in specific events.

4. Are we sure a hackathon is the right event for the API?

   - Hackathons are very en vogue nowadays. Whenever someone has a software product that should be promoted the answer is often to do a hackathon, which can be very effective. However, before running a hackathon it's important to be clear about what you want to achieve (developer portal signups, SDK downloads, new apps, recruiting, brand recognition), and then plan and execute the event accordingly.

5. Should there be an Internal Marketing plan?

- For outward-facing APIs the audience is often defined by the customer set. However, don't necessarily assume there shouldn't be internal marketing: other business units may need to buy in, the marketing department may need in-depth explanations, and product teams must understand how the API benefits existing customers, etc.

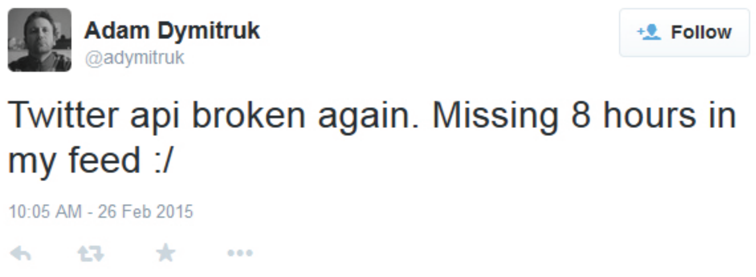
BEST PRACTICE #7

REMEMBER API RETIREMENT AND CHANGE MANAGEMENT

"An API is for life, not just for Christmas."

API advice tends to focus heavily on API design, creation, and operation. However, one of the most critical segments of the API journey is what happens many months after launch and operation—managing updates to the API, and even retirement.

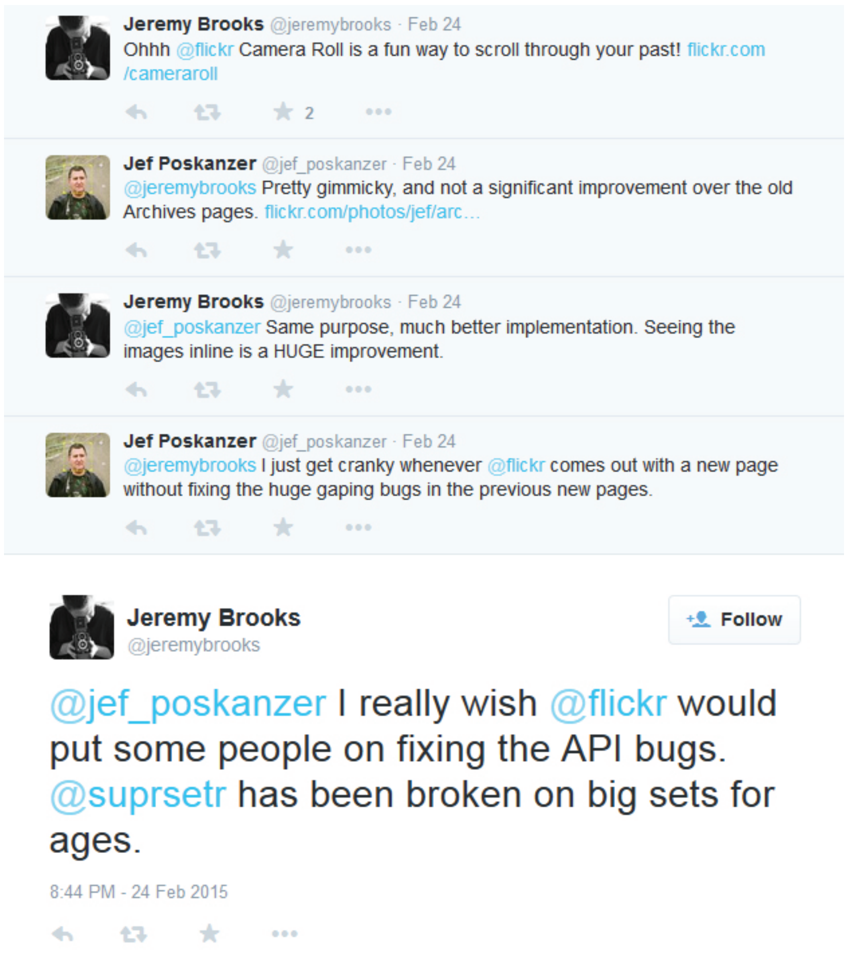
APIs are integration points for software—software that's often written specifically to work with a particular API, and hard wired to operate in a certain way. While there are some techniques (such as hypermedia) which can help loosen this dependency, some dependency will always remain—if an API goes away or changes radically the dependent software will no longer work. Such a breakdown without warning to users will destroy confidence in the API rapidly, and potentially even create legal issues. At the very least it causes a ripple of urgent work for users/customers, and at worst will cause them to go elsewhere.


Adam Dymitruk
@adymitruk

Twitter api broken again. Missing 8 hours in my feed :/

10:05 AM - 26 Feb 2015

Source: Twitter


Jeremy Brooks @jeremybrooks · Feb 24
Ohhh @flickr Camera Roll is a fun way to scroll through your past! flickr.com/cameraroll

Jef Poskanzer @jef_poskanzer · Feb 24
@jeremybrooks Pretty gimmicky, and not a significant improvement over the old Archives pages. flickr.com/photos/jef/arc…

Jeremy Brooks @jeremybrooks · Feb 24
@jef_poskanzer Same purpose, much better implementation. Seeing the images inline is a HUGE improvement.

Jef Poskanzer @jef_poskanzer · Feb 24
@jeremybrooks I just get cranky whenever @flickr comes out with a new page without fixing the huge gaping bugs in the previous new pages.

Jeremy Brooks
@jeremybrooks

@jef_poskanzer I really wish @flickr would put some people on fixing the API bugs. @suprsetr has been broken on big sets for ages.

8:44 PM - 24 Feb 2015

Source: [Twitter]

The cost of such changes climbs rapidly in cases where:

- The developers/apps using the API are unknown (e.g., there are no keys, IDs, or means to contact individuals maintaining client code).
- Apps are deployed on mobile devices which require vendor approval and customer assent to update code.
- Apps are deployed on physical/hardware devices with little or no update capability or UI.

Each of these makes client code updates extremely painful.

However, the reality is that sometimes APIs need to change—new features are needed, and old ones can no longer be supported. Sometimes entire APIs may need to be retired.

Breaking vs. non-breaking changes

When initiating updates to an API it's important to determine what type of change is being made:

1. **New methods**: new methods are being added to the API, but the existing methods are unchanged.
2. **Augmentation of existing methods**: these are changes to existing methods but they are additive—adding new data to existing return types, or allowing additional parameters which modify the return type.
3. **Removal of methods**: some of the existing methods will no longer operate in the new version.
4. **Modifications of existing methods that change current behavior**: changes to existing methods which remove data or options, change payloads, or otherwise change the way things work.

Different organizations use different definitions of types of changes, but typically changes of type 1 & 2 are considered non-breaking changes—in other words, clients using the old API version should in general still function despite the changes (see below however). Changes of type 3 and 4 are breaking changes—in other words, applications using affected methods will clearly no longer function under the new version.

Knowing which type of change is being made is critical. As a best practice we strongly recommend that breaking changes are accompanied by a new major version number change (e.g., v1 to v2) and a migration plan between versions. Whereas non-breaking changes can be addressed by a minor version increment and no migration plan.

A migration plan is a rollout of the new API, which allows for an adjustment period with the old API. In other words:

- The new version is made available for testing and use.
- A notice is released as early as possible warning current API users of a limited lifetime of the old version (if appropriate).
- After assisting users with the transition the old version is retired.

When non-breaking changes break

Unfortunately, changes that often appear harmless and non-breaking ultimately end up breaking applications. This can occur when:

- Developers make unwarranted assumptions on API call returns, e.g., the order of elements in a JSON/XML payload.
- Unexpected additional returned data overloads an application.
- A parameter previously passed by accident by some applications is suddenly used in a new version of the API with a different meaning.
- A format, data type, header or other change which seems innocuous (e.g., shift to HTTPS... you are using HTTPS right?) affects some clients.

It can be extremely difficult to be certain changes won't break some apps. For these reasons we strongly recommend that absolutely any changes to the API, even if categorized as "non-breaking," should be rolled out by: 1) providing a test endpoint with the new version prior to launch and, 2) sending an email or other communication to developers informing them of the change and giving timing/details.

Communication and the contract

Things go wrong — this is quite normal. The most important thing if this happens is to communicate what went wrong and not to leave developers in the dark. Tell them what went wrong, why, and (most importantly) what to do about it. As a preemptive measure, terms of service are critical. These should always include how long you plan to support each version. Of course, try to stick to those commitments, and if that's not possible, communicate it accordingly. Informing developers about changes—positives and negatives—ahead of time builds confidence. It also provides a reliable framework for decisions. One effective way of keeping developers informed about changes is by using automatic tools such as API Changelog.

The end of the line

One day the time may come that an entire API may need to be retired. In many ways this is just a more complex version of a breaking change. However, you may additionally want to consider:

- Even longer lead time retirement notice.
- Considering whether there will be press impact.
- Providing a migration plan for your API users.
- Providing export / extract tools in those cases that the API was a key interface to customer data.

EXAMPLE: STRIPE[8]

by Amber Feng

One interesting implementation of API change management is the Stripe API. Stripe is an online payments platform and, as such, has fairly stringent requirements for API stability—broken integrations are measured in literal dollars lost, not just frustrated developers.

The approach they've taken with their API is to handle change management invisibly. The contract Stripe has with their users is simple: once someone starts using the API they'll never have to worry about their integration being broken and will rarely need to care about what version they're on. Under the hood, Stripe has a separate (dated) API version for each breaking change. Whenever a user makes their first API request, Stripe records what the current API version was and going forward returns requests to that user to the current version.

Of course, you can still upgrade your API version or pass a version override via API headers. However, the majority of users don't really care about doing this and therefore probably won't. By default your API should just **work** while requiring as little from users as possible.

(For more detail how Stripe's API is designed and implemented, they've written several blog posts about it.)

8 This Stripe API example was contributed by Amber Feng.

Critical questions for consideration

Are you ready for the API long haul? Consider the following questions:

1. What is our customer guarantee and how do we ensure commitment?

- This is arguably the most critical question for your API program—what level of service stability are you willing to commit to for your users? This is not only important for potential users who are considering buying into your API, but also for you in terms of fixing change policies.

2. What is our change and breaking change process?

- This is derived from the answer to question 1. Given this user guarantee, what is the release process that ensures this commitment is not violated? Who is involved? What approvals are needed for change?

3. Do we detect, document, and communicate changes to developers?

- Are you using an API definition format such as Swagger or Blueprint? Is the current model checked against the previous version? How is change communicated?

4. How do we detect if and how many user still use older versions of the API, and how do we support retired versions?

- Do you have developer and user IDs to determine version usage among clients? Without a clear detection and retirement support process there may be unfortunate consequences.

5. How do we align API evolution with the evolution of related products?

- Given the guarantee to customers for API stability, how aware is the product organization of these commitments, and what happens if product needs require API change?

BOOSTING YOUR API STRATEGY

The best practices in this book will help define and implement an effective strategy for your API. They also will help evolve your API strategy and spot new opportunities. It's likely many of the questions in the previous sections need to be answered in great detail, and new opportunities or risks may arise over time. For example:

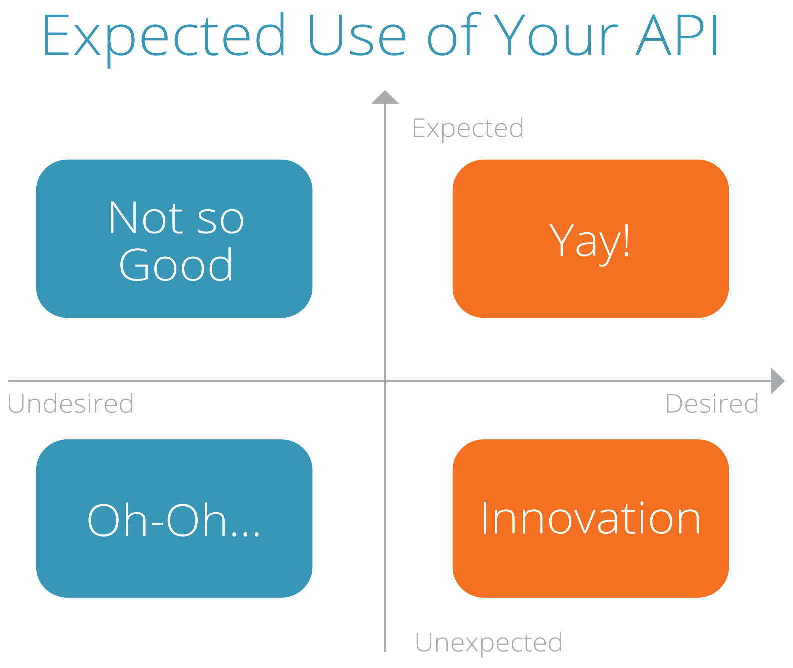
- Are there ways to expand the value of the API to the customer?
- Is future maintenance sufficiently provisioned for?
- Is enough being left on the table for developers to genuinely engage?

It's great to have found and implemented the value proposition and strategy of your API. The best practices from this book will hopefully assist in delivering that value.

Things still change all the time however, and your API and related offerings will have to change too, in order to remain valuable for your users. On a general level there are four main drivers for environmental change:

1. Industry forces: The API space is young and moving quickly. New competitors may appear or provide services that could replace your API.
2. Market forces: The user demands may change. Market segment attractiveness may shrink, or profit margins may decrease.
3. Macro-economic forces: Change in global market conditions or capital markets could affect the propensity of your user's budgets.
4. Trends: Technology trends are changing all the time. XML was the de facto format a while ago. Now it's JSON. What's next? There may be other trends such as regulatory mandates, e.g., demanding higher (potentially more expensive) security standards.

The following framework by Thor Mitchell is useful in planning for evolution and risks, and specifically relates to the ongoing evaluation of your API and its value.



Source: Thor Mitchell (2014)

The graph plots expected/unexpected usage of your API by users on the vertical axis vs. desired/undesired on the horizontal. The usage indicator on the top right hand quadrant is what you are designing and hoping for, but other outcomes could also occur.

Your strategy will be sound and survive longer if:

- You have plenty of strong use cases matching potential customers in the top right quadrant.
- Your API is flexible enough to also allow for innovative uses. And you may choose how much you open the API (in terms of access and functionality) for unbounded innovation in the bottom right quadrant.
- You have considered potentially negative use scenarios and taken API Design, Operations (e.g., access control, rate limits), and other measures to prevent them (top left quadrant).
- You have planned out mechanisms for detecting and blocking behaviors on the "undesired unexpected" side (bottom-left quadrant). Terms and Conditions at the very least allow you to shut down unexpected behavior if necessary.

The strategy for your API may need reevaluating if:

- You are relying too much on “unexpected” innovation (in the bottom right quadrant) to carry the day in terms of value. This “build it and they will come” approach can work if there is also a strong infrastructure to support it. However, if you’re not able to come up with strong use cases you and your users care about, it may be time to rethink the approach.

- If there are constant questions about possible negative behaviors within the organization, then this likely suggests not enough measures have been taken or internally communicated to eliminate undesirable users.

- If your API is constantly compromised and people are attempting to exploit it in unexpected ways, this suggests that there is value in the API but it’s not aligned with the value you were intending to create.

CONCLUSIONS

The purpose of this ebook is to summarize what API teams of successful API programs do and extract those best practices. To help you learn from others who have achieved API success, we identified the following 7 crucial best practices:

1. Focus relentlessly on the value of the API
2. Make the business model clear from the beginning
3. Design and implement with the user in mind
4. Place API operations at the top of the list
5. Obsess about developer experience
6. Go beyond marketing 101
7. Remember API retirement and change management

We hope they will be useful in guiding some choices you make, or at least in guiding some of the questions asked. Here are some final notes from our observations

- Most of the observations are simple to grasp. But it does take discipline.
- Everything starts with being very clear about the value of the effect of your API.
- Make sure you understand your business model and how the API program needs to be established to support it.
- Cover all 7 best practices.
- If you don't have answers to some of the key questions in each section, it's worth taking the time to evaluate and answer them. The results may surprise you.
- Be clear that things change, all the time. Don't make the strategy for your API a static document.

Based on our experience of working with our customers and observations in the API economy in general we can summarize some signs indicating valuable API programs:

- An API is adopted by users because its of value to them: It does an important job. It relieves a pain for them. Or it creates a gain for them.
- An API continues to provide value for users, and delivers a value proposition and strategy that changes according to changing environments.
- An API is valuable to the organization internally. So, it does something important and helps the organization achieve significant objectives.
- As many stakeholders as possible (ideally all) are happy.

Special Thanks

We would like to thank the various people who contributed the case studies to this ebook:

David Meikle and **Steve Griffin** for the Lingo24 case study.

Koen van Erp for the Senzari case study.

Philippe Sultan for the APIdaze case study.

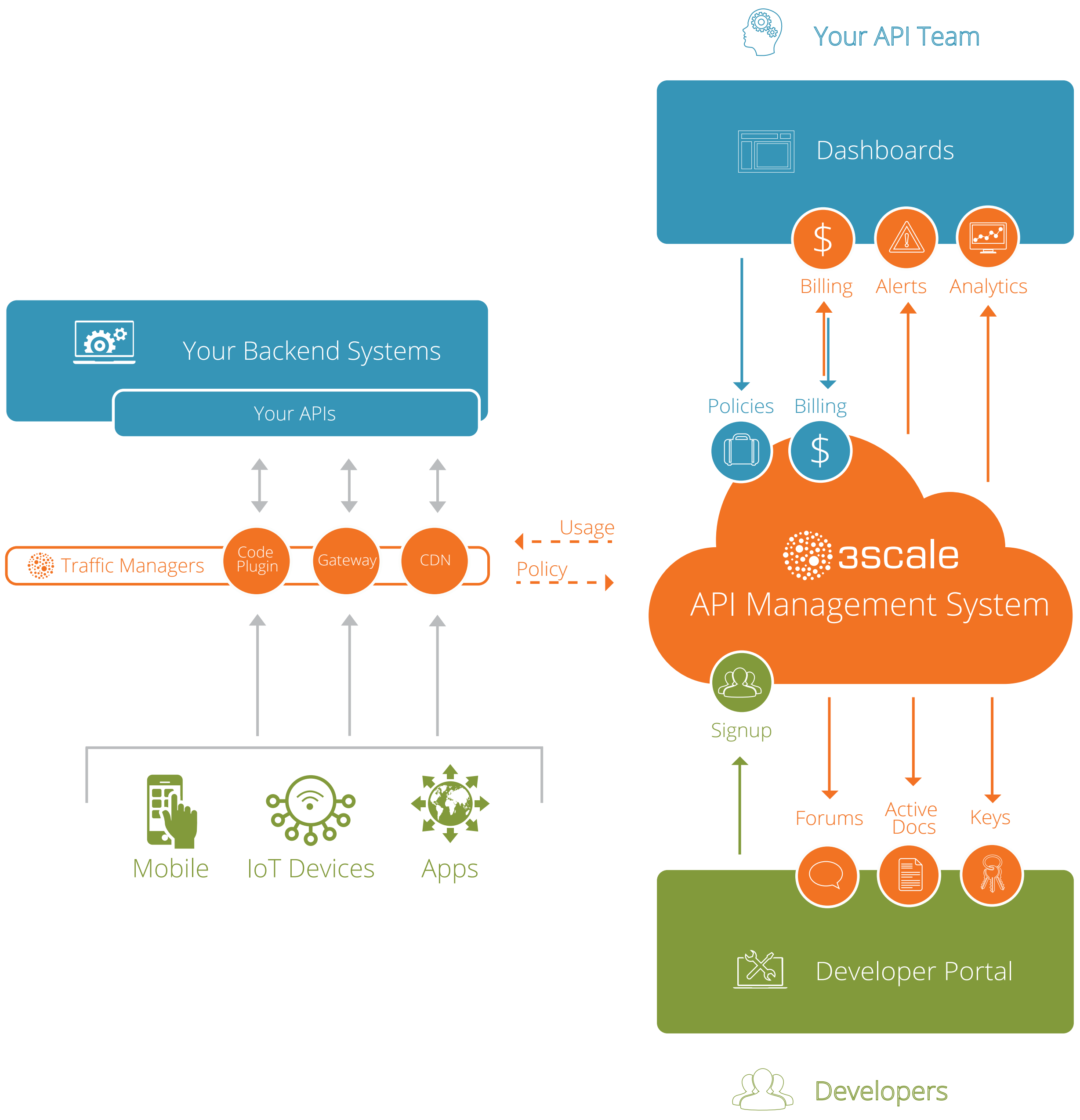
Victor Osimitz for the Slice case study.

Tony Blank for the Context.IO case study.

Amber Feng for the Stripe case study.

ABOUT 3SCALE

3scale is the leading self-serve API Management Platform built with performance, customer control, and excellent time-to-value in mind. 3scale makes it easy to open, distribute, control, and monetize your APIs. No other solution delivers so much power, ease, and flexibility in such a cost effective way. More APIs are powered by 3scale than any other vendor because our unique management architecture and self-serve approach serve all categories of APIs, and all types and sizes of customers. Learn more about how it works, or sign up for your free account and start exploring for yourself.



AUTHORS

MANFRED BORTENSCHLAGER

Manfred (@ManfredBo on Twitter) is an API geek and evangelist. He blogs about topics related to APIs and developer evangelism, and curates articles about API strategy and technology for the API Magazine.

In his day job, Manfred is API Market Development Director at 3scale.net. His job involves educating markets about the value of APIs and about how to implement effective API programs.

STEVEN WILLMOTT

Steven (@njyx on Twitter) was previously the research director of one of the leading European research groups in Europe on distributed systems and Artificial Intelligence at the Universitat Politècnica de Catalunya in Barcelona, Spain. He brings 10 years of technical experience in Web Services, Semantic Web, network technology, and the management of large-scale international R&D teams to his work.

www.3scale.net

450 Townsend St.
San Francisco, CA 94107
+1 (415) 671-6432