E-book

Red Hat

The API owner's manual

Best practices of successful API teams

Manfred Bortenschlager
Director, Business Development for API-based Integration Solutions and API Management, Red Hat

David Codelli
Senior Principal Product Marketing Manager, Red Hat

Steven Willmott
Senior Director and Head of API Infrastructure, Red Hat

Red Hat

Table of contents

facebook.com/redhatinc
@redhat
linkedin.com/company/red-hat

Executive summary

Application Programming Interfaces (APIs) have become the digital connective tissue of modern organizations, adding new capabilities to everything from their operations and products to their partnership strategies. It's no longer a stretch to say that most organizations don't ask whether to engage in API programs, but rather how to do so. This e-book aims to answer this question by drawing on seven best practices 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

This book provides an overview of 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 APIdaze, eBay, Flickr, Lingo24, Netflix, SendBird, SendGrid, Alpaca, Rakuten Slice, Stripe, and Twilio. Throughout this e-book, we hope you ask yourself key questions about your own plans—and gain the right focus to create a valuable API.

Introduction

APIs have quickly become critical for businesses, and API use is increasing. APIs are used for many purposes—developing internal agility between teams, underpinning mobile or Internet of Things (IoT) initiatives, enabling customer integration, and 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, teams can use a few common practices to evaluate their own approach and ensure success.

This book brings together the seven most common factors we've seen adopted by successful API teams. These practices may appear simple at first—and indeed, they are. However, once adopted, we often see organizations skip these steps or focus on the wrong targets or solutions. These failures can easily occur because APIs inherently mix technical and business concerns, and with many stakeholders involved, it is often difficult to determine the key issues.

The team running an API might range from a single member tasked with implementation and operations to multiple teams that span 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 form the foundation for an entire company. Whatever the case, many of the key success factors 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 our experience, there are seven best practices that are found in almost every successful API program:

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, which we will detail in this e-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, along with real-life examples. You'll also find 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 Red Hat's extensive experience with hundreds of production APIs, as well as industry-wide best practices from a range of companies.

API strategy

This book focuses on the "how" of APIs rather than the "why," and it should not be considered an API strategy guide. However, it can still help you formulate some of the questions that need to be answered about your strategy. You should establish a clear, overall corporate API strategy before you start, including key goals and metrics.

Before reading this e-book, we recommend noting the key objectives of your API program for reference. Some of the questions asked later may help change, validate, or provide more substance to your objectives.

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 clearly answer the following three questions:

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?

You should establish a clear, overall corporate API strategy before you start, including key goals and metrics.

The why

People often misinterpret this question. First, rather than focus on the value of the API, it's helpful to think of the value of the effect the API will have on the business. An API is valuable when it becomes a channel that provides new access to the existing value an organization delivers.

Another common error is the belief that in order for an API to be valuable, API users must pay for it. This is true only if the API itself is the product. Red Hat® internal data, for example, suggest that just over half of API projects are internal only. APIs are usually generating 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* e-book, we examine typical use cases for API providers:

1. Enable mobile as an additional channel
2. Grow ecosystems: customer (B2C) or partner ecosystems (B2B)
3. Develop massive reach for transaction or content distribution
4. Power new business models
5. Generate internal innovation

As McKinsey concluded, "APIs allow business to monetize data, forge profitable partnerships, and open new pathways for innovation and growth."[1] In addition, the authors noted that APIs have a internal benefits as well, specifically to:[1]

1. Simplify back-end IT systems.
2. Enable delivery of personalized products and services.
3. Allow for participation in [a] digital ecosystem.

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

1 *Iyengar, Keerthi, et al. "What it really takes to capture the value of APIs," McKinsey Digital, September 2017.*

Example: Flickr

Originally created as an online game, Flickr quickly evolved into a social photo sharing sensation. Flickr's API quickly allowed the company to 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 inspiration behind the concept of BizDev 2.0, a term coined by cofounder Caterina Fake to describe their policy of requiring companies to use the API to develop applications.

The company would only contact companies that 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 the wider business strategy?" In theory, both the concepts of the internal view and the external view of an organization can help define the "what" of an API.

The internal view refers to specific and valuable assets that an organization possesses. The more valuable, rare, inimitable, and nonsubstitutable—also often referred to as VRIN assets—the more suitable they are for the "what" of an API. An organization that has unique data, for example, could allow access to the data via an API. Facebook is one of the world's most popular media owners. Their content generates a lot of "likes," and Facebook is the only provider that can open up access to the number of likes of a piece of content. This combination makes Facebook's content, and the API that allows access to it, extremely valuable.

The external view is related to everything outside of an organization, such as market dynamics, trends, competitors, and customer behaviors—all of which are macro-environmental drivers and industry forces. Macro-environmental drivers include political, economic, social, and technological forces, as well as industry forces such as competition, buyers, suppliers, substitutes, or new entrants (cf. Michael E. Porter's *Five Forces*).[2] This external view affects every business strategy—including thinking about what an API should do.

An example of how external market dynamics can influence strategy is the success of mapping APIs. Early mapping technology companies invested heavily in building mathematical models encoding geographic location, infrastructure, image, and elevation data. They sold these models to many organizations, including regional and national governments, military organizations, and logistics

2 *Michael E. Porter, "How Competitive Forces Shape Strategy," Harvard Business Review: May 1979 (Vol. 57, No. 2), pp. 137–145.*

companies, so these groups could do things like spatial planning, navigation, and instrument (or even unmanned) flight. However, the early companies delivered their models in batch, as monthly file transfers or even as physical media. The companies were slow to respond to the real-time revolution that was sweeping the software world, and they did not anticipate the sudden demand for real-time access to this data via APIs.

This failure opened the door for small, nimble startups like Waze to use the same models to offer APIs and developer programs, capturing mindshare and market share. Waze was later bought by Google, which has incorporated the technology into its own very successful API. Another supplier of geographic information was Navteq, one of the earliest and most successful mapping technology companies and arguably a Google Maps competitor. The Google Maps API is open, but Navteq did not have an API at a time that the market was clamoring for automated interactions with mapping systems. Navteq had been struggling before it was bought by Nokia and rebranded as HERE Technologies. HERE provides an open API and a full-fledged developer program. Waze, a small company since acquired by Google that, in contrast to Navteq, provided a public API for its traffic and navigation data from the beginning, used it as the backbone of its growth strategy.

Fitbit provides an example in the fitness domain. The company offered an activity tracker and fitness products with a public API to foster innovation and grow the Fitbit ecosystem. Fitbit disrupted market giants such as Nike and Adidas, which soon decided to use APIs to support their strategies.

When deciding what an API should do for your business, examine both internal and external views. The decision about the "what" is then usually a combination of the two. The illustration below shows how internal and external views can help find the right "what" for an API program.

Figure 1. External and internal dynamics factor into API execution

While the "why" is unlikely to change often, the "what" may vary significantly based on external factors such as markets, technical considerations, or economic conditions. 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 design the API program to achieve what we want?" is all about implementation and execution. The how is really what this piece 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?

The answers to these questions will be different for each organization, but the seven best practices provide a framework for making these decisions.

The API team

An API team is really most like 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 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 code API technology, and operations folks who ultimately run the API. Over time, you may also have others involved, including support and community team members, API evangelists, and security representatives.

While this team could be a large number of people, some individuals may wear many hats, especially in smaller organizations. It is important to ensure that all stakeholder opinions are represented—even if that consists of just having one of the team members check in on their concerns.

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

No matter the size of your organization, the seven 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 structure can make it particularly challenging to define a common vision for the API.

Best practice #1: Focus relentlessly on the value of the API

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.

John Musser, Director of Platforms for Ford Motor Company and CEO of API Science, Inc., offers five keys to a great API.[3] He suggests your API program should:

1. Provide a valuable service.
2. Have a plan and a business model.
3. Be simple, flexible, and easily adopted.
4. 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 concept in detail, the main takeaway being that it can be really challenging to find the right value proposition.[4]

The value proposition is the main driver for API success. 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 if no user group finds the API valuable. Conversely, you may find users with needs that could be met with a certain proposition, but if the value proposition is not aligned with corporate objectives, it will be difficult to sustain. In this scenario, 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–that is, if the API links to existing offerings and enhances them. As long as the API is structured in 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 the Value Proposition Canvas,[5] created by Alex Osterwalker, cofounder of Strategyzer. This model 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" between what a customer wants and what your product or service can offer, and the canvas can be used to define and refine the value proposition.

---

3 *Musser, John, "What makes a great open API?" O'Reilly Open Source Convention 2012.*

4 *Guerrero, Hugo, and Steven Willmott. "Building great APIs, part 1: The gold standard," Red Hat developers blog, November 2, 2012.*

5 *Strategyzer, "Value Proposition Canvas."*

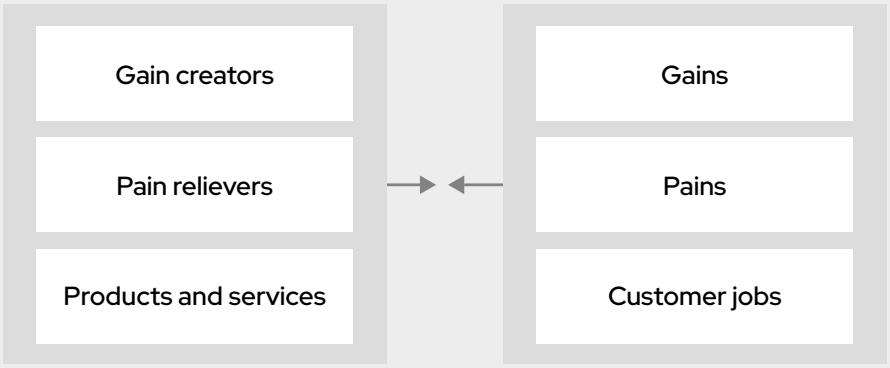

Figure 2. Value Proposition Canvas

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, such as:

- Automatically sending urgent communications to team members in an emergency.
- Backing up critical files to ensure they are never lost.
- Sampling data to detect certain events.

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

- Ensuring reliability of sending with multiple attempts, 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 while minimizing the amount of transfer bandwidth.
- Dealing with massive amounts of data and attempting to correlate it in real time.

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

- Sending other types of notifications, which create opportunities rather than warn of threats.
- Eliminating other storage equipment if reliability is good enough.
- Automatically triggering actions based on 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 nonfunctional aspects and additional services. Think rather broadly and list things like support, documentation, or developer portals–everything that a user could use. 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 any issues that may prevent them from doing so. In his model, Osterwalder refers to these features as pain relievers. Then, describe how you intend to create gains for your API users.

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

- A multichannel 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 can 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 that need to be added, revised, refined, or eliminated. It could also be that your API offers 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 the value proposition for your APIs. In the case of the messaging API example, this statement might be something like:

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

You may be thinking "this seems like complete overkill – ours is just an internal API." This reaction is natural, but such a focus on value is key even in internal use cases. A poorly determined value proposition will lead to a lot of time spent educating and pitching the API to other teams. A well-defined value proposition will make the API program a key contributor to the business.

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

Example: Lingo24

Tech-savvy translation agency Lingo24 offers a good example 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 via two APIs:

- The Business Document API provides access to Lingo24's professional human translation services. These offer 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. This access is 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 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 ecosystem, and allow those partners to embed translation within their solutions as a value-added service. This ability would allow the development community to build on Lingo24's service.

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 allow Lingo24 to integrate directly with customers to automate translation workflows and work closely with them as partners – which is what their enterprise customers were really seeking.

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 – providing easy access for customers to use these services.

How did Lingo24 design the API program?

Lingo24 determined how their resources and services matched with the market by exploring various customer profiles, what those customers 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 to not only 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 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 because the two services had different pricing structure needs, sales and marketing challenges, and scalability requirements. Although they split their offering across two APIs, they developed a shared developer portal using Red Hat 3scale API Management. This portal created a clear point of access to Lingo24 development resources, as well as 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: OpenCage

Often, the value of personalized service and open standards are overlooked in API programs. OpenCage offers an API that provides sophisticated geocoding, reverse geocoding, and other map features—features offered by Google, Bing, and other giant technology companies. OpenCage, in contrast to those players, is a small, self-funded startup. How is this recent market entrant able to compete? OpenCage customizes its service and delivers it in a way that the bigger companies aren't able to do.

OpenCage uses public data and open standards to deliver data for every country in the world. Because of the terms of service of the open data, and OpenCage's own terms of service, OpenCage customers are able to use the data in ways not possible for Google or others. For example, OpenCage customers can display map information on any mapping technology (Google maps can only display geocoded location data on Google maps). Furthermore, customers can retain the data as long as they like, and the APIs are significantly cheaper than the competition. Also, OpenCage offers live human support to all customers regardless of level, something that Google and similar companies cannot do.

Critical questions for consideration

To help define the value of your API program, 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, or external developers), their role (are they data scientists, mobile developers, or operations people), and their requirements or preferences.

2. **What user pain points are we solving or what gains are we creating for the user?**
   This question should be answered in relation 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?**
   With the help of the Value Proposition Canvas, identify the 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. Identify important upcoming milestones relating to internal or external changes (such as trends or technological innovations.)
5. **What value is the API creating 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 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 rapidly accelerating cost. As a result, in a worst-case scenario, your API project 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."[6] 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 – in addition to the required cost structure. For a business model to work, an adequate and addressable market needs to be available. It should also describe served customer segments, the distribution channels to reach them, 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 the existing core business of the organization is, 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.[7]

6 *Osterwalder, Alex and Yves Pigneur, Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers. Strategyzer: July 2010.*

7 *Strategyzer, "Business Model Canvas."*

The Business Model Canvas

Using the Business Model Canvas will help 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 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 rapidly accelerating cost. As a result, in a worst-case scenario, the API project may ultimately have to be shut down or refocused.

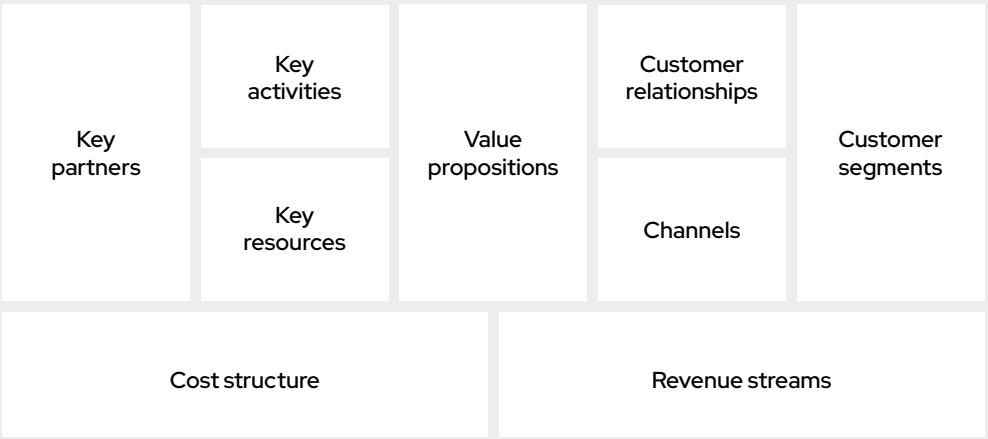

Figure 3. The Business Model Canvas, adapted for readability from Strategyzer

In some cases, APIs can lead to entirely new business opportunities for an organization – but even in that case, they generally take advantage of existing assets or expertise.

In summary, it is important to determine the right business model for APIs for these reasons:

1. It brings the value of the API into focus, making it easier to determine long-term commitments to the API program. Without that commitment, the program will usually lack the resources to establish and run effectively.
2. It helps define the functionality of the product, which is needed to satisfy third parties and execute the business model.
3. It ensures consideration of organizational roles and responsibilities and determines who retains the value generated by the API. It also defines benefits gained by the API users and how that compares with benefits gained by the API provider.

A clear business model should be considered internally and communicated to customers and partners. Without this business model, the API risks being an appendage that doesn't deliver value – and is unlikely to be sustained.

Example: Netflix

Netflix is a pioneer of APIs and API strategy. It is notable how Netflix used APIs to support their business model. When the company first launched its API program, it decided to open the API to the public. It seemed to do everything right: provided a solid developer portal with sample code, documentation, forums, and showcases. It had people covering developer relations and engaged in events such as hackathons.

However, Netflix noticed that a public API program did not contribute to its business model in a significantly valuable way. Early in 2013, the company decided to shut down the public API program.[8] Its decision was to use the power of APIs primarily for internal use and only open it to very few trusted partners. Netflix created a modern API and one of the most cited microservices architectures.[9] The company scaled into the cloud, grew internationally, and now serves millions of consumers over a wide array of devices and operating systems.

The decision to close the public API program and use APIs 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, but it's not. Having a public API program initially helped them increase brand awareness, which clearly spurred its current success. Also, other companies competing with Netflix started to adopt APIs. As a result, APIs are now common in the media and entertainment sector.

It is notable how Netflix used APIs to support its business model. Netflix was a pioneer in opening its API to the public at the program's inception.

8 *API Evangelist, "Netflix API is much more than a public API." March 2013.*

9 *Mauro, Tony. "Adopting microservices at Netflix: Lessons for architectural design." NGINX. February 2015.*

Example: Alpaca

Alpaca's API has been likened to "Strype for stocks." Essentially, Alpaca provides an API-driven platform permitting any organization to launch a commission-free trading platform with minimal infrastructure costs.

Alpaca's founders envisioned a world where buying and selling securities would be so simple that the transactions could easily be added to multiple contexts, not just banking and investment companies, and not just in the developed countries. To that end, Alpaca built APIs that allow developers to create rich transaction experiences easily without worry of regulatory or security complexity.

Alpaca was founded in 2015, and as of 2019 it has achieved US$1 billion in transactions.[10]

Critical questions for consideration

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

What value does the API create for the organization?
The value of an API may include more than monetary value. This question can be answered by thinking about how the API helps the organization. It could include analyzing how to increase reach or innovation or using network effects for content distribution.

How do we capture that value?
Once you understand what the value is, think about the best way to capture the value — and how to lower the barriers for doing so as much as possible.

What costs are occurring and how do we cover them?
Think about which costs are related to the API. These costs will most likely lie outside of the API team, such as engineering or marketing efforts.

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.

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 take advantage of complementary offerings.

Best practice #3: Design and implement with the user in mind

Good API design has some core principles, which may differ in implementation. The authors of *APIs: A Strategy Guide* have a great analogy:[11]

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.

---

10 *Constine, Josh, "Alpaca nabs $6M for stocks API so anyone can build a Robinhood." TechCrunch, November 8, 2019.*

11 *Woods, Dan, et al. "APIs: A strategy guide." O'Reilly Media. December 2011.*

This level of "ready to drive" design is what great API teams strive for – APIs that require little or no explanation to the experienced 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 *RESTful Web APIs*, or *APIs: A Strategy Guide*. The principles of good API design are closely aligned with 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 Extensible Markup Language (XML), JavaScript Object Notation (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 determine the base of the potential space of operations, including the underlying systems and data models, and define what subset of these operations is feasible and valuable. To find the right balance between simplicity and flexibility:

1. 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 combines several atomic operations to serve these use cases.

Arguably, the concept of Hypermedia as the Engine of Application State (HATEOAS) can further improve flexibility because it allows runtime changes in the API and in client operations. HATEOAS increases 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 API

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 phone 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 phone 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, representational state transfer (REST), and webhooks are familiar to anyone who develops web applications.

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 Document Object Model (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 Session Initiation Protocol (SIP) 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. For simplicity, it has not been considered in the original development of this part of the API.

APIdaze also provides a way to relay events that happen within the infrastructure. If a phone call is coming into 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 voicemail. XML-based HTTP webhooks come into play. 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, and eventually get engaged enough to run their applications. Working on having clear documentation is a day-to-day task for the technical team.

Critical questions for consideration

To think through your API design, consider the following 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 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 popular. However, you should not follow this trend just because they are popular. There are use cases that are very well suited for RESTful APIs, but others favor different architectural styles.
3. **Did we just expose our 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 that require such an approach.
4. **Which geographic regions are most important, and have we planned our datacenters accordingly?**
   API design must also cover nonfunctional elements such as latency and availability. Make sure to choose datacenters 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. You may decide to completely decouple API design from other products. If you do, the structure must be communicated internally and externally.

Best practice #4: Place API operations at the top of the list

API operations manages APIs once they are live to make sure that they are accessible and deliver according to developers' expectations. API management involves two main functions:

1. Streamlining internal processes to be efficient and reduce cost.
2. Making operations effective to meet the expectations of developers who are external to the program.

This notion ties in to Musser's fourth key to a great API: It should be managed and measured.[12] In the API Gold Standard (III) article, we analyzed in detail what and how that can be achieved.[13] There is an additional tool that should help with API operations: the API Operations Donut.

12 *Musser, John, "What makes a great open API?" O'Reilly Open Source Convention 2012.*

13 *Guerrero, Hugo, and Steven Willmott. "Building great APIs, part 1: The gold standard," Red Hat developers blog, November 2, 2012.*

API Operations Donut

Operations management theory suggests five key performance objectives:

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

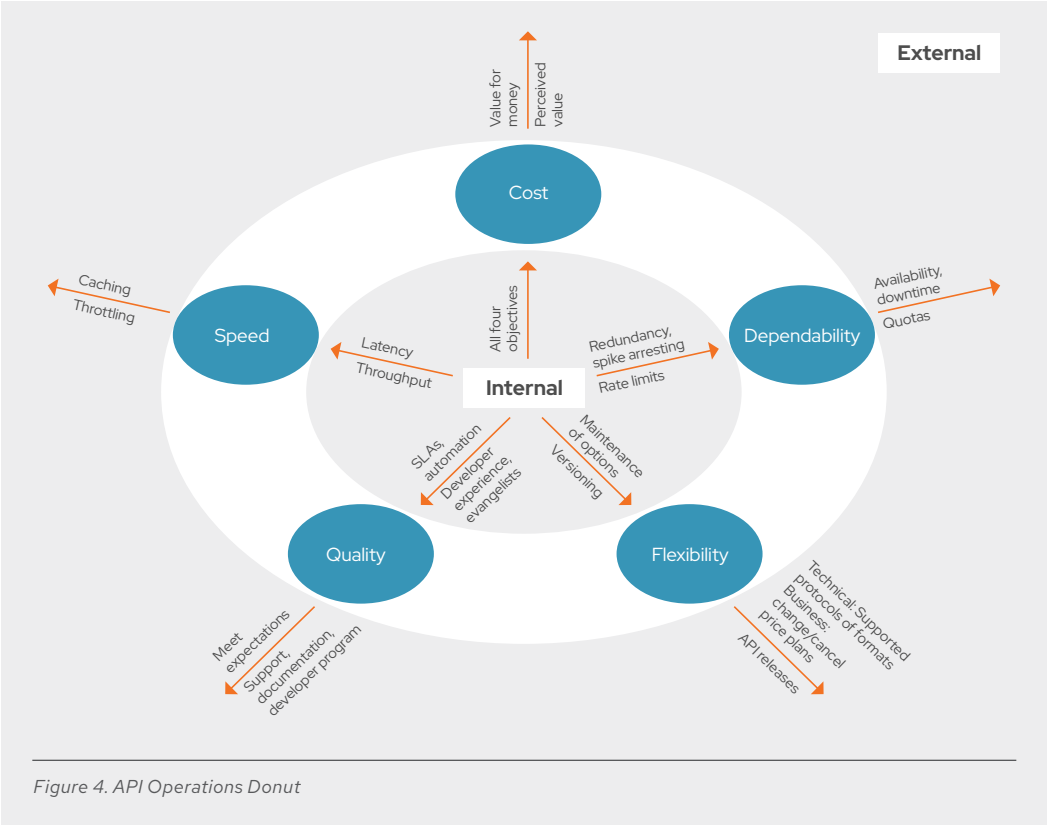

Figure 4. API Operations Donut

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, and everything outside of the ring shows 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 timeframe. 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 describes the options developers have in adopting APIs. This flexibility 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 ring refers to the variables that are in the realm of control of internal API teams, such as 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' expectations, and it influences developers' 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 (SLAs). Streamlined and purposeful automated processes can improve internal efficiency and contribute positively to quality.

Speed

Latency and throughput are important to achieving speed in API operations. Both can be internally influenced by techniques such as throttling or caching. Throttling, like quotas, can also be used for defining an API providers' business models.

Cost

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

At a minimum, we suggest having the following in place for your API management:

1. Access control: Authentication and authorization systems to identify the origin 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 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 – Red Hat included. In many cases, using a vendor is a cost-effective way to address these problems, but the strategy must be thorough.

Example: Rakuten Slice

Rakuten Slice built a powerful data-extraction engine that connects to any email inbox, identifies the e-commerce receipts contained in that inbox, and extracts item-level purchase information from those receipts. This data-extraction engine powers the Rakuten Slice consumer apps (available from www.slice.com). Rakuten Slice officially launched the Slice API, opening the same engine to third-party developers building new experiences around their users' purchase data.

Rakuten Slice's key performance objective in building the API was flexibility. Due to the diversity of the applications of this technology, 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.

Rakuten 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 platform, and others that wanted quick integration and were comfortable using OAuth to "link an existing Rakuten Slice account." Initially, Rakuten 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 needed a way to retrieve orders, purchased items, and shipment information from specified users that had authorized Rakuten Slice to share their data.

Ultimately, Rakuten Slice decided to support two types of authorization: vanilla OAuth 2.0, and a signature-based method for power white-label integrations. This decision added significant complexity to the API, and Rakuten Slice implemented its developer portal in such a way that most developers would only be aware of the OAuth integration method.

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, and how to enforce limits.
2. **How do we capture metrics and handle alerts?**
   Gaining visibility about what's happening with your API during operations is key for API success. This is where analytics can help. 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 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

While developer experience 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 a negative developer experience.

An 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, well thought out API design has a considerable impact on developer experience and adoption. Adoption is an essential part of the developer experience.

Musser summarizes how to get developer engagement:[14]

- Make it very clear what the API does
- Provide instant signup
- Provide free access
- Be transparent about pricing
- Have great documentation

A key metric to improve API design for easy adoption is the Time To First Hello World (TTFHW). This metric is a great way to put yourself in the shoes of a developer 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.

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

TTFHW should be the main driver when building developer experience 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 Musser's fifth and last key for a great API: "provide great developer support."

---

14 *Musser, John, "What makes a great open API?" O'Reilly Open Source Convention 2012.*

Crafting a developer program

The aim of an effective developer program is to provide outstanding developer experience. Rahul Dhide puts it this way:

DX [developer experience] is not just about APIs and CLI [command-line interface], they are small parts of [a] developer's overall experience with a product or service. While designing for developer products, designers use UX methods and principles to deliver a positive experience. DX design is about understanding the context of use, understanding what developers need to complete their tasks, underlying technology, integration points, and focussing on how developers feel while using a product or service.[15]

Developer experience includes two tenets:

1. Design a product or service that provides a clear value to developers and addresses a clear pain or gain. The value can be monetary or another type, 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. It should have a lightweight registration mechanism (or none at all), access to testing features, great documentation, and a lot of free and tidy source code.

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

- Developer portal
- Community building
- Evangelists
- Events
- Communications and social media
- Pilot partners and case studies
- Acceleration via ecosystem partners
- Measurement

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

15 *Dhide, Rahul, "Building the developer experience (DX) from the ground up." Argo blog, October 4, 2017.*

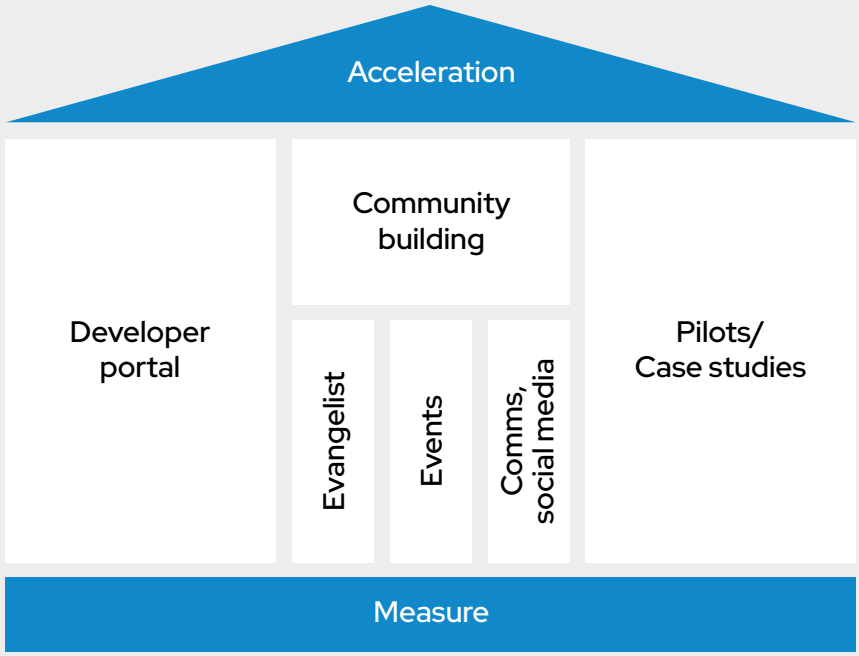

Figure 5. Elements of a developer program

Developer portal

The developer portal is the key element of a developer program. It is the core entry point for developers to sign up, access, and use your APIs. Getting access to your API should be simple for developers so they can get started quickly. TTFHW is the best metric to measure this capability. You should also consider streamlining the sign-up process – the simpler and quicker, the better. A recommended best practice is allowing developers 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 helpful. Also, supplementary content, such as getting started guides, API reference documentation, or source code, helps reduce the learning curve.

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

Community building

Community building has two main aspects: a physical presence 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 promote the benefits of your API program and increase adoption. Events are also valuable to get in touch with your community and meet the developers in person, which often leads to useful insights that 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, and training sessions.

Developer evangelists

The developer evangelist role is a key element of a successful developer program. The onion model[16] summarizes some of the most important activities of a developer evangelist. This model is also scalable – it can be applied to one evangelist or a team of evangelists.

Pilot partners and case studies

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

Acceleration via ecosystem partners

As an API provider, you are operating in an ecosystem of partners and vendors. These external resources 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. Such alliances can often be found when APIs are complementary and provide value to developers when combined.

Measurement

You can only manage what you measure. Measuring is important to the developer program to understand its effectiveness and 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 helpful to create correlations, such as "did a talk at an event trigger more API signups?" Swift, CEO and Co-founder of Major League Hacking, has an insightful presentation about the Nuts and bolts of developer events.[17]

---

16 *3scale presentation at API Strategy and Practice Conference Chicago, "How to use donuts and onions for scaling API programs." September 25, 2014.*

17 *Swift, "The nuts and bolts of developer events," APIdays conference. January 22, 2015.*

Example: SendGrid

SendGrid, an email services company that uses APIs to send and manage emails, offers another example of good developer experience. The SendGrid developer portal incorporates most of the best practices we summarize in this e-book. SendGrid understands the power of communities. SendGrid's four layers of focus are: developer education, startup outreach, developer experience, and events.[18] The company determines the effectiveness of their activities using seven metrics. Finally, developer evangelists are crucial for the SendGrid API success, as noted in a blog post about SendGrid hiring practices.[19]

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 evaluation improves the developer-friendliness of your API by thinking about the end-to-end TTFHW. We recommend considering TTFHW and TTFPA with all elements that are added to the developer experience (like portals) — and every aspect that changes.
3. **What is the onboarding process for developers, and is it as painless as possible?**
   The onboarding process needs to be in line with the use cases of your API, and it needs to be appropriate. The level of security naturally needs to be higher for more sensitive APIs or data access, and it often needs more formal agreements. For other APIs, it should be simple and straightforward to allow for early developer success.
4. **Are we providing enough value 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. Helping developers gain success will retain and grow their numbers.
5. **How do we support developers if they face problems?**
   We believe in the self-service approach. Many developer questions can be covered by good documentation, FAQs, or forums. But self-service has its limits. For more in-depth questions–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?
Comprehensive documentation for fast-evolving technology is challenging, but the best products find a balance with good coverage of general principles, feature exploration, and end-to-end examples.

---

18 *Jones, Carrie Melissa, "From startup to rapid growth: How SendGrid scaled their developer evangelist strategy as they reached critical mass." CMX.*

19 *SendGrid blog, "How I evaluate a developer evangelist candidate." January 1, 2014.*

Best practice #6: Go beyond marketing 101

Marketing APIs can prove challenging: Developers often reject the idea of marketing, and the strategies applied to marketing APIs can sometimes be overly simplified. (Hackathons aren't the only way to attract API users.) APIs should be marketed just like any other product.

Marketing should bring the right product to the right customer in the right way. The same theory is true for marketing an 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 use when marketing APIs is segmentation, targeting, and positioning (STP).

Segmentation

Customers of an API provider are people or companies that develop software, regardless of the scope of an API. Various audiences should be considered when marketing APIs:

1. Internal usage
2. 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 IDC indicate that there were 22.3 million software developers in the world at the start of 2018. This number does not include quality assurance (QA) engineers, deployment engineers, technical writers, or product managers.[20] The latter group, which also could use the API portal, is probably at least as large as the software developers group itself, but for simplicity, consider just the developers as the total addressable market. Clearly that group is too large and diverse to engage with a compelling product offer. The total market needs to be divided into smaller developer segments based on characteristics and behaviors.

A great way to establish developer segments can be done via the jobs-to-be-done method. This method focuses less on conventional segmentation ("If customer buys A, they are likely to buy B, because other buyers of A bought B"). Instead, the jobs-to-be-done method calls for analyzing responsibilities of the buyer that may only relate peripherally to the purchase. Harvard Business Review has done an interesting report of this method. This segmentation could be used as part of your API marketing strategy.

20 *"IDC's Worldwide Developer Census, 2018: Part-Time Developers Lead the Expansion of the Global Developer Population," doc #US44363318. October 2018.*

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: Platforms, operating systems, programming languages, or tools.
2. Individual: Skills, experience, or persona of developers (similar to the psychographic dimension mentioned above).
3. Business: The types of companies or organizations, their market position, supply chain position, or financial strength.
4. Market: Secondary markets dependent on the prime market, such as suppliers, buyers, or other depending verticals.

Determine whether the resulting segments are relevant, large enough, or valuable to your API program, and whether there are 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 segments, which is often 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, consider the following characteristics:

- Accessible. The segment 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.
- Substantial. There should be a critical mass of developers in your segments. Also check if the community is active and growing. You don't want to waste your efforts on a segment that is disappearing.
- Differentiable. The segments you choose should be sufficiently different, with a differentiated set of tactics.

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

Positioning

Positioning is making sure that your product occupies a distinctive – and desirable – place in the minds of your customers, compared to other products from your competitors. By now, you 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 pain points, and creates important gains that users care about, as we described in the Value Proposition Canvas.

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 often put off by marketing that lacks substance. As a result, it's important to use developer-specific tactics. Getting the developer experience right, as described in the previous section, is a crucial element. Some of the most effective tactics include:

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

Examples: Twilio, Braintree, and SendBird

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 across the globe. 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 for programming languages such as PHP, Ruby, Python, C#, and Node.js. That's why Twilio provides helper libraries for exactly these languages. Twilio is also known for its 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."[21]

PayPal is another example of a company that targets different segments with different API programs. PayPal provides and operates two payment APIs as products: the PayPal API and the Braintree API. Braintree has gotten a lot of positive developer attention, especially among long-tail developers, via a public API. The PayPal API remains the first choice for close enterprise customers, and it is accessible via an API partner program. Tactics for targeting developers in these two different segments are completely different. For the PayPal API, it's often 1:1 marketing and sales engagements. For the Braintree API, it's more like 1:n, similar to what Twilio is doing in working with developer evangelists. In the 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.

SendBird has succeeded with careful segmentation and a focus on developer experience. While consumers now have many free options for messaging on their phones, computers, or tablets, SendBird recognized that businesses needed to embed messaging as part of a customized experience. It attracted blue-chip media companies like NBA, Yahoo! Sports, GO-JEK, and Virgin Mobile UAE as early customers. SendBird's popularity with developers propelled it to 180% growth in 2018, to jump to world leadership in embedded messaging.[22]

---

21 *Robinett, Ricky. "Being Alfred: Serving developer communities and making heroes." API Strategy and Practice Conference Amsterdam, March 2014.*

22 *"SendBird raises additional $50 million Series B to ramp up global demand for in-app messaging; now surpasses 1 billion messages per month." TechStartups, May 6, 2019.*

Example: eBay

The eBay API ranks as one of the longest-standing API programs, and it is often cited as an example of an API success story. And for eBay, exposing APIs to its platform to allow developers access to executing e-commerce functions was a huge success. eBay recently crossed a threshold of US$1 billion in merchandise bought entirely via APIs, transacted between parties that never visited the website.[23]

eBay was also one of the first API providers that planned and executed a dedicated developer marketing program. eBay's developer segments and targets have evolved since its API launched 20 years ago. It first targeted a few selected partners who had to obtain a special license to participate. eBay soon noticed that to achieve the desired reach, it needed to open up access, which also required a stronger self-service model for their developer program. The result was the eBay developer portal.

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 that 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 versus vertical and industry events, global versus local, conference versus hackathon, formal versus informal. 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 have become popular API events. These events can be effective, but 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 external APIs, the audience is often defined by the customer set. However, internal marketing is still an important factor to consider. 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.

---

23 *Santos, Wendell, "How eBay's buy APIs hit $1B in gross merchandise bought." ProgrammableWeb, January 1, 2020.*

Best practice #7: Remember API retirement and change management

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) that 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 rapidly destroy confidence in the API and could potentially create legal issues. At the least, such a breakdown causes a ripple of urgent work for users and customers. At the worst, it could cause customers to go elsewhere.

The cost of such breakdowns climbs rapidly in cases where:

- The developers or 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 that require vendor approval and customer permission to update code.
- Apps are deployed on physical or hardware devices with little to no update capability or user interface.

Each of these considerations make client code updates 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 versus nonbreaking 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 are additive—adding new data to existing return types or allowing additional parameters that 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 that 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 types 1 and 2 are considered nonbreaking changes—in other words, the old version of the API should still function for clients despite the changes. Changes of type 3 and 4 are breaking changes—in other words, applications using affected methods will 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 be accompanied by a new major version number change (e.g., v1 to v2) and a migration plan between versions. Nonbreaking 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 this plan:

- 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 nonbreaking changes break

Unfortunately, changes that often appear harmless 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 that seems innocuous affects some clients.

It can be extremely difficult to make certain that changes won't break some apps. For these reasons, we strongly recommend that any changes to the API, even if categorized as "nonbreaking," should be rolled out by:

1. Providing a test endpoint with the new version prior to launch.
2. Sending an email or other communication to developers informing them of the change and giving timing and details.

Communication and the contract

Things go wrong – this is quite normal. When they do, it is important to communicate what went wrong. Don't 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 terms 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 in advance – positives and negatives – builds confIdence. It also provides a reliable framework for decisions.

The end of the line

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

- An even longer lead time retirement notice.
- Possible negative media coverage.
- A migration plan for your API users.
- Export and extract tools in cases where the API was a key interface to customer data.

Example: Stripe

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 it's taken with its 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 breaking and will rarely need to care about what version they're on. 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, read its blog.[24]

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 question is arguably the most critical for your API program—what level of service stability are you willing to commit to for your users? This commitment 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 question 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 OpenAPI? Is the current model checked against the previous version? How is change communicated?
4. **How do we detect how many users 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 could 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?

---

24 *Feng, Amber. "Move fast, don't break your API." Stripe blog. October 10, 2014.*

Boosting your API strategy

The best practices in this book will help you define and implement an effective strategy for your API. They will also 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?
- Have we provisioned sufficiently for the future?
- Is the value proposition attractive enough 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 to remain valuable for your users. 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 for many years. 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.

Building an API program that lasts

API programs typically fall into one of these four groups:

1. The API deploys and thrives as expected.
2. While well conceived, the API did not thrive.
3. The API program deploys and realizes value, but in a way that was not predicted at program inception.
4. The API program fails due to poor planning, execution, or faulty assumptions.

If your program falls into the first two categories, it will continue to thrive if:

- You have plenty of strong use cases matching potential customers in the target market.
- Your API is flexible enough to also allow for innovative uses. You may choose how much you open the API (in terms of access and functionality) for unbounded innovation.
- You have considered potentially negative use scenarios and taken API design, operations (e.g., access control, rate limits), and other measures to prevent failure scenarios.
- You have planned out mechanisms for detecting and blocking unexpected negative usage. At the very least, terms and conditions allow you to shut down unexpected behavior if necessary.

You may need to reevaluate your API strategy if:

- You are relying too much on "unexpected" innovation 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 that 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 e-book is to summarize the best practices of successful API programs. To help you learn from others who have achieved API success, we identified the following seven 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 these best practices will help guide some choices you make, or at least the questions asked. Here are some final notes from our observations:

- Most of the observations are simple to grasp, but it takes discipline.
- Everything starts with being very clear about the value your APIs effect.
- Make sure you understand your business model and how the API program needs to be established to support it.
- Cover all seven 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 working with our customers and making observations in the API economy, we can summarize some traits of a valuable API program:

- An API is adopted by users because it's of value to them: It does an important job by relieving pain or creating gain.
- An API continues to provide value for users, delivering a value proposition and strategy that changes according to the environment.
- An API is valuable to the organization internally. It does something important and helps the organization achieve significant objectives.
- As many stakeholders as possible (ideally all) are happy.

Red Hat Integration

Red Hat Integration provides developers and architects with cloud-native tools for integrating applications and systems. The product offers capabilities for application and API connectivity, API management and security, data transformation, service composition, service orchestration, real-time messaging, data streaming, and cross-datacenter consistency.

Red Hat Integration was built with cloud-native development in mind so developers can use the same advanced build, management, and runtime platforms for both new service development and integration. The tools create artifacts that are deployable on cloud-native platforms in any combination of public and private cloud, or on-premise for scalable, highly available microservices using the most powerful container management tools on the market.

Red Hat Integration embeds easy-to-use IT productivity tools into the developer toolchain, simplifying integration for microservices teams and enabling them to participate in high-velocity development.

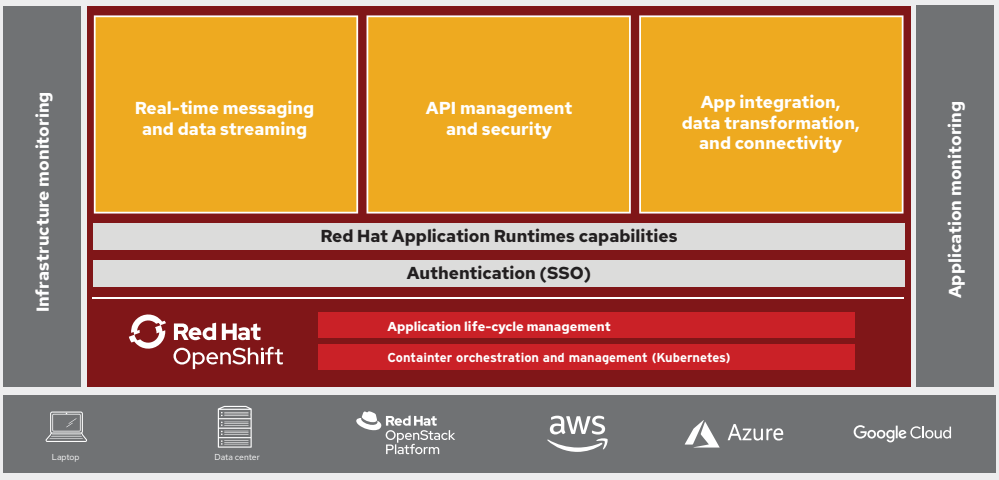

Figure 6. Red Hat Integration overview

Authors

Manfred Bortenschlager, Business Development Manager, Red Hat
Manfred 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 API Magazine. In his day job, Manfred is Business Development Manager at Red Hat. His job involves educating markets about the value of APIs and about how to implement effective API programs.

David Codelli, Senior Principal Product Marketing Manager, Red Hat
David is a Product Marketing Manager in Red Hat's Middleware Group. Previously, he spent many years of development, product management, and marketing with SeeBeyond and Sun Microsystems, rising to Director of Product Marketing. During his career, he has written software for Verizon, McKesson, TIBCO, and General Atomics.

Steven Willmott, Senior Director and Head of API Infrastructure, Red Hat
Steven is the Senior Director and Head of API Infrastructure at Red Hat. Previously, he was 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 15 years of technical experience in web services, semantic web, network technology, and the management of large-scale, international research and development teams.

Special thanks

We would like to thank the various people who contributed to the case studies in this e-book:

Kin Lin: Flickr
David Meikle and Steve Griffin: Lingo24
Philippe Sultan: APIdaze API
Victor Osimitz: Rakuten Slice
Amber Feng: Stripe

About Red Hat

Red Hat is the world's leading provider of enterprise open source software solutions, using a community-powered approach to deliver reliable and high-performing Linux, hybrid cloud, container, and Kubernetes technologies. Red Hat helps customers integrate new and existing IT applications, develop cloud-native applications, standardize on our industry-leading operating system, and automate, secure, and manage complex environments. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500. As a strategic partner to cloud providers, system integrators, application vendors, customers, and open source communities, Red Hat can help organizations prepare for the digital future.

facebook.com/redhatinc
@redhat
linkedin.com/company/red-hat

North America
1 888 REDHAT1
www.redhat.com

Europe, Middle East, and Africa
00800 7334 2835
europe@redhat.com

Asia Pacific
+65 6490 4200
apac@redhat.com

Latin America
+54 11 4329 7300
info-latam@redhat.com

redhat.com
#F21435_0320