Microsoft Azure

APIs

+MICROSERVICES

The ultimate guide to creating an enterprise API platform

TABLE OF CONTENTS

3 Intro

4 Understanding web APIs

11 Benefits of moving to an API-centric enterprise

14 The shift to API platform thinking

24 The emergence of microservices

30 Components of an API platform

36 Planning your move to an API-centric organization

47 Conclusion

INTRO

APIs exist at the intersection of business, products, and technologies. They power customer experience, business relationships, and internal innovation.

Unlike the many choices that a technical leader must make regarding programming languages, libraries, and infrastructure, APIs have a direct impact on the speed of software delivery within a business. Therefore, leaders must not leave an organization’s API strategy up to their developers as they build new APIs. Instead, it requires a thoughtful and well-planned approach across the entire organization.

This e-book is designed for technical leaders tasked with establishing a new API program for their organization or maturing an existing program. It will provide insights and decision factors based on established practices in organizations with successful API programs.

What are web APIs?

As a review, an application programming interface (API) specifies how software components and systems should interact with each other. Web APIs extend this interaction beyond a single application by using HTTP, the language of the web, as the network protocol.

A web API doesn't have to be RESTful. It doesn't have to use SOAP. It doesn't have to use JSON or XML or OAuth or be built in a specific programming language or framework. It doesn't have to have pretty URLs. Web APIs may exhibit some, all, or none of these traits. The only requirement for a web API is that it allows one program or software component to interact with another in a repeatable way over HTTP.

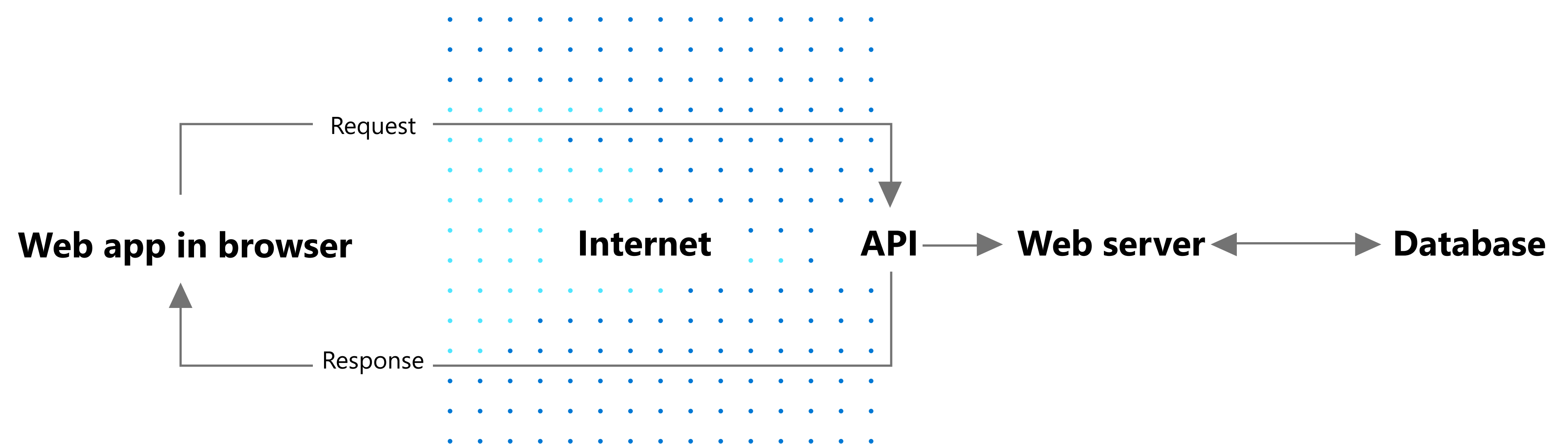


Diagram of a simple web-based application that retrieves data from a database via an API.

THE GROWTH OF WEB AND MOBILE APIs

There are three primary reasons why web APIs have gained traction over the last decade:

Simplicity

Web APIs speak a simple language: HTTP, the language of the web. This promotes easy integration from any programming language—no special libraries required.

Lower cost

Open source technologies have reduced the cost of building web APIs compared with the previous generation of technologies, which required expensive middleware and training to deliver them successfully.

Higher demand

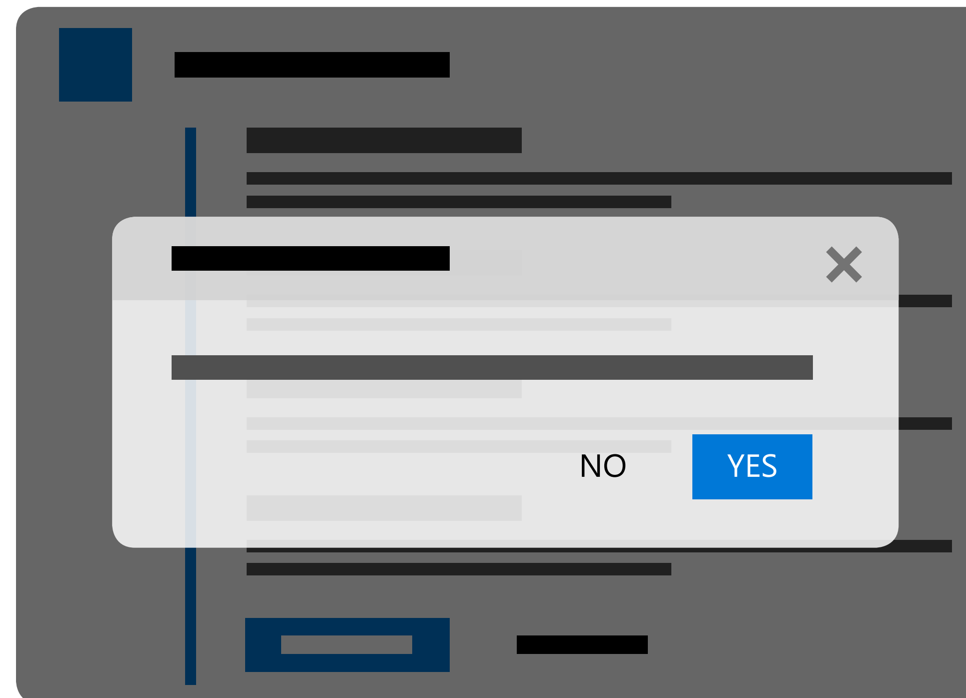
From single-page applications (SPAs) to native and web-based mobile apps, the need for mobile accessibility and anytime, anywhere access has shifted the focus from HTML-based delivery to the separation of data and capabilities from the presentation layer.

APIs as capabilities

THE MOST IMPORTANT PRODUCTS

in your organization are the business and technical capabilities your applications deliver. These capabilities, when combined into solutions, enable your customers, your partners, and your workforce to solve everyday problems and achieve their desired outcomes.

With the emergence of messaging platforms, bots, and voice interfaces, things are dramatically changing. We are starting to see the focus shift from building applications to delivering capabilities via APIs that are then integrated into these platforms. Rather than users going to an application to get things done, we are now experiencing the shift to applications going to the user through these third-party platforms.



Applications are now going to the user through messaging platforms such as Slack, Cisco Teams, and Microsoft Teams.

Web APIs deliver capabilities in a variety of roles

PRIVATE APIs

Hidden behind the firewall of many companies are private APIs that power day-to-day operations. These private APIs may range from APIs that power mobile apps to internal ones used by the workforce. Examples include: managing inventory, booking hotel rooms, and account management.

PARTNER APIs

Partner APIs, such as those offered by Sabre and Capital One, are focused on an organization's set of development partners. By limiting an API to partners, companies can tightly control who uses an API and how it is used outside of their organization.

SAAS APIs

Companies such as GitHub, Stripe, and SendGrid take advantage of software-as-a-service APIs to access their hosted data. They may choose to use SaaS through the existing user interface, through the API, or a combination of both. SaaS vendors use APIs to reduce customer churn by helping their customers integrate SaaS into their daily workflow.

PLATFORM APIs

Platform APIs extend the reach of SaaS providers like Microsoft Dynamics, Fitbit, and API2Cart by bringing together multiple parties within a marketplace. By reaching beyond the typical provider-consumer relationship found in SaaS, platforms often provide increased value through innovation and a greater overall network.

APIs as products

Not only are web APIs used to provide capabilities within an organization, they are also becoming products themselves. A recent example is Twilio, which started by offering a simple way to automate text and voice interactions. Over time, it has expanded to support a variety of capabilities from fax to email. Through the application of rigorous product management techniques, Twilio took what seems like a developer-focused product and became one of the first organizations to go public with an IPO on the US stock market while only offering an API-based product.

Organizations can use the same techniques that have led Twilio, Stripe, and other companies to deliver API products to market. Ones that want to solve problems that matter to business developers will craft APIs that address the desired outcomes of users. They seek to apply product ownership to their APIs, soliciting feedback from stakeholders and continually improving the API design and documentation to make it easier for API adoption.

Those that do not treat their APIs as products will produce APIs that are limited to serializing data over HTTP or providing system-to-system integration. While there is a time and place for this style of API, organizations won't experience the economy of scale that API products thinking offers: the ability to reuse them, increase velocity of delivery, and consistency of customer experience.

API business models

An effective API strategy must first define the objectives of the API program.

Common objectives include:

Accelerating

mobile strategy by making data and services more accessible.

Fostering

technical and business innovation by reducing technical barriers to the delivery of new solutions.

Adapting

to changing customer relationships that go beyond web and mobile to new devices and experiences (omnichannel).

Breaking down

silos to facilitate easier and more consistent data sharing among internal teams and systems.

Transforming

partner integrations by improving efficiency and freeing up resources.

Increasing

revenue directly or indirectly by reducing customer churn through deeper integration.

Here are three examples
of how businesses have used APIs to accomplish one or more of these goals:

FEDEX

The FedEx API increases partner efficiencies while simplifying their own logistics by building on those offered by FedEx. The initial release of their API offered organizations a great deal of insight into shipping logistics, which had been an opaque process just a few years ago. Now, every FedEx Office location is effectively "powered by an API" that allows anyone, anywhere to upload documents for printing at a nearby location.

BEST BUY

Best Buy has opened an API to adapt to changing customer relationships. They have massive stores with vast inventories and they suffer from the "showrooming" problem. Potential customers come in, browse the merchandise, and once they find the item, they use a site like Amazon to find a better price and place the order. Best Buy opted to open its API to developers, making huge amounts of data open to third parties. The company has been able to build a community that is willing, and even excited, to explore the data and ways that it can be used.

NETFLIX

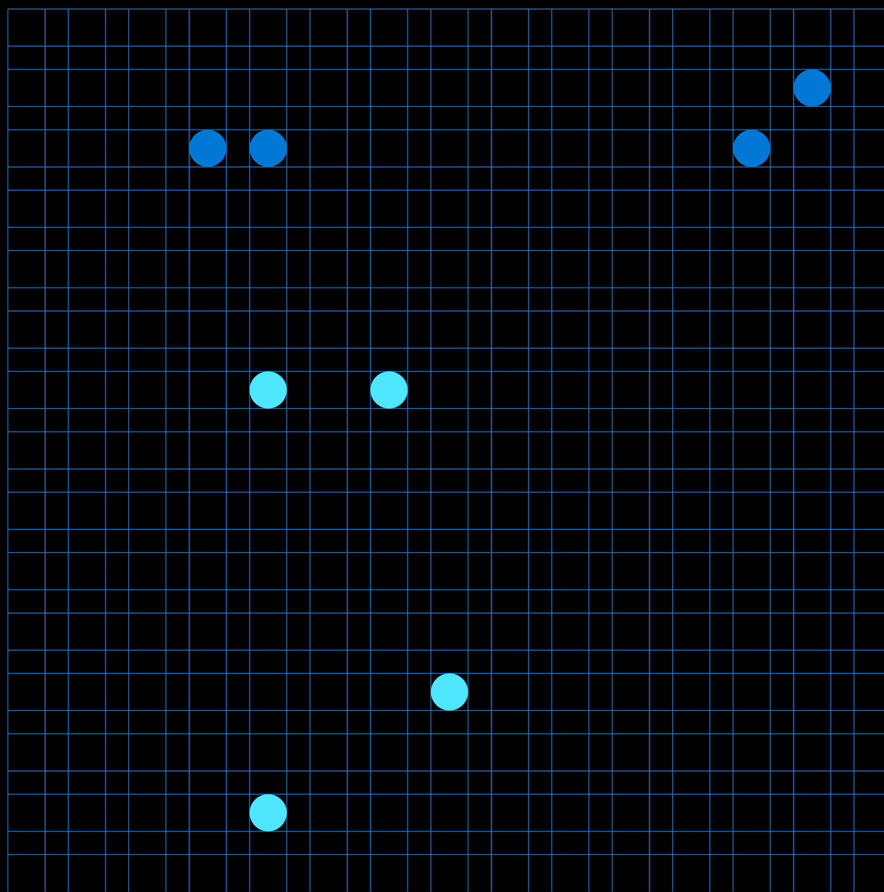
When Netflix originally started mailing DVDs in 1997, its competition was Blockbuster and the cable providers. Over the next few years, Netflix quietly consumed the DVD rental market while preparing to launch its streaming service in February 2007. Today, it is difficult to find a media device that doesn't have Netflix streaming support. This is in no small part the result of a strategic API initiative. Netflix ensures that its API is easily integrated onto any devices that connects to a screen. The next time you use Netflix, think about each interaction you have up to the point of streaming a show or movie. Each interaction is powered by an API capability offered by its device integration API.

APIs encourage reuse and increase velocity

APIs are the ultimate multiplier for organizations because they can lead to reuse and increased velocity by avoiding the need to rewrite the same thing over and over. What one team creates once can be used by many developers to produce solutions used by hundreds or even thousands of end users.

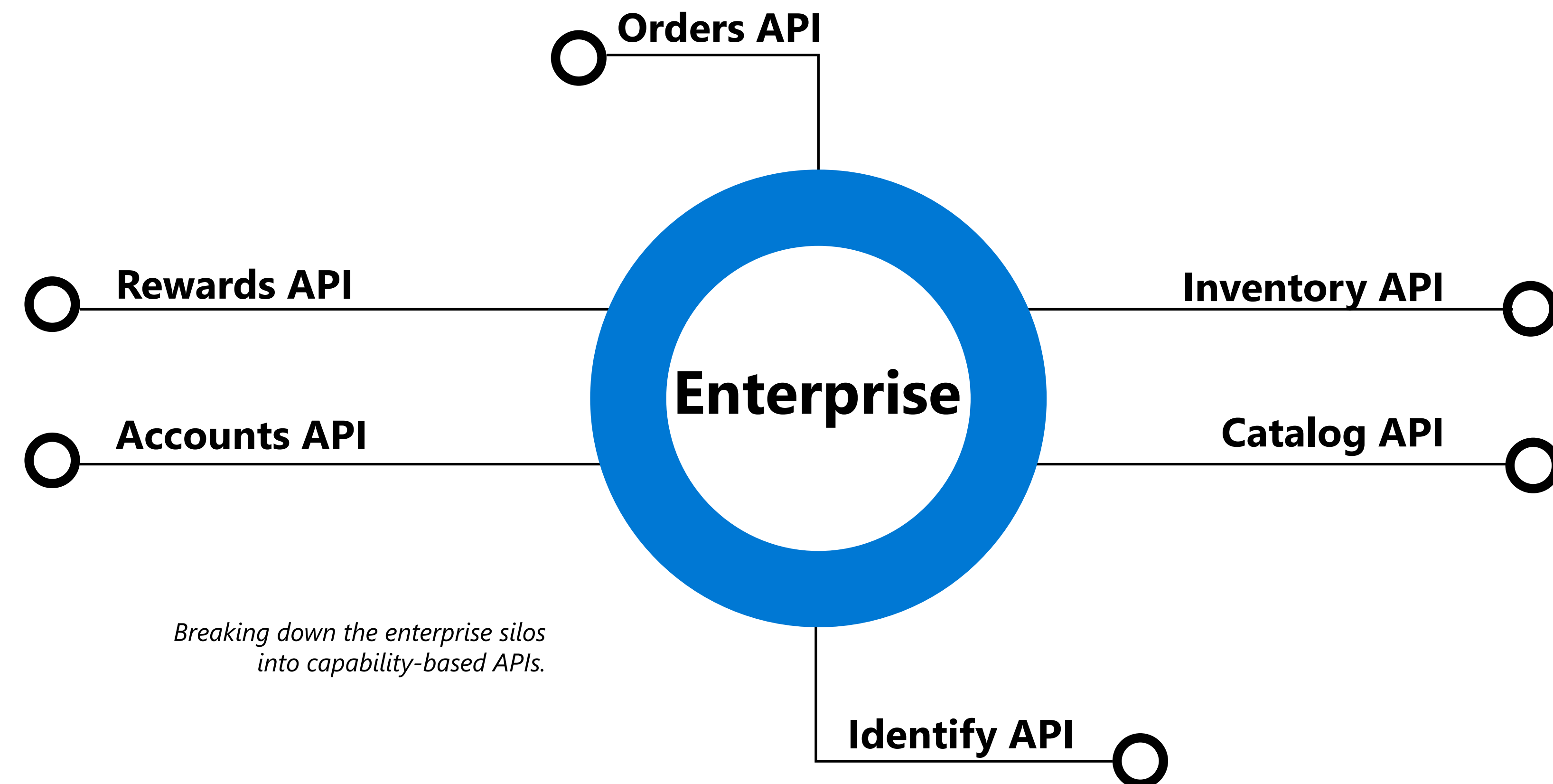
Reuse of APIs often starts with internal developers. Rather than sharing libraries in a single programming language, APIs enable code reuse across a variety of languages. Some organizations have even gone to the extreme of calculating the number of lines that didn't need to be written as a result of adopting an API.

While not all APIs produced by your organization will have hundreds of consumers, it only takes one team to re-implement a capability—but with slightly different behavior—to negatively impact your customers and reduce organizational velocity.



APIs break down silos

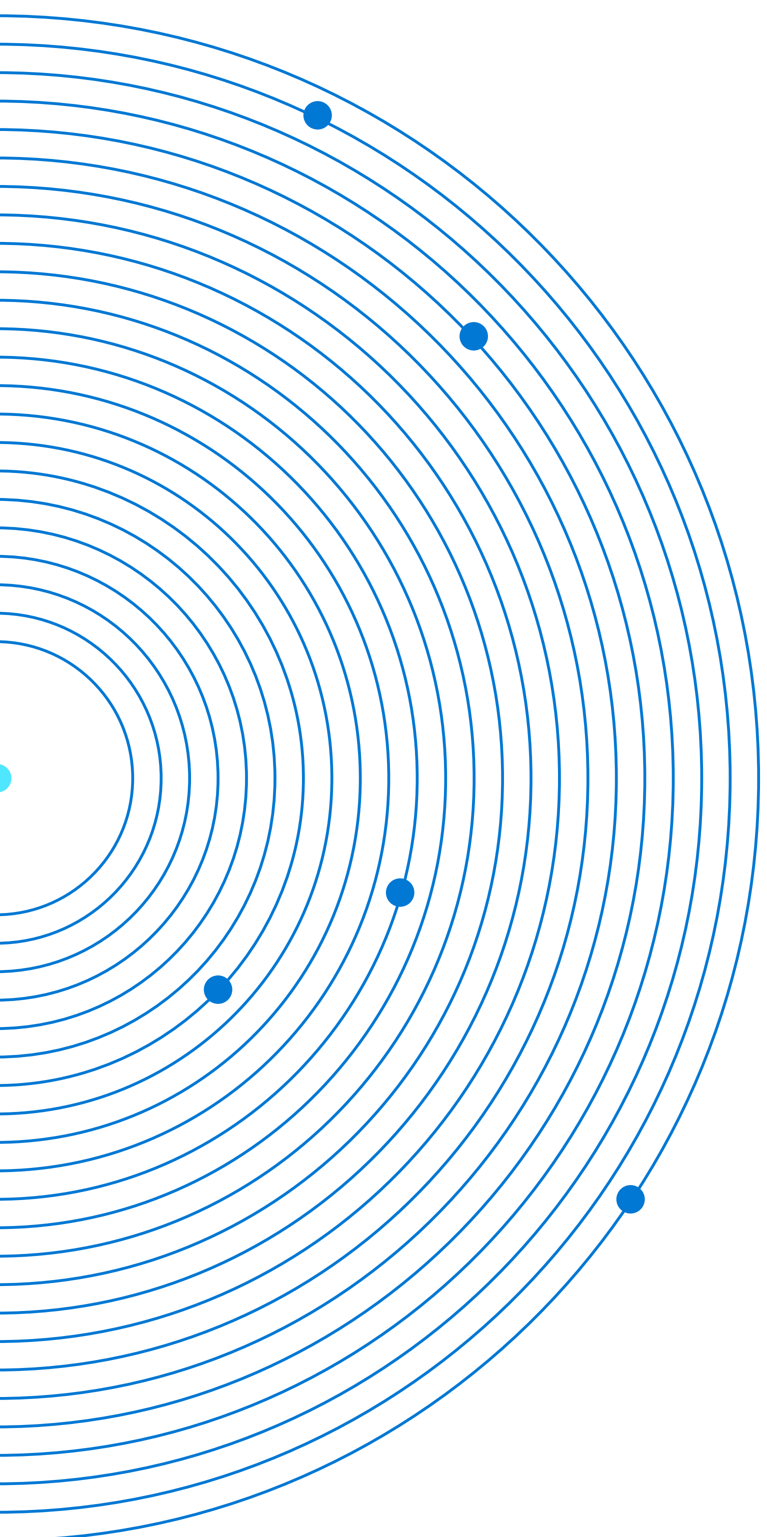
Organizations continue to struggle with siloed applications that contain both data and capabilities. With an API-centric organization, the API owns the data and capabilities instead of these siloed applications. The result is that businesses become more modular, able to unbundle and re-bundle their capabilities to innovate and engage customers in new and exciting ways.



Breaking down the enterprise silos into capability-based APIs.

Effective API programs help to break down these siloed systems by decomposing them into smaller areas of concern (sometimes referred to as "bounded contexts").

These areas offer a clear API that provides the capabilities and data for a specific business concern. Solutions are then built on these APIs to solve the needs of the internal business, partners, app developers, and third-party solution providers.



APIs create a consistent customer experience

You have probably encountered an inconsistent customer experience. Perhaps the mobile app lacks the same features as the website. Or perhaps the website doesn't work exactly the same as a recently updated mobile app. Whatever the cause, the customer experience is inconsistent and may result in customer churn.

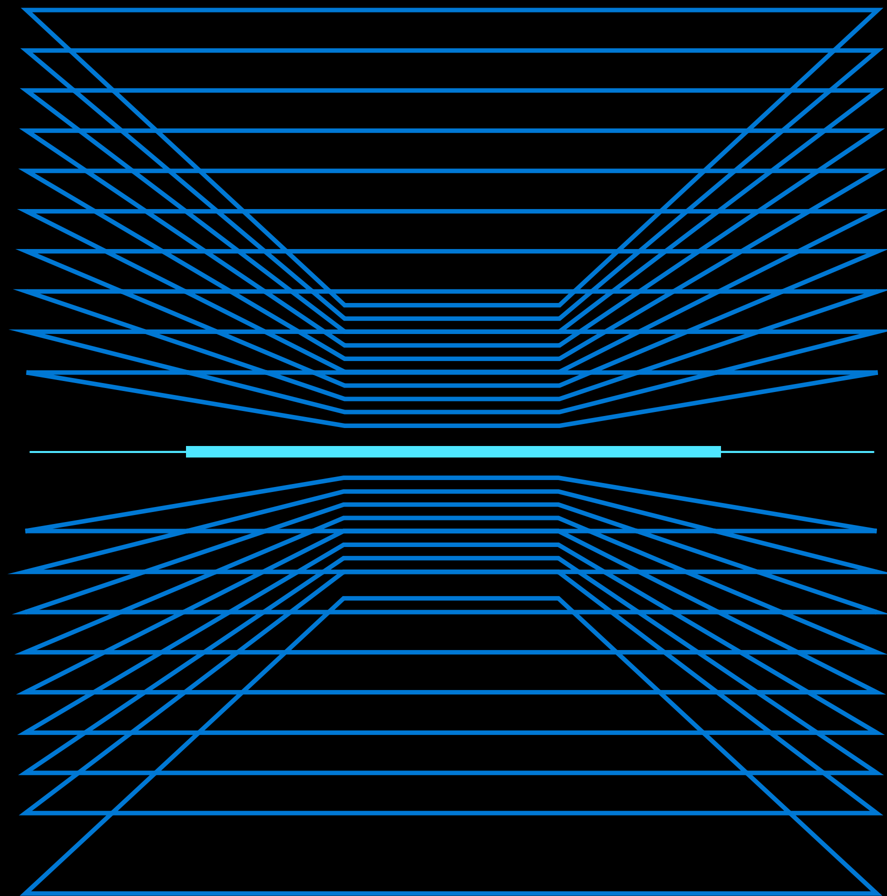
We need to remember that APIs exist to help complete a job-to-be-done for customers. Either the job requires the use of the API to start and finish the work, it requires the API for some portion of the work, or the job is to bridge systems and organizations through machine-to-machine communication. Therefore, APIs must be designed to deliver outcomes with the customer experience in mind. Remember, one poorly designed API may have a negative impact on many developers and even more customers.

APIs are the ultimate IT do-over

No matter your business model or size, APIs are becoming the ultimate do-over for the IT department. Organizations are able to design their APIs based on the business capabilities and jobs-to-be-done, and then adapt the implementation of these APIs to integrate the appropriate systems and necessary data sources. These APIs represent the future state of enterprise IT—even if the current state isn't quite there yet.

Through well-designed APIs that represent the desired future state, systems that were previously unable to be sunset are now hidden behind these APIs. Organizations taking an API-centric approach are more likely to migrate to newer systems and add new capabilities while sunsetting older systems that are currently relied on every day. Use your emerging API initiative to paint a picture of where the organization needs to go. Then map out the processes and effort required to get there.

CHAPTER THREE



What is an API platform?

Platforms create an ecosystem for orgs, customers, and partners. They connect everything that the company does, both internally and externally. Until recently, most platforms were built by vendors connecting a multisided marketplace such as supply chain management, customer relationship management (CRM), and enterprise resource planning (ERP). Now API platforms are emerging within organizations for the purposes of building internal, partner, and public solutions to meet the demands of the marketplace.

The four elements of an API platform

APIs provide interfaces to data and behavior to deliver digital capabilities, typically over HTTP. Your API portfolio likely spans multiple lines of business to address a combination of internal, customer, and partner needs.

SUCCESSFUL API PLATFORMS MOVE BEYOND SIMPLE APIs.

An effective API platform includes the following elements:

1.

A combination of private, partner, and public APIs.

2.

A mixture of API styles.

3.

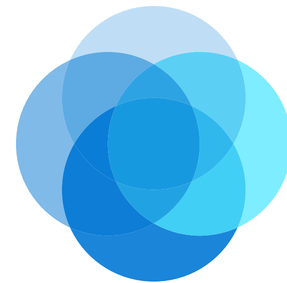
Event notification for reacting to change.

4.

Message streaming to act as a backbone for data.

Let's take a look at each of these elements and how they turn business and technical capabilities into a robust API platform.

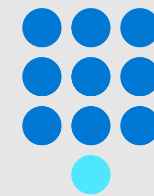
Platforms contain private, partner, and public APIs

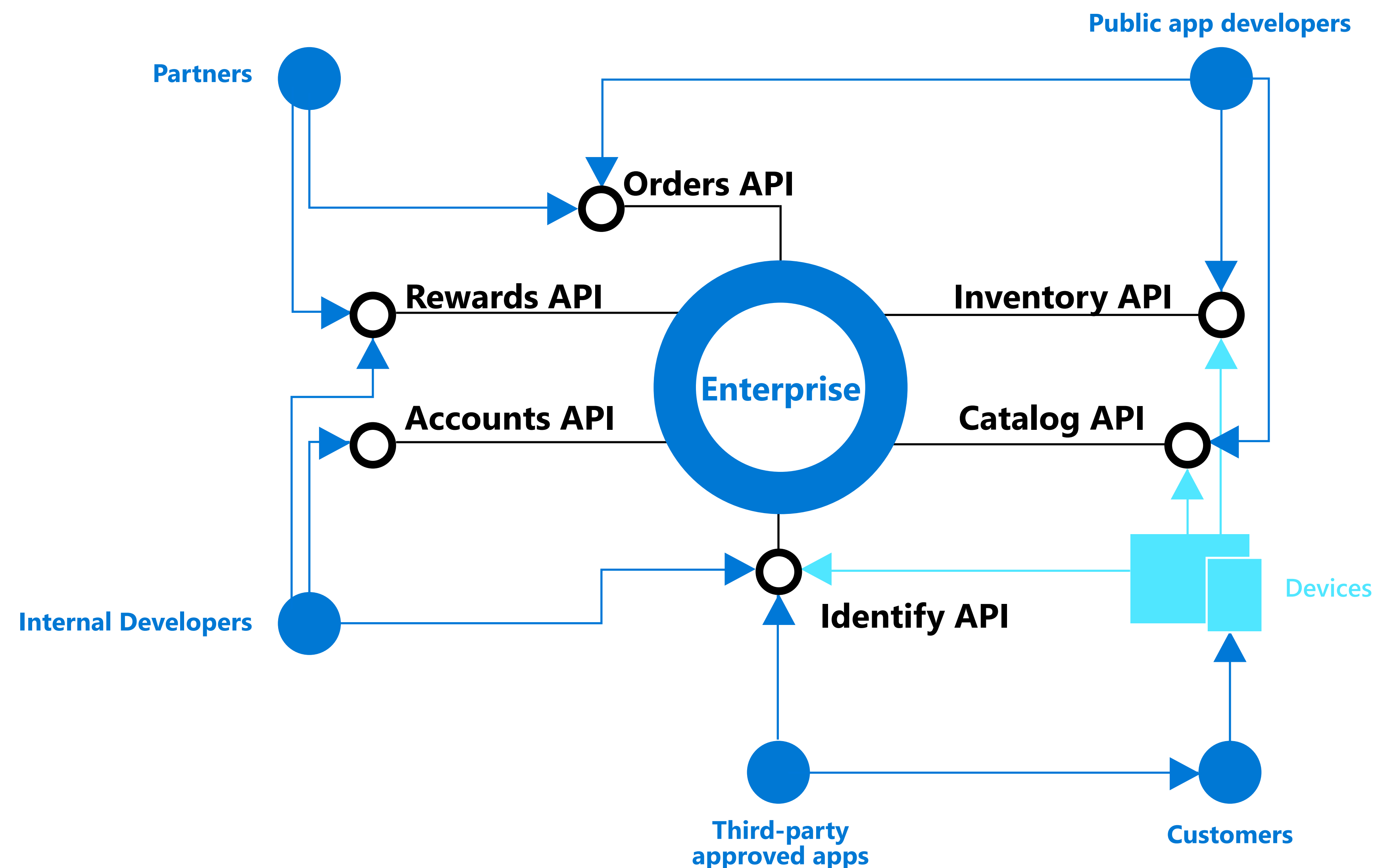


While public APIs from companies such as Twilio and Stripe often dominate headlines, many businesses restrict their public API exposure to a limited set of offerings. Companies such as Capital One offer a few productized APIs for developers to use. Their APIs are confined to a sandbox environment, with production access limited to those willing to engage in a more formal partnership. Others, such as Best Buy, make it easier to use their APIs.

Not all APIs should be externalized for public or partner consumption. Private APIs enable a business to deconstruct their operations into digitized capabilities through APIs built on internal systems. Some API platforms today are constructed purely from private APIs that are only accessible within the organization.

It is sometimes estimated that there are 10 times more private APIs than publicly available APIs. It is likely considerably more, since many organizations prefer to start with private APIs before opening them up to partners and public developers.





The enterprise API portfolio addresses the needs of the business, partners, customers, and public app developers.

While private APIs make up the majority of the API portfolio, organizations often require integration with their partners. What may have once been accomplished through batch files and SOAP-based integrations is now accomplished through web APIs. These partner-focused web APIs are a subset of the API portfolio, productized and supported for efficient partner onboarding and integration.

To date, most APIs have focused on integrating one or more systems. These APIs act as the glue that drives the day-to-day operations and support the workforce. However, APIs are more than an integration technology—they digitize capabilities that represent the business. These digital capabilities may be used internally, to drive partner integrations, and to power web and mobile applications.

Platforms require a mix of API styles

The three most common API styles today are REST, GraphQL, and gRPC. Each one has strengths and weaknesses, summarized here:

REST

stands for Representational State Transfer, an architectural style detailed by Dr. Roy Fielding in a 2000 dissertation that outlines the architectural decisions and constraints applied to the HTTP specification. While not a standard, it has been used to provide an architectural guidepost for designing web-based APIs.

Over time, REST has come to stand for some as JSON over HTTP, commonly through the create, read, update, and delete (CRUD) pattern. While this may be a common pattern, there are a variety of approaches available to building web-based APIs over HTTP. REST-based APIs are easily managed because each capability has its own URL that may be secured and monitored through API Management (APIM), as discussed later. Therefore, offering a REST-based API is a great starting point, allowing any application or automation script to interact with your API over HTTP.

GraphQL

originated at Facebook as a method of optimizing HTTP-based interactions between their mobile app and their graph-based back end. It has since become a common choice by front-end developers that desire a consistent way of expressing queries for the exact data desired. It is also used to aggregate multiple back-end REST APIs into a single request. GraphQL requests resemble SOAP-based remote procedure call (RPC) requests, with a single endpoint that accepts a specific request format. They are therefore more difficult to secure, monitor, and manage because they are managed as a single URL by APIM.

gRPC

started at Google as a high-performance way to send RPC requests between services. gRPC is built on the newer HTTP/2 specification, which supports performance improvements and multiplexing communications. The result is a protocol optimized for service-to-service communications within a microservice-based architecture.

Choosing the API styles that your enterprise uses helps to support the various use cases they solve:

SOLUTIONS	API STYLES AND PROTOCOLS	
Shop Web App	Shop REST API	Shop GraphQL API
Third-Party Marketplace API	Orders REST API	Inventory Mgmt gRPC API
Alexa Voice shopping skill	Inventory Mgmt gRPC API	Catalog Mgmt gRPC API

Organizations have multiple options for exposing their API-based capabilities. These choices allow API platforms to address the specific needs of developers by offering one or more API styles.

For most organizations, using REST as a foundational API style is wise because it has an approachable management, security, and tooling history. gRPC may be used to optimize internal service-to-service communication where appropriate. GraphQL is a good choice in addition to REST for reporting and front-end APIs that require optimizations for the customer experience.

Platform event notifications create extensibility

One of the most powerful elements of an API platform is event notification support. For every API capability that creates or modifies data, there is at least one event that another system or solution may be interested in receiving a notification about when it occurs.

Events allow solutions to be built on top of the platform without the platform knowing they exist. These emerging solutions are notified when state changes occur. It also removes the need for systems to constantly poll an API to see if any data has changed, using resources more efficiently.

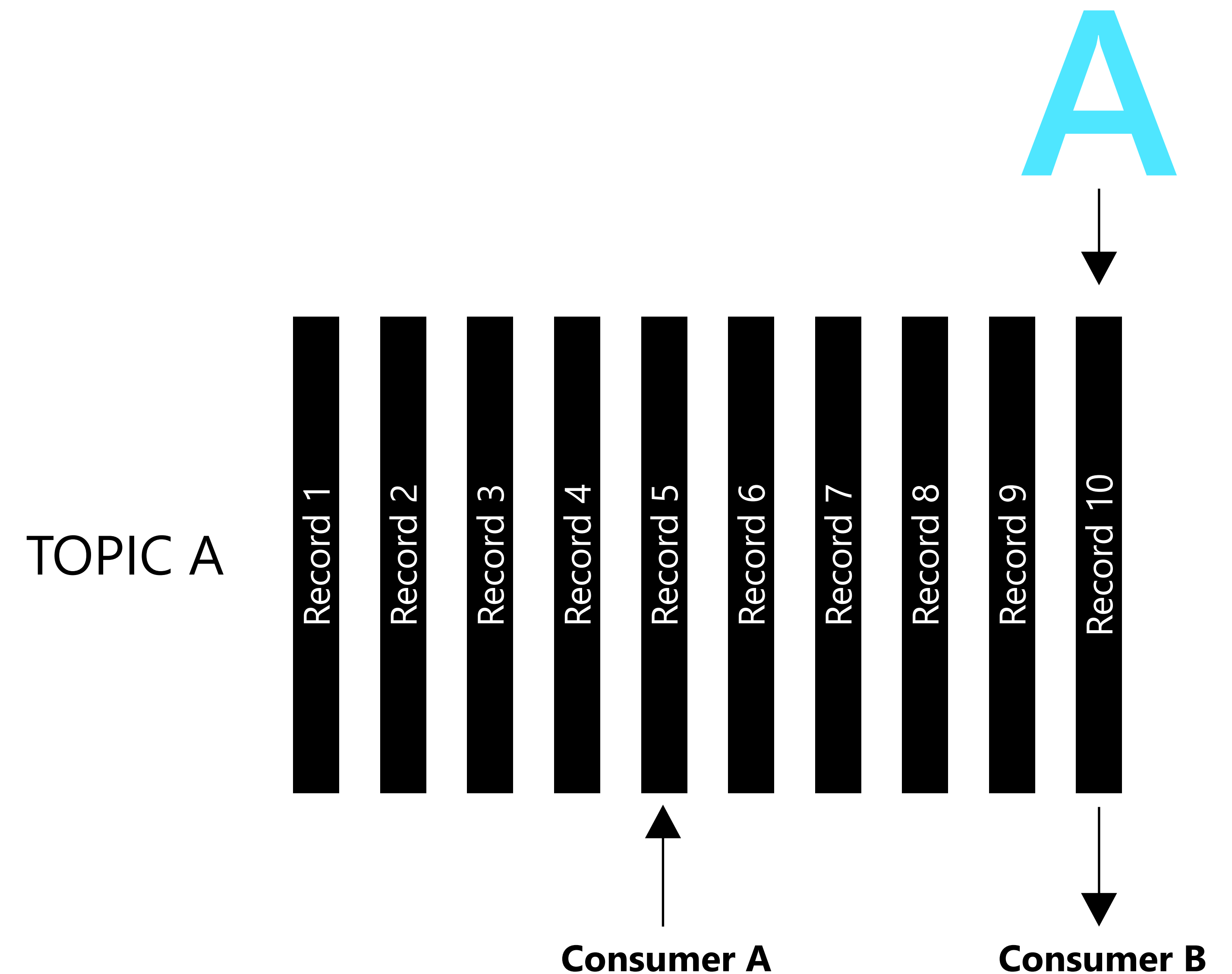
Those familiar with event-driven architectures (EDAs) know the power of events for integrating services and scaling systems. Exposing some internal events or emitting course-grained business events as part of an API platform reduces system-to-system coupling and creates opportunities for innovation in an agile way.

Platforms offer data streams

Data streaming provides continuous, ordered delivery of atomic messages that represent state change in data. Unlike web APIs that deliver data and behavior, data streaming focuses on raw data.

Traditional message brokers are the backbone of most enterprises, delivering transactional messages between systems. Data streams differ from traditional message brokers in that they de-emphasize transactions in favor of append-only, immutable streams of messages. This shift from message brokers supports high-velocity data and even allows for replaying message history. Consumers can revisit past messages in order of publication as new needs emerge or corrective action is required.

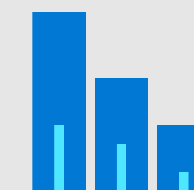
Message streams are a recent addition to the API platform. They are used to enforce data governance and data lineage requirements of regulated businesses. In addition, they are becoming the preferred way for data analysis and machine learning solutions that need to act on data as it becomes available, rather than the more traditional batch-based approach through technologies such as Hadoop.



Data streams allow consumers to process new and historical messages for analytics.

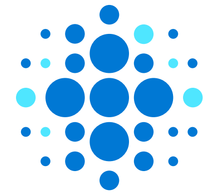
Solutions	Capabilities	Operations	Data mgmt.	Data processing
Shop Web App	Search products	Query available products	Available products DB	Top-Selling product analysis
	View product details	Query product catalog detail	Product catalog DB	
Third-Party Marketplace API	Place order	Place order command	Orders DB	
	View order satus	Order placed event	Browse history stream	Products recommendation engine
	Ship order	Query inventory levels	Recomm products DB	
Alexa Voice shopping skill	Update product desc	Query recom products	—	
	Update product inventory levels	Product ordered event		

Enterprise data management powers APIs through message and data streaming.



Being able to act on data in real time is a key focus of a maturing API platform. By being able to scale how you use your own data, you will be able to analyze and act in a more agile way.

Mesh app and service architecture



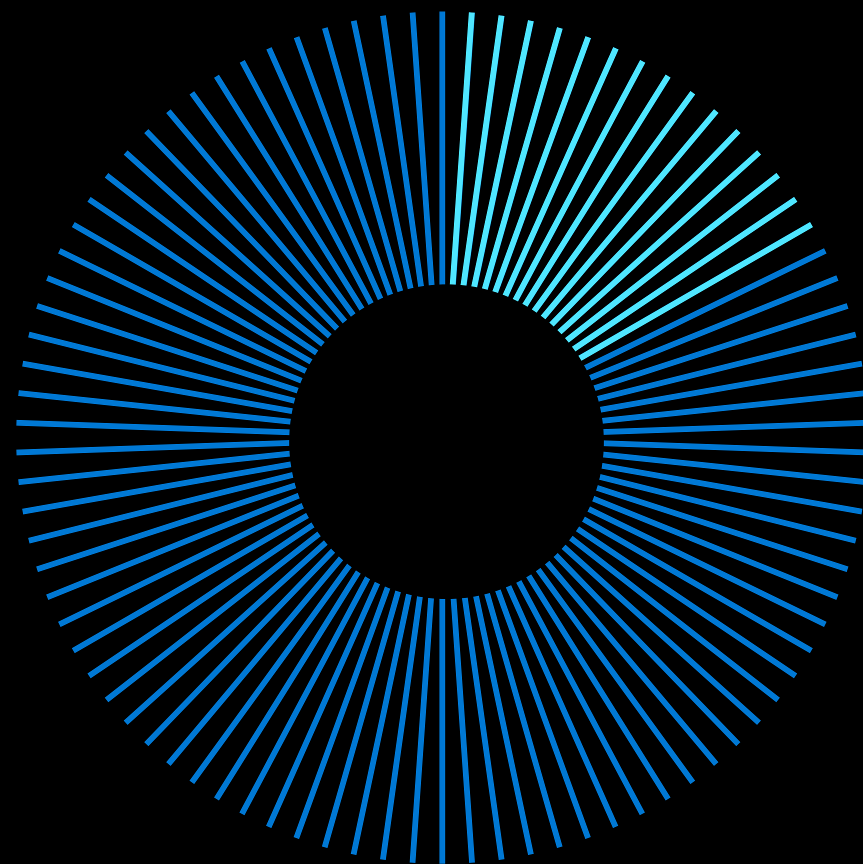
Mesh app and service architecture (MASA), a new architectural model introduced by Gartner, reflects the shift over recent years from a focus on applications to digitizing capabilities and constructing API platforms. MASA enables dynamic connections between services, people, and processes across channels and devices. It incorporates the API platform with external APIs into new solutions, allowing organizations to react to changes in the marketplace at a much faster pace than previously possible with siloed applications.

A common enabler of MASA is the integration platform as a service, or iPaaS. An iPaaS helps integrate APIs and events together into integration solutions that connect systems, resulting in new capabilities. These new capabilities may be deployed as services or new APIs that may be consumed by other applications.

With the emerging growth of MASA and iPaaS, there is now a possibility to enable business developers to construct applications with limited code requirements. This will allow software developers to focus on the more challenging tasks of building digital capabilities as APIs, while business users, comfortable with things such as scripting Excel using Visual Basic for Applications (VBA), are able to build complete applications without waiting for available resources from the IT department.

There is even an opportunity for marketing departments to build ephemeral applications that live for a limited period of time, such as for supporting a conference, and then are thrown out once they are no longer needed. Until now, this hasn’t been possible because the cost of development has been too high to build ephemeral applications. We are still in the early days, but this is becoming more of a possibility.

CHAPTER FOUR



The return of modular software

Over the last decade, we have seen new development frameworks emerge to make it easier than ever to build and deploy web-based applications. The downside of these frameworks is that they rarely encourage modular design. The result is quick development at the start of a new project, but over time complexity increases beyond the ability of the framework to maintain the efficiencies it first enabled.

At the same time, containerization tools such as Docker and orchestration layers such as Kubernetes have emerged. These tools encourage developers to easily break unwieldy applications into smaller components and deploy them in a predictable way.

The intersection of these circumstances has led to a renewed interest in modular software design. Unlike past modular efforts through component-based development, we are seeing these modules built as services and externalized on the network. This has resulted in the rise of microservices.

What is a microservice?

Like many terms in the software industry, “microservices” brings to mind a variety of things. Some think of microservices as small bits of code that offer a web-based API. Others think of microservices as a promise to speed up their struggling development teams. Both of these things are true, but there is more to microservices than many realize.

A microservice is a self-contained, independently deployable unit of code that focuses on doing one thing well. Microservices have no specific line count, contrary to what some may believe.

A microservice architecture exhibits the following traits:

Applies
bounded contexts to limit cognitive load.

Encourages
right-sizing through the continual splitting and recombination of services.

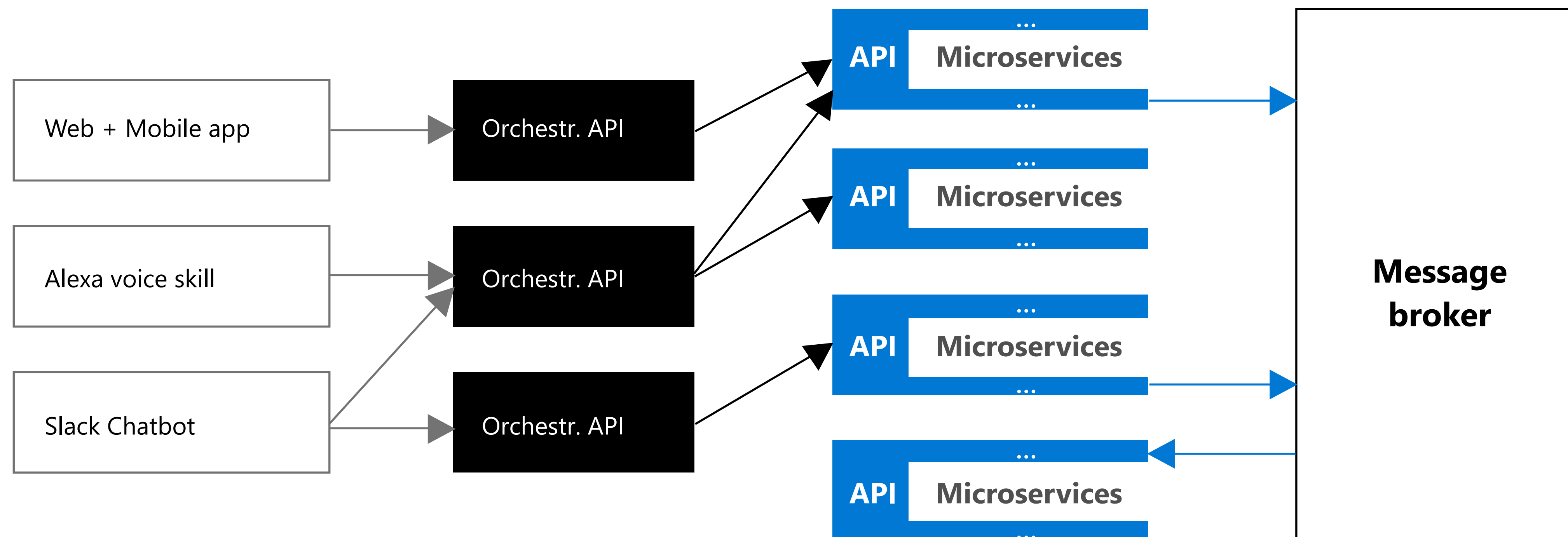
Deploys
independently via CI/CD automation.

Manages
its own data sources and does not share them with other services.

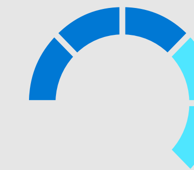
Enables
replaceability and experimentation by limiting the scope of each service.

A microservice architecture

is made up of a combination of API-based services and event- or message-driven services. A common practice is to include front-end orchestration APIs that make calls to one or more microservices. This enables organizations to offer stable contracts through the orchestration APIs, while replacing or splitting microservices behind the scenes and with no impact to API consumers.



A common microservice architecture using orchestration APIs that call one or more microservices behind the scenes.



An organization embarking on a microservice journey may not be able to realize the benefits immediately. Therefore, organizations moving in this direction should plan for the time required to mature their microservice approach as they prepare themselves for the journey ahead.

Benefits of a microservice architecture

All organizations want to build software faster than they do today. However, they must ensure their software works as expected, consistently and accurately. As a result, processes are installed to prevent moving fast and breaking things. These processes often include meetings that coordinate within and across teams. Every time there is a need to make a change to how an application or service works, coordination meetings are required. This results in a slower organization.

This tension between speed and safety is common. Organizations must find the balance between delivering software as fast as possible and the risk of introducing bugs that impact the safety and stability of customer and partner interactions.

A microservice architecture is primarily about reducing coordination costs of aligning efforts across multiple teams/developers working on a complex system. This is accomplished by limiting the scope of a single service while ensuring it is independently deployable. When an enhancement or fix is required, it is made within the microservice quickly and deployed with confidence. Data sources are assigned to a single service and cannot be shared across services, otherwise the coordination cost of the microservices is higher when changes are applied to a shared data store.

When microservices might be the wrong choice

Although microservices have been gaining in popularity, they may not be the right choice for all organizations. Microservices were originally intended to separate highly complex systems by distributing these complexities across the company, rather than having them concentrated in one area. They are best for larger organizations with a large number of developers. Smaller organizations, with systems of lower complexity, often find that the extra work of managing independent services outweighs the benefits of a microservice architecture.

Other considerations include:

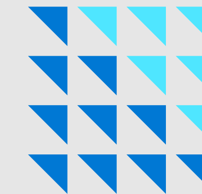
Microservices require a fully automated system. A developer should be able to design a microservice, register the service with an automated deployment system, and deploy code as needed without any manual processes or approvals. Organizations that have not fully automated their deployment pipeline and removed all or most of their manual processes will encounter considerable friction. Existing microservices will need to be used to deploy new capabilities, resulting in a few siloed services.

Microservices must be independently deployable. Some organizations opt to use their existing deployment processes, resulting in a coordinated deployment of all microservices at once. The outcome is a distributed monolith that prevents organizations from realizing the full potential of speed and safety.

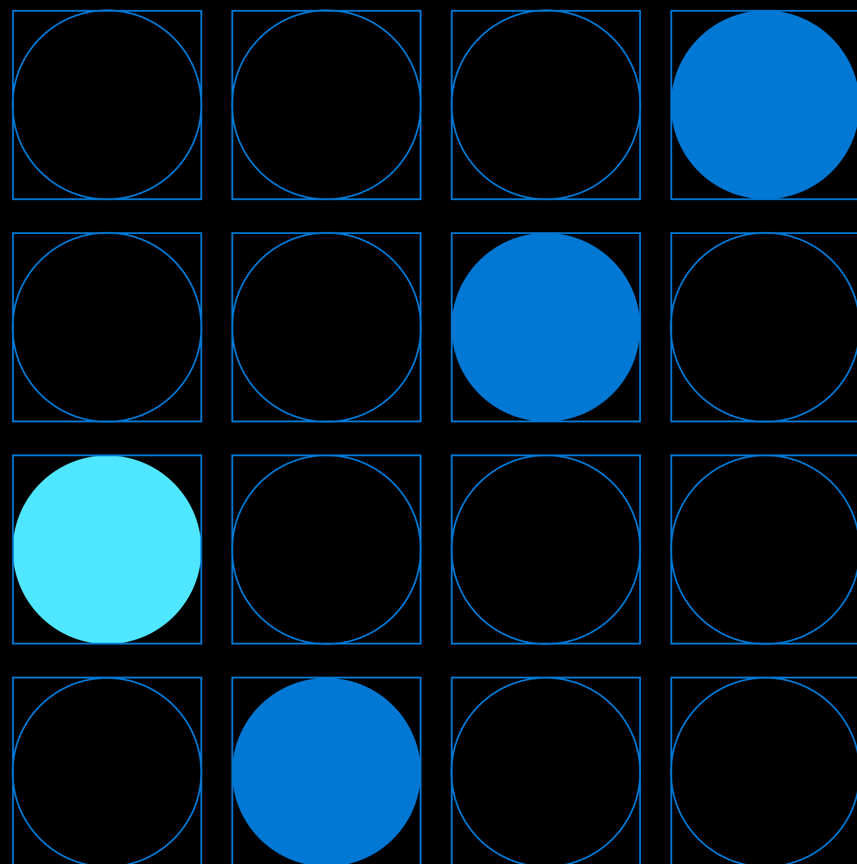
Microservices should be owned, monitored, and managed by a single team. A team may own a few microservices but should not be responsible for a large number of services. Sometimes, organizational structure and culture will be at odds with this style of service ownership, making it difficult to successfully deploy a complete microservice architecture that supports the desired speed and safety. Keep this in mind before shifting to microservices.

Data sources will be distributed, because each service must own its data. This requires heavy investment in distributed data management through data streaming or ETL processes to bring together data from multiple services for the purposes of reporting. Organizations with large databases must use caution when migrating to a microservice architecture.

The journey toward microservices requires a deep understanding of distributed systems. Those not as familiar with the concepts of distributed tracing, observability, eventual consistency, and distributed sagas will encounter a more difficult time with microservices.



In general, organizations should think smaller. They should seek to decompose solutions into smaller units of independently deployed code. However, a complete shift to a microservice architecture may not be the right answer. Instead, perhaps decomposing internal systems into modular monoliths with clearly bounded contexts that expose web APIs and business events may be sufficient for the needs of the organization.

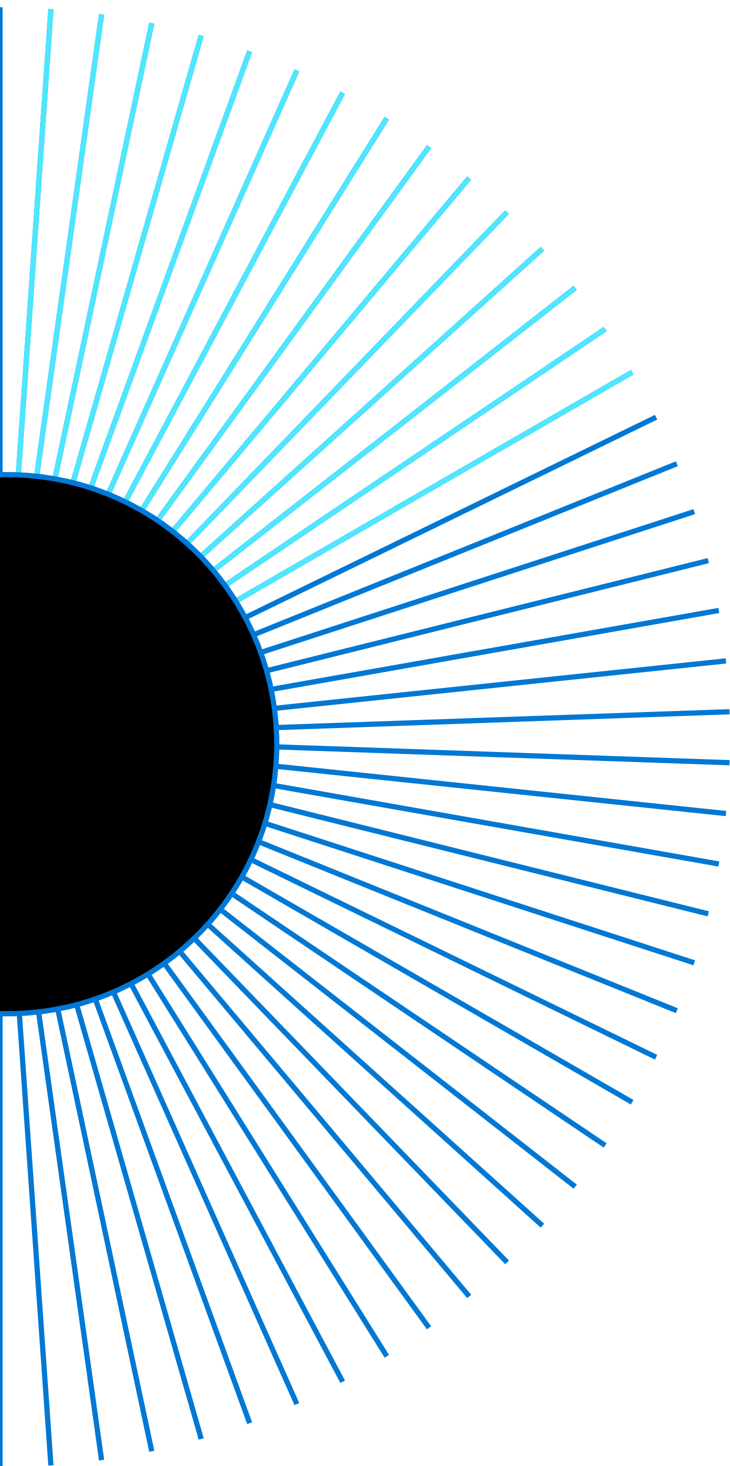


API management and security

API Management (APIM) layers accelerate the deployment, monitoring, security, versioning, and sharing of APIs. They are often deployed as a reverse proxy, intercepting all incoming API request traffic and applying policies to determine if requests are authorized to be routed to the API. The APIM layer may be fully hosted in the cloud—as with Azure API Management or the Tyk open source API gateway—or on-premises, or companies can use a hybrid approach.

Additionally, APIM enforces rate limits to prevent unlimited API usage, helps ensure security through API tokens and/or OAuth 2, and protects against a number of attack vectors. While some offer identity management as a built-in feature, most organizations will configure APIM to work with a third-party identity provider to offload user management and authentication.

Keep in mind that security is a process, not a product, and a continual one at that. Even with the ever-changing security landscape, we can still lay down an evolving set of best practices and test against them. Don't forget to develop healthy practices for monitoring your API surface area for malicious attacks.



Identity provider

An identity provider (abbreviated IdP) creates, maintains, and manages identity and access information for your API. They also provide authentication services to your APIM and support single sign-on (SSO) through technologies such as Security Assertion Markup Language (SAML). Some IdPs offer OpenID Connect (OIDC) support, which extends OAuth 2.0 with additional authorization support using web APIs and JSON identity tokens.

Content distribution network

A content distribution network (CDN) is a distributed cache at the edge of the internet that helps to reduce the load on origin servers. This helps cacheable API responses to be returned to web or mobile apps quicker because the content is often closer to the device than the origin servers. CDNs also offer additional protection from distributed denial-of-service (DDoS) attacks by monitoring layer 3 and layer 4 traffic, detecting problems, and preventing those problems from impacting customers, partners, and internal systems.

Deployment automation

Automating the build and deployment process is critical for a high-value API program. While most organizations have manual checks and verification steps as part of their release process, fewer manual steps means faster time-to-production. This is accomplished through a robust CI/CD pipeline.

Continuous integration (CI) requires developers to merge their changes back to the main branch as often as possible. Changes are validated by building a deployment package, and then running automated tests against the build. Continuous integration places heavy emphasis on automated tests to verify that the API is not broken on each new merge to the code trunk. Automated test strategies for APIs are discussed later.

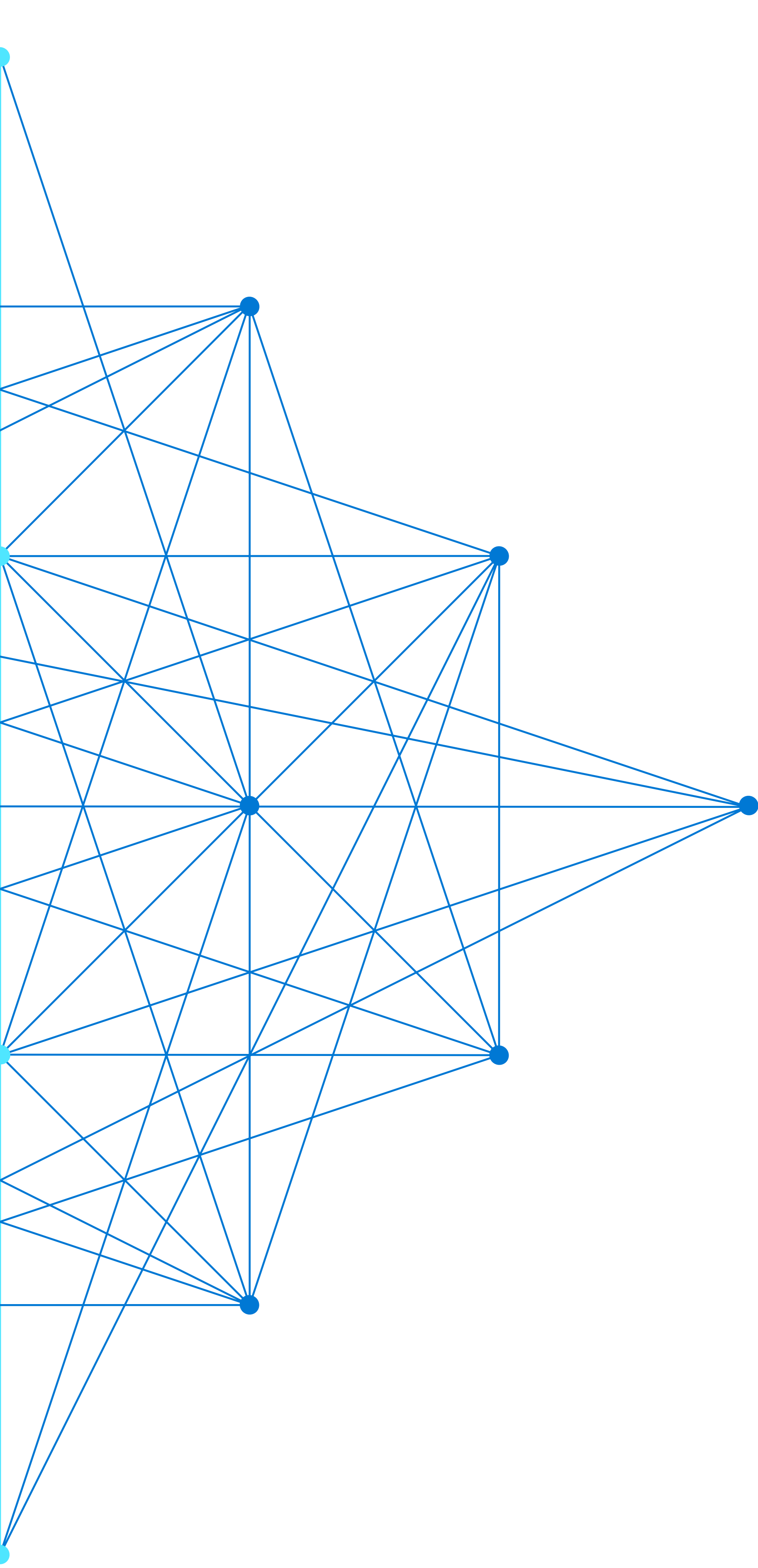
Continuous delivery (CD) extends continuous integration to include automation of your release processes through predictable, scripted processes. Many organizations still prefer a routine release cycle—for example, every two weeks—and therefore opt for this approach. Those that prefer full automation to production select a continuous deployment approach. Continuous deployment supports full release automation as soon as code is checked in, validated, and deployed.

Open source solutions, such as Ansible, Chef, and Puppet, along with managed solutions such as Azure DevOps services provide all of the necessary components to support your CI/CD pipeline needs, including agile tools, Git repositories, and configurable pipeline capabilities.

```
1  swagger: '2.0'
2  info:
3    title: Lab 1
4    version: '1.0'
5  host: example.com
6  paths:
7    /projects:
8      get:
9        summary: List Project
10       operationId: findProjects
11       responses:
12         '200':
13           description: ''
14           schema:
15             type: array
16             items:
17               $ref: '#/definitions/project-summary'
18     post:
19       summary: Create Project
20       operationId: createProject
21       parameters:
22         - in: body
23           name: body
24           schema:
25             $ref: '#/definitions/new-modify-project'
26       responses:
27         '201':
28           description: ''
29           schema:
30             $ref: '#/definitions/project-details'
31   '/projects/{projectId}':
32     parameters:
33       - name: projectId
34         in: path
35         type: string
36         required: true
37     get:
38       summary: Get Project
39       operationId: getProject
40       responses:
41         '200':
```

Example of an API definition using OpenAPI v2

Microsoft Azure



API definition and documentation

Effective communication is a critical factor for API adoption. In fact, your documentation is the primary method for communicating with developers on how to use the API. Your API documentation is your API's user interface.

API documentation was previously captured in a static document using HTML or PDF, but new options are now available. Tools such as Swagger, RAML, and Blueprint are just a few of the formats available to capture the definition of your API in a machine-readable format. These formats support the generation of reference documentation for your API.

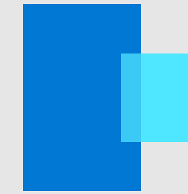
Many organizations are also benefiting from the integration of these definitions into their software development life cycle (SDLC), helping to drive automation and configuration of their APIM and other infrastructure.

The developer portal

Complete API documentation requires more than just your API reference documentation. A developer portal brings together the different styles of communication that you need to ensure that APIs can be discovered and the benefits of using your API are understood. It offers guidance to developers on how to get started integrating your API.

A robust developer portal will include the following sections:

1. **Overview** that describes the use cases that the API solves and basic connectivity details.
2. **Authentication and authorization** that covers how the API is secured, how to obtain an API access token, and details about token expiration.
3. **Example workflows**, commonly in HTTP request/response format, and code examples to help developers understand how to solve common problems with the API.
4. **Rate limiting and related SLA details** to understand how an application may be limited in its use of the API, including pricing (where applicable).
5. **API reference documentation**, typically generated from your API definitions, for developers who need to understand the details of each API endpoint.

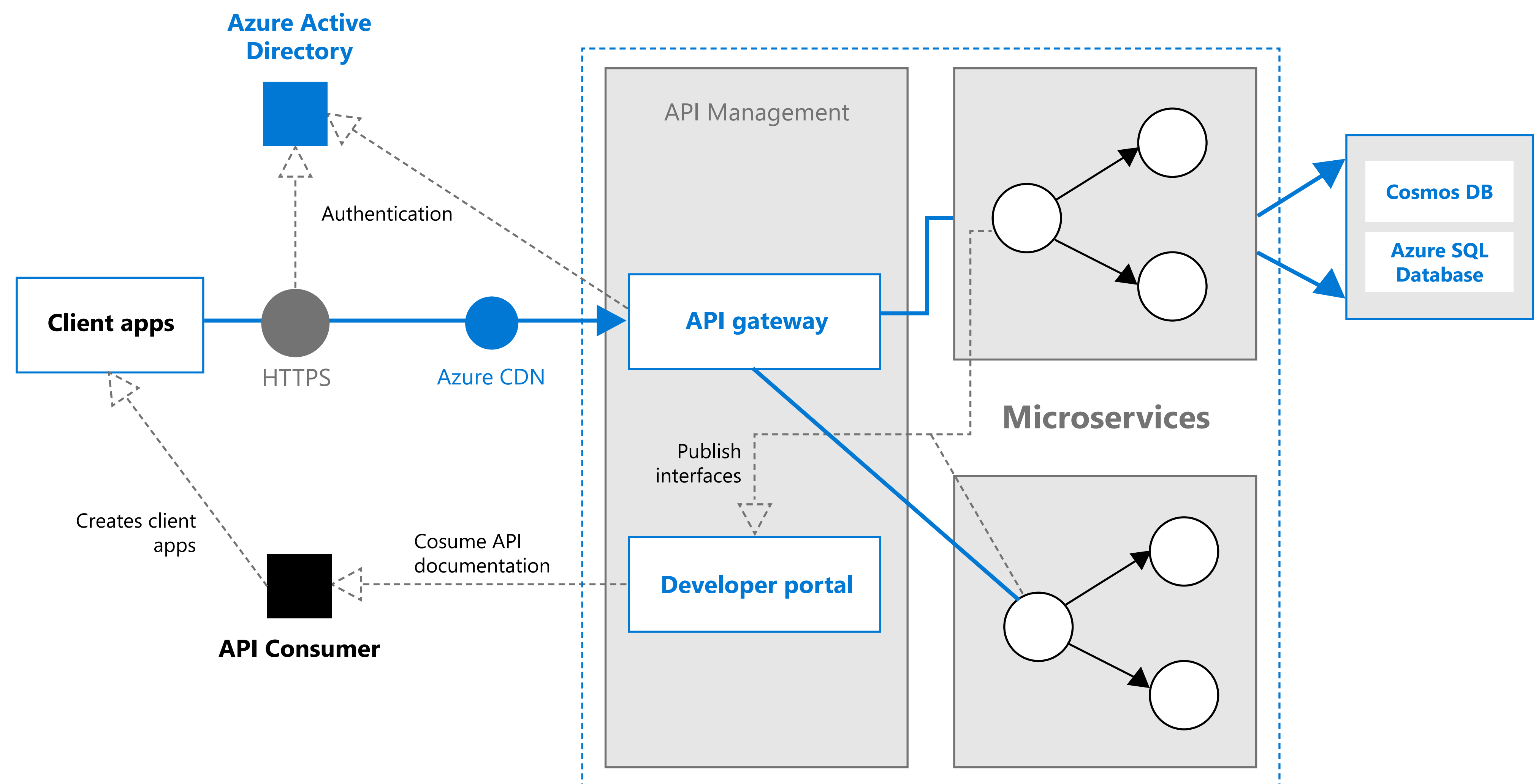


While this list may look daunting, it is recommended to add minimal content into each section to address the common needs of your target audience. You can then grow the content over time until the developer portal addresses many of the questions and common use cases associated with your API.

Some open source and commercial APIM vendors support publishing your documentation artifacts to a developer portal, making it easy for developers to have important documentation on hand.

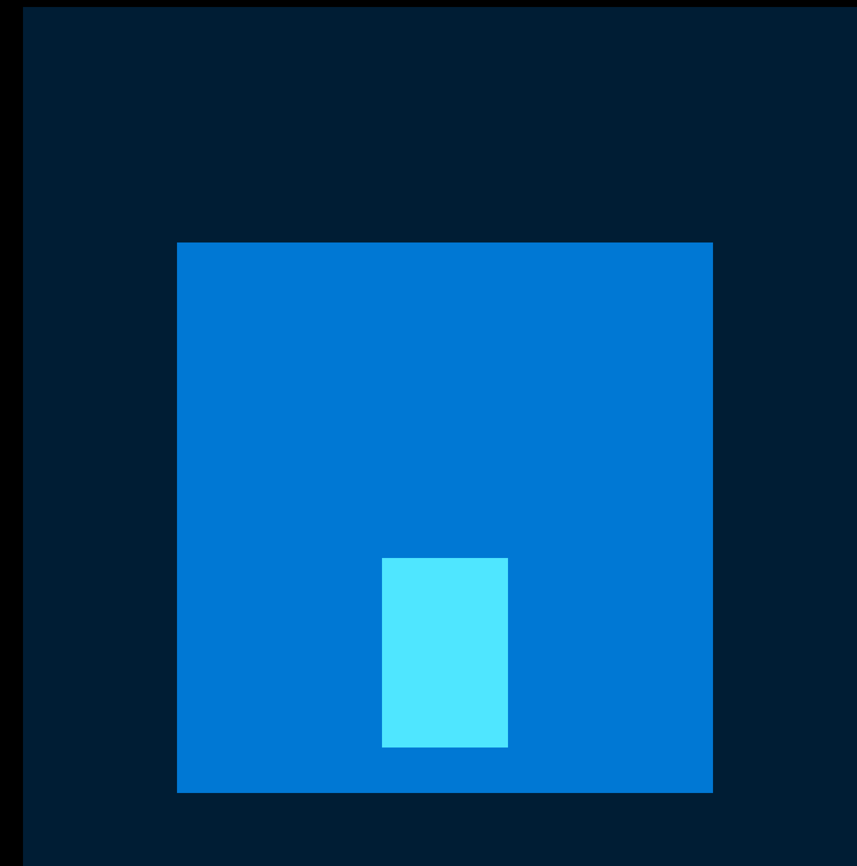
Putting it together:
A modern API architecture

Below is an architecture diagram that illustrates the previous components working together to support client applications using APIs and microservices:



An API and microservice architecture built on Azure Cloud services.

Microsoft Azure



Developing your API strategy

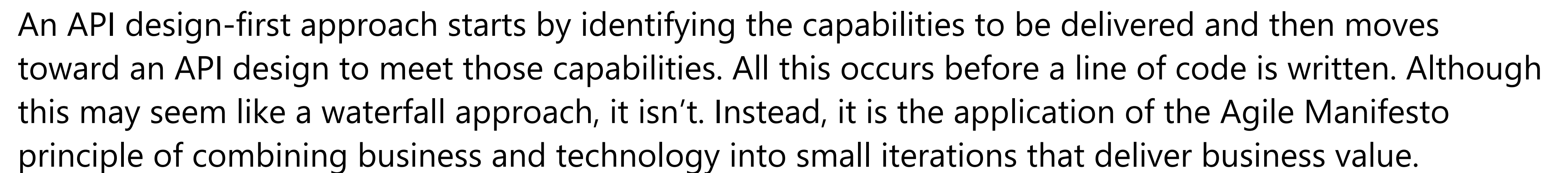
APIs extend beyond a set of technologies with developer interest. They power customer experience, business relationships, and internal innovation. A clear API strategy transforms your data and processes into digital capabilities with a programmable interface.

Unfortunately, some organizations realize too late that teams are already designing and deploying APIs. These APIs are built to solve a specific problem, are often designed in isolation, and lack the necessary consistency for reuse.

By clearly defining the objectives of your organization's API program first, you can keep teams focused on delivering business value that meet these goals. After establishing your API program objectives, clearly document them and share them throughout the organization. Use key performance indicators (KPIs) to measure the success of your program, track them over time, and adjust your execution accordingly to meet your objectives.

Designing an API design-first process

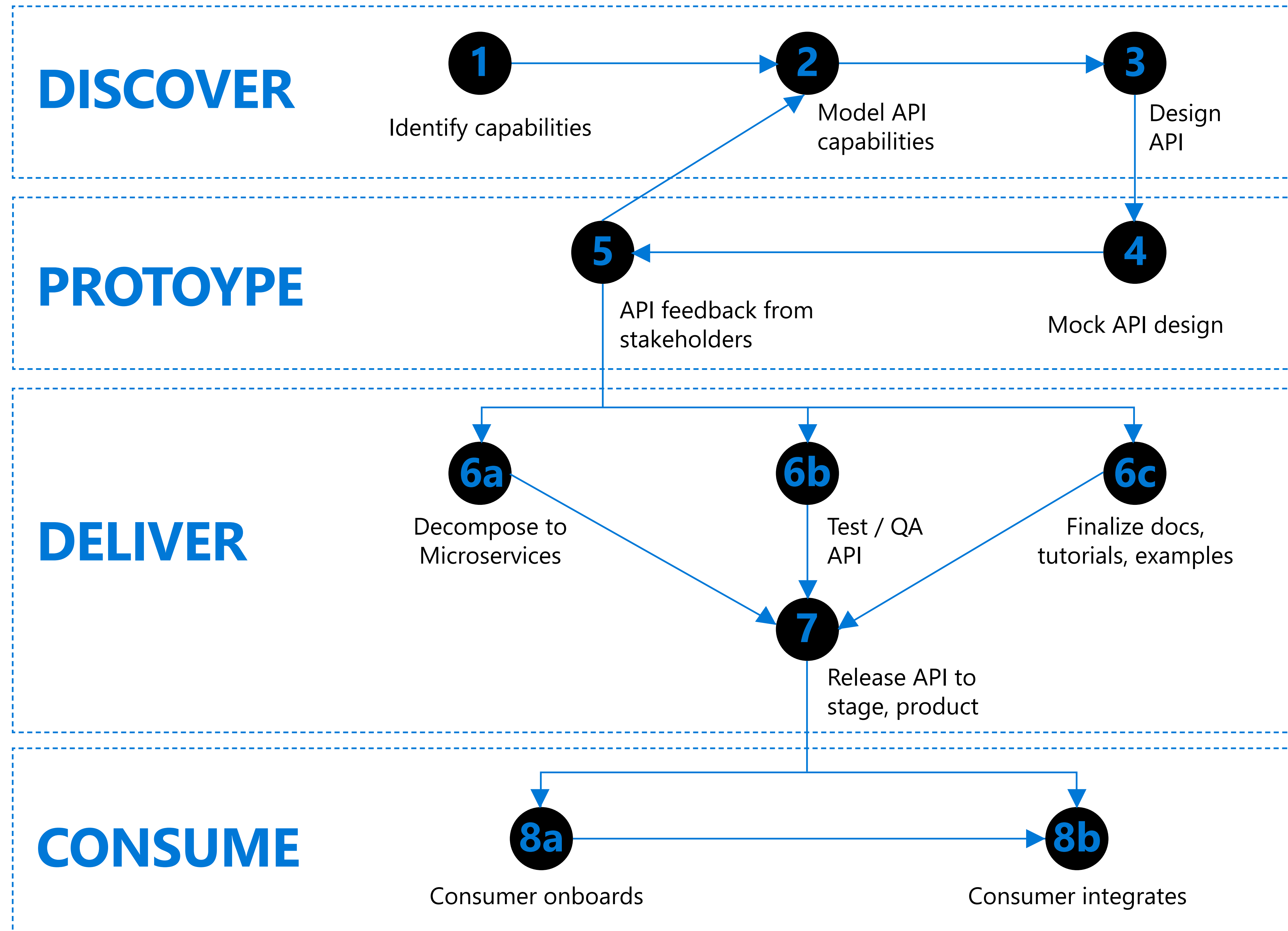
Too often, API design is either ignored or rushed and therefore fails to take advantage of the opportunity to learn and improve on an API design before release. Taking an API design-first approach allows for incorporating feedback before it is released, preventing the need for breaking changes immediately after release.



An API design-first approach starts by identifying the capabilities to be delivered and then moves toward an API design to meet those capabilities. All this occurs before a line of code is written. Although this may seem like a waterfall approach, it isn't. Instead, it is the application of the Agile Manifesto principle of combining business and technology into small iterations that deliver business value.

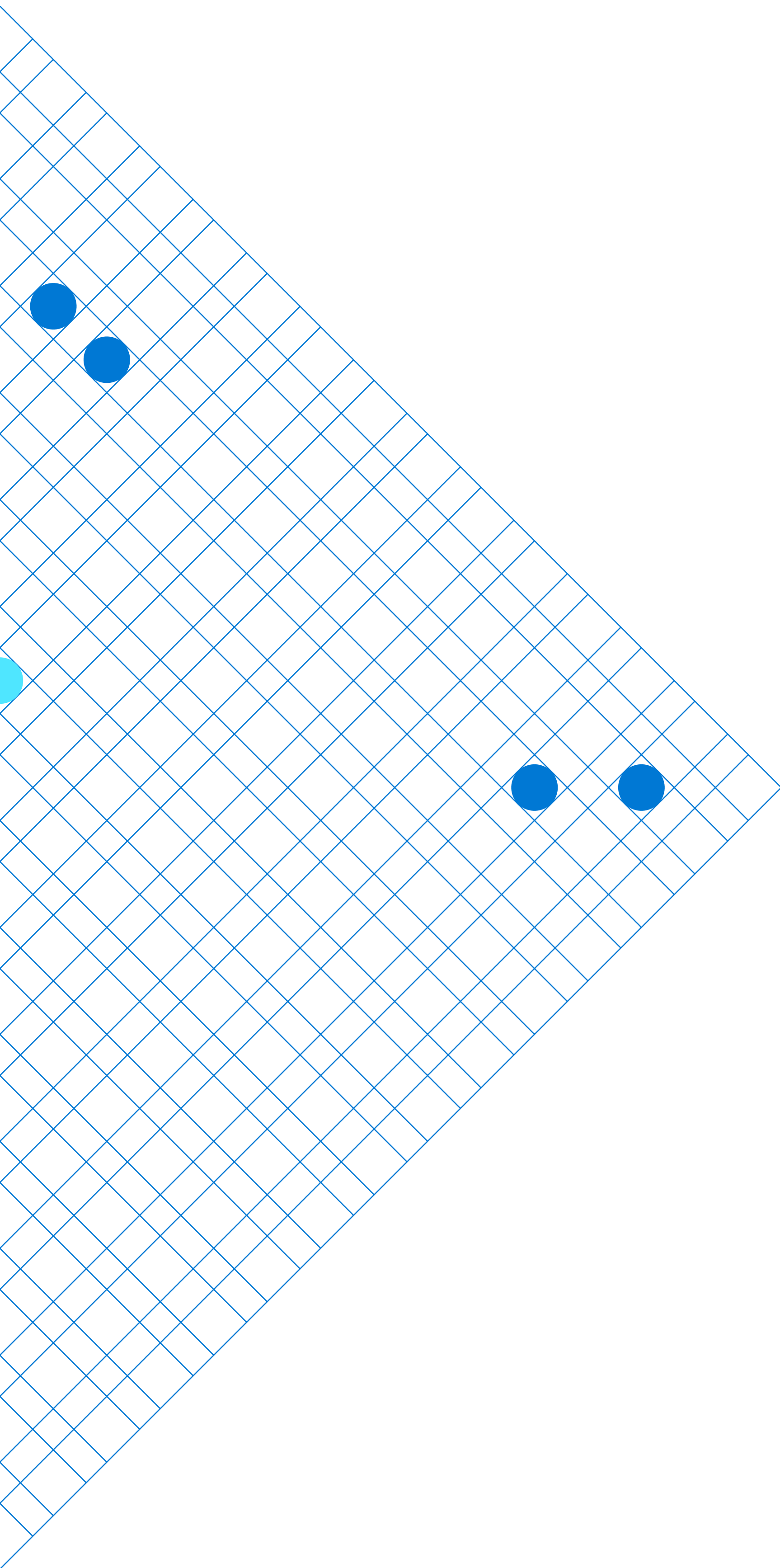
Microsoft Azure

The process consists of the following steps, executed in parallel when possible:



An API design-first process includes stakeholder feedback.

1. **Discover the capabilities that need to be delivered**. Focus on behavior and desired outcomes first, before focusing on the code.
2. **Model and design the API**. Transform the desired capabilities into a resource-based approach using the power of HTTP.
3. **Document the initial API design**. The documentation should provide enough understanding to demonstrate common use cases.
4. **Seek feedback from stakeholders**. Use feedback to validate that the API design meets stakeholder needs.
5. **Update the API design**. Incorporate feedback as appropriate into the design before code has been be updated. Repeat steps 1 through 4 as often as needed to refine the API design.
6. **Produce a mock version of the API**. As the design begins to take shape, use the captured design definition, typically using the OpenAPI format, for your stakeholders to review.
7. **Release the implemented API as a preview release**. During this preview release, additional insights and incorrect design assumptions will emerge. Make necessary changes before officially releasing the API into production.
8. **Promote the release to production**. Once the preview release meets the needs of stakeholders, release the API to production. At this point, breaking changes are not allowed.



Keep in mind that this process may be used incrementally to deliver read-only endpoints, followed by endpoints that manipulate data or support more of the workflow process. Take advantage of the preview release to work out trouble spots, improve documentation, and expand code examples that demonstrate API usage.

By focusing on constant communication and feedback from stakeholders, you avoid the need to redesign and reimplement the API under time constraints while continually delivering business value to stakeholders.

Managing your API portfolio

Some organizations have efficient processes that help them produce large quantities of APIs and microservices, yet they have no clear organization of their overall portfolio. An organization's lack of portfolio management results in a spiraling effect that can result in rebuilding capabilities that already exist.

API portfolio management includes the following steps:

PARTITION

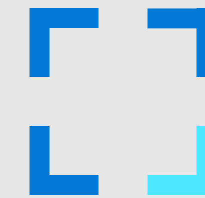
the portfolio into domain areas. Not all APIs are related (e.g., customer accounts, orders, and inventory for an e-commerce platform). Separate your APIs and microservices into domain areas and assign product ownership of each area.

DEFINE

a clear process to add new APIs. Setting up a clear process for adding APIs and microservices into the portfolio will prevent confusion about where new APIs belong and make it easier for API consumers to find the APIs they need.

MANAGE

URL paths like domain names. The URL paths under your API host name (e.g., api.mycompany.com are just like domain names—they are real estate for developers. Manage your URL paths through the use of well-known prefixes that support your domain-partitioning scheme. This will prevent scattered API endpoints, duplicate or confusing resource names, and conflicting paths.



Properly executed API portfolio management ensures that teams across the organization can contribute to the overall portfolio. It will also make it easier for consuming teams to look for an API to address the needs of their apps. Be sure to nominate a product manager or small team to own the API portfolio, perhaps as part of your API governance strategy.

Establishing an API life cycle

An API program should establish a clear and comprehensive API life cycle. The seven common stages for an API life cycle are:

1. **Discover**
   Identify new API capabilities as needs emerge from stakeholders.

2. **Design**
   Refine those needs with the stakeholders, identify capability requirements, and design the API accordingly.

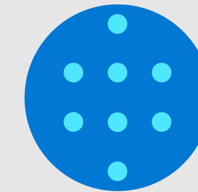
3. **Deliver**
   Implement, test, and verify that the new capabilities meet stakeholder expectations.

4. **Release**
   Deploy, monitor, and manage the API at runtime.

5. **Share**
   Inform your stakeholders and other developers of the new capabilities.

6. **Enhance**
   Incorporate consumer feedback and use proper versioning techniques to enhance the API without disrupting existing consumers.

7. **Retire**
   Inform consumers of any plans to stop supporting an existing API, migrate consumers to a new API (if available), and retire the API.



Integrate existing software processes throughout the API life cycle as necessary, but be prepared to adjust processes to prevent negative effects for existing API consumers.

Defining an API test strategy

When building an API, a proper test strategy ensures that your API both works correctly and meets the promises of its definition. **A complete API testing strategy covers the code, the API functionality, API contract testing, and acceptance testing.**

Unit testing

focuses on whether the code that powers your API is working properly. It helps to reproduce a discovered bug, verify the bug is fixed, and prevent regression of the bug as the code continues to change and evolve.

Contract testing

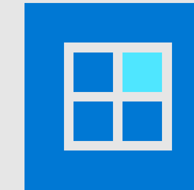
ensures that the implementation of the API matches the specification. Compare your runtime API request/response traffic against your API definitions (typically in OpenAPI, API Blueprint, or RAML format), reporting anything that deviates from the expected contract.

Functional testing

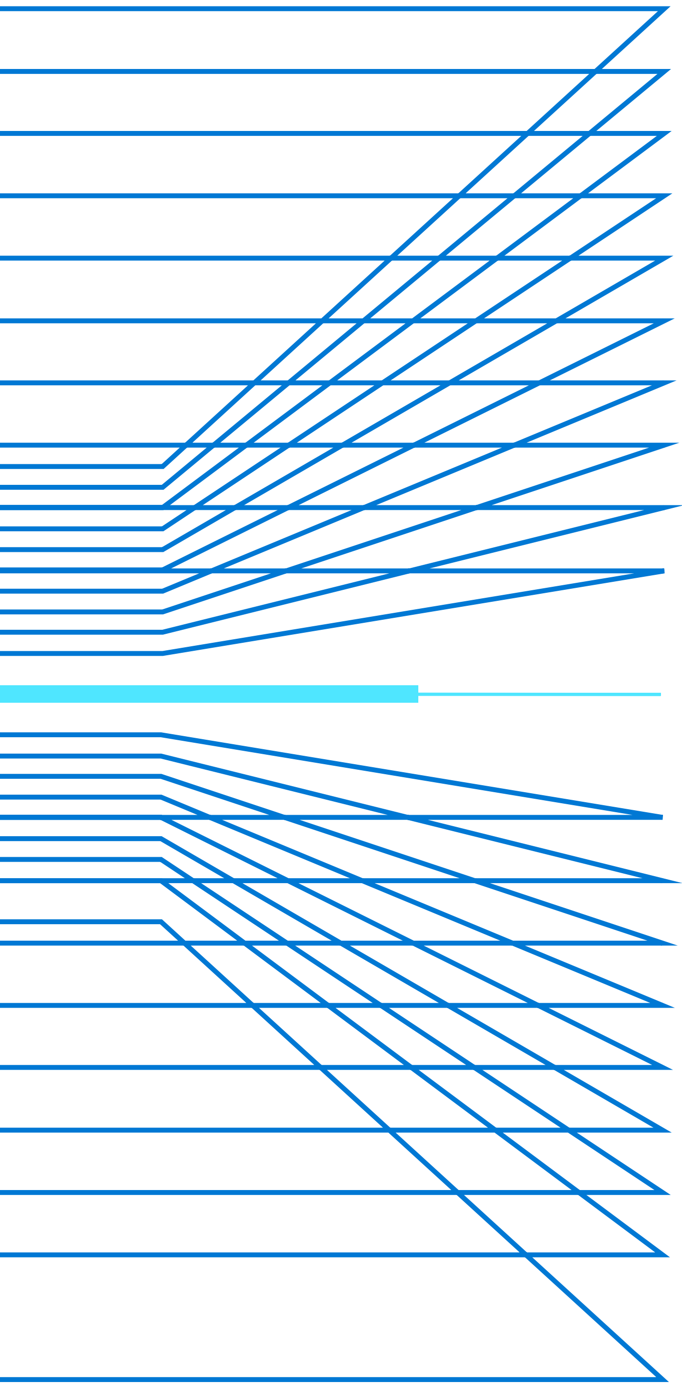
verifies that each API endpoint meets the expected behavior. Functional tests should exercise each endpoint for success and error cases to ensure that the API was coded defensively in the face of bad or malicious client code.

Acceptance testing

executes a series of predetermined paths that best represent the kinds of interactions your end users will have with your API, verifying that a specific end state is achieved.



For each of these aspects of your testing strategy, seek to find tools that help to support an API design-first approach. For organizations that prefer more code-driven testing, drive your tests from code generated by your API definitions and perform manual tests as appropriate. Tool-driven QA teams may wish to consider tools that can help to build test suites automatically from your API definitions, with additional QA-defined test cases as needed. When possible, avoid manual testing of your API beyond exploratory testing.



Monitoring your APIs

Monitoring today's robust applications requires more than just making sure your processes are running. It requires tracing an incoming web request across all the endpoints and services that support that request. Observability ensures that you can detect when recent deployments result in an increased number of errors—something the CI/CD pipeline may have missed.

Modern architectures often require the combination of distributed logging, distributed tracing, and API monitoring support (often offered as a feature within APIM) to properly troubleshoot and assess runtime errors. Some organizations are finding that the instrumentation available via a service mesh is a nice complement to these tools when adopting microservices.

Governing your API platform

Many historical SOA governance programs were internally focused to remove operational redundancies, resulting in slow processes that seemed to add little value to teams producing new services. Modern API programs have shifted this to an externally focused approach that strives to focus on delivering business value as a priority.

A healthy API governance initiative should encourage consistency across the organization, mixed with flexibility to support changing requirements. This is accomplished through the following governance actions:

Coach

teams on API modeling and design techniques, resulting in self-sufficient delivery teams.

Define

a clear API style guide with design patterns that support the organization's common use cases.

Produce

educational material, training, and other resources to communicate shared learning.

Create

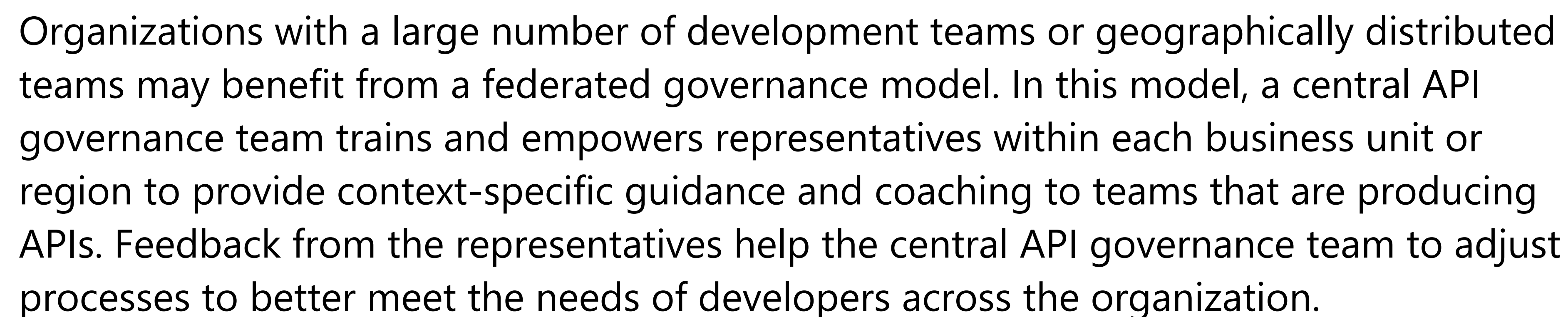
policies for onboarding, rate limiting, and access control.

Empower

solution teams to discover and consume existing APIs, increasing the likelihood of reusing an API rather than duplicating effort.

Craft

flexible processes and practices that encourage innovation.



Organizations with a large number of development teams or geographically distributed teams may benefit from a federated governance model. In this model, a central API governance team trains and empowers representatives within each business unit or region to provide context-specific guidance and coaching to teams that are producing APIs. Feedback from the representatives help the central API governance team to adjust processes to better meet the needs of developers across the organization.

API governance can be centralized and managed by a single team for the life of the API program. For larger organizations, governance may be centralized at the start but evolve to a more federated approach over time to scale the process.

Incorporating existing APIs into your platform

Not all your APIs will be designed under an established API program. Some may have been designed and deployed before the program is formed. Others may be designed in parallel to the initiative, before there are enough processes and people in place to assist them. Expect this to be the reality of your API program for a while.

One approach is to separate these APIs through specific prefixes or hostnames—for example, `GET /legacy/accounts/1234` or `https://legacy-api.mycompany.com/accounts/1234`. This provides a clear path for in-flight projects, while keeping these APIs isolated from your core API portfolio.

For APIs already deployed without APIM, start identifying the APIs that need to be cataloged and managed by reviewing request logs. Identify the highest-risk APIs, work with the team that built the APIs, and develop a plan to migrate them behind APIM as soon as possible.

If existing APIs do not have API definitions already created, there are two common options: use tools to generate your API definitions from code, or use traffic capture tools to generate HTTP Archive (HAR) files that are then parsed by tools capable of generating API definitions from runtime sources. Both options will help jump-start your documentation and governance efforts.

CONCLUSION

While it may be challenging at times to launch and sustain an API program, it is ultimately rewarding when you see the organizational shift from a set of ad hoc APIs to a healthy API program. By incorporating these practices into your API program, you will create a predictable API design and delivery process, alongside a complete API architecture that ensures your APIs are monitored, managed, and secure.

Back to top

Additional reading

"Cloud Design Patterns" – design patterns are useful for building reliable, scalable, secure applications in the cloud.

"The API gateway pattern versus the direct client-to-microservice communication" – optimizing client interaction with a microservice-based architecture.

"A Practical Approach to API Design"– a guide to understanding and designing REST-based APIs.

"Designing APIs for microservices" – an introduction to API design considerations for microservices.

© 2019 Microsoft Corporation. All rights reserved. This document is provided "as is." Information and views expressed in this document, including URL and other internet website references, may change without notice. You bear the risk of using it. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

Microsoft Azure