Compliments of IBM

IBM Limited Edition

APIs

FOR DUMMIES®

A Wiley Brand

Learn to:

- Apply the power of APIs to business challenges
- Define the product nature of modern APIs
- Manage which APIs to provide or consume
- Build an effective API technology platform

Claus T. Jensen

APIs

FOR DUMMIES®

A Wiley Brand

IBM Limited Edition

by Claus T. Jensen

FOR DUMMIES®

A Wiley Brand

APIs For Dummies®, IBM Limited Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN: 978-1-119-04116-0 (pbk); ISBN: 978-1-119-04117-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Editor: Carrie A. Johnson
Development Editor: Kathy Simpson
Editorial Manager: Rev Mengle
Business Development Representative: Sue Blessing
Production Coordinator: Melissa Cossell

Table of Contents

Introduction

APIs are a hot topic, energetically debated by business-people, IT managers, and developers alike. Most of the excitement in the public space is about open public APIs. To some degree, not having a public API today is like not having a website in the late 1990s. Yet for many enterprises, public APIs are really the least of their business concerns. More important concerns include building omnichannel solutions, innovating faster than the competition, becoming a mobile enterprise, or operating in a hybrid cloud environment.

APIs are fundamental enablers for all these initiatives and more, which is why so many different types of stakeholders are interested in them. But what is an API, really; why is it different from an old-school application programming interface; and why should you care? In principle, the acronym API stands for *Application Programming Interfaces,* but the notion of what API means has evolved significantly. APIs today are quite different from the application programming interfaces of old. Read this book and see what that change entails.

About This Book

Modern business ecosystems need to rethink their approach to innovation and integration. This book is your guide to applying the power of APIs to business challenges ranging from changing business models to embracing a world of devices and sensors. Experiencing the power of APIs involves much more than simply data monetization. Whether you're an API provider or an API consumer, you need to make smart business and IT decisions. This book first defines the basic nature of modern APIs and then leads you through several necessary decisions, ranging from which APIs to provide or consume to how you can build an effective API technology platform.

A handful of important mottos are also touched on multiple times throughout the book. These include the following:

- **Think of an API as a product.** It represents something you have chosen to share with a particular audience under defined terms and conditions.
- **Try early, learn fast, scale easily.** An experimenting approach to APIs is effective for most use cases, and many APIs will be opportunistic in nature.
- **Always use APIs as the boundary of your domain.** That gives you control and visibility of traffic going in and out.
- **API platforms are specialized for the task.** You want the creation, operation, and sharing of APIs to be dead easy and 100 percent secure.
- **Monetizing public APIs is not the only entry point.** Many enterprises use APIs for collaboration and innovation across their IT and business ecosystems.

Icons Used in This Book

Like all *For Dummies* books, this one has a few icons in the margins. Here's what they mean.

The Tip icon points out helpful information.

Anything marked with the Remember icon is good to keep in mind.

If you're not interested in the nitty-gritty technical details, you can skip anything that has a Technical Stuff icon.

Take care when you see the Warning icon, which alerts you to things that could harm your business.

Beyond the Book

This short publication can't offer every detail about a topic. So for more information outside the realm of this book, check out the following links:

- `developer.ibm.com/api/blog`: IBM's API blog
- `www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/sg248188.html?Open`: In-depth asset for integration middleware
- `ibm.com/soa`: Download the *SOA Design Principles For Dummies* eBook

Chapter 1

The Anatomy of an API

In This Chapter

- Knowing what an API isn't
- Seeing how developers and consumers want to use APIs
- Categorizing APIs
- Making APIs part of the business model

Modern APIs are flexible ways of projecting your capabilities to an audience outside of your own team. When done right, APIs enable enterprises to innovate faster and reach new audiences. That is the value of APIs, but what is their basic nature, and what key questions should you ask when embarking on an API journey? This chapter attempts to answer those questions.

Looking at What an API Is Not

Sometimes, the best way to explain something is to explain what it's not. With that in mind, here are a few things that APIs aren't:

- **A piece of software:** Software isn't an API (but it may render itself as an API to ease consumption of its capabilities).
- **A user interface:** A user interface isn't an API (but it may be built on top of one).
- **A server:** A server isn't an API (but it may host one or more APIs that expose the data and functions provided by the server).

The Product Nature of APIs

Think of an API as a product. You carefully craft it so it's attractive to the intended consumer — so that it sells. It neither matters whether that consumer really pays for the API nor if she's outside your enterprise or part of a different team inside it. You want her to use your API, because she creates value for both of you when she does.

The product nature of APIs is fundamental to their power. It also makes them very different from old-school application programming interfaces. An old-school application programming interface represents a piece of software that you have built and deployed. A modern API represents a package of capabilities that's attractive to an audience independent of any specific piece of software running in your back end. So although modern APIs do include a defined programmatic interface, they're deliberately designed from the perspective of the intended consumer.

Because an API is a product, before you develop one, you should ask yourself these key questions:

- Who are the intended consumers?
- How are you going to reach these consumers?
- Under what terms and conditions can consumers use this API?

From APIs to API Economy

The API economy emerges when APIs become part of the business model. Public and partner APIs have been strategic enablers for several business models, such as Twitter's and Amazon's.

The Twitter APIs, for example, easily have ten times more traffic than the Twitter website does. The company's business model deliberately focuses on tweet mediation, letting anyone who wants to do so provide the end-user experience.

From the get-go, Amazon chose to be not just an Internet retailer, but also a ubiquitous merchant portal. Amazon's merchant platform is deliberately built on APIs that allow easy onboarding of new merchants.

APIs as business network enablers aren't new. Banks have built payment infrastructures and clearinghouses based on well-defined APIs for decades. Modern APIs, however, are built explicitly for an open ecosystem (internal or external), not for closed private networks. Furthermore, the consumption models for APIs are standardized with a focus on ease of consumption rather than ease of creation (see "Understanding What Developers Want" later in this chapter).

Some people use the term *business APIs* for all modern APIs. The term is certainly fitting in the sense that APIs, as products, should be an integral part of your business strategy. Just be aware that launching a public or partner API isn't the only way to make APIs part of your business model. There are many use cases for internally consumed APIs, perhaps the most common such use case being the need to provide a differentiating *omnichannel customer experience.*

Whether your business was "born on the web" or has been around for 100 years, you're living in the age of cloud, analytics, mobile, and social computing where omnichannel has become table stakes. To differentiate yourself from your competitors, you have to give customers an immediately engaging experience. To deliver that experience, you need freedom to experiment and innovate. Seize the opportunity and try early, learn fast, and scale easily.

Understanding What Developers Want

Developers want to use APIs for innovation and experimentation. To them, reuse is about speeding time to delivery, sharing is about expediency, and encapsulation is about having little to learn. They're not as interested in how APIs were created (and at what cost) as they are in how easy the APIs are to consume.

Easy consumption doesn't refer only to what the API looks like. To the developer who is API savvy, easy consumption also means that an API must be easy to find and easy to register to use, and it must be clear how much the API can be trusted in mission-critical solutions.

Ideally, an API ecosystem should be community-centric. An effective API community shows developers the exact APIs available to them for their current tasks. Self-service registration and preapprovals are already in place for the APIs that are visible to the community. The community's social features allow people to like or dislike a particular API, and consumer-centric analytics show the expected operational behavior of any API of interest. These capabilities historically aren't part of IT governance, but they're core features of good API management solutions. Good API management solutions also add value for API providers, making it easier to create APIs and improving control of their runtime behavior (explained in more detail in Chapter 2).

Four categories of APIs

When you're starting on an API journey, deciding which APIs to build first can be daunting. A good API should make a difference to the business. In some cases, industry specific use cases can drive the creation of APIs. But to truly understand what makes a good API for a particular enterprise, you need to understand the nature of its unique business situation.

In this sidebar you find an approach that IBM has found useful in practice. A good way to start is to ask yourself, "Which business situations would I like to improve, and how can I do that?" When you answer this question, you'll likely end up choosing your first APIs from one of the following four categories:

- **Detection APIs:** Detection APIs help you identify opportunities to engage customers, employees, partners, and devices. They include mechanisms such as mobile location detection, sensor monitoring, predictive analytics, and human observation.
- **Enrichment APIs:** Enrichment APIs improve understanding of the situation with historical data from customer relationship management (CRM) systems, account records, demographical analysis, health records, and the like.
- **Perception APIs:** Perception APIs provide dynamic context for the current situation and enable

you to understand what may be on the minds of the people you want to engage. Examples are social APIs (people sharing future plans or current interests) and sensor analytics (for the overall system state, such as global resource consumption or traffic congestion). After all, someone who's going on a trip to the Caribbean next Monday is probably interested in different things from someone who's staying in winter-cold Chicago.

- **Action APIs:** Action APIs enable you to take action in near real time. Examples of action-type interfaces include push notifications, instrumented devices, and human-task management systems.

Chapter 2

Managing APIs — And Not

In This Chapter

- Reviewing API management roles and tasks
- Establishing API governance
- Seeing when to use unmanaged APIs

Part and parcel of many API conversations is the notion of API management. Even though APIs aren't pieces of software in the traditional sense (see Chapter 1), they're important elements of both business and IT operating environments, so they need to be managed appropriately. In this chapter, you find out how.

Seeing What It Means to Manage an API

To an API consumer, a great developer portal is everything. To an API provider, managing the API externalization and sharing processes is only the tip of the iceberg (see Figure 2-1). Under the waterline are the business and IT concerns that make APIs practical to create, deploy, and operate. These concerns include data mapping, security, rate throttling, monitoring, and version management.

A managed API not only has a well-defined interface and a defined target audience but also is under appropriately enforced business and IT controls. Different groups have specific parts to play in API management, as you see in this section.

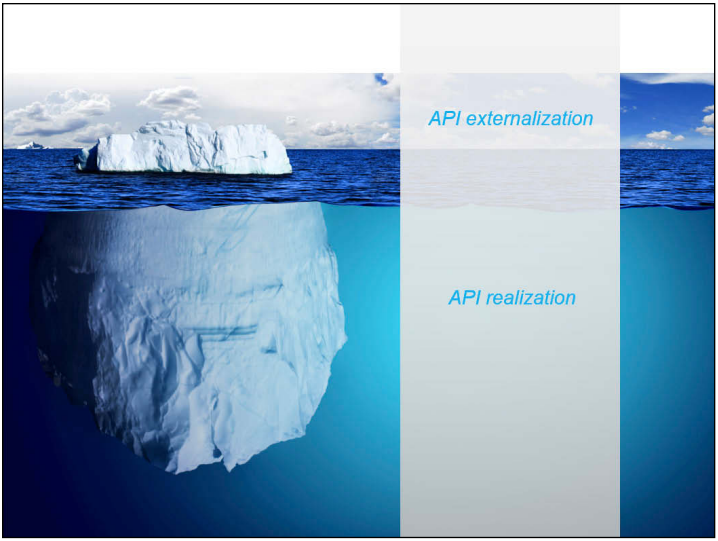


Figure 2-1: Managing APIs requires more than API design and externalization.

For maximum effectiveness, all three roles addressed in this chapter — business owner, IT operations, and API designer — need their own user-experience designs. Their concerns are sufficiently different that one role's tools are inefficient for another role.

API business owner

The API business owner decides the following:

- The plans (terms and conditions) under which the API can be consumed
- The communities that the API will be shared with
- Whether the API is succeeding in its objectives (if not, the business model needs adjustment)

All this can be done without changing the API definition or implementation in any way. For more about the business owner's role, see "The Need for API Governance" later in this chapter.

IT operations

IT operations must ensure certain operational characteristics, all of which can also be done without changing the API definition or implementation in any way. These characteristics are as follows:

- The runtime that hosts the API can be operated securely and robustly.
- The API is properly authenticated, and authorization is in place for anyone who uses it.
- API traffic is optimized and prioritized according to business needs.

There's a fundamental difference between what the API owner sells in terms of API plan access and what the underlying IT infrastructure can provide. Having spare capacity to support the full potential of traffic corresponding to API plans sold can be very costly.

To prevent prohibitive runtime costs, make sure that your API runtime is highly scalable (so that actual loads matter less) or apply traffic throttling when current load surpasses available capacity to smooth out traffic spikes. These capabilities should be available in your selected API runtime.

For more about IT operations' role, see "The Need for API Governance" later in this chapter.

API designer

The person that holds the API designer role physically creates and deploys the API. She needs to do the following:

- Define the API interface
- Discover back-end endpoints that may provide the data or function required to implement the API
- Configure the mapping between the API interface and the back-end data or function sources

The API designer must be able to perform these tasks without doing a lot of coding. As soon as creating an API becomes code-intensive rather than a matter of dynamic configuration, the rate of innovation inevitably slows, even for the most agile

development teams. The distinction between configuring an API and developing the back-end data or function embodying that API is fundamental to API thinking. As pointed out in Chapter 1, modern APIs aren't a piece of software; modern APIs are a flexible way of projecting capabilities to audiences outside your own team.

The Need for API Governance

A common myth in API circles is that governance bogs everything down. But governance is about making good decisions — making sure that the right people make the right decisions at the right time and for the right reasons, based on the right information. So if an API is important to your organization, you want to make good decisions about it. As an API provider or an API consumer, you have several decisions to make about APIs.

This governance is a different kind of governance from the type that's routinely applied as part of a software delivery life cycle; nevertheless, it's important.

API provider decisions

Deciding who can use the API under which business terms and conditions is the job of the API business owner. This business operational decision applies to all APIs, whether they're the APIs that your mobile development team uses to build mobile apps, the APIs that you use to integrate systems across various lines of business, or the APIs used by external consumers, so you probably want to make different decisions for each of these audiences in terms of what APIs they may use under which conditions.

IT operations also needs to make appropriate API provider decisions — typically, in the form of security and traffic policies — to protect the infrastructure from misuse or overload.

The governance regime needs to be very lightweight, and the decisions must be operational in nature as opposed to the typical life-cycle decisions made during a software delivery life cycle. If the right decisions can't be made and enforced easily, the open and dynamic nature of APIs is compromised

(remember, good API implementations are configurations, not code). Business and IT decisions are both part of good API management discipline and should be supported by the chosen API platform. (For more about API middleware, see Chapter 5.)

API consumer decisions

API consumers also need to make good decisions. In particular, they need to decide which APIs they're willing to use for what purposes and then ask the following questions about each API:

- What is the payment model for using the API and is that acceptable for your purpose?
- Will you need a corporate proxy in front of the API to handle licenses, payment, and the like, or will every developer register independently?
- Is the API secure and reliable for mission-critical purposes? Any historical records about how the API has behaved over time may add to consumer confidence in using it.

When the APIs being consumed are your own, these decisions are pretty straightforward, being mainly about overall business design. When the APIs are third-party APIs, the decisions become more complicated. Ultimately, the end-user experience and responsibility for maintaining business integrity can't be delegated. You need someone in your own organization to be responsible for the end-user experience and to make the right decisions about which APIs it's appropriate to consume as part of your delivery model.

Making the Case for Unmanaged APIs

Everyone knows that modern APIs must be powered by an API management solution, right? Not so fast. Not all APIs are necessarily managed. Throughout this chapter, I've defined good API management, but what does it mean for an API to be unmanaged?

Here are the key differences between managed and unmanaged APIs:

- An unmanaged API may have an intended target audience, but this target audience is rarely precisely defined, let alone enforced. If a user has network access to the API, generally, he can invoke it.
- An unmanaged API doesn't have independently enforced business and IT controls. Any control is provided via logic in the API implementation, usually in the form of code.

In other words, unmanaged APIs still have a well-defined interface, but there's no way to enforce controls on their runtime behavior or even on who may use them. So why would you want any API to be unmanaged?

If an API is a direct part of your business model, you probably don't want it to be unmanaged. Having said that, I give you a few examples of situations in which having unmanaged APIs is appropriate or even unavoidable:

- A device or sensor has a defined API as part of its physical reality, such as a home thermostat that can be programmed remotely or a Fitbit (a wearable device that monitors physical activity) that synchronizes data with a computer via a defined interface.
- An existing software system — perhaps a standard system such as SAP or a mainframe system with a native REST interface — exposes a micro API.
- The API sits inside your own domain, and all you really need is connectivity to access it.

Unmanaged APIs can be important resources in many ecosystems, making key functions and data available in a uniform fashion. The uniform consumption model is the reason why you still want to think of these interfaces as APIs. Often, you even want to catalog all the unmanaged APIs that are available to you, to make them easy to find and as simple as possible to use in a particular programming model.

Turning everything into an API, as seen from the consumer, is the easiest and most effective way of innovating and collaborating in a hybrid environment. This means that thinking APIs isn't just about API management and managed APIs. Thinking APIs should be part of a bigger integration strategy for turning your enterprise into an innovation engine.

Chapter 3

Discovering the Nature of Good APIs

In This Chapter

- Seeing why APIs are like race cars
- Using opportunistic APIs
- Combining APIs with Service Oriented Architecture
- Understanding good API design

Rapid innovation is enabled by good design — which, for any given API, includes its interface and technical characteristics. More important, good design is about which APIs to provide and when. If all you are trying to do is provide a handful of stable public APIs, perhaps the question of which ones is not so complicated. But what if you are trying to do that, as well as create a partner ecosystem, as well as use APIs to fuel internal innovation? There are many different kinds of APIs and uses for them, so in this chapter, you learn what makes a good API.

Comparing APIs with Race Cars

You could make an apt analogy between APIs and how Formula 1 racing teams build and evolve race cars (see Figure 3-1). In Formula 1 racing, every single car ever raced is a prototype. No team takes the same car to two consecutive races. A race car is built from rapidly replaceable components with well-defined interfaces, and the car itself is instrumented with built-in controls and analytics.

Although parts of the car may remain stable throughout the season, some component is always optimized based on lessons learned in the previous race.

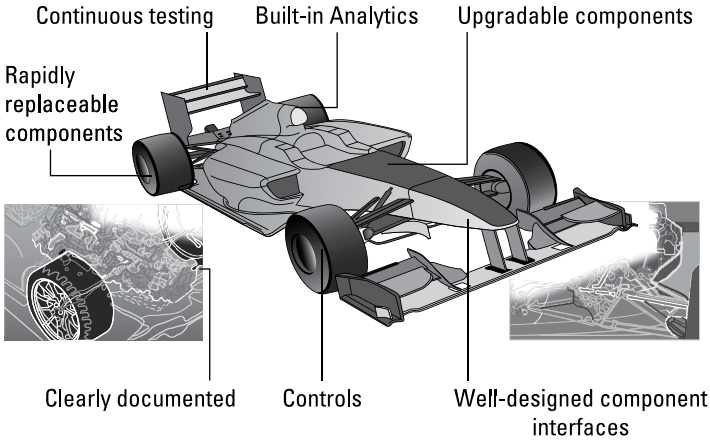


Figure 3-1: Race cars and APIs should be built to the same design principles.

Modern enterprises are in many ways like Formula 1 teams, always trying to optimize the business model and always looking for the right balance between change and stability. APIs are one way in which experimentation can be harnessed for enterprise advantage. "Try early, learn fast, and scale easily" is a good principle to apply to the world of APIs.

Making the Case for Opportunistic APIs

Should APIs always be designed to be reusable? Reusability implies stability over a relatively long period, and stability is appropriate if an API is to be used for partner integration or exposed externally as part of your business model. But if the API is created simply to improve collaboration between, say, a mobile development team and the team that maintains a back-end system of record, reusability may be neither desirable nor appropriate.

An API needs to be attractive to use, and for a developer, it must be faster and more expedient to use that API than to code a different solution. If the needs of a mobile developer change rapidly, the APIs must change just as fast to keep up.

This situation makes the case for *opportunistic APIs* — rapidly created, rapidly changing APIs defined to meet a specific consumer need. For more information about the developer point of view, see Chapter 1.

Nothing in the concept of an API requires it to necessarily be reusable or stable over time. The importance of reusability and stability depends solely on your business purpose for having the API. If that business purpose involves rapid change, opportunistic APIs are appropriate.

Providing opportunistic APIs requires that creating and maintaining APIs is both easy and cheap. Otherwise the cost of opportunistic change becomes impractically high. So API management software focuses on this challenge.

Good API management solutions create APIs via configuration rather than coding, and the task of creating or changing an API usually takes only minutes. The nature of an easily managed API is simply that it is both defined and controlled by configuration. Regardless of the cost of creation and maintenance, there's significant value in managing even opportunistic APIs properly (see Chapter 2).

Managing opportunistic APIs provides the following benefits:

- Definition and enforcement of business and IT controls
- Global insight into how an API performs from a business perspective
- IT operational flexibility for moving and dynamically scaling API workloads

These benefits are important aspects of try early, learn fast, and scale easily and are critical for optimizing change in a world of opportunities and innovation.

Thinking of APIs and Services

Service-oriented architecture (SOA) has been mainstream for about a decade; modern APIs are more recent. Both approaches to integration have their proponents and address business and IT concerns alike. What's the real difference between these approaches, and do you need to choose between them?

APIs versus services

The core concept of SOA is the notion of a service. The Open Group, for example, defines a service as "a logical representation of a repeatable activity that has a specified outcome." Services are self-contained and opaque to their consumers, and they have well-defined interaction contracts. From a technical perspective, these characteristics also apply to any well-designed API, so technically, an API *is* also a service.

In that case, are APIs just another name for services? Well, there's one important difference between services and APIs, however, and that's the goal behind their design (see Figure 3-2). APIs are always designed to be attractive to the intended consumer, and they change as the needs of the consumer change. Services, in contrast, are generally designed with global cost and stability as the most important concerns. In the car analogy, the API is the race car designed for looks and consumption, but the service is the regular car designed for cost and mass production.

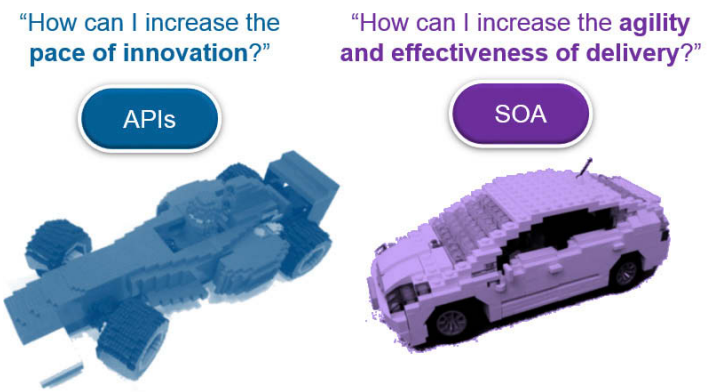


Figure 3-2: APIs and services address different concerns.

Chapter 1 describes the product nature of APIs and states that they should be aimed at the needs of a particular group of consumers. To a consumer, using an API is about speed, expediency, and having little to learn. Those design criteria

are the fundamental differences between APIs and the classical notion of services, at least different from the perspective of the service provider:

- To a service provider, reuse is about effort involved in API delivery. To API consumers, reuse is about speed of delivery of their software, no matter the cost to provide the APIs consumed as part of that sofware.
- To a service provider, sharing is about effectiveness. To an API consumer, sharing is about expediency (if it isn't expedient, the API will not be used).
- To a service provider, encapsulation is about having little to *change*. To an API consumer, encapsulation is about having little to *learn* (if the interface is complex, the API will not be used).

How often have you not seen an SOA initiative slowed down by conflicts between service providers and service consumers on what constitutes a good service interface? On the one side, a mobile developer just wants it to be simple for her particular app. On the other side, the back-end team wants everyone to use the same standardized service and data model. Instead of forcing a resolution of this conflict, is there a way to meet both needs without incurring prohibitive cost?

A historical analogy exists in the evolution of databases. The first generations of databases were focused exclusively on the internals, the tables, schemas, and data procedures. Quickly though, it became obvious that there was a need to expose controlled subsets of data in a particular form, fitted to a particular group of data consumers — and the notion of a data view emerged as a core capability in most modern databases, a lightweight proxy on top of the data domain represented by the internals of the database.

APIs are controlled (proxy) views of the data and capabilities of a domain, optimized for the needs of API consumers. As long as it's dirt cheap to create and maintain proxy APIs, you can use them to render a domain in multiple forms, optimized for each group of API consumers. (After all, you probably want to give external partners a different view of your capabilities from the view your internal developers have.)

APIs teaming with services

SOA emerged as a means of shielding service consumers from changes in the back end. But who protects the service providers from the churn of changing needs in omnichannel front-end solutions? Applying APIs and services together lets you create an eye of calm in a hurricane of change. Services are the means by which providers codify the base capabilities of their domains. APIs are the way in which those capabilities (services) are repackaged, productized, and shared in an easy-to-use form. APIs and services are complementary rather than contradictory, and applied together, they dramatically increase the overall effectiveness of enterprise innovation.

Recognizing Good API Design

No doubt the technical design of APIs is important, but it also varies widely with the technology choices made for the design and implementation of any particular API. For instance, what constitutes a well-designed REST interface is very different from what constitutes a well-designed SOAP interface. Entire books have been written about interface design, providing a level of detail way beyond what this small book can offer. Suffice it to say that interface design of APIs is generally well understood, as witnessed by these examples:

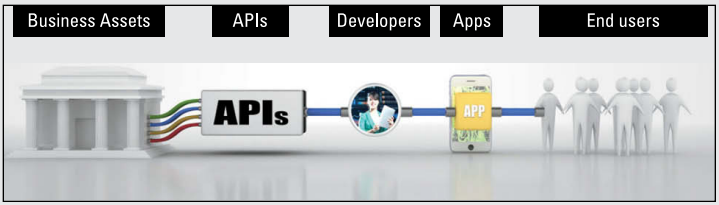
- REST interfaces are resource-based. The most important aspect of the interface design is the URI structure that allows a consumer to navigate the object graph embodied by the API.
- SOAP interfaces are method-based. The most important aspects of the interface design are the set of supported methods and the data structures of each method.
- MQTT interfaces are event-based. The most important aspects of the interface design are the set of events (emitted or received) and the associated event messages.

Not all APIs are REST. Generally, REST interfaces are excellent for human consumption, and they are the current preference of mobile developers. But REST interfaces tend to be chatty, and though they're extendable, they don't carry strongly

typed complex data structures. SOAP interfaces are great for system-to-system integration, and IT operations teams prefer them due to their less chatty nature and more precise data structures. MQTT interfaces are preferred for communicating with the Internet of Things, in which bandwidth and battery life are key concerns, and guaranteed delivery may be the difference between preventing accidents and inadvertently letting them happen.

There is more to understanding API value than the so-called API value chain

Much of the hype about APIs is centered on the API value chain illustrated in this sidebar figure — the way that APIs extend a business model into an open ecosystem.



Unfortunately most of the examples discussed in the industry at large are exclusively around public APIs, and this is by far not the only use case for APIs. Even more unfortunately, the generic value chain picture is uninteresting from the perspective of which APIs to provide and why. The reason being that it says nothing about the different types of business objectives that may have led you to consider APIs in the first place. Chapter 4 addresses that conceptual gap by defining the typical API entry points with associated decision criteria.

Chapter 4

API Entry Points

In This Chapter

- Monetizing your data
- Taking advantage of innovation
- Speeding the transition to mobile
- Going hybrid
- Programming everything

What does it really mean, to "think APIs"? Some people simply point to the so-called API value chain (see Chapter 3), but is that all there is to it? Is that what the excitement is all about? I believe that there is more to understanding APIs. As discussed in Chapter 2, decisions have to be made regarding the balance between enterprise and opportunistic APIs. There are decisions around the terms and conditions under which APIs can be shared, and there are decisions regarding how to map required APIs to existing assets during API implementation. All of these decisions and more depend on the desired business outcome.

The key part of the phrase "think APIs" is the first word: Think. Think about what you're trying to achieve business wise, which audience to engage, what kinds of APIs are required to engage the audience, and how to curate your data and application assets (as services) to support those APIs that you need to provide. In the process, don't forget to think about what APIs you're going to consume yourself and from whom. Thinking APIs is not just about being an API provider; many organizations consume several times the number of APIs that they provide. These concerns are the core elements of an effective API strategy.

This chapter defines five API entry points that, in my experience, are representative for the business and IT agendas driving API thinking. An enterprise may have multiple agendas at the same time, but each agenda leads to different decision criteria for API adoption. You can read the chapter from beginning to end or you can jump to the entry point that best matches your business needs.

Monetize Your Data

Monetizing your data is based on externalizing insights or functions in a form that entices third parties to use those insights or functions. Monetization can come in many forms. The most obvious example is when the third party is paying to use your API. In other cases, you may actually be the one paying the API consumer in return for a broader business reach and a stronger ecosystem. Or you may be onboarding partners via APIs without any direct payment involved at all. The key business objectives for monetization are pivoting your business, changing your value chain, and increasing your reach and influence.

Success with this entry point likely requires careful planning. Although you can and should do some experimentation, "opportunistic style," the final product must be a set of stable enterprise APIs that a third party can depend on for a prolonged period.

Thinking about APIs in the context of monetizing your data, here is some guidance:

- **Goal:** The desired outcome is monetary or based on a nonmonetary value such as increasing your influence.
- **Audience:** The audience is inevitably a third party. Typical cases are partners and external developers (not developers in your own organization or developers hired by your own organization). Treating a different line of business within your enterprise as a partner or third party isn't uncommon, particularly when the different lines of business are treated as economically independent entities.
- **APIs to provide:** The APIs required to engage the audience must provide the value you want to sell. This choice

requires careful thought — not just in terms of the root value provided, but also in terms of the form that makes consumption attractive.

- **API terms and conditions:** Don't forget to include in your considerations the terms and conditions under which API consumption may happen, such as freemium, pay as you go, or prepaid contract.
- **API implementation:** The way you curate the data and function to implement the APIs comes down to quality and reliability. Some people say that cost of implementation is the most important factor, but implementation cost won't make or break a data-monetizing strategy. What decides long-term viability is whether the intended API consumers experience both value and trustworthiness.
- **Consuming somebody else's APIs:** In some cases, you also need to consider which APIs to consume. Although you generate value primarily by providing APIs for others to consume, implementing those APIs may involve creating higher-level composites of existing APIs, most often by blending those existing APIs with something that's uniquely yours.

The data-monetizing entry point may be the one you've heard the most about, but it's not the most common one. Currently, a significant majority of API initiatives are for internal use cases, and this may remain the case even when public API exchanges become fully mature.

Freedom to Innovate

Freedom to innovate is the most important imperative for many businesses today. Try early, learn fast, scale easily — key characteristics of a dynamic, engaging enterprise. The focus of this API entry point is to chase business opportunities aggressively and to make innovation a learning process through the following model:

- Everything is a prototype until it's proved in practice.
- Discovering that something didn't work isn't failure; it's just learning.
- Standing still guarantees decline.

As discussed in Chapter 3, the role of APIs is to provide the calm center in a storm of change. This role has two aspects:

- Delivering quickly what the experimenting API consumer needs and removing it when it's no longer needed
- Protecting the provider from churn (see Chapter 3 for a refresher on why defining and deploying new API "views" on existing assets should be easy and cheap)

The Freedom to Innovate entry point is perhaps not as glamorous as Monetize your Data (see the preceding section), but it's by far the API use case that IBM sees the most in major enterprises. The ability to compose new innovative capabilities, internal and/or external, without breaking the bank in terms of cost, is something that everyone is struggling with. To accelerate innovation, a blend of careful planning and opportunistic reaction is required.

Thinking about APIs in the context of freedom to innovate, here is some guidance:

- **Goal:** The desired outcome is to quickly discover what works and what differentiates in the market and then to scale successes.

  Even when you believe that you know what the market needs, a learning attitude can deliver a healthy reality check. It's easy to get symptom and cause mixed up when you're dealing with complex market dynamics.
- **Audience:** The audience is primarily internal developers, in-house or outsourced. Sometimes, external agencies are hired to be part of an innovation effort. In these cases, more formal agreements about which APIs to provide and when are required, but even then, it's best to maintain some opportunistic behavior to maintain learning speed. Contract negotiations are anathema to fast innovation.
- **APIs to provide:** The APIs required to engage the audience are a blend of predefined enterprise APIs and opportunistic APIs (see Chapter 3) derived from the needs of an innovative app. Mixing in some enterprise APIs to expose core data in an easily accessible fashion can kick-start not only development, but also idea generation.

Keep the predefined APIs small and simple. Exposing the full data structure of a customer back-end system probably wouldn't be easy for an innovative channel developer to consume.

- **API terms and conditions:** The terms and conditions under which API consumption happen remain important — not in terms of payment, but in terms of protecting the security and stability of back-end systems. After all, innovation is unpredictable.
- **API implementation:** The way you curate the data and functions required to implement the APIs is different for preplanned enterprise APIs and opportunistic, demand-driven APIs:
    - *For preplanned APIs,* you must make careful decisions about which data segments to expose at an organizational level. Also, your implementation (typically proxy'ing enterprise services) needs to take ultimate runtime cost into account. Preplanned APIs will be used in unpredictable ways, so runtime cost must not be a preventing factor for such use.
    - *For opportunistic APIs,* the most important considerations are development speed and development cost. If you have to do a hack to get the API out the door today rather than next week, do so as long as you have a viable plan for cleaning up the API implementation if and when that API becomes a success.

      Many opportunistic APIs may not live very long. If it turns out that you need something different, just scratch the opportunistic API and start over in the next iteration of the learning process.
- **Consuming somebody else's APIs:** Considering which third-party APIs to consume is important for this entry point. As a simple example, building social mobile apps is difficult without accessing APIs from well-known public social services such as Twitter, Facebook, and LinkedIn. These third-party APIs should be part of the API catalog that you provide your internal developers; they shouldn't have to go to some external site to find things themselves. You may even want to curate the third-party APIs into your own, simpler version, as some of the public APIs are quite complex in their native forms.

Innovation is never easy, but it can be aided in various ways. Proper use of APIs can make the corporate back end an integral part of your innovation engine rather than a millstone around your neck. Enterprises with a long history have the advantage of having more assets to expose as APIs. But even for startups, an API-driven approach to implementing innovative solutions provides more flexibility in terms of sourcing data and function. The use of APIs frees channel developers to focus on the user experience rather than integration. The use of APIs also promotes an omnichannel experience, as the data and functions behind the API by definition are remote and can be accessed from any channel or application across the enterprise.

Mobile In Ten Minutes

This API entry point is strongly related to the freedom-to-innovate entry point (see the preceding section) and can be considered to be an extreme variant. The difference is that for this entry point, everything is about opportunistically supporting what your mobile development team needs here and now.

Enterprises today need to create many small, self-contained apps, rather than traditional comprehensive portals. I strongly believe that mobile consumers prefer to orchestrate their own omnichannel experiences instead of having someone else predefine the process. I see it in my own behavior where I rarely spend more than a couple of minutes in a given mobile app before moving on to something else.

The mobile in ten minutes entry point involves improving collaboration between back-end owners and mobile development teams. Those mobile development teams can be internal or external, but in all cases they need to leverage existing data and functions easily to provide an engaging experience. Being able to see stock prices is useful, for example, but being able to do something with them in the context of your own investment portfolio is the true differentiator.

The focus of this entry point is controlled simplicity. Hide complexity, simplify what the mobile developer sees and consumes, and at the same time provide appropriate business and IT operational control. Also, the process has to be fast in order to not slow down mobile innovation.

Just for the fun of it, some IBM developers tested whether it was possible to take a piece of data on a mainframe and put it on a mobile device, using an API approach, in 10 to 15 minutes. It was indeed possible! There wasn't any pretty API design, but it proved that the complexity of integration logic can largely be taken out of the equation by appropriate use of API and integration technology.

Mobile in ten minutes is all about opportunistic innovation. Thinking about APIs in that context, here is some guidance:

- **Goal:** The desired outcome is immediate support for the needs of your mobile development teams. The mobile team is responsible for figuring out what data the end user experience requires, and the back-end owner builds the APIs that deliver exactly that data. Mapping to existing data and functions (services) is part of the API implementation, not the responsibility of the mobile developer.
- **Audience:** The audience for the APIs is your mobile development team, period.
- **APIs to provide:** The APIs required to engage the audience are almost exclusively opportunistic. You may be lucky enough to be able to apply existing APIs to a new mobile app, but you want to avoid creating too many cross-app dependencies on particular API contracts.
- **API terms and conditions:** The terms and conditions under which API consumption happen are about protecting the security and stability of back-end systems. The same is true for Freedom to Innovate (see the preceding section).
- **API implementation:** As the APIs produced are very opportunistic and rarely live long, the most important considerations are development speed and development cost. If you have to do a hack to get the API out the door today rather than next week, do so as long as you have a viable plan for cleaning up the API implementation if and when that API becomes a success.
- **Consuming somebody else's APIs:** Engaging mobile applications typically need a social element, so it's crucial to consider which public social APIs (such as Twitter, Facebook, and LinkedIn) to consume and how to control that consumption. You may want to curate the third-party social APIs into your own, simpler version

(refer to "Freedom to Innovate" earlier in this chapter). At minimum, you must consider the dark side of APIs (see Chapter 6) and try to minimize the impact on your business reputation if any stability or ethical issues arise from the public APIs that you end up using.

The opportunistic approach of this API entry point may sound unmanageable, due to the number of APIs involved and their inevitably limited reuse — and indeed with the technology available even three to four years ago, it would have been practically unfeasible for many large enterprises. Not so today! API management software changes the equation with its ability to easily delineate a lightweight consumer-centric API from the portfolio of software-based enterprise data and services. If it's dead easy to create a new API, and if it's simple to manage and share large numbers of APIs with various communities, reuse and architectural rigor on the API interfaces are much less of a concern. Instead, the main concern is what will make the API consumer successful.

Living in a Hybrid World

No book about APIs would be complete without discussing how APIs relate to cloud. The fourth API entry point focuses on using APIs as the uniform consumption model in a hybrid ecosystem of on-premise systems and private and public cloud environments. The business mantra is "freedom of choice" — freedom to choose how to source any function or data and freedom to deploy solutions to any desired form factor.

An API-centric approach to integration is driven by the following:

- A disconnected enterprise isn't competitive, so cloud equals hybrid solutions across cloud parts and on-premise parts, and integration needs to be managed at scale.
- Cloud is about capability (business and IT), not location, so any ubiquitous consumption model needs to be location-independent.
- In a "system of systems" world, there's no traditional network perimeter to enforce, so interactions need to be controlled at the application level.

If to a consumer everything is a (remote-able) API, then the consumer doesn't need to know anything about where and how that API is hosted. Syndicated API catalogs can and should provide visibility across domain and provider boundaries. In this API entry point, one of the most important aids for a developer is the catalog of APIs that are readily available for consumption. Don't show every single API out there (there are too many); show only the ones that are relevant to the developer in question. That developer shouldn't have to care about how and why the API is procured; she should focus solely on what she can do with the API after it has been made available to her.

In a hybrid world, the domain structure is inevitably complex. Public marketplaces, private API catalogs, partner portals, and more are part of the API fabric. Consequently, a well-defined community structure is more important than ever, with direct correlation between community design decisions and API design decisions.

Figure 4-1 illustrates how a developer can use APIs to securely access any part of a hybrid environment. At development time, the API marketplace provides information about the APIs that are available to the community of that developer. At runtime, the cloud gateways secure the communication between the API consumer environment and any API endpoint, independently of location.

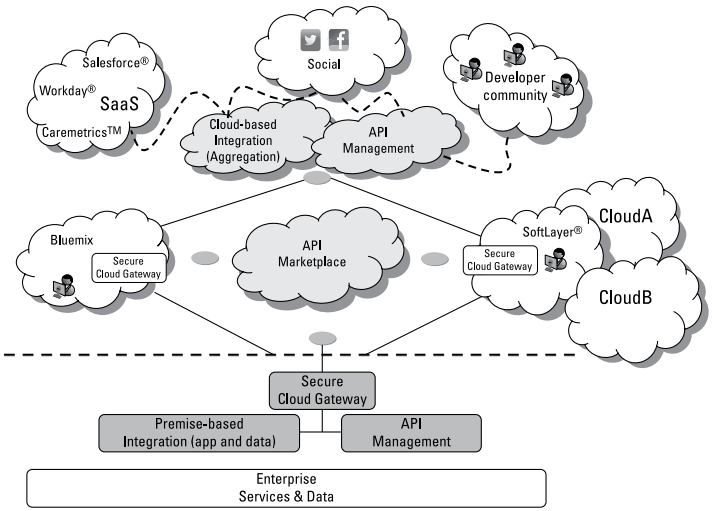


Figure 4-1: Using APIs as the lingua franca for a hybrid environment.

Thinking about APIs in the context of living in a hybrid world, here is some guidance:

- **Goal:** The desired outcome is to empower both developers and IT operations. Careful controls need to be placed on the APIs made available, and any communication on an open network needs to be secure and appropriately managed.

  The most-often-cited reason for not adopting a hybrid cloud approach is security concerns. The second-most-often-cited reason is fear of losing operational control.

- **Audience:** The audience can be any mix of internal and external developers who are part of the hybrid ecosystem, so designing appropriate community structures is very important. Making decisions at the single-person level is highly impractical; you need a structure that allows you to make and enforce API sharing decisions at the community level, treating all developers in a particular community in the same fashion.

- **APIs to provide:** The APIs you need to engage the audience depend on the audience community structure. For external audiences, you probably need a good set of predefined enterprise APIs; for internal audiences, you need some opportunistic APIs to support rapid creation of innovative apps. No hard-and-fast rule applies to all hybrid cases; each hybrid case tends to be different.

- **API terms and conditions:** The terms and conditions under which API consumption happens can be complicated in a hybrid world — business-wise as well as IT-wise. This holds true whether the APIs are managed or unmanaged (see Chapter 2), yet the implementation mechanisms usually are very different for the two:

  - *For managed APIs,* you apply business and IT policies in the normal fashion on the API platform — this is one of the advantages of using APIs as the uniform consumption model. Just make sure that your API platform is community-aware so that you can make decisions at the appropriate community level.

  - *For unmanaged APIs,* the only vehicle for enforcing business terms and conditions is community visibility, along with any formal agreements that you establish through side channels. Where unmanaged APIs

are concerned, IT operations has no built-in way of enforcing API-level security and traffic controls; unmanaged APIs must instead be invoked through secure tunnels established at the network level.

- **API implementation:** The way that you curate the data and functions to implement the APIs differs for preplanned enterprise APIs and opportunistic, demand-driven APIs. As described in "Freedom to Innovate" earlier in this chapter, the key differences are robustness and runtime cost versus time and development cost.
- **Consuming somebody else's APIs:** Considering which third-party APIs to consume is harder for this entry point than for any of the other entry points because over time, the number and variety of available APIs is dramatically greater.

The best advice is to start simple on API consumption. Pick a small number of important APIs that you want to consume: social APIs, analytical APIs, mobile back-end APIs, or something else, depending on your most immediate business need. You also have the option of curating third-party APIs into your own simpler or more controlled version, so take that option into account in your decision-making.

Hybrid environments are intrinsically complex. Using APIs as the lingua franca can make that complexity much more manageable from a developer perspective. From an IT operational perspective, complexity may become manageable due to the evolution in hybrid cloud platforms.

Program Your World

The final API entry point focuses on the world of devices and machinery. It's based on two beliefs: The consumer-driven mobile revolution is about more than just phones, and manufacturing and logistics will drive the intelligent corporate Internet of Things.

Healthcare, utilities, cities, manufacturing — practically everywhere you turn, you see the need for a more intelligent approach to blending people, software, and machines. Granted, the idea of programming everything into a single intelligent experience is still emerging, but signs of progress

toward that goal are seen every day in such examples as smart cars, interactive retail experiences, and ecofriendly buildings.

One important difference separates this entry point from the others: You generally can't change a device after it exists in its physical form. That fact in turn means that the device APIs remain what they are. Therefore, the focus shifts to providing an enriched experience with optimized execution, as follows:

- Extend software into the physical realm.
- Control any component, software, or device through programmable APIs.
- Use device-driven feedback and insight to optimize the behavior of the entire system, not just a single component.
- Where appropriate, monetize your high-level control capabilities as your own set of APIs (refer to "Monetize Your Data" earlier in this chapter).

The programming-the-world entry point usually is much more about API consumption than it is about providing new APIs.

Thinking about APIs in the context of programming the world around you, here is some guidance:

- **Goal:** The desired outcome is a completely programmable environment. Look for a programmable physical infrastructure, and build your own software to be controllable through APIs as well.
- **Audience:** The audience is mostly internal. Your own developers are the ones who'll be building smart systems on top of device and software APIs. If you're a producer of devices yourself, you do need to make sure that your own devices have APIs intended for consumption by others as required by your business model.
- **APIs to provide:** The APIs you need to engage the audience are largely defined by whatever interfaces are designed into your physical devices and machinery. You don't have a whole lot of choice, as the majority of the APIs you'll be using are predefined.
- **API terms and conditions:** If you have network access to a device, you can most likely communicate with it

because its APIs are by definition unmanaged. Terms and conditions are not as important for this entry point as they are for the others. Much more important is having a good library of available device APIs.

In some cases you do want to control access to the device-level APIs and can do so by adding in front of the device APIs a layer of managed proxy APIs with built-in security controls.

- **API Implementation:** For device APIs, the methods used to curate the data and functions to implement the APIs aren't your concern (unless, of course, *you* are a producer of physical devices).
- **Consuming somebody else's APIs:** Considering which third-party APIs to consume is very important for this API entry point. Having the right agreements in place with third-party suppliers of devices and machinery is critical. If you don't have updated API documentation, you simply can't communicate effectively with the device.

As the world gets smarter and more instrumented, the need for Program your world capabilities will continue to rise. Because this environment is quite different from a classical software-only environment, getting an early start may be necessary. Embarking on this journey requires developers and businesspeople alike to get experience in what it means to seamlessly program interactions among devices, software, and people.

Chapter 5

API and Integration Middleware

In This Chapter

- Understanding API middleware
- Choosing a domain topology

As I discuss in earlier chapters, there are key distinctions between managed and unmanaged APIs. A managed API has appropriately enforced business and IT controls, which in turn begs the question of where and how those controls are enforced at runtime.

Enforcing the controls in code generally is a bad idea. For one thing, it's notoriously unreliable, depending on the code quality of every single API implementation. Furthermore, an organization really needs to be able to change the controls without having to change and redeploy the API itself. In other words, the controls should be policy-driven, with independently managed policies defining the (business and IT) operational intent and the API runtime enforcing that intent. Examples of common policies are the level of security required and the amount of traffic allowed.

A piece of middleware that hosts APIs and enforces API policies is commonly called an *API gateway.* Don't forget that API platforms aren't just about the runtime; they need development time capabilities as well.

API Middleware Isn't "just another ESB"

Have you ever heard someone say, "I already have an enterprise service bus (ESB), so I'm all set for APIs"? The implication is that API middleware is "just another ESB." In fact, there are significant differences:

- Ideally, APIs are defined by configuration rather than coding. Defining the API by configuration preserves the lightweight nature of the API proxy and supports fast turnaround for new and changed APIs. By contrast, general-purpose ESB tools are flow- or code-centric, and service implementations are part of the enterprise software delivery life cycle. Even with continuous delivery, a code-centric approach is difficult to maintain while attempting to meet the need for rapidly changing opportunistic APIs (see Chapter 2).
- API runtimes need to be lightning-fast, completely secure, robust, and highly scalable. Ideally, the API gateway has routerlike network characteristics, so adding an API proxy as part of an end-to-end interaction never creates problems with response time or latency. The fastest and most scalable API gateways are based on domain-specific, lightweight configuration languages with stateless execution. Although general-purpose ESB runtimes also need to be fast, robust, and scalable, the execution engines must be able to support full-scale composition and some amount of statefulness. Consequently, general-purpose ESB runtimes can't be as highly optimized for throughput as API runtimes.
- APIs have policy-driven business and IT controls ranging from authentication to traffic controls and the business terms and conditions under which the API may be consumed (the API plan). Although some comprehensive ESBs include traffic-management policies, few have the security-policy capabilities and certification of a full-scale API gateway, and none has the separate business controls embodied in an API plan. Furthermore, the software nature of ESB executables typically provides less room for operations to control runtime objects independently after deployment.

- APIs must be made available to app developers in a self-service fashion. Any kind of post-deployment approval process slows the adoption rate and ultimately, at scale, generates significant organizational costs.

  The preferred sharing mechanism — one that has proved to be highly effective — is publishing APIs to developer portal environments. In particular for internal API use, sharing is preferably managed on a community basis, making sure that given developers see only the APIs that their community is supposed to use.

  General-purpose ESBs include none of these features, and even service management solutions (whether embedded or separate) are aimed more at service development time controls and operational controls than at optimizing the developer sharing process.

- Finally, API owners need business statistics about who uses their APIs and how much. These statistics measure success against the business objectives for the portfolio of APIs rather than focusing on IT concerns such as the operational dashboards typically included in ESB platforms.

Some people may argue that you need only one bus, which can be repurposed to support classical integration needs, service-oriented architecture (SOA), and also API management. Yet without providing dedicated experiences for the API developer, API owner, and API consumer, you risk engineering the middleware for the lowest common denominator. Not to mention that the classical integration developers also need their separate optimized experience. Further, if you need only API management or general-purpose integration capabilities, why pay for both?

Modern enterprises need API management as well as enterprise integration, but as two separate platforms aimed at different target audiences. Do remember, though, that if you are already doing SOA, you have a good set of assets that you can quickly discover, assemble into (proxy) APIs, and expose through an API management platform.

The (Repeatable) Topology of Integration

An API gateway controls traffic across the boundary between API consumer and API implementation; it also enforces the terms and conditions for API consumption defined by the API owner. Historically, we've tended to think of different types of gateways — be they web, security, messaging, or API gateways — as being distinct "boxes" in a deployment topology. In a hybrid world of devices, people, and software, however, the perception of distinct gateway boxes is rapidly evaporating. As the scale and variety of interaction grow by orders of magnitude, it's operationally advantageous to have a unified gateway strategy with a single command-and-control center for any traffic that crosses internal or external boundaries.

As a good practice, traffic across a domain boundary should always happen through an API. This approach to integration creates a desirable level of visibility and control. Furthermore, it provides maximum flexibility in terms of balancing enterprise and opportunistic implementations in a way that's completely invisible to the API consumer.

The choice of a coarse-grained or fine-grained domain topology is up to the individual enterprise. Typical domain boundaries are driven by the following:

- Organization (such as mobile developer versus back-end team)
- Ownership (such as line-of-business boundaries)
- Security (such as demilitarized zone [DMZ] versus internal network)
- Quality of service (such as mainframe versus distributed)
- Ecosystem (such as public or partner)

It's important that this suggested integration topology model be both repeatable and composeable. For each domain within an ecosystem, good practice dictates the same topology structure, with cross-domain interactions at runtime always going through a (API) gateway as illustrated by Figure 5-1. In the figure each separate domain has the same structure of

concentric circles with different types of integration capabilities. The standard integration topology for each domain can be composed to a system-of-systems topology across multiple domains simply by repeating the pattern. In other words, the domain structure can be defined by the needs of the enterprise, yet the API sharing and consumption mechanisms remain the same. This structure crisply addresses the system-of-systems nature of hybrid integration environments and is a good fit for any API entry point defined in Chapter 4.

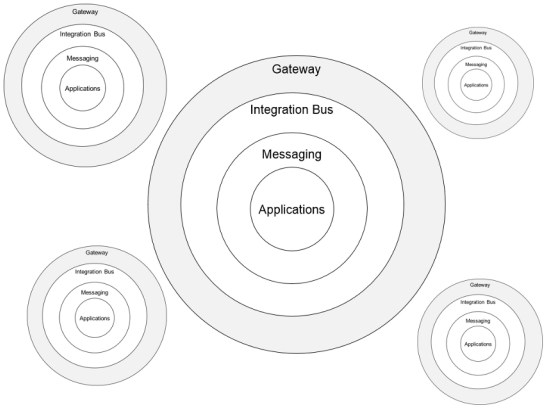


Figure 5-1: The topology of integration is a repeatable pattern.

Some people would argue that latency or response time is an issue in a pattern that always goes through an API for cross-domain interaction. Good API gateways make latency and response time nonissues, as outlined in "API Middleware Isn't Just another ESB" earlier in this chapter. As an analogy, how often do you ponder the number of routers between a browser and a web server? Hardly ever. You just assume that the router infrastructure is fast enough and scalable enough that the number of routers isn't an issue. Granted, the actual network distance between two points can and will add some amount of latency, but this latency is due to network distance rather than the number of network routers involved. Good API gateways are like routers in this respect; adding one of these gateways on the path from API consumer to API provider makes no noticeable difference.

Use general-purpose ESBs in the integration bus role, curating data and function into well-formed services. Then define (proxy) APIs to control the productization of those services in the ecosystem beyond your own domain. Finally, run all API traffic through a highly scalable, robust, and secure API gateway. That way, each part of the end-to-end integration platform is used in the area of its main strength (refer to Figure 5-1).

If you're interested in further details on the nonoverlapping capabilities of an end-to-end integration topology, see the IBM Redbook *Integration Throughout and Beyond the Enterprise* at `www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/sg248188.html?Open`.

API and Service-Economy Reference Model

Back in the 2000s, IBM defined an SOA reference model that outlined the middleware capabilities required to create, deploy, and manage services effectively. In the API economy, those capabilities are still important, as services are the basis on which APIs can be defined, but they're no longer sufficient. Embracing an open ecosystem, delivering APIs as part of a business product, and carefully promoting and managing your cross domain business persona require more than classical integration middleware.

The reference model for this extended middleware platform is defined in Figure 5-2. It is a practical guideline for the set of middleware capabilities to seek across your integration and API platforms.

The integration-centric elements of the original IBM SOA reference model are rendered in compressed form as the bottom layer of the API and service-economy reference model. In addition, the new reference model has capabilities for defining and managing APIs, for developer portals and marketplaces, as well as capabilities that accelerate the consumption of APIs.

It's important not to make decisions about the parts of the reference model in isolation. Instead, use an integrated middleware strategy to turn assets into business advantages.

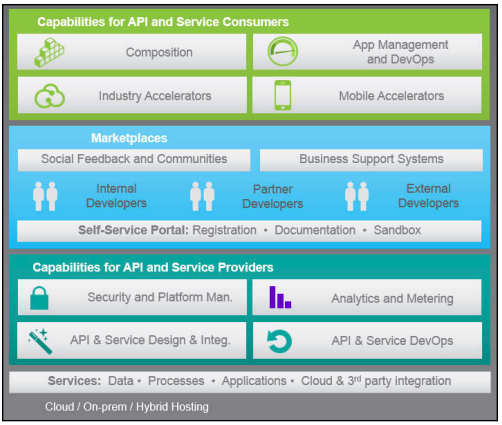


Figure 5-2: The API and service economy reference model.

Chapter 6

Ten Things to Know about APIs

In This Chapter

- Seeing that APIs are products
- Recognizing that design never stops
- Understanding that every API needs an owner

Part of an API journey is the mental mindset that lets an organization think about APIs in an effective fashion. This chapter contains some of the things that I have learned on my own journey through the world of APIs.

The Omnichannel Experience Drives the Need for APIs

The basic concept of APIs isn't new. The difference now is that modern users (consumer and enterprise) expect an omnichannel experience that's both social and personal. To be truly personal, the experience must be self-orchestrated, at least to some degree. No longer can an enterprise define a "one size fits everything" channel process.

Self-orchestration inevitably points in the direction of micro apps, which in turn lead to the need for purpose-built APIs. An omnichannel experience implies an ecosystem that includes people, software, and devices, which again leads to the need for purpose-built APIs.

APIs Are Business Products

Thinking of APIs as business products makes it easier to tell the difference between an API-centric approach and a classical software-delivery approach. For products, you have several key questions to ask:

- Who is the audience?
- What do they want to buy?
- What are the terms and conditions under which I'm willing to sell?

The terms *buy* and *sell* are used deliberately even though the economic models behind APIs vary widely. Whether the "price" is cash or influence, whether the model is consumer-paid or provider-paid, the product nature of the API persists.

Business Design Is an End-to-End Endeavor

APIs are no longer just IT concerns. APIs should be part of your end-to-end business design.

Consider a traveler who shares her experiences on social channels. One day, she tweets about a really bad experience with Airline A. Ten minutes later, she receives an email from the airline that says: "We're sorry about your bad experience. Here is what we can do for you." The next week, she has a great travel experience with Airline B, and as usual, she tweets about it. Five minutes later, that airline retweets her tweet with this added text: "Happy you had a good experience. See you next time."

Both these airlines considered in advance how to weave social channels, through the use of APIs, into their overall business operating models.

You Can Gain Insight from API Instrumentation

Try early, learn fast, scale easily — part of that recipe is the ability to learn fast, and the best way to learn fast is to tap into information already flowing through the business operating system. You can access this information easily via instrumentation of APIs and use of associated business analytics — capabilities that should be part of a fully functional API middleware platform.

Not All APIs Are REST

A common myth is "SOAP is dead; APIs are always REST." Although most modern APIs are currently based on REST/JSON rather than SOAP, this fact doesn't mean that there's no use for the formalism of SOAP — merely that such formalism isn't necessary for human consumption of APIs (the focus of most current API discussions). There are still plenty of uses for SOAP in machine-to-machine communication and formal composition tools. There are good reasons why the industry has invented more than one binding protocol; good designers choose the right one for the purpose.

Every API Needs a Business Owner

Simply put, no business API owner equals no responsibility and no decision-making power. It's human nature for people to resist owning things that they don't control completely (as in "Only this part is my responsibility"). Nevertheless, you must assign a business owner to every single API. That person then becomes the focal point of decisions about productization and sharing.

APIs Do Need to Be Versioned

Saying that APIs aren't versioned is like saying that you don't need to change a baby's diaper. Obviously, disruptive changes happen, and when they do, you have to handle them. Claiming that versioning isn't necessary merely means letting the consumers of an API figure out the versioning themselves.

Because APIs are business products, you should version them wisely. Reduce version churn by coining a new official version only when an update isn't backward-compatible.

APIs Are Easy to Control with Policies

Policies are the traditional ways that business and IT operations intentions get codified; therefore, they're vehicles for business consistency and integrity. The terms and conditions under which APIs are consumed and managed should be codified as policies, which in turn are enforced by API platforms.

It's important to make sure that policies can be changed independently of the logic embodying the API itself (such as interface and data mapping). This supports dynamic change of operating behavior.

APIs Have a Dark Side

Innovation without business integrity is a fragile value proposition. Whether the breach of integrity is a broken promise, exposure of sensitive information, or inappropriate behavior, the result is largely the same. It takes only one bad experience for someone to lose faith in you.

When you use a third-party API, take care that your own business's integrity won't be negatively affected. The vehicle you use — formal agreements with penalties, compensation mechanisms, or judicious evaluation of API robustness and security — matters less than the fact that you've taken proper precautions. Remember to include ethical concerns in your consideration.

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.