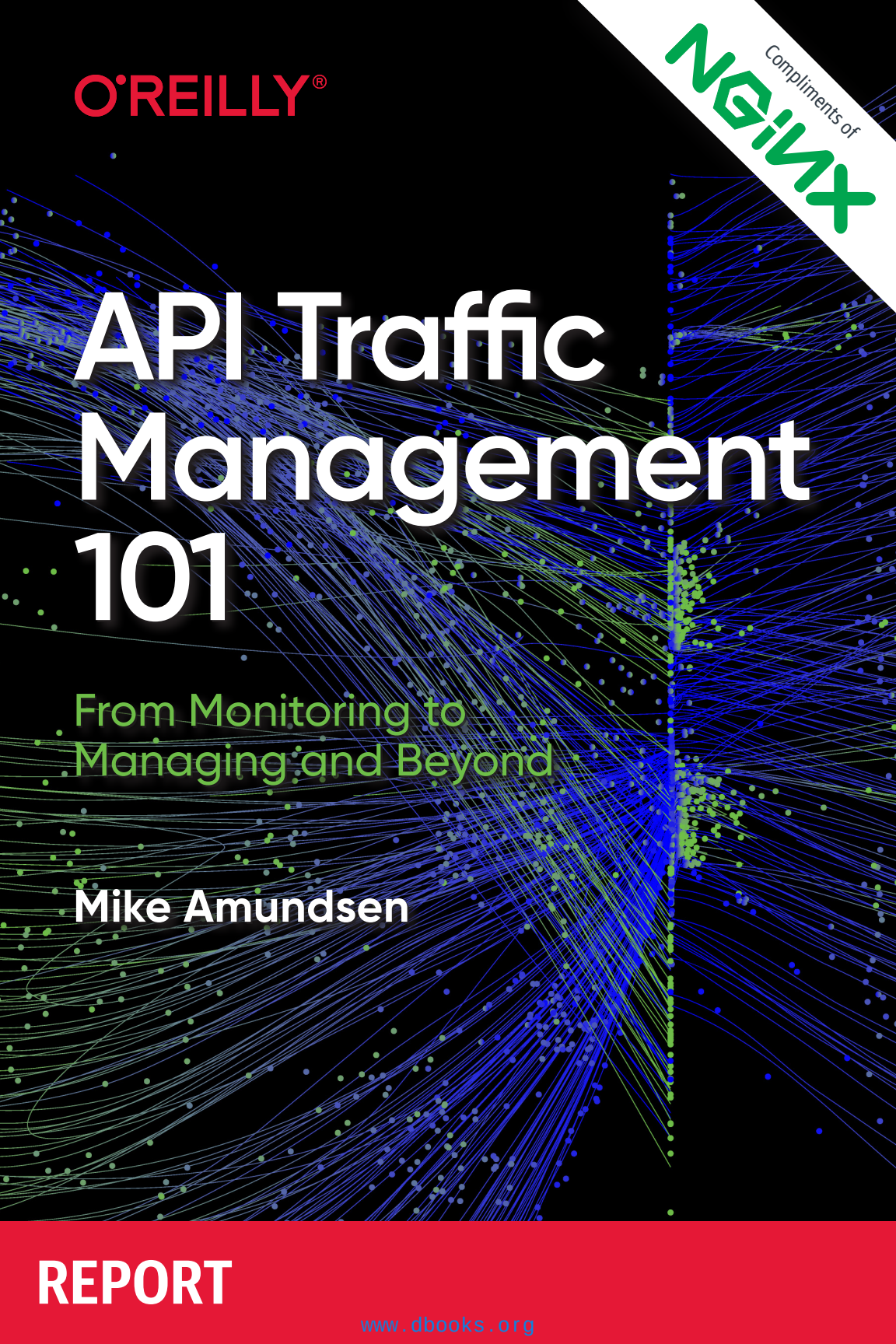


Why Trust Your APIs to Anyone Else?

Traditional API management tools are complex and slow. As the most-trusted API gateway, we knew we could do better. NGINX has modernized full API lifecycle management.

API Definition and Publication

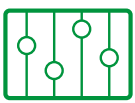
Define APIs using an intuitive interface.

Rate Limiting

Protection against malicious API clients.

Authentication and Authorization

Applying fine-grained access control for better security.



Real-Time Monitoring and Alerting

Get critical insights into application performance.

Dashboards

Monitor and troubleshoot API Gateways quickly.

Learn more at **nginx.com/apim**

NGINX

API Traffic Management 101

From Monitoring to Managing and Beyond

Mike Amundsen

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

API Traffic Management 101
by Mike Amundsen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Elizabeth Kelly
Copyeditor: Octal Publishing, Inc.
Proofreader: Kim Wimpsett
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the First Edition
2019-08-28: First Release

This work is part of a collaboration between O'Reilly and NGINX. See our statement of editorial independence.

978-1-492-05636-2

LSI

Table of Contents

Preface

Welcome to *API Traffic Management 101*!

The aim of this short book is to introduce the general themes, challenges, and opportunities in the world of managing API traffic. Most of the examples and recommendations come from my own experience (or that of colleagues) while working with customers, ranging from small local startups to global enterprises.

Who Should Read This Book

This book is for those just getting started in API traffic management as well as those who have experience and want to review the basics and take your work to the next level. Developers who are responsible for creating and maintaining APIs will learn how network admins and those charged with enabling API traffic collection identify and track key API activity. And admins who design and maintain API traffic metrics can learn how to align and enrich traffic collection to support and inform API developers.

And you don't need to be a traffic management practitioner to extract value from this book. I also spend time focusing on the business value of good API traffic practice, including the ability to connect your organization's business goals and internal progress measurements with the useful traffic monitoring, reporting, and analysis.

How to Get the Most from This Book

I've included ways in which you can adopt well-known engineering principles from DevOps and Agile practice as a way to add rigor and

consistency to your API traffic program. That includes references to test automation, continuous delivery and deployment practices, and even engaging in site reliability engineering (SRE) and chaos engineering as part of your traffic management practices. The chapters are arranged to focus on key aspects of every healthy API traffic program cutting across important practices, including the role of traffic management in your company (Chapter 1), types of traffic to consider and how to approach basic monitoring (Chapters 2 and 3), security concerns (Chapter 4), how to use traffic metrics to improve system resilience and scaling (Chapter 5), and how you can use your API traffic management program to support advanced efforts like SRE and chaos engineering.

Whether you are a veteran of network and performance monitoring or just getting your feet wet in the field, this book is designed to provide you important insight into patterns and trends as well as pointers to specific tools and practices that you can use to build up your own experience and grow an API traffic management practice in your own company.

This book is meant to be read straight through, but if you want to jump directly in at some point in the book, that's fine, too. I made sure to write up clear introductions and summaries so that, even if you don't want to spend time reading the entire book, you can get the big picture by skimming the table of contents and reading the beginning and end of each chapter.

Additional Reading

Most chapters have footnotes to point you to related material that is referenced throughout the text. Each chapter also has an "Additional Reading" section at the end. Here, you'll find references to handy books that expand on the concepts covered in the chapter.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

No project like this happens without lots of hard work and contributions from many sources. First, I'd like to thank the NGINX team for sponsoring the project and allowing me to be a part of it. Additional thanks to the folks at O'Reilly Media including Eleanor Bru, Sharon Cordesse, Chris Guzikowski, Colleen Lobner, Nikki McDonald, Vir-

ginia Wilson, and countless others. I am especially indebted to those who took the time to read my early drafts and provide important and valuable feedback to help me improve the text including Liam Crilly, James Higginbotham, Matthew McLarty, Scott Morrison, and copyeditor Bob Russell. Finally, thanks to Ronnie Mitra for creating the illustrations for this book.

CHAPTER 1

The Power of API Traffic Management

As Application Programming Interfaces (APIs) and their associated traffic become more common in organizations large and small, new challenges emerge in how to both monitor and manage this new data. For the purposes of this book, *monitoring* is the act of collecting and observing your API ecosystem's behavior (e.g., the number of API calls per minute), and *managing* is the process of analyzing, reaching conclusions, and acting on those conclusions (e.g., whether your API calls are unusual and what is needed to change that). As I travel around the world and talk to companies about how they are dealing with this new flood of data within their organization I note that most of them are struggling just to grasp the job of monitoring itself. And it is no small job.

In this chapter, we explore the challenge and the advantage of a broad-ranging API traffic management approach. This includes the work of monitoring traffic in order to collect data and improve the observability of your APIs as well as the notion of actively managing your API program using the insights gained from your traffic metrics (see Figure 1-1). Getting a handle on these two concepts will help you place API traffic management as a central pillar in your healthy API program and set you on the path to determining just what to track and how to use that information to improve your company's API program.

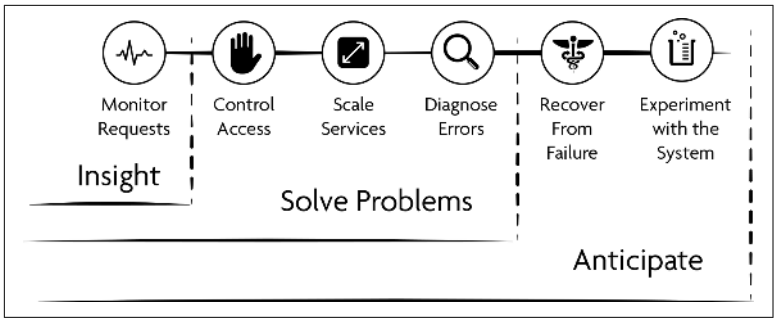


Figure 1-1. Progressing from monitoring to managing API traffic

Knowing what data points are worth monitoring can be a challenge. Just about any out-of-the-box traffic tooling will offer up values like CPU, memory, and disk space for individual machines as well as other values such as request/response length, message counts, and, usually, error rates. But these rarely tell you what you need to know. Instead, you often need to monitor not just the length and count of messages, but also the *types* of messages, their routes through our system, and the ultimate dispensation of any data sent or requested. And that's just the beginning of the role of monitoring in making your system more visible and your services more observable.

But, monitoring is not enough. Good traffic management means just that—managing the traffic you observe. That requires the ability to categorize some traffic as "good" or "bad" and having a sense of what "healthy" traffic looks like for your business. These are things no tool or service can offer. Instead, you need to apply strategic and tactical thinking to the traffic you see and then design data collection and displays that give you a broad sense of the health and welfare of your business as a whole. Like any complex system (and most API-driven ecosystems qualify as complex), turning raw data into valuable *information* means applying experience and judgment. And that is a level of API traffic management that can help you gain an advantage over competitors in your industry as well as give you an opportunity to anticipate important changes in customer habits and your market segment in general.

You can combine the two key perspectives on monitoring and managing your API traffic to gain business insight. In this chapter, you learn how to use Key Performance Indicators (KPIs) to identify and track important monitoring metrics and how to combine those with Objectives and Key Results (OKRs) to create a comprehensive high-

level view of your company's API activities and their effect on your overall business. Let's take a look at each of these in turn and see how KPIs and OKRs offer a vital perspective on API traffic monitoring and management.

Monitoring with KPIs

Monitoring is the act of observing; of collecting and collating data points that can help you make assessments and act on your judgment. But what do you monitor? Although the usual health of a machine's processor, memory, and disk is a start, the distance, for example, between the remaining space on a server's disk drive and the total sum of latency between components handling a single transaction spanning from your network to the system-of-record in a distributed storage installation is, pardon the reference, quite a long way. It is important to focus on observing key indicators of your ecosystem that can help you to track the progress of both your day-to-day online operations as well as the behind-the-scenes work of building, testing, and deploying updates to your API landscape. Especially as your system grows in size, variety, and complexity, correlating the effects of newly updated parts of the system with the character of daily business transactions can be a key to success.

The concept of performance indicators dates back quite a while. Business people have, almost since "the beginning," been looking for ways to identify and reward good performance and reduce unwanted elements. The notion of "key" performance indicators alludes to the idea that not just any metric or measurement will do when trying to get a proper view of your IT operation or your business itself. There are, essentially, *key* performance indicators that can represent important activity and processes. I dig into some common methods for identifying and collecting KPIs in Chapter 3. For now, it is worth pointing out there is more than one way to think about KPIs and how you need to use them in your organization.

Management Challenges

For many organizations it is still difficult to wrangle all of the disparate internal and external traffic into a coherent traffic platform. This is especially true for companies with offices around the globe. This is primarily a *management* challenge. As mentioned earlier, the growth of APIs and the explosion of small, focused services within a

company's ecosystem means the traffic between those services often represents a critical aspect of the health of not just your IT operations but also your day-to-day *business* operations. The quantity of API traffic and the quality of that traffic reflect as well as affect the quantity and quality of the company's business.

This means that managing the business requires an understanding of the types and meaning of your API traffic. Just as sales traffic monitoring and market intelligence are essential elements in top-level business decisions, understanding which API calls represent real revenue and which ones are indicative of your organization's place in the wider market of ecommerce is part of managing your business in the twenty-first century. In Chapter 3, we examine the details of how you can use API traffic monitoring to get a handle on the day-to-day operations of not just your IT operations but your business, too.

Traffic Challenges

The very nature of API traffic has been changing in the past few years. As more companies adopt the pattern of smaller, lightweight services composed into agile, resilient solutions, the amount of interservice traffic (typically called "East–West" traffic) is growing. Organizations that have spent time and resources building up a strong practice in managing traffic from behind the firewall to the outside world (called "North–South" traffic) might find that their tool selection and platform choices are not properly suited for the increased traffic between services behind the firewall.

Many traffic programs have focused on the very important "North–South" traffic that represents both the risk and opportunities of API calls that reach outside your own company and into the wider API marketplace. And this will continue to be important as more and more organizations come to rely on other people's APIs in order to keep their own business afloat.

At the same time, paying close attention to your "East–West" traffic will provide you key information on the spread and depth of your own service ecosystem and can help you better understand and even anticipate how changes in interservice traffic will affect your company's IT operations and spending.

In both cases, the added use of external APIs means that you'll always be dealing with dependencies on external services over which

you have little to no control. This can affect your own infrastructure choices, too.

Monitoring Challenges

Even when traffic is properly corralled and effectively routed, the landscape of monitoring this traffic continues to evolve over time. To gain the degree of observability needed to make critical decisions on how to grow and change your service and API mix, you'll need to be able to monitor at multiple levels within the ecosystem. *Proxy-level monitoring* gives you a "big picture" of what is happening on your platform. But you also need *service-level monitoring* to gain insight into the health and usage patterns for individual components in your system.

Often organizations start by focusing on monitoring traffic at the perimeter of operation. This *top-level monitoring* can act as a bellwether to let you know what kind of traffic is entering and leaving your company's ecosystem. This kind of monitoring is also usually "noninvasive." You can establish data collection points in gateways and proxies far from the actual source code of the services that handle this traffic. But there are limits to the kind of information proxy-level monitoring can tell you.

Especially in companies that rely on a microservice-centric implementation model, understanding what happens to API traffic after it passes your proxies and enters your ecosystem can be daunting. What you need to know is not just the general flow of traffic. Sometimes you need to know exactly where a particular request is going and how that requests is processed throughout its lifespan. Typically this means that you need to employ shared transaction identifiers—ones that are retained as the traffic passes through various network boundaries.

Microservice implementation eventually means leveraging monitoring information at the service level, too. You might not always be able to control the source code of all your services or be able to inject monitoring directly into a single service component. In that case, you need to find lightweight monitoring solutions at a smaller-grained level. We look at how you can do this in Chapter 3.

From Managing to Understanding

From actively managing to understanding traffic types to getting a handle on the types of monitoring access you need within your company, all these elements are critical to designing an API traffic control system that gives you a good picture of what is going on within your service ecosystem. But there is another perspective, too. Why are you monitoring your traffic? What is the connection between API traffic and business goals? And, if knowing your API traffic can help you know your business better, how does that work?

One of the ways that I see companies moving from monitoring to management is through the application of OKRs in addition to their KPIs.

OKRs

Along with the typical API traffic management practices around monitoring, securing, and scaling requests and responses, there is another perspective on your APIs—the business perspective. APIs exist only to serve business objectives. If they're not contributing to the fundamental goals of your organization, APIs are not properly aligned.

It is through the use of traffic management and analysis that companies can gauge the business-level success of their APIs. That means setting traffic metrics that matter for the business (number of customers gained, number of shopping carts abandoned before checkout, failed uploads per hour, etc.) and then using your traffic management platform to monitor and report on those key metrics. These metrics are usually based on your company's OKRs.[1] We dive into the details of how this works in Chapter 6.

Andy Grove's Management Objectives

OKRs were first described by Intel's Andy Grove, in his 1983 book *High Output Management*. They were motivated by the writings of Peter Drucker and the concept of "Management by Objectives." OKRs were Grove's way of applying management practice in an engineering environment. Google adopted OKRs within the first

1 "What are OKRs"

year or so of its founding, and it has been credited as being one of the important elements of the company's overall success.[2]

Your traffic management platform is a treasure trove of data spanning from the rhythm and stability of your internal deployment processes to the behavioral aspects of your API consumers. By investing in a robust API traffic management platform, you have—at your fingertips—an incredibly valuable tool for gaining insight into your company, solving the problems of your consumer audience, and even anticipating the needs of your customer base.

Gaining Insight

As API traffic—and the services and tools that are used to support and build them—continues to grow, it is important to have a handle on how your organization is spending its time and money to design, implement, and deploy API-based services. Most of this activity happens "outside" the typical production API traffic management zone. However, while teams are selecting which processes to automate via APIs and which interfaces will be most effective to add to your growing list of APIs and services, these additions need to be based on business-level goals and tracked by tangible metrics. Just as the programming side of digital transformation means increasing the amount of early test-driven development, the traffic management side of business demands more attention to establishing business metrics early in the process and baking those metrics into the deployment and monitoring process.

For the past decade or so, most organizations have been learning to rely on Agile, Scrum, and other models to shorten the iteration cycle of design, build, test, and release. And it is the last step in that process where a robust traffic management practice can yield results. This means combining your key traffic metrics with measurements that reflect your company's own internal behaviors around test and deployment cycles.

We dig deeper into techniques you can use to apply your traffic practices to your business in "Business Metrics" on page 60.

2 "How Google sets goals: OKRs"

Good API traffic management practice can improve the overall visibility of your system and allow you to better understand just *what* is going on throughout your organization. And, after you have a better sense of your system's activity, you'll have an opportunity to use that information to solve problems directly.

Solving Problems

As APIs become more prevalent within companies, they also become a more vital element in the success of the business. In its 2018 Tech Trends Report, accounting and consulting company Deloitte predicted: "Over the next 18 to 24 months, expect many heretofore cautious companies to embrace the API imperative."[3] When this happens more and more business-critical traffic is carried throughout the company via API gateways, proxies, and routers. The proxies themselves—the intermediaries between various services spread throughout the organization—become a key link in the process of completing business transactions, generating revenue, and reducing costs. In this case, understanding your traffic patterns gives you an opportunity to identify heavy API usage, recognize pockets of inefficiency, and work to solve these problems in ways that accrue to the company's bottom line.

For example, as your API traffic grows over time, some cross-service connections can become saturated with traffic, causing slowdowns, more denied requests, and a drop in business-critical transactions. Good traffic management systems will allow you to redesign traffic patterns to eliminate these bottlenecks; for example, when "hot spots" appear where traffic degradation is likely. In this case, that might mean standing up more instances of transaction-processing services in another location, adding routing rules that break up heavy traffic into separate zones, and route the transactions throughout the system in ways that balance the load and therefore speed processing for critical portions of your system.

And when you have reached the stage at which your API traffic management system affords you visibility into day-to-day operations and allows you to identify and remedy API traffic jams, the next stage is to design and build traffic management infrastructure that allows you to anticipate your system's traffic needs.

---

3 "API imperative: From IT concern to business mandate"

Anticipating Needs

The state of most API-driven systems today is often based on piecemeal growth and haphazard management. Getting a handle on your organization's API traffic means improving overall visibility as well as learning to solve business problems through the routing and shaping of the very traffic your APIs depend upon. The next level of traffic management is *anticipating* the needs that are likely to arise and providing solutions before the problems are significant enough to be noticed.

An excellent way to do this is to conduct *runtime experiments* on your ecosystem. The ability to safely and consistently create and test your traffic hypothesis is something we explore in "Runtime Experiments" on page 66.

Summary

This chapter covered both the challenge and the advantage of a broad-ranging API traffic management approach. The work of *monitoring* traffic with meaningful KPIs in order to collect data and improve the observability of your APIs means that you'll need to deal with the challenges of taking control of the monitoring process, focusing on the different *types* of API traffic you experience (e.g., "East–West" as well as "North–South") and dealing with the challenges of collecting data in the proper locations (network-level proxies as well as service-level components).

You also learned how to move beyond monitoring API traffic and into actively *managing* your API program. Well-designed and implemented API traffic management can give your organization the opportunity to gain insights into daily business activity, identify and solve traffic problems before they grow to a critical level, and even anticipate needs within your own ecosystem and your business market in general.

In the next chapter, we look at the basics of traffic in general and how your knowing the types of traffic you're dealing with can help you determine just how and why you need to establish your IT ecosystem's KPIs and relate them to the OKRs of your overall business.

Additional Reading

- Continuous API Management, Medjaou et al.
- Production-Ready Microservices, Fowler
- Building Evolutionary Architectures, Kua et al.

CHAPTER 2
Managing Traffic

In this chapter, we get a chance to look at just how you can model your traffic patterns and how those patterns can help you to focus on what's important.

The two approaches covered here are controlling external traffic (also known as the North–South model) and optimizing for internal traffic (called the East–West model). Both of these approaches have their place, and most companies need a mix of North–South and East–West configuration and controls in place in order to both protect your network from external problems and support your internal service ecosystem as it grows and changes over time.

The North–South approach is typically associated with traditional monolithic implementations. However, as more and more microservices are entering company IT operations, the North–South model may not always be the best options. Let's start by exploring the North–South model first, and then we can deal with the alternative.

According to the "Cisco Global Cloud Index" (updated in late 2018), the East–West (server-to-server) traffic was estimated to reach 85% of total datacenter traffic by 2021. The total portion of North–South traffic (traffic exiting the datacenter) is expected to amount to about 15% of the total traffic. Despite this lopsided distribution, North–South traffic has received the bulk of traffic management's attention in recent years due to the increased vulnerabilities and lack of predictability when dealing with traffic that originates (or terminates) outside your own network boundary.

We devote time to both approaches in this chapter.

Controlling External Traffic

The most common way to diagram API traffic is in what is called the "North–South" pattern. It looks similar to the diagram presented in Figure 2-1. We've probably all seen a diagram that looks like this. At the top of the image is the world outside your network; that's where the API requests originate. These incoming requests then move "south" through your API gateway—a gateway or cluster that performs various tasks to inspect, shape, and route the incoming request—on the way to the target destination where some work is performed. After that work is done at the component level, the resulting API response makes its way back up "north" to finally pass through the perimeter and back out in the world beyond your network.

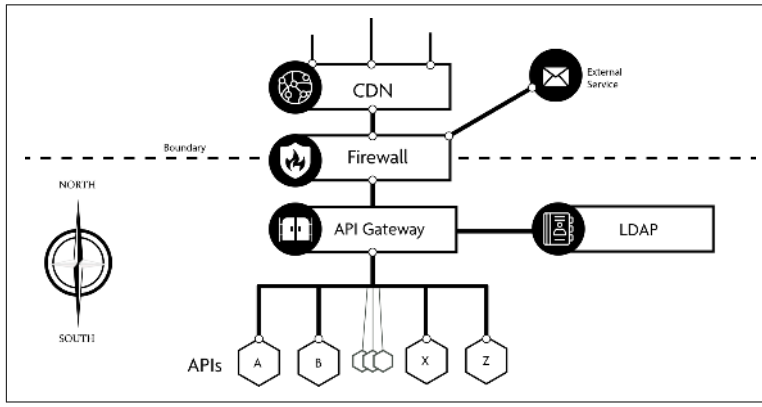


Figure 2-1. Example of north-south traffic

This North–South model focuses on the traffic that comes from "outside" your domain or network zone. That traffic is, by default, unknown and untrusted. For that reason, much of the North–South model emphasizes the work of validating the message, authenticating its requester, filtering, shaping, and finally routing the request to the proper place.

This is especially true in a monolith-style implementation for which all the incoming message processing (authentication, routing, and transformation, etc.) happens only at the outer boundary of the network. However, there are other cases for which the North–South

approach falls short. We look at that possibility in the next section (see "Optimizing Internal Traffic" on page 18).

Crossing Boundaries

A critical aspect of North–South traffic management is the work of "crossing boundaries"—passing data safely and efficiently from behind the firewall out to other domains, datacenters, or the cloud. The details of sharing data securely (e.g., authentication, authorization, and encryption) when crossing from one system to another is important, and we cover it in Chapter 4. It is also important to focus on efficient data sharing across domain boundaries. Usually, cross-boundary application-level protocols like HTTP focus on interoperability at the expense of efficiency. Effective traffic management makes sure to avoid unnecessary boundary crossings in order to limit the loss of efficiency when sending data outside the firewall.

Your network perimeter is typically not the only boundary that needs to be managed. You might have separate zones for mobile, web, partner, and other kinds of traffic that you need to manage. It is at the zone boundaries that traffic is identified, shaped, and routed to the proper location within your network. What follows is a quick review of typical responsibilities when handling this kind of traffic pattern as well as a few challenges.

Typical Responsibilities

Along with the notion of enabling cross-boundary access, there are a set of common tasks associated with North-South API traffic management. These tasks are usually handled at the boundary perimeter by API gateway servers. But the work can also be done at various points within the network. Following is a quick rundown of typical responsibilities for this class of API management.

Authenticating ingress

One of the first tasks for handling inbound (or *ingress*) API traffic is to associate an identity with each request. Even though robust authentication is unnecessary, API requests usually have some assigned identifying key as part of the request metadata (e.g., HTML headers). You can use this identifying information to track the progress of the request through your system, and, when needed, you can use it to limit access, throttle requests, and more.

After the request is authenticated, it needs to be *authorized*. It is important to keep in mind that *each request*—even requests from the same API identity—needs an access control check. The same request identity is likely to have differing rights levels for differing URLs. The access control might even be dependent on the type and location of the device making the request (e.g., a never-before-seen mobile phone connecting from a country not normally associated with the requester's identity). For more on access control details, see Chapter 4.

Even in cases for which a unique identity or shared API key is not required to access the network, some form of identifying data should be *assigned* to each request. This will ensure that all requests can be tracked and managed.

Finally, the process of authenticating and authorizing inbound messages can also inject additional metadata into the request such as the identity of the server doing the authentication, date/time data, information on the initial location of the incoming request, and so on.

Limiting ingress

Another important task for perimeter gateways is limiting inbound traffic. It is a good idea to have thresholds for all API traffic. These thresholds can take the form of request limits or quotas for inbound traffic associated with a particular identity.

Gateways can enforce traffic limits based on request/response counts and/or payload size. These limits can be associated with request identity, request targets or routes, or just boundary level request counts. And this associated data is usually computed over a "time-window" (per minute, hour, day, etc.). This means that gateways need to maintain persistent data covering all of the managed request parameters (count, size, time-window, identity, etc.).

For more on dealing with threshold and limits, see Chapter 3.

Routing ingress

The work of routing incoming requests is also another key task for API gateways. Typically this is handled by writing declarative rules that run at the boundary gateways to inspect incoming request metadata (URL, HTTP request method, HTTP header, etc.).

Although routing might be just a simple scan matching the URL with an existing rule, API traffic platforms might also be tasked with rewriting the URL to convert it from a simple external address (`api.bigco.com/customer/1q2w3e`) to an address that is better understood by services within your internal network (`south-region.bigco.com/services/cust/v2?id=1q2w3e`).

API gateways might also inject request metadata such as timestamps, information identifying the proxy that performed the routing, and so on.

Transforming ingress

One more task that API gateways might perform is to request body transformation. For example, an incoming request might be formatted as a JSON message, but the request is targeted at a service that only understands XML-formatted bodies. You can use a gateway to transform that JSON into XML before passing it along to the target service.

Another kind of transformation is one that deals with minor differences in layout and content due to changes in payload specifications over time. For example, a request might have arrived marked as version 1 of the customer update payload. However, the internal component that is running in production supports only version 2 of the customer payload. If the differences are minor, an API gateway might be tasked with modifying the incoming request to conform to the backend component's needs.

Dedicated to Transformation

Payload transformation is a costly and sometimes buggy process. One way to deal with the challenges of payload transformation is to stand up a dedicated transformation service within your network. This could be a shared service that can be called upon to convert a message body based on rules or scripts stored at the transformation server. This will allow you to optimize the transformation work and even customize it for your own specific use cases. Using a dedicated service within your network can also make it easier to test, deploy, and manage transformation scripts. Finally, it will be easier to monitor and isolate errors introduced through transformations, too.

Like other tasks, transformation proxies can also inject additional metadata into the request, including a reference to the original formatted payload.

Transforming egress

Just as you might be transforming the payload of an incoming request (ingress), you can also transform outbound responses (egress). Using the example from the previous section, you might need to write a transformation rule to convert internal XML payloads into JSON before passing the response out of your system. You might even need to modify the layout slightly to support the initial API requester's specifications.

As mentioned earlier, payload transformation can be time-consuming, complex, and error prone. Whenever possible, you should limit the use of transformations as part of your API traffic management. When you need it, use a dedicated transformation service as described in "Dedicated to Transformation" on page 15.

Filtering egress

You might need to filter the outbound message to remove some payload and/or header metadata before it leaves your network. For example, your message might include several metadata fields used to track the request internally, identify it for security and access control needs, or other things that should not be seen outside your boundary.

It might also be necessary to remove or modify data in the payload for security reasons. Like the topic of transformation, filtering payload data can be tough to program, control, and debug, so it is best to not rely on your north-south gateway system for this work and instead use dedicated services (see "Dedicated to Transformation" on page 15).

Encoding and encrypting egress

One more task common for north-south traffic is encoding the outbound message. There will be cases for which the message needs to be encrypted (based on request/response metadata) for security reasons. You might also be able to compress the payload (if the requester supports it).

In both cases, after the message is encoded, it will be difficult to make additional changes (filters and transformations), so you should perform this step only after any required changes to the payload or metadata of the message.

Caching

One more step that you should consider is caching the outbound response for later replay. This can improve the perceived performance of your network services and reduce load on the individual components within your system. I cover caching in more detail in Chapter 5. For now, you should keep in mind that you can take advantage of shared cache specifications like those in the HTTP protocol in ways that can reduce both your internal traffic (when you replay the cache response to new requests) and the overall network traffic (when you mark the response cachable by the party receiving the response).

Common Challenges

Along with the long list of common tasks performed by the traffic platform in a North–South model (see "Typical Responsibilities" on page 13), there are some challenges. Let's take a moment to highlight a few of these before taking a look at another possible approach for modeling and managing API traffic.

Long routes

One challenge for the North–South model is that all of the important traffic management is located at the boundaries. This works fine if you need to deal with the request only *once* when it enters and *once* when it leaves the network. This is usually the case for monolith-style implementations. However, as you add more services to your network and as those services start to call each other more often, you'll find that you need traffic management capabilities in many parts of the network.

Creating more "zones" within your network boundary can help with this problem, but the nature of most cross-boundary designs is that they are usually inefficient in handling simple traffic flow. This is because the North–South model is generally optimized for secure boundary crossing, not fast message passing.

Central failure points
: Another problem with the North–South approach is the reliance on a single perimeter of traffic services. If this boundary service fails, it will have wide-ranging negative effects throughout your entire network. This problem can be alleviated by creating gateway clusters and using other high-availability techniques, which I cover in Chapter 5. However, combining the possible single point of failure with the growing intercomponent traffic can add up to a unhealthy mix of internal and external traffic all passing through the same traffic platform servers. Again, adding zones (e.g., internal-gateway and external-gateway) can help, but it adds complexity and continues to rely on inefficient boundary-crossing implementation patterns.

Reduced observability
: The last drawback that I cover here is the challenge of reduced observability on what is actually happening to API traffic as it travels the network. First, many of the tasks cited in the previous section can be performed on a single instance of the perimeter gateway (or within a cluster). Although it is possible to log each task separately for later analysis, the logs can grow large quickly, and it can become quite complicated to parse out just the activity related to filtering, transformation, and so on. Relying on separate running traffic services (authentication, access-control, transformation, etc.) can definitely help in this area.

So what is the alternative? It turns out that you can overcome many of these challenges by implementing a different traffic management design, one based on optimizing traffic *between* components operating within your network boundary. This approach is known as an East–West model, which we look at next.

Optimizing Internal Traffic

Another important pattern in managing API traffic is the work of handling internal or component-to-component traffic. This is often called *East–West* traffic because most diagramming of API traffic shows server-to-server traffic running from left to right. For example, in Figure 2-2, you can see that component A (`user-management`) communicates with component B (`data-services`), which communicates with the data storage server.

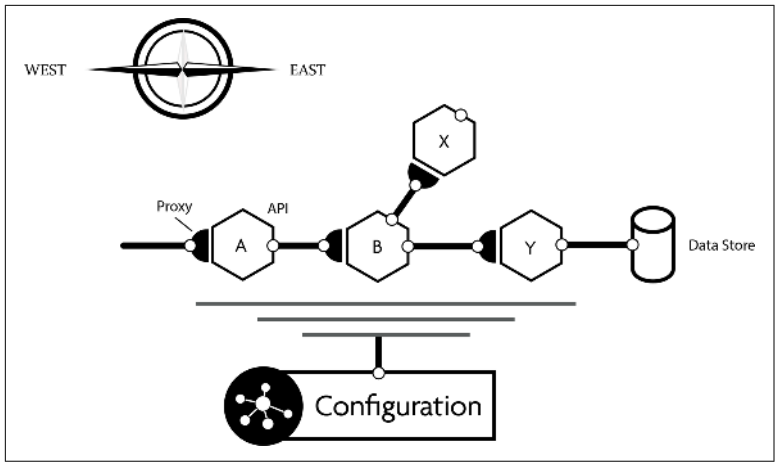


Figure 2-2. Example of east-west traffic

In this diagram, the component traffic is illustrated as running "east and west" on the page. In reality, the cardinal direction of the traffic is not important. And there are lots of diagrams that show internal traffic running "north and south." The key element is that the traffic is focused on cross-component connections. These connections are almost always within the same subsystem, too. That might mean components running on the same server instance (e.g., within a single virtual machine or container). Internal traffic might also be running between machines in the same server rack or even within the same physical datacenter. There are also cases in which the internal traffic runs across datacenters over a dedicated line—as part of a virtual private network (VPN).

Another key aspect of east-west traffic is that it often runs behind the company firewall and, for that reason, can run without extensive identity and access control metadata. This can be a big problem because, in the best cases, the API traffic usually lacks important metadata needed to properly monitor and manage that traffic. At worst, the traffic is running without sufficient security metadata, which can make it easier for someone to spoof or simply bypass important security controls. We talk more about the security aspects of API traffic management in Chapter 4. For now, let's focus on the details around observing and supporting intercomponent traffic behind your company's firewall.

Enabling Services

One of the important jobs of managing east-west traffic is to optimize service-to-service communications. That means making it easy to *find* services within your system. It also means making it easy to *bind* to services together—for one service to call on another service in order to solve a problem. It also means streamlining the actual service-to-service communication; in other words, the data packets that are sent back and forth.

Most of the work of *finding* services behind the firewall is done by humans at design and build time using a catalog or registry. The service identifying information is then added to configuration files and passed to the build stage where it can be resolved when the component is deployed into the ecosystem. That's the *bind* step; resolving the identifiers in configuration files into actual pointers to running services. This name resolution works very much like Domain Name System (DNS) services at runtime. Platforms like Kubernetes, Mesosphere, and others are often good at this name resolution work.

A common way to optimize component-to-component traffic is to use a binary, strong-typed communication protocol such as Apache Thrift or general-purpose Remote Procedure Call (gRPC) over Transmission Control Protocol (TCP)/Internet Protocol (IP) or User Datagram Protocol (UDP). Thrift and Avro work best for communications that stay *behind* the firewall because they do not provide built-in support for security metadata. You can, however, add security and other metadata into the Protocol Buffers (Protobuf) object definitions used by Thrift and Avro.

Good API traffic management systems allow you to place proxies throughout your organization to help ease the process of finding and binding services as well as optimizing the traffic between them.

Typical Responsibilities

A key difference in the way East–West traffic designs differ from the way North–South traffic is handled is that, for the internal (East–West) model, there are usually many more instances of traffic management proxies and gateways to deal with. They are usually more lightweight and reside close to the services that are meant to protect and support. This will have a number of advantages (and a couple challenges).

Also, you'll find this list is a bit shorter than the one for North–South tasks and that some of the responsibilities listed here are the same (or quite similar) to those in the North–South list. However, the minor variations in the list point out big differences in the way East–West traffic is handled.

And that starts with the way security is approached in the internal model.

Validating Identity

As in the North–South approach, one of the first things East–West traffic proxies deal with is identity. However, in the East–West model the proxies don't usually *establish* the identity of the requester. Instead they are designed to *validate* the request's identity. Lightweight proxies can pick up the identity token (often a header in HTTP or some other message metadata) and make a call to an internal service that can check the validity of that identity token.

As long as the token is still valid (or is automatically refreshed by the validation service), the request can continue. If the validation fails or the token is missing, the API gateway can reroute the request to a nearby perimeter server for full authentication.

Good API traffic programs will make it quick and easy for internal requests to be authenticated and/or validated to optimize for speedy handling of the within-network request.

Controlling access

In the North–South model, access control information was loaded at the perimeter and followed the request down and up through the network. This works fine for monolith-style systems but can be problematic for microservice-style implementations.

In the East–West model, local proxies concentrate quite a bit on resolving access control and doing this close to the service it is meant to protect. A common approach is to apply an implicit access control token at the perimeter—one that points to a dynamic list of access claims associated with the validated identity. When a request arrives at the local proxy, that proxy makes a request to an access control service that resolves the rights list just for this one service. This results in a more accurate reading of current rights and a smaller, faster payload in response. I review the details of this

implied access control in "Managing Access with Tokens" on page 40.

Ensuring ingress

Another very important task in the East–West model is protecting individual services from inbound traffic surges and network-level problems. Local proxies can be set up to do their own routing and optimizations specifically for a single associated service. This is sometimes referred to as the *sidecar* approach. Each service is deployed to run *behind* one of this small traffic proxies, and it is the proxy's responsibility to inspect incoming traffic, throttle it if needed, reroute traffic to other instances of the service if this instance is failing, and do other similar problem-solving work. I'll show you a list of things these "smart proxies" can do in Chapter 5.

Transforming ingress

It is also possible for these small proxies to do some transformation work on incoming messages. However, the same drawbacks apply, too. While local proxies can be customized to perform detailed message transformations for the associated service, this work doesn't scale well over time and is best handled by a standalone transformation service before the request is routed to this proxy.

Transforming egress

It is also possible that the service response needs some transformation work done. And, again, this is best handled by a standalone transformation service (see "Dedicated to Transformation" on page 15).

Caching egress

While transformation is best handled elsewhere, these smart proxies are great places for dealing with caching of responses. The proxy should be set up to mark responses using astandard HTTP caching directive and, in some cases, might want to keep its own cache of common responses to play back when the same inbound request appears. That can increase reliability *and* reduce load on the service at very little added cost to set up and maintain your traffic platform. We look at this in more detail in "Caching" on page 54.

Common Challenges

Just as in the North–South model, there are cases where the East–West approach will meet challenges such as 1) having to deal with lots of proxies in your network, 2) getting comfortable with lots of very flat (point-to-point) internal routing, and 3) dealing with the management burden of all your additional proxies and associated rules. Let's take a closer look at each of these:

Lots of proxies
: One of the more obvious challenges to working in the East–West model is that it results in many more proxy machines all over our network. This is especially true when you have lots of microservices running internally *and* you adopt the sidecar approach of pairing each service (or service cluster) with a traffic management proxy. However, local sidecar proxies are typically very lightweight and usually relatively easy to program and monitor. That helps in dealing with the added instances on your network.

Flattened routes
: Another noticeable change is that you're likely to see lots of point-to-point (East–West) traffic between internal components. In fact, another name for the East–West approach with local smart proxies is a *service mesh* because the traffic starts to look like a mesh or web rather than an orderly north-south traverse. There are lots of infrastructure products dedicated to designing and implementing these mesh-like networks, too.

Increased management
: Finally, as the first two drawbacks imply, having more machines and more traffic means more management work for your traffic platform. However, with so many emerging products dealing with service mesh and microservice security and traffic routing, there are quite a few options that you can explore to find the tooling that is right for your network mix. And most all of these new products adopt an engineering approach and follow the DevOps-style continuous deliver/deployment approach. That can do a great deal to improve the reliability of your system in a scalable way. We examine some examples of this in Chapter 6.

That covers the East–West approach to building out your API traffic platform. As you add more and more services to your ecosystem

and enable more and more internal traffic, the East–West model that favors optimizing component-to-component interactions can greatly improve both the reliability and the scalability of your network.

Summary

This chapter covered the two common models for handling API traffic in your network: the traditional North–South approach, which focuses on securing network boundaries against untrusted traffic; and the East–West approach, which works to optimize internal service-to-service interactions. As you might guess, a healthy ecosystem has a mix of both approaches. A typical scenario is that companies start by focusing on protecting their perimeter and then, as their internal ecosystem grows, they add more East–West-style traffic support to the network.

Now that we covered the basic types of traffic management, the next chapter goes into the details of just *how* you can start tracking and collecting monitoring data to help you get a better picture of your current network traffic and where you can spend the best time improving the safety and reliability of your service ecosystem.

Additional Reading

- SDN, Gray & Nadeau
- Cloud Networking, Lee

CHAPTER 3

Monitoring Traffic

In this chapter, we drill down into the basics of traffic monitoring. This means understanding the typical metrics we can monitor in order to understand that overall health of our system. We also spend some time reviewing some common traffic formulas that help you better understand the runtime condition of your service ecosystem. With these elements in hand, we can custom-build suites of monitoring metrics that will assist you in observing and monitoring the API traffic that's important to your business.

Rather than just focus on low-level metrics like CPUs and memory, it is also important to monitor service-level metrics (e.g., number of queries processed per second) and even business-level metrics (e.g., number of new customers added this week). Being able to separate monitoring into levels helps us better focus on getting the right metrics into the right dashboard at the right time.

Monitoring Levels

An important aspect of monitoring is to ensure that you know what level you are observing and what kinds of information you want to collect at that particular level. A useful way to think about levels is to carve up your observation space into three levels: infrastructure, service, and business, as illustrated in Figure 3-1.

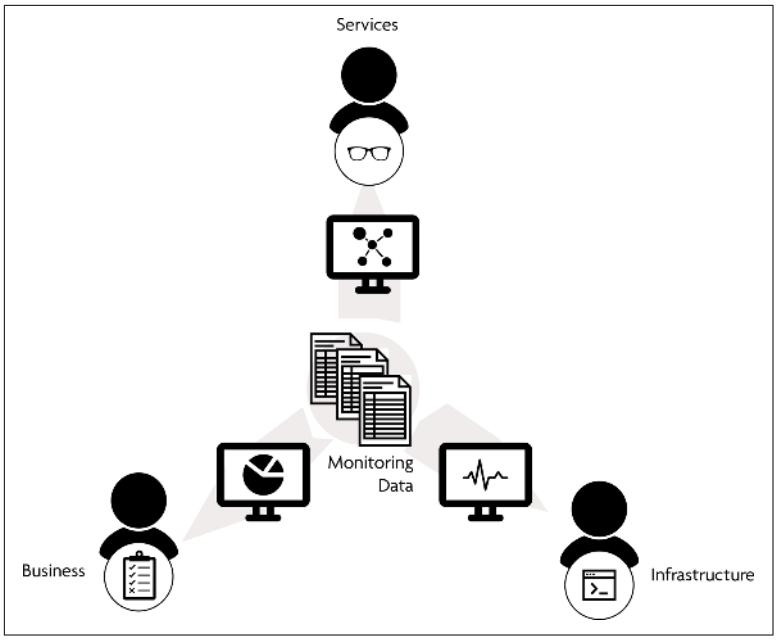


Figure 3-1. Three traffic monitoring levels

Infrastructure Metrics

At the infrastructure level, the focus is on the health of the hardware and basic software elements (memory, queues, threads, etc.). Infrastructure metrics focus on basic values such as CPU, memory, and disk space at the hardware level. Knowing when your infrastructure is operating above or below normal trends is an important way to diagnose, possibly even predict, problems within your system.

Service Metrics

At the service level, the focus is on whether the important tasks of that component are in good shape (data reads, writes, files created, etc.). This level of monitoring helps you to understand the health of a particular service or, in the case of monoliths, instances of your application. It also allows you to focus on critical service-level metrics beyond generic reads and writes.

For example, you might have released an updated service component to improve throughput and reduce latency for handling complex computation for your online investment service. To confirm

that your update really *does* reduce latency, you can establish a metric for a single endpoint within your service. This metric can help you better understand the payback on your company's efforts to improve performance within your ecosystem.

Business Metrics

Finally, at the business level, the focus is on key metrics for the company (new customers, whether the latest marketing campaign is actually increasing sales, etc.). These kinds of metrics can provide details about fundamental process flows experienced by end users, partners, and other parties critical to your company's overall success.

You might, for example, want to track the average amount of time it takes for a new customer to complete their onboarding process. You can select metrics that focus on all the transactions in your system that are involved in onboarding and monitor them in real time and, when needed, adjust your services and routing to make sure you can reduce the time it takes to improve completion rates and increase monthly onboarding of new customers.

So, what types of metrics can you use to monitor at all three levels? It turns out there is a small set of metric types that you can focus on to help you gain the insight you need to manage your API traffic.

Typical Traffic Metrics

Just a small handful of traffic metrics can provide you with a wealth of data to use when managing and diagnosing the health of your system. These metrics fall into a few simple categories: rate limits, request latency and duration, and error rates.

By monitoring the rate at which traffic flows (and where it becomes stuck), the typical time it takes for requests to be processed (latency and duration), and the rate at which errors occur in the system, you can get a solid picture of the strengths and weaknesses within a running system.

Rate Limits

Rate limiting is a way to control the pace of requests in a system. The request-handling limit of a system is sometimes called the *saturation point*. There are actually two types of rate limiting to consider

when establishing and enforcing limits. The first type of limiting focuses on the rate at which a backend process (the service receiving requests) can successfully process requests. The second is the rate at which a frontend process (the service sending requests) is permitted to initiate requests.

All services have a limit to the speed at which they can successfully handle requests. After that point, the service begins to fail in some way. It might be unable to respond in an adequate amount of time, causing other parts of the system to slow or fail. Or, the service might start to run out of resources like memory or even local disk space. In the worst case, the service will simply crash abruptly, possibly ruining stored data.

To prevent these kinds of problems, you can establish a backend rate limit. With backend rate limiting, the service must protect itself against an unhealthy flood of requests from other parties. When the service itself does not have this protection built in, it is important to place that service behind a proxy or gateway that can provide the same protection.

Within a service, a common way to manage rate limits is to simply monitor requests per second. Additionally, if the service is using some type of inbound queue for requests, you can monitor the depth of the queue.

Request Latency and Duration

The latency of a request is the time it takes to complete a request. This is sometimes called *duration*, too. In both cases, the measurement is based on the cost of the wait time and the execution time needed to complete the action. Depending on how your system is designed, it is important to know what portion of your duration is due to execution and what portion is due to waiting or routing costs.

Request monitoring is usually done at the proxy or gateway level because that is where the "total cost" of the request is most easily computed. This is especially true in microservice-style implementations for which it can be difficult to trace and correlate all of the hops of a request from start to finish.

Monitoring the average request latency/duration (in milliseconds) can help you identify where requests are stalling, figure out which services are eating up most of the execution time, and provide clues

on how you can improve your network and service topology in order to reduce latency in your system.

Error Rates

Tracking errors in your system is an important element of traffic management. Aside from the annoyance that errors bring to a running system, they can also wreak havoc on your monitoring efforts all by themselves. Therefore, it's wise to keep a close eye on errors as a separate category along with rate/saturation and latency/duration metrics.

Here are some examples of how error requests can skew your rate limit and latency monitoring:

Fail-fast errors
: When errors occur on the service side due to missing network connections or badly formed internal request payloads or responses, your overall request duration can appear to be quite short. Requests that "fail fast" can give you the false impression your system is running very efficiently.

Slow connections
: At the same time, errors that are caused by slow connections or mistakes in routing can add time to your request latency and lead you to think your *service execution* is taking too long.

Flooding
: Finally, a high rate of malformed requests can confuse your rate-limiting and saturation metrics. This can lead to misdiagnosing load management and rate prediction.

For these reasons, it is important to monitor and report errors separately from, or at least *in addition to*, your typical rates and duration metrics.

Other Considerations

Along with rates, duration, and error monitoring there are a few other things to keep in mind when setting up your monitoring metrics. One of them is how to deal with proactive rate limits in general, and the other is how to approach the notion of reporting metrics in averages or percentiles.

Throttling, bursting, and quotas
: A common monitoring practice is to apply proactive rate management through the use of throttling, burst management, and/or request quotas. These are all ways to protect your system from experiencing over-saturation and the resulting increased request latency. Throttling is also a "friendly" way to lessen the impact of reduced access for a single requester (end user, partner, peer service, etc.). *When using your traffic management platform to deal with these types of problems, be sure to keep in mind that your proactive measures have the effect of skewing your monitoring results, too.*

Averages versus percentiles
: Most monitoring and reporting operates on *averages*. Data is collected over a time window (e.g., every minute or so), and then that data is reduced to a single number that is usually the average of all the values. That works fine when *most* of the data points are within a small range. However, when a small set of data is wildly out of the norm, reporting the average can hide important outliers.

  One way to deal with this masking of outliers is to report *percentiles* instead. Percentiles give you more information about the distribution of the requests, not just the general average. This helps you identify rare outliers that sometimes plague a system.

If you're interested in learning more about the pluses and minuses of using percentiles in your monitoring, check out the blog post "Everything You Know about Latency Is Wrong."

Now that you have some ideas about what metrics can be helpful, let's talk about how to collect those metrics into *traffic formulas* that you can use to derive some conclusions about the health of your API traffic.

Common Traffic Formulas

After you know the kinds of things that you can access as monitoring metrics *and* you have a notion of the types of metrics you want to observe (infrastructure, service, and business level), it makes

sense to work up patterns or formulas for creating sets of metrics that allow you to collect data in order to derive results.

Traffic formulas are ways to tackle the monitoring problem in an organized way. They provide a kind of checklist you can use to identify values to observe. Instead of saying "you should monitor the following elements," traffic formulas guide you to think about how to solve a problem with monitoring and how to think about which values to consider when creating your monitoring suite.

Let's review three rather popular traffic formulas, their origins, and their strengths: USE, RED, and LETS.

The USE Method

The USE (Utilization, Saturation, and Errors) formula for system monitoring was developed by Brendan Gregg (formerly of Oracle, now with Netflix) around 2012. As he explains it, "I developed the USE method to teach others how to solve common performance issues quickly, without overlooking important areas."[1]

Gregg tells us that USE is an aid for asking three questions about any part of the system you want to better understand: "For every resource, check utilization, saturation, and errors."

A *resource* is any physical component such as CPUs, disk space, I/O bus, and so on. By *utilization* Gregg means the average amount of time (as a percentage) the resource is busy doing work. An example would be the percentage of time a component was in a `processing` state versus when it was in an `idle` or `waiting` state. For some resources (e.g., CPUs) utilization might be expressed as a percentage such as 100% utilization of the CPU, and so on. The *saturation* of a resource is the degree of `extra work` that is backing up for that resource. Examples of saturation include a large number of requests waiting to be handled, too many files on disk awaiting processing, and so forth. Finally, the *errors* element is simply the count of error events for that resource (usually computed over a sliding window as in `errors per second`, etc.).

1 *http://www.brendangregg.com/usemethod.html*

NOTE

Check out Brendan Gregg's website for more on the USE method and how you can apply it.

Although Gregg initially wrote about what we identified as hardware metrics (CPUs, memory, storage, etc.), he points out that we can also apply the USE method to software such as threads, locks, file handles, and more.

I find that the USE method works very well for infrastructure-level metrics (see "Infrastructure Metrics" on page 26). For more component-level monitoring, I prefer another method: the RED method.

The RED Method

RED (Requests, Errors, and Duration) was developed by Tom Wilkie of Weaveworks around 2015 and is discussed in detail on the Weaveworks blog. The RED method was created, as Wilkie tells it, as a kind of response to Gregg's USE method. In particular, Wilkie said that the increased use of small services (e.g., microservices) made applying the USE method more difficult.

Because Wilkie was interested in improving monitoring for ecosystems that relied on lots of small services, his RED method focuses not so much on *resources* but instead on the *requests* handled by the *services*. Essentially, this is the RED approach:

> "For every service, check the request rate, request errors, and request duration."

It is pretty easy to see that RED is aimed at the service-level metrics we discussed earlier in the chapter (see "Service Metrics" on page 26). What is interesting is that Wilkie uses the requests that travel between services as a way to build up his monitoring suite. That reflects some important realities of microservice-style ecosystems where components might be installed at any time during the day, written in multiple languages, and using a wide range of deployment tooling. As Wilkie says in his GOTO talk, "In the old [world] your primary key in your monitoring system was *host* metrics. In the new world the primary key should be *application services*."

NOTE

You can learn more about Wilkie's RED method by watching his 2016 GOTO Stockholm talk, "Monitoring Microservices."

An important point that Wilkie makes is that his RED method is not designed to be applied to each individual service. Instead, you should apply this approach to your overall system. The real value of monitoring requests is following those requests throughout the system from initial entry to final response.

Although USE is good for code-centric and low-level monitoring and RED is good for request monitoring at the proxy or gateway, there is another approach that blends a bit of both of the two: the LETS formula from Google.

The LETS Method

The LETS (Latency, Errors, Traffic, and Saturation) formula from Google was developed around 2003 when Google created its site reliability engineering (SRE) program. We learn more about Google's SRE program in "Site Reliability Engineering" on page 67. LETS is a mix of code/machine-centric and service/request-centric metrics. The LETS values are sometimes called "The Four Golden Signals."

In fact, in the O'Reilly book *Site Reliability Engineering*, they say:

> "If you can only measure four metrics, focus on these four: Latency, Errors, Traffic, and Saturation."

The *Latency* value represents the time it takes to service a single request. It is important to treat failed requests separately. Failures due to missing connections might end quickly, and failures due to lack of memory might take several seconds. The *Traffic* metric measures the demand on your services, such as the requests per second for a service. The *Errors* value represents the number of failed requests, both at the system level (e.g., HTTP 500) and at the component level (e.g., internal failures). And the *Saturation* measurement is used to track the pressure your services are under at runtime. In systems for which memory is important, you can track the use of memory. In systems for which request handling is critical, you can monitor the length of incoming request queues, and more.

For more on LETS and other SRE topics, check out Google's free HTML version of its *Site Reliability Engineering* book.

The LETS formula is a good model for cases in which you want to be sure to cover a kind of *minimum-viable* set of metrics. Cover each of these four, and you're likely to catch most problems. However, LETS doesn't really give you as much guidance on how to troubleshoot internal issues (like USE) or general external issues (like RED).

Summary

In this chapter we covered three key topics related to monitoring API traffic. First, we covered monitoring levels. Understanding the various monitoring levels (infrastructure-, service-, and business-level) helps us better focus on selecting metrics that will help us better understand and manage our API traffic.

Second, we covered typical traffic metrics. There are a small set of metric types such as rate limits, latency, and errors that we can use to quantify and observe API traffic in our system. Finally, we reviewed three common traffic formulas: USE, RED, and LETS. Each has its strengths and weaknesses, and knowing what they are helps you to determine when and, most important, where to use them when designing your API traffic-monitoring system.

Now that we've covered the basics of traffic monitoring, in the next chapter, we review several security-related aspects of API traffic management, including security basics and the challenge of access control in distributed systems.

Additional Reading

- Lightweight Systems for Real-Time Monitoring, Newman
- Practical Monitoring, Julian

CHAPTER 4

Securing Traffic

A comprehensive API traffic management system includes robust security features. This means a reliable authentication system as well as a scalable authorization strategy. Each aspect of security (authentication and authorization) is essential for a healthy API ecosystem. In this chapter, we cover API security basics such as API keys, authentication, authorization, and encryption.

Access control (or authorization) is a particularly important security element in API systems that rely on microservices. As your service collection grows and becomes more adaptable at runtime, it becomes increasingly difficult to know—ahead of time—just which services your request is likely to encounter. We devote some additional time in this chapter on designing and implementing a scalable and reliable authorization system based on access tokens.

Security Basics

The basics of API security (see Figure 4-1) center on authentication (the requesting identity) and authorization (the identity's access controls for this request). API keys are another important element of API security because they help identify API usage independent of the requesting identity. There is also the matter of data encryption for messages in transit.

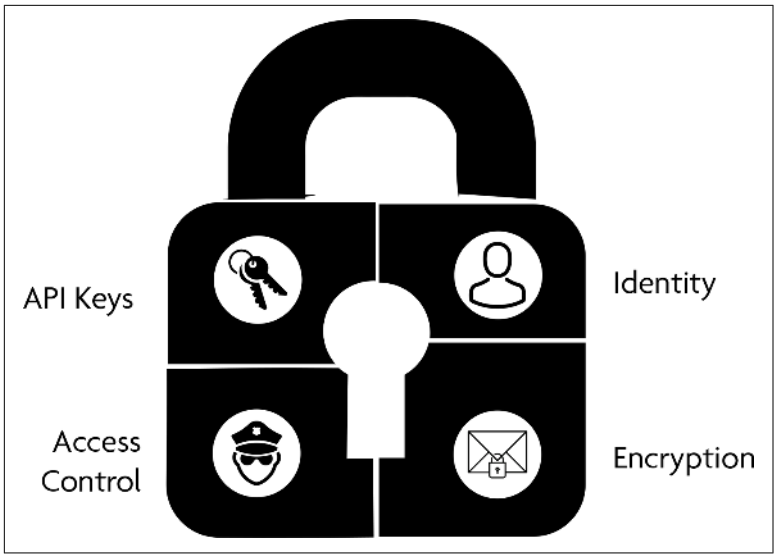


Figure 4-1. API traffic security basics

Also, a robust security implementation is able to deal with identity and access control between separate systems. For example, when APIs from your own system need to access services from an external API ecosystem such as Salesforce, SAP, and other so-called third-party services, your API traffic management needs to be able to negotiate identity and support access controls between your own API ecosystem and that of other, external systems.

API keys

API keys are a simple, low-level way to track and control how an API is used. All good API traffic management systems have a way to generate and track API key usage. The actual format of the API key is not very important—often they are just a universally unique identifier (UUID).

It is important to *not* use API keys as authentication or authorization keys. API keys are just a way to control access to the API, not proof of identity or access rights. To manage identify and access, you need other elements of your API security platform.

No matter how they are created, API keys need to be passed as part of any API request. That allows API proxies and gateways to track the appearance of these keys, validate the key against a list of authorized keys, and log them for monitoring and analysis. Typically, if you don't have a valid API key, your request is rejected by the API traffic management platform. Good API platforms allow you to cancel or *revoke* an API key if you discover any sign of abuse, too.

It is also important to keep in mind that API keys are *not* authentication or access control tokens. Because API keys are usually static strings that contain no identifying information themselves, they are not the same as authentication or authorization *tokens*. We cover authentication and authorization next.

Identity/Authentication

After your system validates a request using an API key, the next step is to confirm the requester's *identity*. That might be a human operating an application that calls the API, or it might be another internal service calling the API on its own behalf or as part of a more involved set of calls to solve a particular problem.

End-user API calls are often authenticated using some form of mutual authentication such as a security certificate or a three-legged authentication protocol (more on this in a moment) such as OpenID and OAuth. In all cases, both the client and the server need to *share* some identifying information (usernames/passwords, certificates, etc.). This means that there is some setup needed ahead of time—before the first API request—to make sure both parties trust each other at runtime.

The OAuth protocol is a common API authentication protocol because it offers quite a number of authentication flows (called *Grant Types*) to fit various levels of security and interaction needs. An important feature of OAuth is that it supports what is called *three-legged authentication*. This means that the actual authentication event happens at a *provider* service, which provides an authentication token that is then used by the client (application or API) to present to the actual service. Although this is bit more complicated than simple username/password or certificate authentication, three-legged models make it possible for people to build API consumer applications that do not ever see the requester's authentication credentials.

Your API traffic platform should make it possible to manage and support multiple authentication schemes without any change to the related API or the services behind that API. It should also make it easy to collect logs and related information at the authentication level given that this is your first line of defense when it comes to recognizing intrusions.

Identifying the API requester is just part of the job of completing a secure API transaction. It is also important to understand the access control limits each request has for any services it attempts to contact.

Access Control/Authorization

Knowing the identity of the entity initiating the request is just the start of the process when engaging in secure API transactions. The next step is establishing the access control rights that identity has for the life of the API request. This is usually called *authorization*.

The act of authorizing a request is, essentially, associating access rights to the request. Access control can be applied and validated a couple of different ways. For example, we can associate identities with *roles* (e.g., `admin`, `manager`, `user`, `guest`, etc.). You can apply access control directly to identities, too (e.g., `margaret_hamilton`, `frederick_jones`, etc.)

A reliable API traffic platform is able to quickly and easily associate validated identities with the proper roles for existing services. This work is usually not a problem for ecosystems for which there is a limited number of services and all the users are managed with the ecosystem (e.g., local user accounts). However, as the number of services and/or the number of roles within an ecosystem increases, it becomes difficult to scale access control. In the next section ("Managing Access with Tokens" on page 40), we look deeper into how to deal with this scaling challenge to your API ecosystem.

Encryption Considerations

Encrypting traffic offers a level of security for messages as they pass from ecosystem to ecosystem and between components within the same ecosystem. A good API traffic program includes the ability to employ message-level encryption and, if needed, field-level encryption.

The most common message-level encryption implementation is to use Transport Layer Security (TLS). The goal of TLS is to prove what is called a "secure channel" between two parties. TLS requires a bit of setup or *handshaking* to establish the identities of each party and a mutual encryption scheme. When that is done, the two parties use the same encryption details for each message they pass back and forth.

It is also possible to send messages "in the clear" *and* include field-level encryption. In this case, the data in sensitive fields such as personally identifying information (PII) is encrypted using a shared key (one that both the sender and the receiver already know ahead of time). Field-level encryption requires additional setup between parties and is challenging to scale. Often field-level encryption is handled by data storage systems (databases, filesystems, etc.) and is not something API traffic platforms need to deal with directly.

Managing Authentication Risk

Another important aspect to authentication is quantifying the *risk* associated with a particular authenticated identity. To do that, you need to know not just the identity of the requesting party ("Hi, my name is Mike!"), you also need to know several other things:

- How the identity was authenticated (username/password, active directory, Facebook, etc.)
- The device used to log in (mobile app, desktop, VM in the head end, etc.)
- The location of the authentication point (e.g., geo-code information)

You can use these factors (method, device, location, along with actual identity) to create a *risk score* associated with the authentication. For example, "Mike logged in using a valid certificate from a building on the company campus using his company laptop." This probably merits a relatively low-risk scoring. However, if the scenario were "Mike logged in using a social media account from Singapore using a mobile device we've never seen before," this deserves a high-risk score and might mean adding a two-factor authentication (2FA) to continue or might simply mean denying the login, alerting the security team, and locking the account to prevent further attempts to log in.

Your API traffic management program should support some form of risk scoring and mitigation to protect your system, your data, and your users.

Mediating External Security Systems

One more topic worth discussing when reviewing API security basics is the work of mediating security details *between* separate systems. As APIs become more ubiquitous in enterprises, it is increasingly likely that your API platform depends on the services of other, third-party APIs over which you have no control. And many of these external third-party APIs have their own security details including authentication and authorization requirements.

When an API client is making a call to another service that resides outside your ecosystem, they need to supply the proper security credentials. In many cases, these credentials are not the same ones used within your own API ecosystem. Thus, there is a need for a mediating layer to handle the security "hand-off" between systems.

The simplest approach is to use a single, shared set of credentials when making calls to a third-party API. The advantage is that only the component that makes the third-party call needs to know that third party's credentials. The downside is that this sharing of credentials results in a built-in *privilege escalation*. To avoid this, a good API traffic platform will provide a way to associate each API call with the appropriate level of security when reaching out to third-party APIs. This improves the overall security of your platform and provides more accurate usage and monitoring data when reporting on your API traffic both within and beyond your own ecosystem.

Managing Access with Tokens

The work of dealing with access control (or authorization) can be a challenge. This is especially true for organizations that 1) implement a microservice style behind the APIs, 2) have lots of external users (e.g., identities not managed within your own Lightweight Directory Access Protocol [LDAP] or Active Directory), or 3) depend on third-party/external APIs in order to complete requests.

In a monolithic environment in which all user identities are managed by the local user directory, where there is a small number of services, and where there are no external services to access, it can be

sufficient to assign every identity a single set of roles ("guest," "payroll," "sales," "sysadmin," etc.) for the life of *all* requests for that identity. However, as the number of services increases, as you add external users (e.g., partners, end users), and you add third-party API usage, assigning one set access control *profile* for each identity becomes difficult to manage and scale.

In this section, we talk about the importance of adopting an access token approach for authorization as well as the two ways to grant an identify access control: *grants by value* and *grants by reference*. A good API traffic management program supports both approaches, and with just a bit of planning and design of your actual tokens, you can easily switch between implementations at runtime.

The JWT Specification

JWT, or JSON Web Token,[1] is a specification designed to make it easy to transfer access control rights between parties. It is part of a collection of standards for representing security elements in the JSON format. Following is the full set of related specifications:

- JWS: JSON Web Signature
- JWE: JSON Web Encryption
- JWK: JSON Web Key
- JWA: JSON Web Algorithms
- JWT: JSON Web Token

For now, we'll focus only on JWT.

The JWT series and the JSON Object Signing and Encryption (JOSE) specification is not the only way to handle access tokens. There are some similar specifications including Branca, PASETO, and Macaroons.

For our purposes here, I review the basics of token-based access control using JWT as the example. Your API traffic management platform should guide you through the details of implementing a secure, reliable, and scalable JSON-related token support process,

1 *https://tools.ietf.org/html/rfc7519*

and you should check out how your platform approaches token-based access control and how you can observe and manage token lifetimes.

The JWT

JWTs are compact ways to share data between parties. Each token has three distinct parts: header, body, and signature.

Header

The header holds *preamble* information—the information needed to understand the rest of the token. JWT headers have two fields. The `typ` field identifies the token type, and the `alg` field identifies the hashing algorithm used to encode and sign the token:

```
{
    "typ": "JWT",
    "alg": "HS256"
}
```

Body

The body holds the *claims* information—the data that is to be passed between parties. There are a series of predefined properties for JWT claims outlined in the specification. In the example that follows, the three predefined properties, called reserved claims, are `iss` (the issuer), `iat` (the expiration time), and `sub` (token subject). Then there is a series of private claim names. These are not standard and are understood only by the two parties sharing the message:

```
{
  "iss" : "bigco",
  "iat" : 1516239022 ,
  "sub" : "authorization",
  "http://users.bigco.org/" : "admin",
  "http://accounts.bigco.org/" : "user",
  "http://products.bigco.org/" : "guest",
  "http://identity.bigco.org" : "q1w2e3r4t5y6"
}
```

Signature

The signature holds the *check-value*—the information that you can use to verify that the body you received is actually the body that was originally sent. This signature string becomes part of the token sent between parties in the form of `header.body.signature` and is wrap-

ped using the hashing algorithm identified in the header. Following is an example using the encoding string of "big-co-is-awesome" as the secret shared between both the sender and the receiver:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  "big-co-is-awesome"
)
```

The resulting hashed string would look something like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJqdWxlcyIsImlhdC
I6MTUxNjIzOTAyMiwic3ViamVjdCI6IkJpZ0NvIiwiaHR0cDovL3VzZXJzLmJpZ
2NvLm9yZy8iOiJhZG1pbiIsImh0dH6Ly9hY2NvdW50cy5iaWdjby5vcmcvIjoid
XNlciIsImh0dHA6Ly9wcm9kdWN0cy5iaWdjby5vcmcvIjoiZ3Vlc3QifQ.RnMdw
Bv3t4_tK5ZZy0nDQWQ7hKFtkGn5rmchZSDzuN8
```

It is important to keep in mind that the JWT specification is designed as a way to reliably encode messages that are shared between two parties to ensure data integrity and prevent tampering. This encoding, however, is not at the level of encryption. The contents are not *encrypted* in a way that secures the data from prying eyes (see "Encryption Considerations" on page 38).

Grants by Value

The most common way to ship authorization information via JWTs is to use the *grants by value* method—you load the JWT with all the grant information. The previous example (see "Body" on page 42) illustrates this approach. This approach works well for cases in which your API traffic platform knows ahead of time all of the services to which you'll need access. For example, when you log in to a system, that platform is able to locate all of your access control grants and load them into a single JWT, which is then carried with the request as it runs through the system "talking" to services along the way. As the request reaches a service point (e.g., a function in the code), that end point can check the JWT for the appropriate grant property and act accordingly.

In monolith-style architectures, this usually works quite well. There is a fixed number of possible services (often just one big one) and a fixed number of possible access control grants for each user. When the system doing the authentication work (e.g., handling the user login) is the same system hosting all of the components, the access control data is usually easy to find and load for each user request.

Grants by Reference

In the *grants by reference* model, the JWT does not contain the actual access control information. Instead, it contains a single pointer to a *claims store* that contains the list of all possible grants for this authenticated identity. Following is an example of a JWT that relies on the grants by reference model:

```
{
  "iss" : "bigco",
  "iat" : 1516239022 ,
  "sub" : "authorization",
  "http://identity.bigco.org" : "q1w2e3r4t5y6",
  "http://claims.bigco.org" :
  "http://api.bigco.org/claims-store/"
}
```

In this example, you can see that instead of a series of rights claims, this token contains two private name claims. One represents the authenticated identity for this request (`q1w2e3r4t5y6`), and the other points to an endpoint for the claims store service (`http://api.bigco.org/claims-store/`). Now, when a request reaches a service endpoint (e.g., a microservice entry point or monolith function), that endpoint can use the supplied identity and some local information (e.g., the function name, action associated with the request, etc.) and send all that to the claims service identified by the `http://claims.bigco.org` property. The claims service can then evaluate the request and return a response indicating the actual access control rights this request should be granted.

The key advantage for this approach is that as the number of services and/or grants grows, the size of the JWT does not need to change. This is especially handy in a microservice-style architecture in which the number of grants is often changing and can be difficult to predict at runtime. It is also helpful when the party authenticating the request identity is not the same as the party managing access control rights.

There are, however, downsides to this approach. First, each individual service endpoint needs to take on the responsibility of making a call to the claims store for each request. This will add traffic to your internal system. Also, because rights-checking now includes another network call, your access control system is more vulnerable to system outages than when you use a grants by value approach. You can

use cached responses and other techniques to address this problem. I talk more about dealing with the network in Chapter 5.

Summary

In this chapter, we covered security basics, including API keys, authentication, authorization, and encryption. We also covered the notion of authentication risk scoring and mediating identity between systems. Finally, we focused on using JWTs to document and manage access control via two approaches: the grants by value and grants by reference models.

Now that we have the basic elements of securing API traffic covered, we can look at the ways in which you can use your API traffic platform to improve the scalability and reliability of all your services. And that's the topic of the next chapter.

Additional Reading

- Securing Microservice APIs, Morrison, Wilson, McLarty
- Network Security Through Data Analysis, 2nd Edition, Collins

CHAPTER 5

Scaling Traffic

One of the biggest challenges to successfully managing API traffic is dealing with the network on which the API travels. Monolith-style implementations need to deal with the network because every API call starts and ends with a network call. Microservice-style implementations, however, are usually more vulnerable to network failures because there are usually multiple network "hops" involved in completing just one external API call. In fact, this is one of the big drawbacks of adopting a microservice approach—the increased dependency on the network to complete your API calls.

In this chapter, we discuss the importance of dealing with network-related problems and review a handy set of patterns you can use to meet most of the challenges of unreliable networks. It is important that your API traffic platform is able to recognize and mitigate these common network failures in order to keep your system up and running at a level of reliability and resilience you and your API consumers expect.

And the first step in this journey to reliability is to recognize that instead of focusing only on preventing network errors, you should also focus on surviving them when they happen.

Surviving Network Errors

Many of the network problems you'll encounter are things that happen outside your own internal network and events that are out of your control. Your service provider can experience upstream out-

ages, Domain Name System (DNS) routers can be misconfigured, other services that you depend upon might become unreachable or unreliable, and the list goes on. All these things can be compounded if you are relying on a cloud host provider (Amazon Web Services [AWS], Microsoft, Google, etc.) given that they, too, can experience these same problems.

And, to make matters more complicated, many network problems are not out-and-out failures but just issues of latency (slow responses) or unreliability (flaky responses). This all adds up to a bit of a challenge for companies that need to provide highly available and reliable services both within and outside the organization's network boundary.

Because you cannot *prevent* network problems from happening, the logical response is to work to *survive* them. In fact, healthy systems are ones that continue to operate successfully, even when parts of that system are failing. This idea was captured in Richard Cook's "How Complex Systems Fail," in which he states, "Complex [networked] systems run in degraded mode."

Safety I and Safety II

Of course, if there are steps you can take to prevent network errors from occurring, you should take these steps. But they are not enough. It is also important to *assume* network errors will happen anyway and plan to respond proactively to these errors. This approach of using both prevention and proactive attempts is defined by Erik Hollnagel as "Safety I and Safety II." As Hollnagel describes it: "A Safety I approach presumes that things go wrong because of identifiable failures or malfunctions of specific components."

As Figure 5-1 illustrates, in Safety I mode, API platforms are designed and managed to prevent problems from occurring. This means preventing bad requests from getting into the system, identifying and removing misbehaving requests, and blocking API consumers that continue to cause problems for your system. This is how most API platforms approach security, too (see Chapter 4).

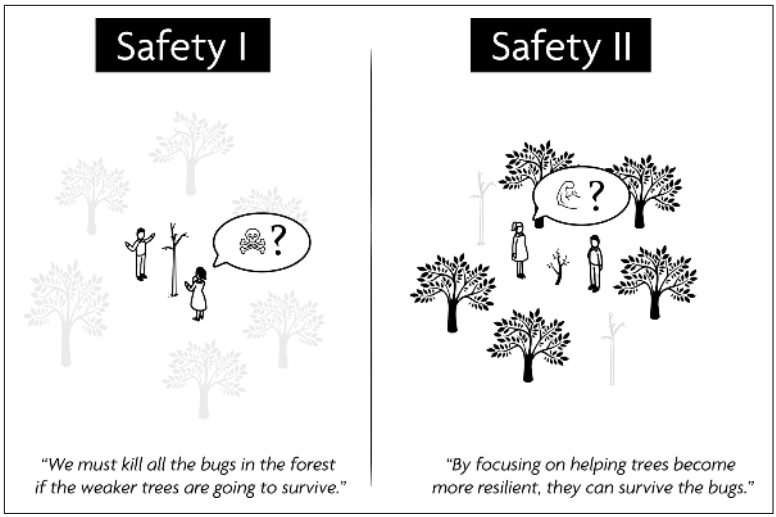


Figure 5-1. Hollnagel's Safety I and Safety II

This works for problems that you know about ahead of time and errors that occur when systems act in a malicious or dangerous manner (e.g., denial-of-service attacks, failure to properly authenticate requests). Many of the topics covered in Chapter 3 fall into this category.

NOTE

You can learn more about Hollnagel's approach to handling errors in complex systems in his book *The ETTO Principle: Efficiency Thoroughness Trade-Off*.

For Hollnagel, "Safety II … relates to the system's ability to succeed under varying conditions." In other words, even when errors occur, the system keeps working. For your API platform, this means implementing a system that *assumes* things will go wrong and works to overcome these problems in a way that allows requests and responses to succeed anyway.

There are lots of possible ways to create a "survivable" API traffic platform. In the next section, I'll review one well-known approach.

Stability Patterns

In 2007, programmer and software architect Michael Nygard published the first edition of the book *Release It!*. Now in its *second edition* (released in 2018), this book teaches the writing of software that works in the face of the harsh realities of runtime systems. In the book, Nygard explains: "Bugs will happen. They cannot be eliminated, so they must be survived instead."

Nygard offers quite a few patterns, antipatterns, and case studies to illustrate the challenges and solutions related to making sure your software stays up and running, even in the face of failures. In this section, I'll review a small collection of the patterns that relate easily to those charged with managing API traffic in a complex and complicated collection of services.

You'll find a wealth of advice and experience in Nygard's *Release It!*, and I encourage you to pick up a copy and share it with others in your organization.

Health Check

One of the things you can to do keep track of the overall performance of your network as a whole and individual services and endpoints in particular is to implement the *Health-Check* pattern. Health checks allow services to report on their current runtime status, and you can use that information to make decisions—even predictions—about the status of the network, too.

There are two key ways in which you can implement health checks for your API traffic. You can use *network-level* health checks, or you can use *service-level* health checks. Each has its advantages and drawbacks. In an ideal scenario, you should use both.

Network-level health checks

You can add network-level health checks (NLHs) at various places within your network infrastructure such as on any proxy or gateway server within your network boundary. NLHs can give you a good view of the overall network status, including the average time it takes for a single request to traverse the network from start to finish. You can also monitor the average number of errors and total request

throughput. We covered examples of these kinds of metrics in "Typical Traffic Metrics" on page 27.

A key advantage of using NLHs is that you can add them to any part of your infrastructure without having to coordinate with any single service running within your system. Another valuable aspect of NLHs is that you can monitor selected transaction paths from start to finish to gain information that no single service within your network has.

The problem with relying on NLHs for monitoring system health is that they don't provide you with much detail. You might notice that some requests are running slower than others, but not be able to determine why. For these cases, you need a different kind of health check—one at the service level.

Service-level health checks

When you need to understand what is happening *inside* a single service, you should use service-level health checks (SLHs). SLHs are implemented at the service level (often within the code) and can be used to collect key metrics about how a particular service is performing, right down to individual functions within a service component. In monolith-style implementations, SLHs are often your best opportunity to uncover bottlenecks and resource-expensive operations.

A big challenge with implementing SLHs is that they become part of the code execution of the service. In other words, adding health-checking code can take up more of your valuable computing resources and degrade the performance of your service. Be careful that in your zeal to add data collection and health reporting to a service that you do not become the source of bad performance for that component!

Health Reporting

The most common way to implement health reporting—the reflection of the health data collected at the proxy or service level—is to implement network endpoints (e.g., `http://api.bigco.org/customer-services/health/`) that respond with the latest status information for the target service or network segment. Typically, the health data is also spooled to disk-level logs that can be collected and used for postmortem checking. That way, you can establish a

trend analysis and look at the data in various ways to find patterns and/or problems not easily discovered at runtime.

TimeOut and FailFast

One of the most common challenges to "surviving the network" is dealing with slow connections and/or responses. An effective solution for slow and long-running requests is to implement the *TimeOut* and *FailFast* patterns.

The *TimeOut* pattern is implemented by incoming API calls to deal with slow and nonresponsive replies from API providers. If the request is made and the response does not appear within a fixed time limit (e.g., 200 ms), the API caller "gives up" and abandons the request. This prevents API callers from waiting too long on services that are unresponsive.

The *FailFast* pattern is a way for those receiving API calls to proactively "cancel" requests that might take longer than the API caller is willing to wait. In this case, the API caller sends a "wait limit" (e.g., 200 ms) in the request metadata. For cases in which an API provider knows (from its own health-check data) that it is only able to complete the request in about 300 ms, that API provider can just skip attempting the call and return a timeout error (e.g., HTTP 504) immediately. This prevents the API provider from taking work that it will not be able to complete in time and thus would be abandoned anyway.

A good API traffic management platform will let you establish timeout values at the proxy level and abandon requests that take too long to complete. You might also be able to set up a fail-fast rule at your proxies and gateways.

In both cases, it is important to log these proxy-level actions and include them in your health-check reporting as errors. As you'll see later, you can combine these patterns with another pattern, "Circuit-Breaker" on page 53, too.

Bulkhead

A pattern that helps services avoid running into the *TimeOut* and *FailFast* situations is to wrap important (and possibly unreliable) services in a cluster. This is called the *Bulkhead* pattern because it allows you to contain any problems caused by a single machine. Any

failures within the cluster are unlikely to "leak" beyond the cluster into the rest of the network.

Using the *Bulkhead* pattern is rather easy for services that are self-contained or stateless. For example, a service that performs address validation by accessing an in-memory data store is stateless. It doesn't rely on other services to do its work (for more on this, see "Machine-Level Caching" on page 55).

It is important to report health checks both on the cluster *and* on each of the individual components within the cluster. That way, even though the cluster reports high availability, you'll still be able to inspect individual machines and notice any changes in the service-level health logs.

As with the other patterns mentioned here, you can implement the *Bulkhead* pattern without having to alter the architecture of the service component. In fact, the component doesn't even need to know it has been included in a cluster at all.

Circuit-Breaker

The *Circuit-Breaker* pattern is actually a more proactive way to respond to the *TimeOut* pattern. With the *Circuit-Breaker* pattern, request timeouts are treated to additional handling that can allow the failed API request to be automatically rerouted and handled without the API caller ever experiencing a timeout. In all cases, the request failures are used as a *trigger* to force the network to treat the failing service as "suspect" for a short time and not send it any requests. After waiting a short period of time (to allow the suspect service to clear up any problems), requests are again routed to the service. If there are still problems, the process of rerouting and waiting begins again.

There are a number of write-ups on the history and implementation of the Circuit-Breaker pattern. Specifically, I recommend Martin Fowler's "CircuitBreaker," Chris Richardson's "Pattern: Circuit Breaker," and NGINX's "Implementing the Circuit Breaker Pattern."

It is possible for some API traffic platforms to implement the Circuit-Breaker pattern without requiring coordination with the service programmers and architects. To do this, your API traffic system needs to be able to associate requests to a service with a "circuit" that includes a timeout handler, at least one alternate service, and a waithandler that is able to flip the circuit back to the target service after waiting the appropriate amount of time.

As usual, you need to make sure to log all activities in order to make them available to health check reporting and for later postmortem analysis, to find larger trends and find and fix repeating problems with failing services.

Summary of Survival Patterns

The patterns listed here from Nygard's *Release It!* book are examples of ways in which you can use your API traffic platform to add resiliency and reliability to your network infrastructure without rewriting your individual services. They all focus on the challenge of maintaining uptime even when parts of the network are failing. In the next section, I talk about ways that you can handle requests for data even when the network is not available through the use of caching.

Caching

A key element that you can use to reduce the effects of network errors is to proactively store network responses in a *cache* for future use. By caching (storing) responses you receive over the network in local storage, you gain the ability to respond to future requests without using the network at all. As Roy Fielding, one of the principal authors of the HTTP specification and originator of the Representational State Transfer (REST) software architectural style, points out: "[T]he most efficient network request is one that doesn't use the network."

There are two main types of caching patterns that you can use to protect your system from failure: *machine-level* and *network-level* caching. Each has advantages and challenges worth considering as you design your API traffic platform to reduce the adverse effects of unreliable and/or slow networks.

Machine-Level Caching

You can use machine-level caching when you want to improve your system's perceived performance for a single machine in the network. For example, you might have a stateless service that handles postal code lookup and validation. That service might support passing in a single postal code and validating it against a small database of all the postal codes in your company's sales areas. Reading the data off the disk for each request adds latency and offers the possibility of a disk read error for every request. This might happen hundreds of thousands of times each day.

You can improve your system's safety and resilience by loading the entire database into server memory on startup and simply using random access memory lookups for each validation request. This can greatly speed up response time and reduce the number of read errors to a small fraction.

Note that this machine-level example can be applied to a collection of postal code lookup servers in a cluster (see "Bulkhead" on page 52), too.

This kind of caching can improve a single machine's ability to continue to work even when that machine experiences disk errors, but it does nothing to help that machine respond when there are network problems. For that, you need to employ network-level caching.

Network-Level Caching

You can use network-level caching to improve your system's perceived performance across machines in a network. To use the previous example, you could set up a postal-code validation service (PCVS) on the network and allow lots of other machines on the network to call the PCVS using a simple API. While the internals of that service might use the aforementioned in-memory technique to make the *machine* work efficiently, that will not help if the PCVS machine is unreachable (e.g., DNS routing problem, broken connection, etc.).

To survive network unavailability, you can use network-level caching. For example, postal codes are unlikely to change over the space of days or weeks, so you might ship metadata with the response

informing any client that receives a copy of the record that the response would remain valid for up 48 hours. Upon receiving the response, the API consumer could then save it to local memory (or disk) and use that as a valid response for up to 48 hours into the future.

The HTTP protocol has extensive support for network-level caching outlined in RFC7234. Your API traffic platforms should understand and honor HTTP caching metadata directives, and most of them will allow you to establish rules and storage locations for caching responses from any other HTTP-aware servers.

Employing content delivery networks (CDNs) is one important way to take advantage of network-level caching. CDNs understand the network caching directives mentioned here and offer caching servers strategically placed around the world to allow you to "stage" cache content close to your target audience. These also support custom caching templates called server-side includes (SSIs) that allow you to optimize the actual content that is cached, and to customize the content for your target audience.

Reducing network traffic is one way to improve overall scalability and reliability. Using caches also improves the *perceived* performance of the network in general. But there are limits to this pattern of caching responses from requests.

Preemptive caching

One way to expand your ability to cache content in case of network failure is to preemptively cache responses. For example, on startup, a server on your network in charge of making requests to the aforementioned PCVS service might automatically fire off a series of requests for the 100 most commonly requested postal codes and keep a local copy ready to respond.

Data snapshots

Another way to improve network resilience is to arrange to keep a copy of the data set locally. This is sometimes called a data *snapshot*. For example, our PCVS might support a special API call that spools

the full postal code validation dataset to another machine within our local network.

Data snapshots work well for datasets that have a fixed size and rarely change. If the data is of a varying size and/or changes often another possibility to improve overall network resilience is to arrange for a data replica.

Data replicas

For datasets that change quite a bit, you can arrange to keep a data replica locally. In this case, all data reads and writes are eventually executed on one or more remote copies of the data store. This works well when you need to keep a more accurate copy of the data than, for example, a daily snapshot. However, data replication has its challenges.

First, supporting data replication increases network traffic. If you are working to improve system reliability in the face of network failures, increasing network traffic is not the right way to go. Also, if you want to support not just reads but also writes to data replicas, you'll need use a data storage technology built for this added functionality.

Choose wisely

It is important to point out that the first two solutions: request-caching and preemptive caching rely on network-level metadata and work between any two servers that support HTTP caching (RFC7234). This means that you can apply these approaches to any interactions over HTTP, including those with third-party services that you do not control.

The second two approaches (snapshots and replicas) focus on passing copies of the target data and require coordination by both the provider and consumer using technologies like Apache Kafka or other protocols. Thus, you will be able to implement these approaches only when both the API provider and the API consumers already agree on the details of the storage formats and data models.

Summary

In this chapter, we reviewed the challenge of maintaining network reliability and resilience even when parts of that network (or components within the network) are failing. This notion of "surviving the network" is an essential aspect of establishing a healthy and scalable infrastructure for your API platform.

We covered Nygard's stability patterns (*TimeOut*, *FailFast*, *Bulkhead*, and *Circuit-Breaker*) and reviewed various data caching options at the machine and network levels. With the exception of in-memory caching and data snapshots and replication, you can implement all of these patterns at the network level using gateways and proxies—all without the need to rewrite or rearchitect individual service components.

Next, we talk about how you can use your API traffic platform to help you track your company's progress on business-level goals, how to diagnose runtime traffic problems, and how to use automation to improve your platform's ability to "fix itself" when typical problems arise.

Additional Reading

- *Architecting for Scale*, Atchison (O'Reilly)
- *Release It!*, 2nd Edition, Nygard (O'Reilly)
- *Intelligent Caching*, Barker (O'Reilly)

CHAPTER 6

Diagnosing and Automating Traffic

In this chapter, we'll take what we've covered so far and use it to begin mapping out what you can do with your API traffic platform to help proactively support and enhance your company's API ecosystem. Building on top of the level of *API Traffic Management* (see the lowest level of Figure 6-1), we dig into three additional topics:

- Supporting runtime experiments and the principles of *Site Reliability Engineering* and *chaos engineering*
- Adding the *automation* of traffic rules and metrics in testing, deployment, runtime
- Dealing with business goals through the use of *Objectives and Key Results* (OKRs)

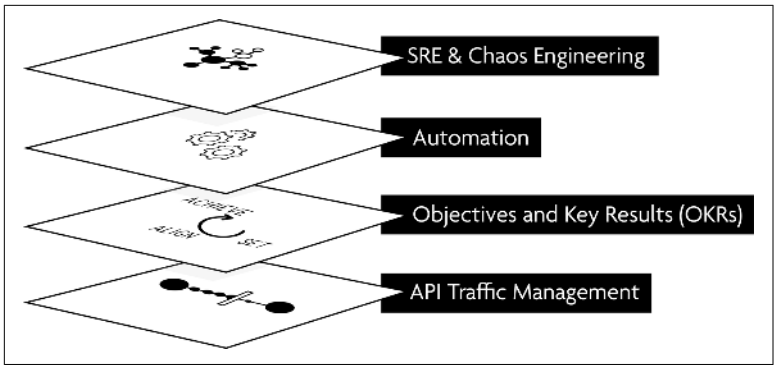


Figure 6-1. Aspects of diagnosing and automating API traffic, showing three new layers added on top of your API Traffic Management foundation

Business Metrics

Most of this book has focused on the network- and component-level aspects of monitoring and reporting system health. But that is just part of the story. It is also important to ensure that your API traffic platform can provide reliable monitoring and feedback on your key business goals and objectives. This focus at the overall business level can help your API program provide timely, real-time data on the company's progress on key business metrics.

There's an old adage by Steven A. Lowe: "You can measure almost anything but you can't pay attention to everything."

As you build up your API traffic practice, it is important to identify the kinds of metrics that matter to your *business*, not just the ones that matter to the network or individual components within the network.

The topic of appropriate business metrics and the process for defining, selecting, implementing, and tracking them is beyond the scope of this book, but some basics can be helpful if you're the person charged with supporting the process in general and implementing the details of an ongoing business metrics initiative.

As outlined in Chapter 1, there is an important difference between the OKRs used to monitor business-level goals and the Key Performance Indicators (KPIs) used to measure progress on those goals.

We focused on KPI-style metrics in Chapter 3. Here, we can dig into the OKR-style metrics.

Pirating Your Business Metrics

In 2007, entrepreneur and angel investor Dave McClure presented what he called "Startup Metrics for Pirates: AARRR!!!" His "AARRR" acronym for identifying key business metrics stands for "acquisition, activation, retention, referral, revenue." Although McClure's talk is geared toward internet startups, his ideas are a great start for thinking about establishing your initial business metrics. Even for internally focused API programs, tracking added "users," how often they actively engage in your company's API platform, how many stick with it over time, and how "viral" your program is (based on referrals to other "users") can be good indicators of the value your API program is providing to the organization.

The OKR Cycle

Like all goal-setting systems, there is an overall cycle to follow in order to get the most out of the process. In his article "The OKR Cycle: Three Steps to OKR Success," Felipe Castro says there are three key parts to the process: *set*, *align*, and *achieve*.

Setting OKRs

Setting OKRs is the process of identifying actions and outcomes that are *valuable*, *engaging*, and *actionable*, meaning that they are more than a list of tasks. They need to be something that everyone understands and finds motivating and doable. Often OKRs read similar to user stories from Agile. For example, "*We will improve API developer experience by reducing mean time to API sign-up by 10%.*"

Note that, in this example, the organization wants to improve *developer experience*. Thus, it is essential to have a baseline on what the *current* developer experience looks like. The API platform needs to be monitoring and reporting KPIs that are related to developer experience *before* you can know whether you've improved them.

The work of identifying important business-related KPIs for your shared OKRs is one of the key contributions your API traffic program can provide to your overall business goals.

Aligning OKRs

Typically, setting OKRs affects multiple teams. It's important to review the defined goals and ensure that they make sense as net positives for all parties involved. This means mapping team dependencies and identifying any roadblocks to attaining the goal. For example, is it possible to reduce the time it takes for a new developer to sign up for an API key without also including the legal department?

This is another important element of a robust API traffic system. You might learn that factors outside typical runtime metric gathering (e.g., interacting with the legal department to improve the developer onboarding experience) will be important for tracking and reporting on business-level OKRs. A robust API traffic program will let you interpolate data from other sources in order to build a good picture of your progress.

Achieving OKRs

The work of setting and aligning OKRs only pays off when you can measure your progress (and hopefully report success) on meeting those goals. In many organizations this process of reporting and evaluating OKRs is a regular ritual, one often done quarterly, monthly, or even weekly. The challenge here is that the time devoted to reporting and review is often time "stolen" from more productive work within the company. A robust traffic platform can help reduce lost productivity by making OKR progress reporting *continuous*.

When the data is continuously updated and displayed, much of the work of reporting and reviewing turns from a corporate chore into a cultural norm. People can come to expect to see these figures and, when the trend looks to be heading in the wrong direction, can be motivated to initiate steps to adjust activities, update targets, and get the trend back on track.

This ability to provide a timely and meaningful reflection of the business's stated key objectives is another way in which a robust API traffic management system can contribute to your organization's bottom line.

Another important way traffic management can help contribute to success is to make it easier for teams to automate metrics creation and evaluation.

Automation

Most of this book has focused on the work of discovering and implementing traffic monitoring details. This work is, for the most part, a "hand-crafted" experience that takes advantage of the curiosity and intelligence of your API traffic team and allows it to apply its skills to your own API traffic platform. Much like programming, this level of API traffic *engineering* is a key element of any quality API management practice. However, there are opportunities to apply API traffic engineering in ways that do not rely completely on individuals designing and implementing the traffic rules.

In this section, we explore three areas where you can introduce automation to your API traffic management in order to improve your system's reliability, resilience, and *testing*, *deployment*, and *alerting/recover*.

Automating Testing

There are two elements to the "testing" of API traffic platforms. First, you need a way to test the various routing, security, monitoring, and resolution scripts and rules used to keep your production platform safe and reliable. The second element is the work of providing enough of a virtualized network to allow service teams to test their own components before placing them in production.

Testing network-level traffic management

Like any testing environment, you'll need to mock or virtualize enough of your production network to make your testing meaningful. But you don't need a complete *duplicate* of production. When you need to test North-South traffic security and routing scripts, you'll need an environment that mimics your production network perimeter and security elements. When you're experimenting with ways to optimize East-West traffic between service components, you'll need a different kind of environment—one that reflects the proper mix of service components and proxy servers needed for your current test parameters.

An important part of your API traffic management platform is the ability to virtualize portions of your network for testing purposes. Automating the process of allocating virtual machines (VMs) and/or containers for a test run, spinning them up, and shutting them down

after the tests are complete is an essential part of your traffic management practice.

Testing service-level traffic management

Your traffic platform also needs to support all of the teams creating service components that will run within your network. They'll need virtualized identity and access control elements and routing support sufficient to validate their own service-level tests. They might also need to spin up instances of proxies to handle the network survivability patterns we discussed in "Stability Patterns" on page 50.

And all this support should appear in the form of an automated process that teams can include in their own build pipeline. The work of spinning up virtualized network elements, installing new traffic rules related to the component's production release, emitting synthetic traffic for test cases, and eventually shutting down all of the ephemeral test elements is all part of the work of a robust traffic management practice.

Finally, in some cases, it might make sense to add API traffic experience to the teams designing and building the components that will end up in production. Just as teams need expertise in designing code-level tests, they can also benefit from the advice and guidance of experienced traffic management staff. This is especially true for the active aspects of API traffic management, such as security (see Chapter 4) and scaling (see Chapter 5).

Automating Deployment

Just like the work of supporting API traffic testing, your API traffic system needs to ensure that deploying updates into production is safe and reliable. Sometimes, a release contains only network-level changes (gateway and proxy changes), and sometimes a release is focused on server-level (component) updates that rely on parallel updates to the traffic system (e.g., new security profiles, routing rules, stability side cars, monitoring definitions).

Whenever you can, make it possible for individual teams to include all of the related traffic changes in their own release packages. This means that your traffic platform needs to support scripted updates and the ability to monitor and coordinate changes at both the service and network levels. Of course, your traffic team needs to make it possible to not only reliably post updates into production, it must

also make sure that it is possible to quickly and safely *back out production changes* when things don't go as expected.

Your API traffic system is part of your change management system.

Automating Alerting and Recovery

The work of analyzing and modifying thresholds for altering is another service your API traffic team can to provide to the organization. Your company may even have dedicated analysts focused on developing reporting/alerting systems. In that case, you need to arrange your API traffic platform to make it easy for the analytics staff to safely gain access to appropriate levels of traffic data, use that data in their analysis, and (where needed) provide your traffic teams updated advice on which values/levels to monitor and at what level (business, network, service) to do that monitoring.

There is another step—one that goes beyond the work of sending alerts when traffic becomes unhealthy. That is the work of actually "fixing the problem" in reaction to the discovered traffic patterns. There are lots of ways in which your platform can provide real-time solutions to problems:

- Spinning up additional instances of services within a cluster when traffic spikes (and spinning them down as traffic subsides)
- Rerouting traffic to different geographical regions to deal with localized spikes in API traffic
- Automatically increasing identity security checks (e.g., requiring two-factor authentication) for a class of users or geolocations that exhibit a spike in risky activity
- Preemptively invoking traffic circuit breakers (see "CircuitBreaker" on page 53) when one or more clusters exhibit a sudden increase in latency or are unresponsive
- Periodically adjusting TimeOut and FailFast values (see "TimeOut and FailFast" on page 52) to better reflect "new normal" traffic loads
- Automatically reversing production updates when a new build shows early signs of major failures

This list contains actions that can be programmed into your API traffic platform as a way to maintain a minimum level of reliability and safety even in the face of unforeseen network problems. The process of going beyond altering to fixing discovered problems is an approach rooted in the principle of "Eliminating Toil," from Google's Site Reliability Engineering program. It assumes that, instead of just alerting a human when things begin to go poorly, your system should be engineered in a way that supports the ability to self-maintain whenever possible. As Google's Carla Geisser describes it: "If a human operator needs to touch your system during normal operations, you have a bug. The definition of normal changes as your systems grow."

And this attention to runtime behavior—and the power to fix it automatically—leads to one more area of diagnostics and automation: supporting runtime experiments as a way to explore and discover weaknesses in your production system *before* they become a problem.

Runtime Experiments

Organizations that have already established a healthy business metrics program, track and report on their build/deploy cycle, and rely on automation for injecting monitoring and tracking metrics can also take their traffic programs one step further: they can help teams in the company implement and monitor runtime experiments on the resilience and reliability of your system.

Using runtime experiments in production is a way of testing the "bad path" (testing cases where things go wrong in your system) instead of just testing the "happy path" (proving that things work as expected). This is a kind of advanced testing regime that can be introduced *in addition to* the typical "happy path" test suites more commonly used in build and production environments.

Following is a quick review of two well-established ways to run these kinds of experiments (Site Reliability Engineering and chaos engineering) along with some suggestions on how you can use your API traffic management platform to help implement and gather the monitoring data needed to run a robust runtime experiments program.

Site Reliability Engineering

Site Reliability Engineering (SRE) is the practice of applying software engineering practices to network infrastructure. The SRE that we recognize today started as a practice at Google around 2003 that involved fewer than 10 people. At last report, Google had more than 1,500 people dedicated to doing SRE work. There is also at least one conference circuit, SRECon hosted by Usenix, that has run continuously since 2014.

Applying software engineering principles to operations means more than automating deployment and monitoring system health. It also means using engineering practices to explore and test the boundaries of your running system. This means collecting data on both individual services and the network that hosts them. It also means using the collected data to establish hypotheses, run experiments, and review the results in order to identify opportunities for improving your system's reliability and resilience—all things that we've talked about in this book.

SRE has a handful of principles. They are:

- *Embrace risk* by measuring and managing the system.
- Rely on *Service-Level Objectives* (SLOs) to define expected user outcomes.
- *Eliminate toil* through the use of automation.
- *Monitor systems* to ensure your agreed SLOs.
- *Engineer releases* to improve reproducibility and reliability.
- Aim for *simplicity* by eliminating accidental complexity.

As you can see, several of SRE's stated principles fall well in line with the kind of work a good API traffic management platform needs to deal with. As you build out your API traffic management practice, you can use it to embrace and promote SRE efforts within your organization, too.

Chaos Engineering

In 2011, Netflix's Yury Izrailevsky and Ariel Tseitlin published a blog post that described their work toward improving the availability and reliability of their systems. In their post they describe something they called *Chaos Monkey*: "a tool that randomly disables our pro-

duction instances to make sure we can survive this common type of failure without any customer impact." This approach of purposefully introducing "bugs" into running production systems has come to be known as *chaos engineering*.

Similar to the work of SREs, chaos engineering is a way to test the resilience of production systems directly. This works only if there is a high degree of observability already in place within the network. And, as we've seen already in this book, API traffic management plays a key role in providing that observability both at the network and service levels.

As you roll out your API traffic program, be sure to consider any current or future chaos engineering practices that you'll need to support.

Summary

In this chapter, we brought together several earlier elements of the book such as going from monitoring to managing, dealing with security risks, and surviving network errors and using that information to lay out things you can do with your API traffic platform to help set and track business metrics; introduce automation of traffic testing, production, and recovery; and even help support runtime experiments based on principles for SRE and chaos engineering.

That's a lot to consider when it comes to establishing and maintaining a robust and flexible API traffic management practice. In Appendix A, we take a moment to reflect on what's been covered here and how you can apply it to your own organization now and in the future.

Additional Reading

- *Introduction to OKRs*, Wodtke (O'Reilly)
- *Site Reliability Engineering*, Petoff, Murphy, Jones, and Beyer (O'Reilly)
- *Learning Chaos Engineering*, Miles (O'Reilly)

APPENDIX A

From Monitoring to Managing and Beyond

This book covered a lot of area in a short amount of time. But it is important to keep in mind that no one can introduce *all* of the things here all at once.

At the start you need to understand what's at stake (Chapter 1) and the fundamentals of establishing your traffic management approach (Chapter 2) and tackling the basics of monitoring and reporting important traffic values and trends (Chapter 3).

Once you have your foundation set, you can spend time shoring up your system-wide security (Chapter 4) and begin to expand your API traffic program's scope from just dealing with day-to-day safety and stability toward "surviving the network" (Chapter 5). Eventually you can begin designing traffic management features that allow you to fix problems automatically and run safe and valuable experiments that help you anticipate the needs of your internal staff as well as your external customers and partners (Chapter 6).

As more companies progress along this path of treating API traffic management as another essential engineering practice, we're bound to see more tooling and API management platforms adopt these same principles, and *that* means everyone gets better at monitoring, securing, scaling, and ultimately managing not just your APIs but also your *business*.

It all begins now with your initial steps to apply what you find here to your own company in your own unique way.

About the Author

Mike Amundsen is an internationally known author and speaker. He travels the world discussing network architecture, web development, and the intersection of technology and society. He works with companies large and small to help them capitalize on the opportunities provided by APIs, microservices, and digital transformation.

Amundsen has authored numerous books and papers. He contributed to the O'Reilly book *Continuous API Management* (2018), his book *RESTful Web Clients* was published by O'Reilly in February 2017, and he coauthored *Microservice Architecture* (June 2016). His latest book, *Design and Build Great APIs*, for Pragmatic Publishing is scheduled for release in late 2019.