

Design and deliver valuable
Enterprise APIs

Enterprise API Management

Design and deliver valuable
Enterprise APIs

API management
with RESTful services

Luis Weir

Packt



EXPERT INSIGHT

Enterprise API Management

Design and deliver valuable business APIs



Foreword by
Zdenek "Z" Nemec

Luis Weir

Packt

Enterprise API Management

Design and deliver valuable business APIs

Luis Weir



BIRMINGHAM - MUMBAI

Enterprise API Management

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Dominic Shakeshaft

Acquisition Editor – Peer Reviews: Suresh Jain

Project Editor: Kishor Rit

Development Editor: Joanne Lovell

Technical Editor: Aniket Shetty

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Sandip Tadge

First published: July 2019

Production reference: 2250719

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-443-2

www.packtpub.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Foreword

Do you remember the days when businesses were considering whether they should invest in a website? Would building an online presence have any impact on their bottom line?

Getting on the web merely involved moving from the Yellow Pages and printed advertisements to a new channel. But this still left the potential of offering every business capability digitally untapped.

The web revolution was a kick-start for the next era for businesses – digital transformation. And APIs are the connective fabric of this much-debated digital transformation. Under API transformation, you deliver your organizational capabilities – your products – via APIs. Your communication, information exchange, innovation, and adaptation to ever-changing market conditions happens through well-laid-out APIs. The APIs become the backbone of your company. You can even leverage your partner sales channels with the right APIs!

API transformation does not start with building APIs, which were previously the technical creations of engineers. It begins with changing the mindset of the entire organization. It means embracing the culture of API first.

API first implies that everything produced and consumed in the organization is, and has, an API. Companies such as Adidas, DHL, or Volkswagen have already embarked on the API-first journey. Do you ever wonder whether there is an API for Adidas Stan Smith Sneakers? There is, and not just one! Adidas product, order, inventory, and other APIs are all about physical products. Every product has an API!

And we are still only scratching the surface. APIs no longer live isolated in silos; they are forming landscapes on every level: team, organization, domain, and even cross-domain. In the coming years, autonomous APIs will completely change how we discover, close, and implement deals. How do you find the best logistics service to ship a container from Wakkanai Port every month? What if you can perform this search and make the order in a fraction of a second? What if you can look for and close better deals a thousand times a day? Do you still think that, in 10 years' time, businesses will be asking whether they should invest in APIs?

Of course, embarking on an API journey requires preparation, learning new skills, and avoiding roadblocks. Many organizations naively start with the purchase of an expensive API solution in the hope of getting on the right track, only to later find themselves in a vendor lock-in trap, with a lack of product ownership and the infamous pitfall of "build it (an API), and they will come."

Executing a successful API transformation is a matter of building upon the three pillars: business, organization, and technology. Ignore one of them, and the project will fail. Each pillar has to understand the role and importance of APIs, rally under the API-first flag and carefully plan the API strategy. Only when all three elements are acting together can you hope for a prosperous API landscape.

This book will take you on an API journey that avoids common traps. It is the handbook for every API program owner, enterprise architect, and forward-thinking business person. Wherever you are, I am sure this book will prove to be an excellent companion.

The author (Luis) does an incredible job of explaining the business aspects of APIs in chapters one, three, and eight, while providing a great deal of technical background in chapter two, and then building on these foundations with architectural and technological concepts in the subsequent sections. The learning process then culminates in chapter seven, which presents the framework for the API life cycle process before closing with the grand finale on API products, business, and organizational impacts.

This book is the missing API manual for everybody interested in executing API transformation. It provides a holistic, concise look at the business, organization, and technical aspects of APIs like no other book before it.

API styles, tools, and vendors come and go. However, the concepts as presented in this book will help you to create a culture and values that last.

Good luck on your API journey!

Zdenek "Z" Nemec

Founder of Good API Consulting, and the author of API Blueprint and supermodel.io

Contributors

About the author

Luis Augusto Weir is a director of software development at Oracle and a former chief architect at Capgemini, Oracle Ace Director, and Oracle Groundbreaker Ambassador. An API management and microservices evangelist, Luis has over 17 years' experience in implementing complex distributed systems around the world.

Having always had a natural talent for software, computers, and engineering in general, Luis' career in software began from an early age. Even before starting university, Luis' entrepreneurial spirit led him to start several ventures, including one of the very first social media websites in his country of origin (Venezuela), as well as a small software development firm. Although none of these ventures turned into multi-million dollar corporations, the experience and knowledge gained during this period led him to develop a passion for distributed software computing that inevitably led to service-oriented architectures (SOA).

In recent years, Luis has been helping some of the largest Fortune 500 companies in industries such as retail, the supply chain, and agriculture to define and implement their API and microservice strategies, his experience of which served as a foundation for this book.

A co-author of three other books as well as numerous articles and white papers, Luis has been a frequent speaker at events such as CodeOne, Devoxx, Gartner AAD&I, Oracle OpenWorld, Java2Days, and many user groups and meetups.

Luis holds an MS in corporate networks and systems integration from the Universitat Politecnica de Valencia (UPV) and a BS in electronics engineering from the Universidad Nueva Esparta.

I want to dedicate this book to my beautiful family: my wife, Elena, and my three gorgeous daughters, Helena, Clara, and Alicia. Thank you once again for allowing dad to be stuck at a computer when I could have been spending time with you. I would also like to give special thanks to all the co-authors and editors of this book.

About the reviewers

Phil Wilkins has spent over 30 years in the software industry, acquiring a wealth of experience in different businesses and environments, from multinationals to software start-ups, and from customer organizations to specialist consulting. He started out as a developer on real-time, mission-critical solutions, and has worked his way up through technical and development leadership roles, primarily in Java-based environments.

He now works for Capgemini's multi-award-winning team specializing in cloud integration and API technologies and, more generally, with Oracle technologies.

Phil has contributed his knowledge and experience by providing input and support to the development of technical books (particularly with Packt Publishing), including co-authoring *Implementing Oracle Integration Cloud Service*, and *Implementing Oracle API Platform*, as well as online training on API best practices.

In addition to this, he has also had articles published in technical journals and is an active blogger. He has presented at a broad range of industry events, from large conferences around the world to user group and developer meetups. Phil's

expertise and contributions to the Oracle community have been acknowledged by Oracle by accrediting him as an Oracle Ace.

I would like to thank Luis Weir for the opportunity to contribute to this book, and for the time we spent working and presenting together at Capgemini. I would also like to take this opportunity to thank my wife, Catherine, and our two sons, Christopher and Aaron, for their understanding, given that many of the contributions I make to books and other activities mean spending extra hours in front of a computer.

Kshitij Mehrotra is an expert and thought leader in "digital transformation" with extensive experience in APIs, cloud applications, SOA, analytics, security, business activity monitoring (BAM), and business process management (BPM). He has helped several customers and employers to define and execute transformation and growth strategies by recommending the right architecture and validating strategic investment in a variety of technologies most relevant to customers' requirements.

Kshitij shares his experience with customers, helping them to shape digital initiatives and highlighting pitfalls that could affect implementation, as well as identifying the digital tools and technologies designed to ensure that the program is aligned with the businesses' strategy.

A blogger and speaker, he is Axway's chief architect for platforms and products. He has more than 19 years' experience of implementing solutions across the world and has successfully delivered large and complex digital and SOA solutions to Fortune 500 companies.

He has led middleware programs for renowned organizations, including Oracle Consulting, Wipro, and HCL Axon.

Thanks to everyone who inspired me to contribute to this book. And special thanks to my parents and family.

Rolando Carrasco is an Oracle Groundbreaker and Oracle ACE specializing in API management, service orientation, digital transformation, and microservices. He has over 19 years' experience and has worked for companies including HP and Oracle. Currently, he is the CTO of a Mexican consulting firm by the name of S&P Solutions, which has a very solid foothold in the Latin-American market.

He has been a constant Oracle Open World speaker and ongoing contributor within the community with blogs, videos, webinars, podcasts, presentations, and event coordination.

Rolando is a certified instructor for Arcitura, in particular providing content around service orientation and microservices. He is both a certified SOA architect and a microservices architect.

Rolando specializes in modern architecture, as well as high-demand and mission-critical applications.

He is a co-author of the book *Oracle API Management 12c Implementation*, published by Packt in 2015, and has contributed as a technical reviewer for at least three books during the last three years.

I thank God for giving me the direction, time, and knowledge to deliver my work. I also wish to thank my wife, Cristina, and my daughter, Constanza, as well as my mom, dad, and my brother, Manuel. These are the most important people in my life.

Preface

Application Programming Interfaces (APIs) can be compared to doors: their main purpose is to provide access to something. Doors come in different shapes, sizes, colors, and materials, and offer different levels of security to protect whatever is behind them.



Figure 1: Different types of door

In the case of APIs, however, that something is **digital assets** such as raw and cleansed data, images, videos, documents, and even functionality that performs complex calculations or data processing based on inputs.

Sometimes, doors are wrongly designed or built:



Figure 2: Real-life door design errors

Source: <http://www.constructionhunter.com.au/blog/industry-news/20-photos-that-will-make-you-question-your-faith-in-humanity>

The same is also true with APIs. **API management** is a discipline that has evolved to deliver the processes and tools required to discover, design, implement, use, or operate **enterprise-grade APIs**. Most importantly, the discipline is responsible for managing the communities around APIs. Such communities may consist of developers building and/or using APIs in their apps, but there are also communities of business and IT executives looking to speed up innovation at a lower cost.

We can conclude that API management's true objective is to **deliver value**. This could be valuable to the business in the form of reducing development effort by using existing APIs (internally developed ones or external ones developed by third parties). Value could also come from monetizing APIs that offer

intangible products (digital assets) that developers and/or executives alike would be willing to pay for.

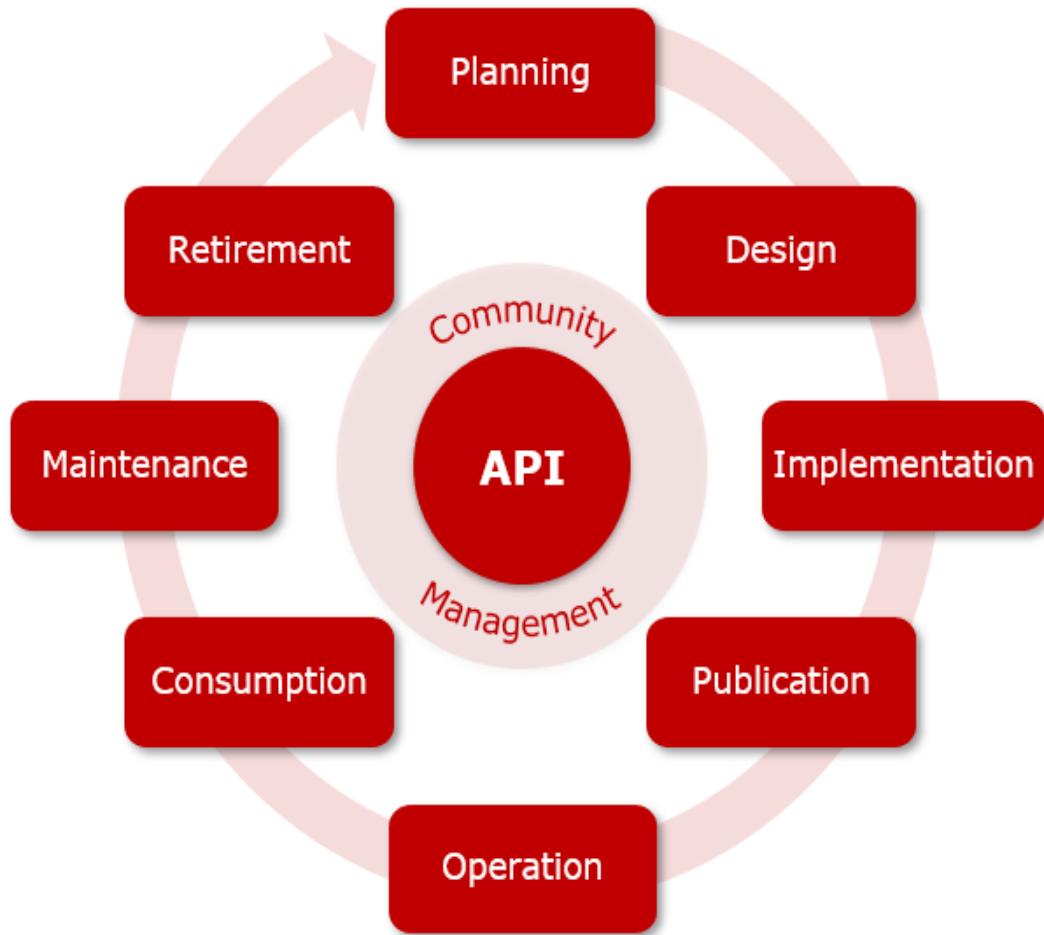


Figure 3: The API management cycle

Value can only be truly delivered when the full cycle of delivering something, in this case APIs, is fully understood, optimized, and overseen. The creation of an API strategy with a clear purpose and objectives is followed by the inception of an API through innovation workshops. Next is planning, design,

implementation, deployment, operations, and monitoring, until the eventual retirement of the API.

API management is no longer just about implementing APIs. Thousands of public APIs (with more being added by the day) are listed in **API marketplaces**, such as programmableweb.com, and RapidAPI.com, each representing a digital door to an organization's digital product offerings. Thinking that all APIs need to be internally developed is a huge fallacy.

To summarize, API management must be as much about providing the means to discover and use public APIs as it is about implementing new ones. At the epicenter of any API management initiative must be the creation of value for the business but also for the users of an API.

APIs at the center of digital ecosystems

As organizations embrace the adoption of public APIs and/or create new API products, there is an interesting effect. The creation of new ecosystems, all enabled through APIs, starts to happen as value comes from adopting and combining someone else's digital assets in the creation of new products.

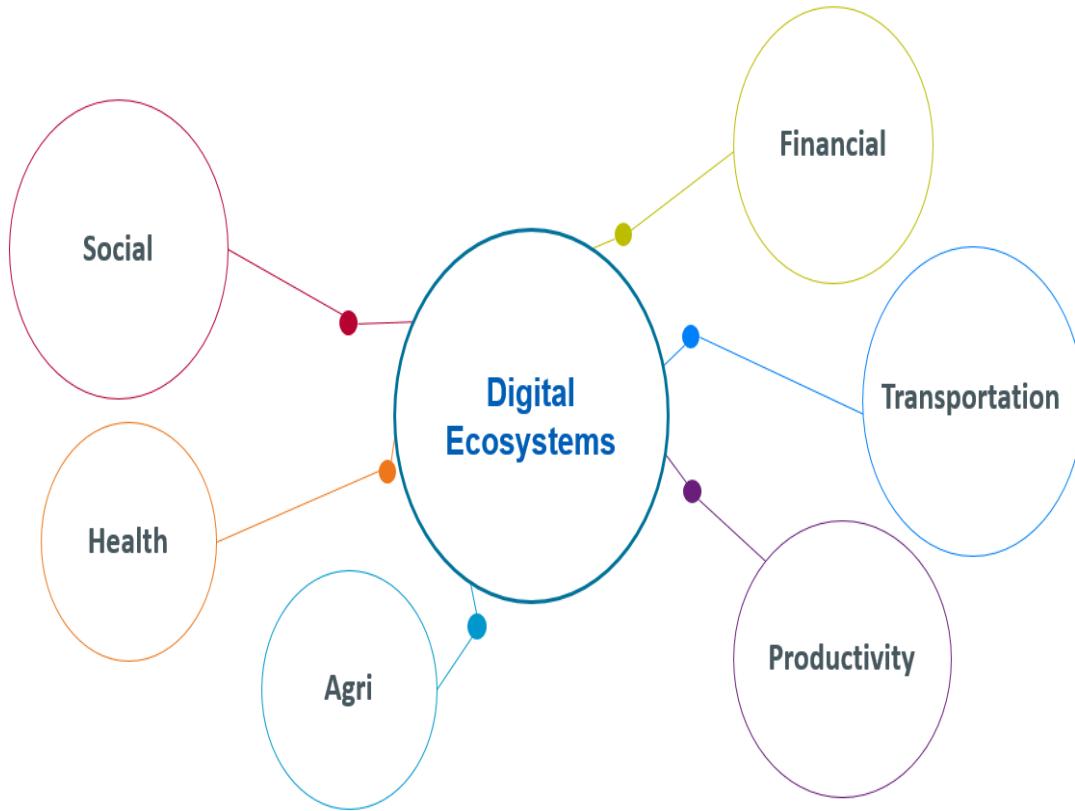


Figure 4: New ecosystems being created

In fact, a study conducted by McKinsey predicts that by 2025, digital ecosystems will account for 30% of global revenues, which according to the firm is about 60 trillion US dollars.

The study is referenced in the following link:

<https://www.mckinsey.com/industries/financial-services/our-insights/insurance-beyond-digital-the-rise-of-economies-and-platforms>

Not only this is huge, but it shows that being part of this digital ecosystem will be a matter of survival for some organizations.

APIs as an evolving paradigm

APIs are not new. In fact, they are far from it. The use of the term API can be traced back to 1968 to a publication titled *Data Structures and Techniques for Remote Computer Graphics* by I. W. Cotton and F. S. Greatorex, Jr. Since then, we've seen the term being born and re-born in proprietary protocols such as Sun Microsystems' **Remote Procedure Call (RPC)**, **Common Object Request Broker Architecture (CORBA)**, and **Distributed Component Object Model (DCOM)**. We've also seen it in public standards, such as XML-RPC, which then evolved to become **Simple Object Access Protocol (SOAP)**, which then, along with the **Web Services Description Language (WSDL)**, became the foundation for Web Services and **service-oriented architecture (SOA)**.

There was then a shift of paradigm into resource-centric and more lightweight APIs based on the REST architectural style. We are now back to RPC with emerging technologies such as GraphQL and gRPC, both of which are rapidly increasing in popularity.

The evolution of APIs is described in more detail in [Chapter 6, Modern API Architectural Styles](#).

However, what we see today is not just a technological shift of API technologies. The emergence of APIs as the means to enable digital ecosystems has created an economy of its own, an API economy, which has a more fundamental impact on how businesses organize their teams.

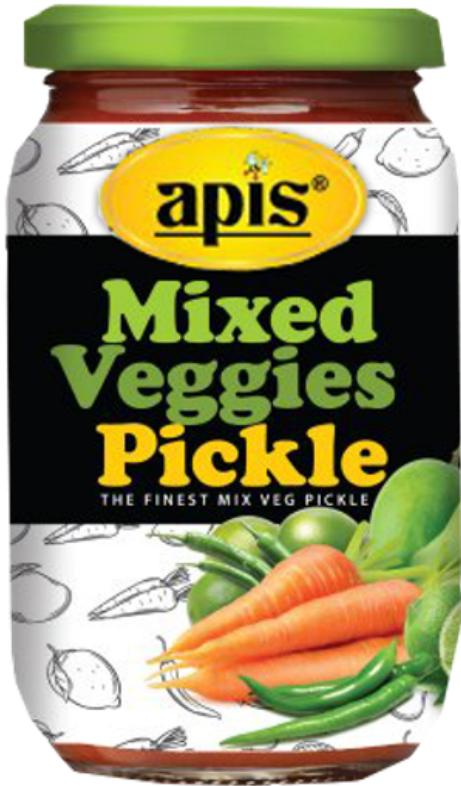


Figure 5: APIs as business products
Source: <http://www.apisindia.com/>

As businesses realize that APIs can truly be business products in their own right, the teams that deliver these products will no longer be simply considered IT teams or, to put it bluntly, cost centers. For businesses to succeed in the API economy, they need to shift away from traditional ways of delivering IT to a product-centric approach whereby the main goal is to make an API product successful and profitable.

Technical capabilities, especially with cloud computing now being the new normal, must too evolve in order to give these teams all of the tools they need to produce products that are innovative and competitive in the marketplace.

Who this book is for

This book will be of great benefit to developers, architects, and even IT/Digital executives looking for sources of inspiration when defining and implementing:

- Business-led API strategies
- API-led architectures and patterns
- API architectural styles to use (for example, REST, GraphQL, or gRPC)
- The full API life cycle, including related cycles such as the service and API consumer cycles
- Target operating models suitable for API products.

Lastly, as the book is technology-agnostic and doesn't offer strong views on tools, it can also be used as a reference to compare different products, whether they are commercial or open-sourced.

What this book covers

Chapter 1, The Business Value of APIs – this chapter gives context to the rest of the book by elaborating on what APIs mean to a business and why they should be embraced. It also talks about business drivers for APIs and how to determine their value based on an API value chain.

Chapter 2, The Evolution of API Platforms – this chapter takes a step back to look in detail at how technologies and platforms have evolved from traditional middleware and enterprise service bus-centric SOA architectures to fully federated, multi-cloud, and microservices-based architectures that enable APIs to exist and be managed wherever information resides.

Chapter 3, Business-Led API Strategy – the main focus of this chapter is to deliver a comprehensive approach to defining API strategies that have clear, concise, and business-centric goals and objectives.

Chapter 4, API-Led Architectures – this chapter walks through a reference architecture and all of the capabilities required to implement modern APIs and fully decouple (micro)services. The chapter is a great reference for what modern stacks should look like.

[Chapter 5, API-Led Architecture Patterns](#) – this chapter extends [Chapter 4](#) by walking through how the different capabilities described in the reference architecture can be combined in order to deliver sound solutions to common problems.

[Chapter 6, Modern API Architectural Styles](#) – this chapter gives a detailed overview of the trendiest API architectural styles (at the time this book was written). The chapter is a great source of inspiration for anyone looking for a point of view on different API styles and their pros and cons.

[Chapter 7, API Life Cycle](#) – this chapter walks through the full API life cycle and also related ones, such as the service and API consumer life cycles. The chapter is a great reference for those wishing to implement end-to-end API processes and tools from scratch or anyone looking for inspiration on how to enhance existing ones.

[Chapter 8, API Products' Target Operating Model](#) – as the name suggests, the chapter walks through what it really means to treat APIs as products and the implications this has for organizations. From core concepts to different operating models with their pros and cons, this chapter elaborates on a topic that is rarely discussed in the world of APIs.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [http://www.packtpub.com/sites/default/files/downloads/9781787284432_Co](http://www.packtpub.com/sites/default/files/downloads/9781787284432_ColorImages.pdf)
[lorImages.pdf.](http://www.packtpub.com/sites/default/files/downloads/9781787284432_ColorImages.pdf)

Conventions used

There are a number of text conventions used throughout this book.

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes. For example: "Modern APIs are broadly considered to be an evolution of the **Remote Procedure Call (RPC)** protocol."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit <http://www.packt.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

The Business Value of APIs

This chapter focuses on the value that **Application Programming Interfaces (APIs)** bring to the business. It begins by describing how digital disruption is forcing organizations to change in order to innovate and therefore avoid being disrupted. To this end, I explain how APIs enable digital strategies and digital transformation by unlocking key enterprise information assets and functionality, which are typically locked in **systems of record**, many of which are legacy. The chapter continues by elaborating on the value chain of APIs and how each level of maturity delivers new capabilities to the business.

Change or die

The world has changed. Information technology has changed every aspect of our lives: from fundamental things, such as how we purchase goods, interact with brands, and even do our jobs, to how we communicate with each other. In fact, a study by British psychologists suggests that around two billion people use smartphones across the globe, with over half the population in developed countries relying on them on a daily basis. That's over half a billion people worldwide using their phones to do all sorts of things, 85 times on average each day, according to the same study.

Refer to the study Beyond Self-Report: Tools to Compare Estimated and Real-World Smartphone Use for further information on the research mentioned.

<http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0139004#pone.0139004.ref001>

However, the aforementioned study only focuses on smartphone usage. If you factor in interactions with other devices, such as tablets, personal computers, wearables, and even machines (that is, smart cars and voice assistants such as Alexa), the number of interactions is huge.

What does this hyperconnectivity tell us?

For a start, it is pushing the boundaries of what we thought was possible and making science fiction seem real. Most importantly, it has opened up new avenues for businesses to innovate and disrupt the market, which is exactly what the so-called "digital disruptors," such as Google, Apple, Facebook, Amazon, and Netflix, in fact did, and continue to do.

Established businesses, such as Blockbuster and Kodak, couldn't cope with the (digital) innovation introduced by Netflix and Apple, and ended up filing for bankruptcy.

Traditional industries, such as the taxi industry and hospitality, are also being severely disrupted by Uber and Airbnb.



Figure 1.1: Apple digitally disrupted Kodak

These companies are just the obvious examples that everyone talks about. With over 100 million new start up businesses launched every year, even if only 10% (as analysts predict) are successful, we are talking about 10 million new companies with the potential to become the new Netflix or Uber, but for different industries.

Further reading: Shocking Number of Worldwide Business Start-Ups Each Year?

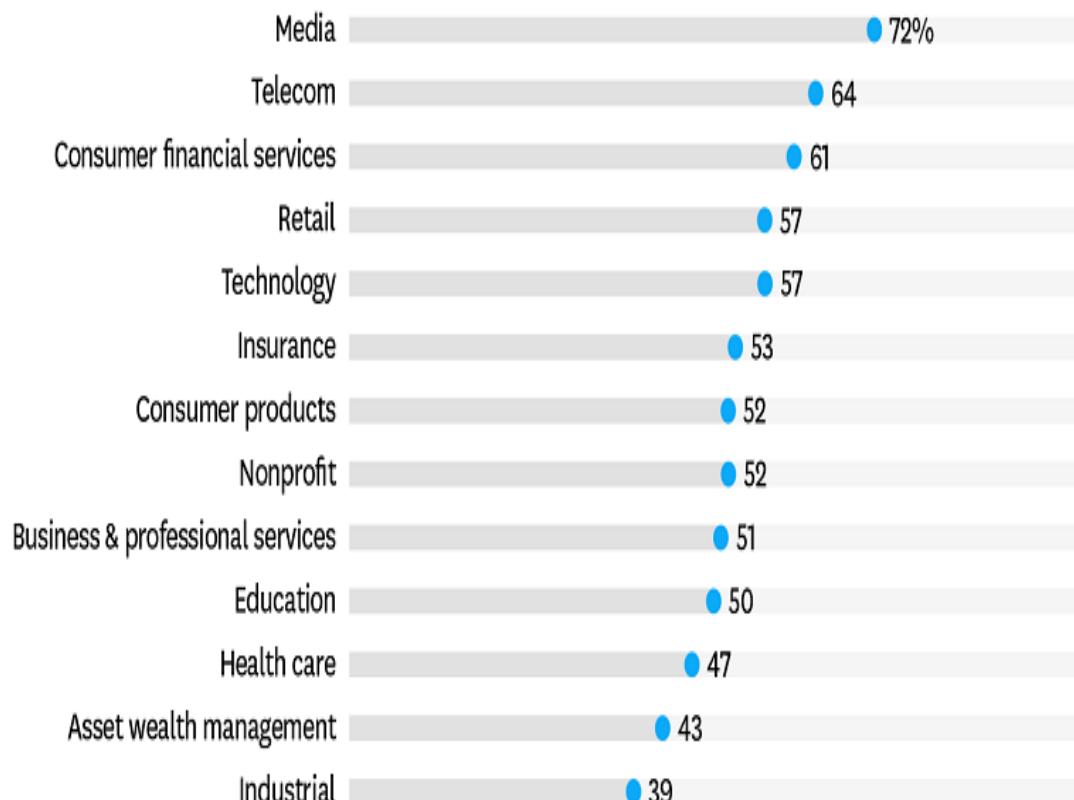
<http://www.lerumba.com/Directory/shocking-number-of-worldwide-business-start-ups-each-year-article-39.aspx#.WTfmxRPytE5>

Now, because of this, it's no wonder that most organizations globally have embarked on digital transformation initiatives in order to avoid being (further) disrupted. As **Harvard Business Review (HBR)** nicely put it:

"Digital is no longer the shiny front end of the organization - it's integrated into every aspect of today's companies."

According to the same article by HBR, the most disrupted industries are those that relate to **Business to Consumer (B2C)**, with media and telecom at the top of the list, closely followed by financial services and retail:

Executives Who Anticipate Moderate or Massive Digital Disruption in the Next 12 Months, by Industry



SOURCE "DIGITAL PULSE 2015," BY RUSSELL REYNOLDS ASSOCIATES

© HBR.ORG

Figure 1.2: Disruption according to industry

However, these figures should not come as a surprise. A closer look shows that there is a direct correlation between the disrupted B2C organizations and the fact that 2 billion individuals are using their smartphones and other devices frequently. Put simply, B2C organizations that haven't been able to innovate and engage customers in different ways, and through digital channels, are more susceptible to being disrupted by newer and more agile businesses:

"The most disrupted industries typically suffer from a perfect storm of two forces. First, low barriers to entry into these sectors lead to more agile competition. Secondly, they have large legacy business models which often generate the majority of their revenue. These organizations, therefore, have embedded cultural and organizational challenges when it comes to changing at the pace required."

Further reading: The Industries That Are Being Disrupted the Most by Digital

<https://hbr.org/2016/03/the-industries-that-are-being-disrupted-the-most-by-digital>

The digital dilemma

These organizations that are more exposed to digital disruption are therefore faced with a dilemma. In order to remain relevant and stay in business, they must create a **digital strategy** that allows the business to innovate and be more agile. However, in order to do so, they can't simply get rid of old systems of record, most of which are legacy and contain critical information assets that support day-to-day operations.



“What if we don’t change at all ...
and something magical just happens?”

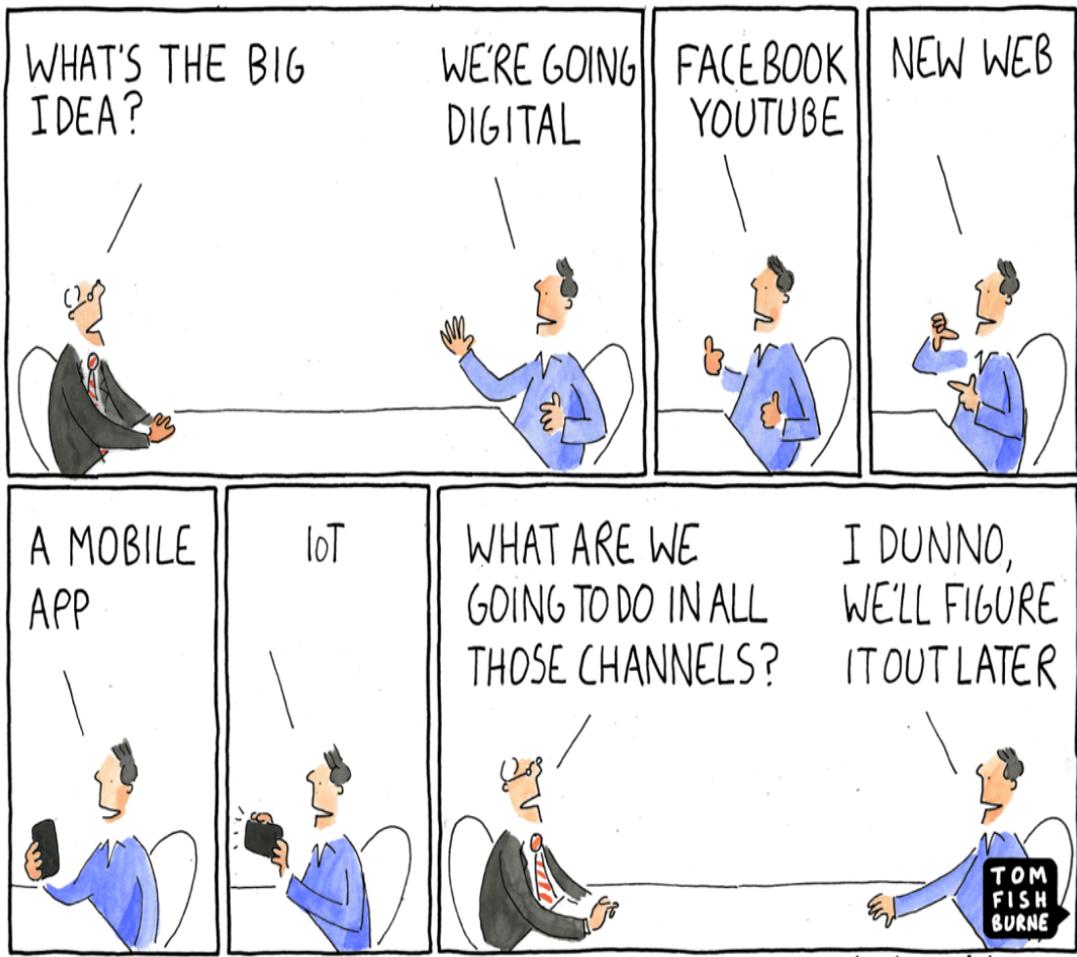
Figure 1.3: The digital dilemma

Bearing in mind that most of these organizations can't afford to start from a white sheet of paper, the digital strategy must therefore cater to the transformation of hundreds (if not thousands) of existing systems, many of which are considered legacy.

Such an undertaking can be huge, not only in terms of costs to the business, but also in terms of the risks. This is exactly where the dilemma lies: do nothing and save costs/avoid risks, and most likely end up being disrupted, or become a disruptor by taking the business on a digital transformation journey, which could be risky and costly.

Access to enterprise information and functionality is king

Is it really that risky and costly to take the business on a digital transformation journey? Well, as with everything, it depends. Organizations that perceive digital transformation as merely an exercise to adopt omnichannel strategies, without first understanding "why" they are doing it, or "what" they are trying to accomplish, will most likely fail to realize any business benefits. For these organizations, such an undertaking will have plenty of unknowns and will therefore be perceived as risky and costly.



© marketoonist.com

Figure 1.4: An accidental multichannel strategy

Organizations that start off with the creation of a digital strategy to articulate the targeted business goals, and also identify what business and technical capabilities are required in order to achieve this, will most likely perceive the digital journey as a key enabler for the business strategy, rather than just another expensive IT project. For such organizations, digital transformation represents a justifiable and calculated risk.

It's no wonder that Forbes listed digital transformation as the #1 priority for **Chief Information Officer's (CIO)** in 2017:

"Either companies figure out how to outsmart, outpace, and outmaneuver their competitors with the clever, customer-focused deployment of digital technologies, or they will be marginalized-sooner rather than later."

Further reading: Top 10 Strategic CIO Priorities For 2017

<https://www.forbes.com/sites/oracle/2017/01/17/top-10-strategic-cio-priorities-of-2017/>

However, in order to achieve this, the devil is in the detail. The "how" question should not be forgotten and must be thoroughly addressed while defining the strategy. For example, one of the biggest challenges faced by organizations undertaking digital transformation is around how to get access to core enterprise information assets, most of which are typically locked in hundreds of systems of record. Therefore, without unrestricted, secured, and reliable access to such systems, introducing any sort of innovation will be nothing more than a prototyping exercise.

*A **system of record (SOR)** is a data management term for an information storage system (commonly implemented on a computer system running a database management system) that is the authoritative data source for a given data element or piece of information. Source: https://en.wikipedia.org/wiki/System_of_record*

What are APIs and why should a business care?

APIs are like doors that provide access to information and functionality to other systems and/or applications. APIs share many of the same characteristics as doors:

- Most of them have locks and, without the key, they can't be opened.
- They come in different types (size, color, material, type of lock, and so on).
- They can serve different purposes. For example, they can be public-facing or just internally accessed.
- They are located in a specific location: an address.
- They can be as secure and closely monitored as required.
- If they don't work, it will affect the experience of their users.

APIs, however, are not new. In fact, the concept goes back a long time and has been present since the early days of distributed computing (some argue even before then).

However, the term as we know it today refers to a much more modern type of API, known as REST or Web APIs.

The term REST APIs was first introduced in the year 2000 by Roy Fielding in his PhD dissertation Architectural Styles and the Design of Network-based Software Architectures. In his dissertation, Roy presented

Representational State Transfer (REST) *as a way for computer systems to interoperate over the internet, by making correct use of the already available Hypertext Transfer Protocol (HTTP).*

For further information, refer to the following link: [*https://en.wikipedia.org/wiki/Representational_state_transfer#History*](https://en.wikipedia.org/wiki/Representational_state_transfer#History)

Modern APIs started to gain real popularity when, in the same year of their inception, eBay launched its first public API as part of its eBay Developers Program. eBay's view was that by making the most of its website functionality and information also accessible via a public API, it would not only attract, but also encourage communities of developers worldwide to innovate, by creating solutions using the API. From a business perspective, this meant that eBay became a platform for developers to innovate on and, in turn, eBay would benefit from having new users that perhaps it couldn't have reached before.

eBay was not wrong. In the years that followed, thousands of organizations worldwide, including known brands, such as Salesforce.com, Google, Twitter, Facebook, Amazon, Netflix, and many others, adopted similar strategies. In fact, according to programmableweb.com (a well-known public API catalogue), the number of publicly available APIs has been growing exponentially, reaching over 20k as of August 2018:

Growth In [Public] Web APIs Since 2005



Programmable Web

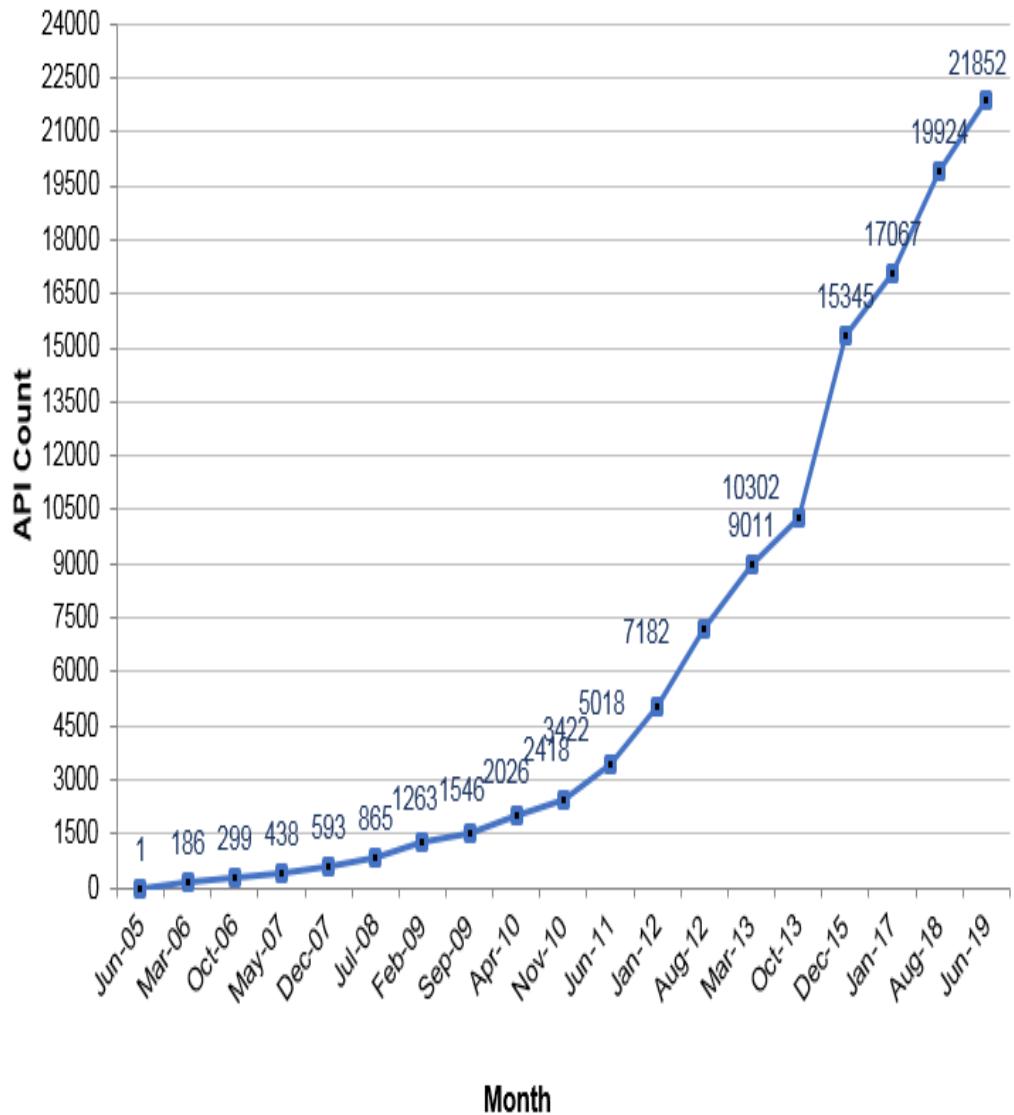


Figure 1.5: Public APIs as listed in programmableweb.com in August 2018

It may not sound like much, but considering that each of the listed APIs represents a door to an organization's digital offerings, then we're talking about thousands of organizations

worldwide that have already opened their doors to new **digital ecosystems**, where APIs have become the products these organizations sell and developers have become the buyers of them.

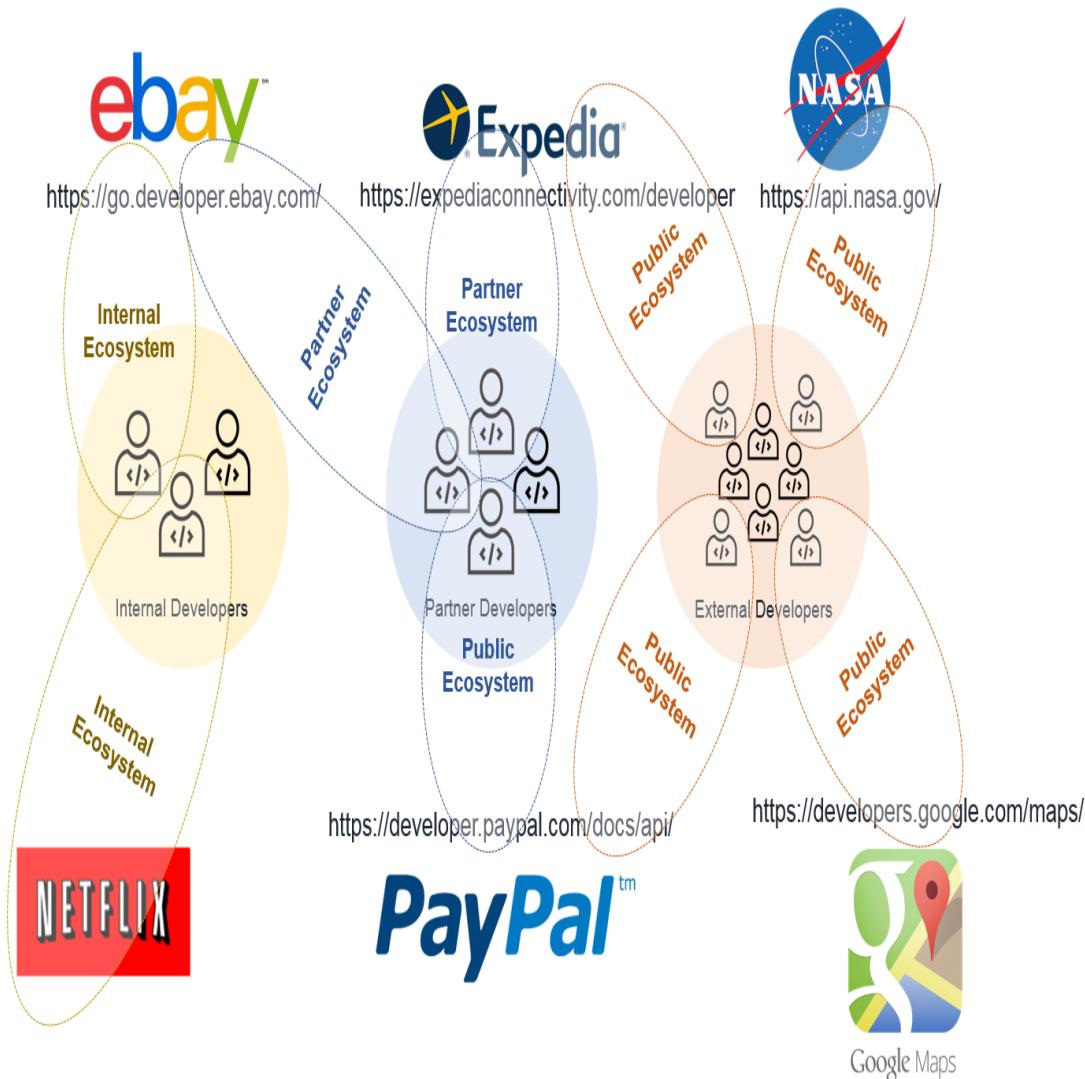


Figure 1.6: Digital ecosystems enabled by APIs

In such digital ecosystems, communities of internal, partner, or external developers can rapidly innovate by simply consuming these APIs to do all sorts of things: from offering hotel/flight booking services by using the Expedia API, to providing

educational solutions that make sense of the space data available through the NASA API.

There are ecosystems where business partners can easily engage in business-to-business transactions, either to resell goods or purchase them, electronically and without having to spend on **Electronic Data Interchange (EDI)** infrastructure. Ecosystems where an organization's internal digital teams can easily innovate as key enterprise information assets are already accessible.

So, why should businesses care about all this? There is, in fact, not one answer, but multiple answers, as described in the subsequent sections.

APIs as an enabler for innovation and bimodal IT

What is **innovation**? According to a common definition, innovation is the process of translating an idea or invention into a good or service that creates value, or for which customers will pay. In the context of businesses, according to an article by HBR, innovation manifests itself in two ways:

- **Disruptive innovation:** Described as the process whereby a smaller company with fewer resources is able to successfully challenge established incumbent businesses.
- **Sustaining innovation:** When established businesses (incumbents) improve their goods and services in the eyes of existing customers. These improvements can be incremental advances or major breakthroughs, but they all enable firms to sell more products to their most profitable customers.
- *Further reading: What is disruptive innovation?*
<https://hbr.org/2015/12/what-is-disruptive-innovation>

Why is this relevant? It is well known that established businesses struggle with disruptive innovation. The Netflix versus Blockbuster example reminds us of this fact. By the time disruptors are able to catch up with an incumbent's portfolio of goods and services, they are able to do so with lower prices, better business models, lower operating costs, and far more agility and speed to introduce new or enhanced features. At this point, sustaining innovation is not good enough to respond to the challenge.

With all the recent advances in technology and the internet, the rate at which disruptive innovation is challenging incumbents has only grown exponentially. Therefore, in order for established businesses to endure the challenge put upon them, they must somehow also become disruptors. The same HBR article describes a point of view on how to achieve this from a business standpoint. From a technology standpoint, however, unless the several systems that underpin a business are "enabled" to deliver such disruption, no matter what is done from a business standpoint, this exercise will likely fail.

Perhaps by mere coincidence, or by true acknowledgment of the aforesaid, Gartner introduced the concept of **bimodal IT** in December 2013, and this concept is now mainstream.

Gartner defined bimodal IT as the following:

"The practice of managing two separate, coherent modes of IT delivery, one focused on stability and the other on agility. Mode 1 is traditional and sequential, emphasizing safety and accuracy. Mode 2 is exploratory and nonlinear, emphasizing agility and speed."

Bimodal IT

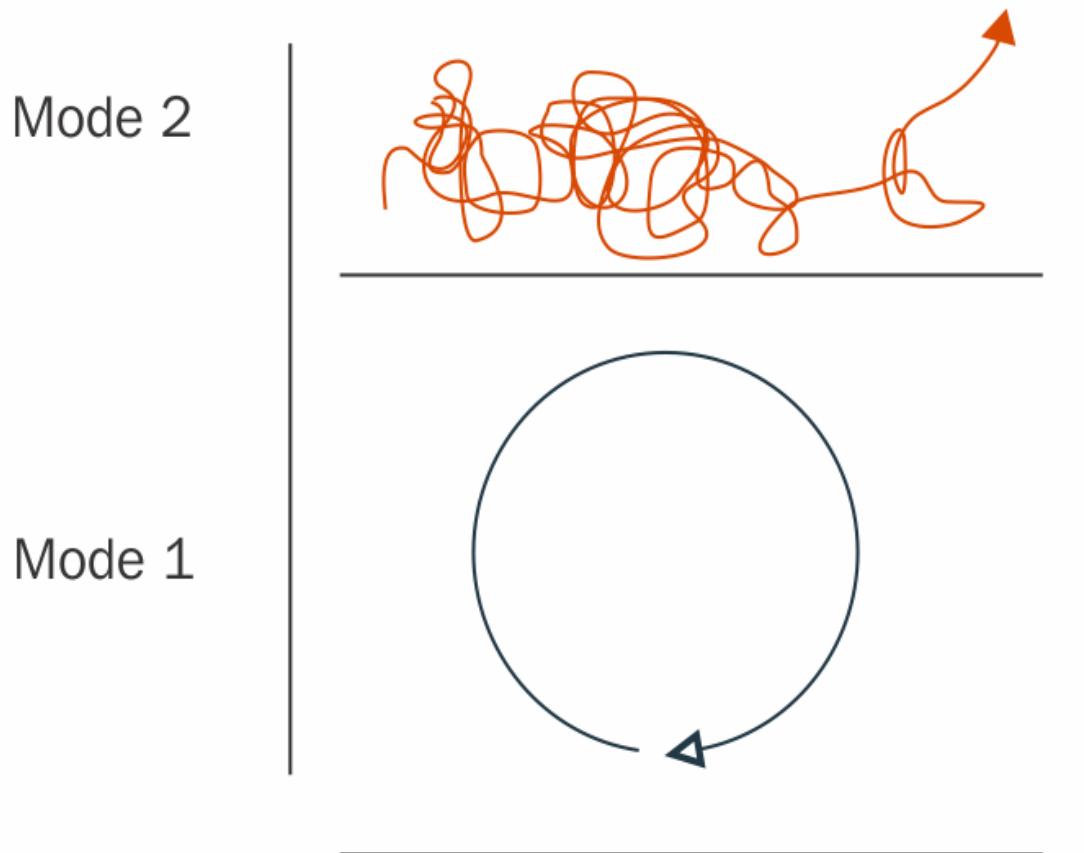


Figure 1.7: Gartner's bimodal IT

According to Gartner, **Mode 1** (or slow) IT organizations focus on delivering core IT services on top of more traditional and hard-to-change systems of record, which, in principle, are changed and improved in longer cycles, and are usually managed with long-term waterfall project mechanisms. Whereas, for **Mode 2** (or fast) IT organizations, the main focus is to deliver agility and speed, and therefore they act more like a start-up (or digital disruptor in HBR terms) inside the same enterprise.

Further reading: Bimodal IT: Business-IT alignment in the age of digital transformation

https://www.researchgate.net/publication/287642679_Bimodal_IT_Business-IT_alignment_in_the_age_of_digital_transformation

However, what is often misunderstood is how fast IT organizations can disruptively innovate, when most of the information assets, which are critical to bringing context to any innovation, reside in backend systems, and any sort of access has to be delivered by the slowest IT sibling. This dilemma means that the speed of innovation is constrained to the speed by which the relevant access to core information assets can be delivered.

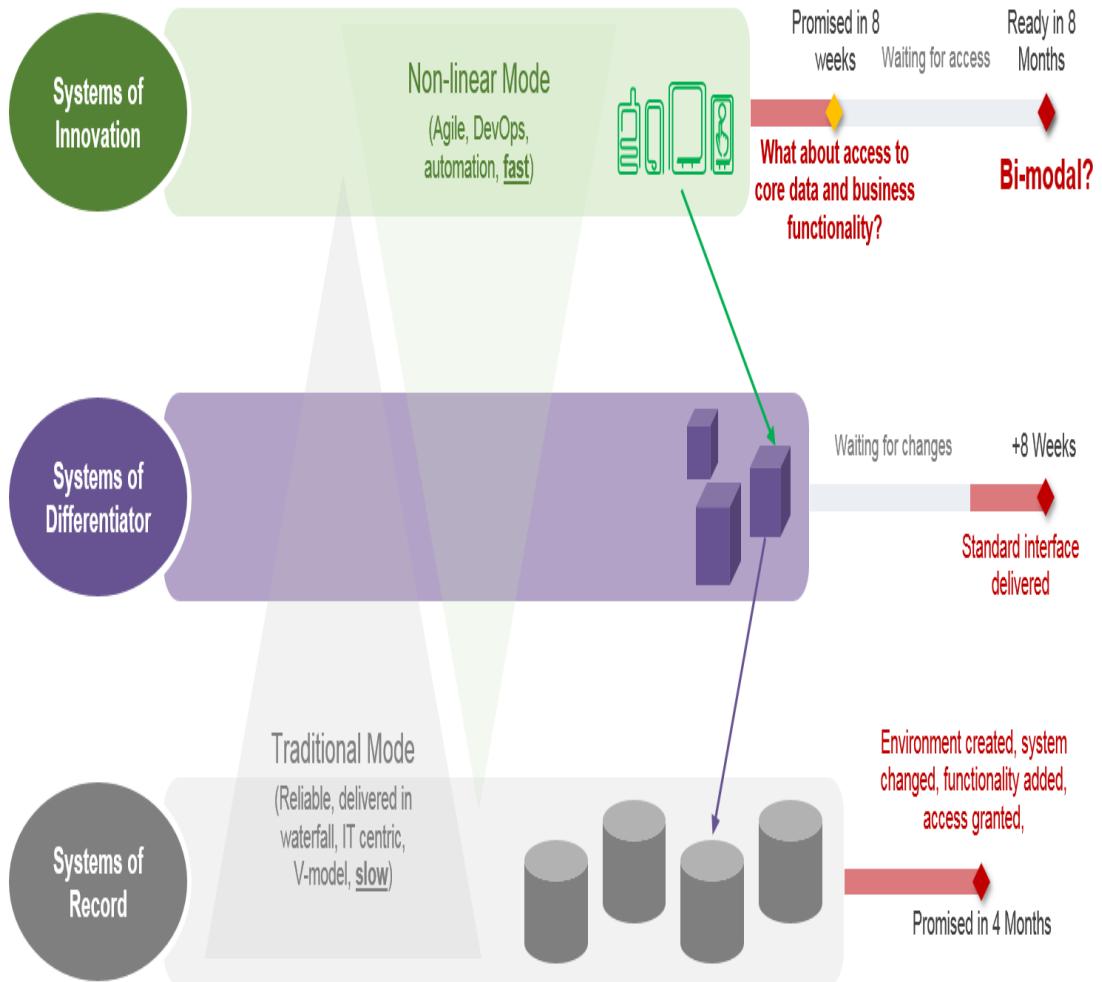


Figure 1.8: Bimodal IT - is it really?

As the saying goes, "Where there's a will, there's a way." APIs could be implemented as a means for the fast IT to access core information assets and functionality, without the intervention of the slow IT. By using APIs to decouple the fast IT from the slow IT, innovation can occur more easily.

However, as with everything, it is easier said than done. In order to achieve such organizational decoupling using APIs, organizations should first build an understanding about what

information assets and business capabilities are to be exposed as APIs, so the fast IT can consume them as required.

This understanding must also articulate the priorities of when different assets are required and by whom, so the creation of APIs can be properly planned for and delivered.

Luckily, for those organizations that already have mature **service-oriented architectures (SOA)**, some of this work will probably already be in place. For organizations without such luck, this activity should be planned for and should be a fundamental component of the digital strategy.

Then, the remaining question would be: which team is responsible for defining and implementing such APIs; the fast IT or the slow IT? Although the long answer to this question is addressed throughout the different chapters of this book, the short answer is neither and both. It requires a multi-disciplinary team of people, with the right technology capabilities available to them, so they can incrementally API-enable the existing technology landscape, based on business-driven priorities.

APIs to monetize on information assets

Many experts in the industry concur that an organization's most important asset is its information. In fact, a recent study by **Massachusetts Institute of Technology (MIT)** suggests that data is the single most important asset for organizations:

"Data is now a form of capital, on the same level as financial capital in terms of generating new digital products and services. This development has implications for every company's competitive strategy, as well as for the computing architecture that supports it."

Further reading: The Rise of Data Capital

http://files.technologyreview.com/whitepapers/MIT_Oracle+Report-The_Rise_of_Data_Capital.pdf

If APIs act as doors to such assets, then APIs also provide businesses with an opportunity to monetize them. In fact, some organizations are already doing so. According to another article by HBR, 50% of the revenue that Salesforce.com generates comes from APIs, while eBay generates about 60% of its revenue through its API. This is perhaps not such a huge surprise, given that both of these organizations were pioneers of the API economy.



60%

Salesforce.com has a marketplace (App Exchange) for apps created by its partners that work on its platform; they now number more than 300 partners.

90%

Expedia's APIs allow people using third-party websites to tap its functionality in order to book flights, cars, and hotels.

60%

eBay to list its auctions on other websites, get bidder information about sold items, collect feedback on transactions, and list new items for sale — all of which give additional exposure to eBay items and increase revenue.

Figure 1.9: The API economy in numbers

What's even more surprising is the case of Expedia. According to the same article, 90% of Expedia's revenue is generated via APIs. This is really interesting, as it basically means that Expedia's main business is to indirectly sell electronic travel services via its public API.

Further reading: The Strategic Value of APIs

<https://hbr.org/2015/01/the-strategic-value-of-apis>

However, it's not all that easy. According to the previous study by MIT, most of the CEOs for Fortune 500 companies don't yet fully acknowledge the value of APIs. An intrinsic reason for this could be the lack of understanding and visibility over how data is currently being (or not being) used. Assets that sit hidden on systems of record, only being accessed via traditional integration platforms, will not, in most cases, give insight to the business on how information is being used, and the business value it adds. APIs, on the other hand, are better suited to providing insight about how/by who/when/why information is being accessed, therefore giving the business the ability to make better use of information to, for example, determine which assets have better capital potential.

APIs for regulatory compliance

Another challenge that is increasingly being faced by organizations concerns compliance and regulation. Let's take, for example, the introduction of the **General Data Protection Regulation (GDPR)**, which, as of May 2018, regulates how organizations worldwide are expected to handle the customer data of **European Union (EU)** citizens, with the risk of being exposed to eye-watering fines. Similarly, the **second payment service directive** by the EU, otherwise known as **PSD2**, will introduce important regulations to open up core banking transactions and information.

GDPR

Superseding the EU Data Protection Directive, GDPR has the objective to give individuals (EU citizens) more control, protection, and privacy over how their personal information is used and by whom.

The regulation is quite extensive and, for many organizations, achieving GDPR compliance will be (or has been) an expensive and long process. The full GDPR regulation is available at

<https://www.itgovernance.eu/en-ie/eu-general-data-protection-regulation-gdpr-ie>

With personal data being at the heart of GDPR, how can APIs help with complying with the GDPR regulation? Although APIs may not be the only answer, a good API management solution will introduce strong access control over who can access what information via APIs, therefore ensuring that personal data is not misused or accessed without prior consent. In addition to these controls, the solution should also provide full visibility and auditability over data access, meaning that any data breach can be notified to customers and authorities as soon as possible, or within the 72-hour period, as indicated in the regulation.

PSD2

PSD2 aims to stop financial institutions' monopoly over the use of customer data and payment services. Before the end of 2018 (when the directive is supposed to come into effect), financial institutions in the EU must open the doors of their customers' data and payment services to third-party providers.

The full PSD2 directive is available at

https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en

In practical terms, what this means is that in the near future, you might be using Facebook, for example, to check bank account balances, do bank transfers, and pay bills.

Another example, in the same industry, is the **Open Banking** initiative being introduced in the United Kingdom as a result of the *Retail Banking Market Investigation* report produced by the **Competition and Markets Authority (CMA)**. In a nutshell, the initiative aims to promote increased competition and consumer choices in the banking industry by forcing banks to securely share their data via an **Open Banking API**.

For further reading on the Open Banking initiative, refer to the following link:

<https://www.gov.uk/government/news/open-banking-revolution-moves-closer>

However, this is easier said than done. According to research, over 75% of financial institutions in Europe still run on

outdated systems. Worldwide, the figure is similar, if not more.

Further information on this research is available at

<http://www.computerweekly.com/news/2240150122/Banks-still-handicapped-by-IT-legacy>

Bearing in mind that making changes to these systems won't be a trivial task, the expectation is that software vendors and system integrators alike will come up with pre-built solutions, which will make the process of creating APIs on top of systems and complying with regulations, such as PSD2 and CMA Open Banking, a lot easier.

Fast Healthcare Interoperability Resources (FHIR)

It is not just the financial industry that's embracing the use of APIs. In healthcare, for example, a newer version of the widely adopted **health-level 7 (HL7)** international standard, known as the **Fast Healthcare Interoperability Resources**, or **FHIR** (pronounced "fire"), defines, in fact, a REST API.

Further information on FHIR is available at

<https://www.hl7.org/fhir/http.html>

In the USA, for example, the healthcare industry is taking a step further and introducing a rule to promote the use of standard APIs to access patient records.

Recommend reading: A Brief Summary of the CMS Meaningful Use Final Rule

<http://geekdoctor.blogspot.co.uk/2015/10/a-brief-summary-of-cms-meaningful-use.html>

Although it is still very early days, the expectation is that this trend will continue, and that more regulation will be introduced that promotes the use of APIs as the means to provide open access to information and enable interoperability.

APIs for the reuse of business capabilities

Just as is the case in traditional SOA, whereby one of the key principles is to build reusable web services and not just to avoid duplication of functionality, but also to reduce development costs, in the case of web APIs, the same principle can apply.

It is possible, and, in fact, recommended, that business APIs are created internally, so business functionality that needs to be commonly accessed is then made available as an API. This will not only allow such functionality to be accessed in real time and in a standard, controlled, and secure way, but it is also a much better alternative to data replication techniques that risk losing visibility and control over who by/why/when/how information is being accessed.

By creating a common business API layer, not only does innovation and bimodal IT become possible (as described previously), but other business benefits can be realized, such as lower development costs by reusing already available APIs, reduced duplication of system functionality, and increased visibility and analytics around the usage of data, which can provide the business with meaningful business insights.

Avoiding a hyperconnectivity mess

With an increased number of public and internally developed APIs offering a wide range of functionality (that is, access to **Software as a Service (SaaS)** applications), bank transactions, artificial intelligence, and address services, to name a few), it can be quite tempting for developers to quickly incorporate the use of all sorts of APIs within their applications.

However, doing so in an uncontrolled manner can, and will most likely, result in what some call a **hyperconnectivity mess**. This is when IT systems are interconnected and dependent on APIs, but no one within the enterprise really has visibility and/or understanding of this. Not only can this result in a serious gap in accountability when issues occur, but, in an even more complex IT landscape, systems can have real exposure to issues outside of the control of enterprise IT.

PETER VAN

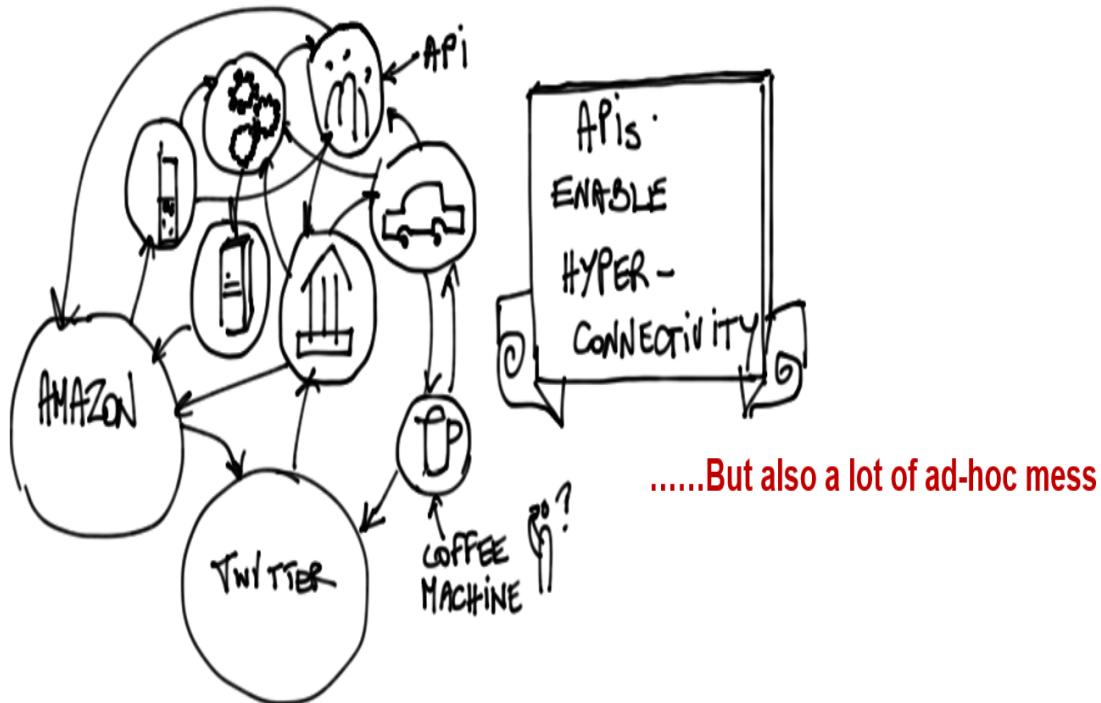


Figure 1.10: Hyperconnectivity can also create an ad hoc mess

A hyperconnectivity mess occurs as a result of APIs being used in an ad hoc manner and without proper governance. At this point, the business benefits that APIs have to offer can be countered by the risks they can introduce to core enterprise systems, and thus business operations themselves.

This is the reason that the management of APIs has become so critical, and this does not just apply to the APIs being internally developed within enterprise IT, but also to the use of public APIs within enterprise systems.

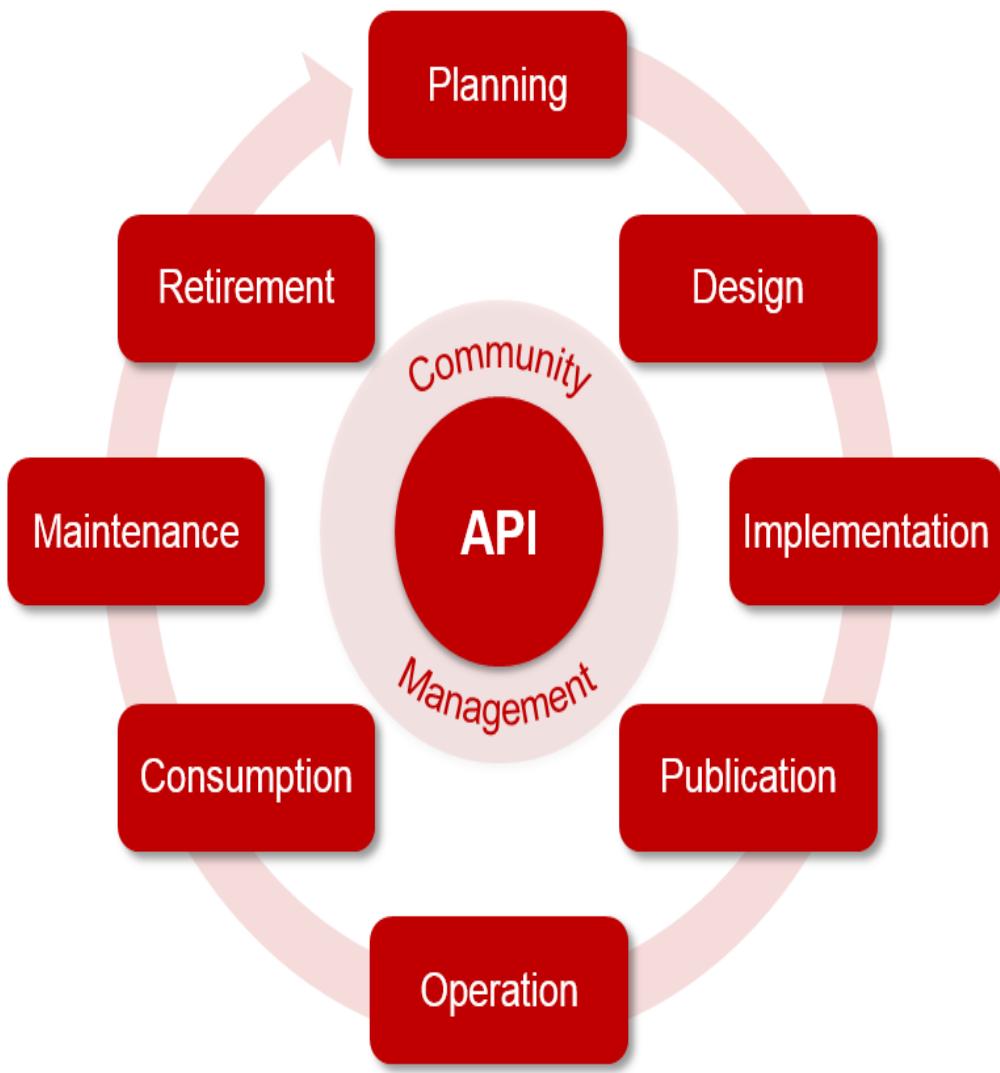


Figure 1.11: API management

API management, therefore, is born as a discipline to manage APIs (both internal and external), meaning establishing the processes, roles, and responsibilities, and the tools required to govern APIs throughout their full life cycle.

API management differs from related disciplines, most notably SOA governance, in that it is much more lightweight and a lot more focused on making the lives of the API consumers (developers) easier, by providing the right tools for the design and run aspects of APIs, and making processes simple to follow. SOA governance, on the other hand, is fine-grained, with

detailed processes and complex tools. Chapter 2, The Evolution of API Platforms, covers this in more detail.

Any API management initiative should focus on at least the following aspects of the life cycle:

1. **Planning:** Provides the required facilities (tools) to plan in advance for the creation and/or modification of APIs. Regardless of the methodology used to deliver the APIs or whether there is one or multiple teams implementing it, there should be a common approach, and ideally tooling to capture which APIs are the priority, and who is responsible for delivering them. This is important as it will provide visibility to any relevant party of the capabilities being delivered, and therefore encourage coordination/collaboration over the duplication of work. The tools used to ensure tracking/status of the teams implementing APIs should also be addressed.
2. **Design:** Design-first thinking is fundamental in any API management initiative. Tools and processes that enable API-first design (covered in detail in subsequent chapters), and that encourage API designers and API consumers to interact during the design of an API, will shorten the development life cycle and therefore reduce costs, as the actual product produced will most likely meet the requirements from the get-go, without having to iterate several times through the entire implementation process to get it right.

- *An important consideration during the design phase is around what level of security controls are to be adopted in the API. Authentication and authorization, for example, should not be an afterthought, as they will have considerable impact on API usability. Therefore, rather than doing this later in the life cycle, security should also be part of the API design.*

3. Implementation: The actual implementation of the APIs requires adequate processes and tools to be in place, such that developers can focus their efforts on producing actual code and not on sorting out life cycle concerns, such as code coverage, continuous integration, regressing testing, and deployment. For this reason, automating and streamlining the implementation cycle of the API, by creating development pipelines that make it very easy for developers to move code from development all the way to production, will deliver considerable results for the business.

- *It's worth highlighting that development pipelines do not mean bypassing quality gates. It is still possible, in fact recommended, to also introduce quality gates. However, if the same can be automated (that is, verifying that the results of code coverage and regression testing are adequate), quality assurance can still be introduced, but without the burden and costs of manually testing the API.*

4. Publication: Making APIs discoverable is fundamental in API management. Providing the

facilities to easily deploy and version APIs, but most importantly to publish them along their relevant (consumer-oriented) documentation in a developer portal, ensures that developers can reuse APIs, rather than reinventing wheels, and ultimately reduces development and operations costs.

5. Operation: Runtime operations is as much about "keeping the lights on" as it is about providing meaningful analytical insight to both the business and IT, so they, too, can make the most out of the operational data being generated. From simple capabilities, such as central operations, API statistics, gateway stats, user management, and system management, to more sophisticated ones, such as **application performance monitoring (APM)**, SLA management, rule-based alerting, predictive analytics, self-healing, and API metering, operations is, without a doubt, a first-class citizen in API management.

6. Consumption: API management is not just about designing and building APIs, but also about consuming them. With the number of public APIs growing exponentially, the expectation is that some organizations will be consuming more public APIs than they will end up building them. The problem is that without proper controls and visibility over who by/why/which/when public APIs are being used and the associated costs, organizations can easily end up in the hyperconnectivity mess described previously. To

prevent this pitfall, API management must equally focus on providing the means and facilities for public APIs to be consumed in a controlled and governed manner. In other words, developer portals should not only allow for internally developed APIs to be published, but also external ones.

7. **Maintenance:** In API management, the life cycle doesn't end when an API goes live. In fact, it only gets started. As it will be better described in the next section, APIs should be treated as products and, as such, the product must be continuously evolved by taking into account evolving consumer needs and expectations. For an API to become a good product, it must undertake a series of iterations and changes. API management should therefore make it easier to do so.
8. **Retirement:** When an API has served its purpose and there is a need to decommission it, it should not be the case that doing so is complex and cumbersome. API management should also take care of the process and capabilities needed to retire an API and handle (minimize) the impact that this may cause to any existing consumers.
9. **Community management:** As previously described, APIs also open the door to new digital ecosystems. In such ecosystems, the main actor is the developer. With thousands of developers worldwide, managing communities of internal (known) developers, partner

developers, and external (unknown) developers is another fundamental aspect of API management. Self-service facilities for development onboarding and developer portals, whereby developers can search for APIs, subscribe to them, read their documentation and even comment and rate them, are some of the capabilities that API management should offer.

The API value chain

Realizing the benefits that APIs have to offer to a business can't be completed in a day, and organizations that think that monetizing their information assets will be a simple and straightforward exercise are going to be in for a surprise. Rome wasn't built in a day, or so it goes.

Like most things, there is always a journey and a path that, when followed, will guide us toward getting to an end goal or a target. It will not necessarily be quick, as that pretty much depends on the pace an organization can deliver, but at least there will be the certainty of avoiding common pitfalls. That is not to say that some organizations might not opt for a different (and perhaps shorter) path.

That said, the following API value chain illustrates both a path and a maturity model to help organizations of all sizes to embark on the journey of API management maturity:

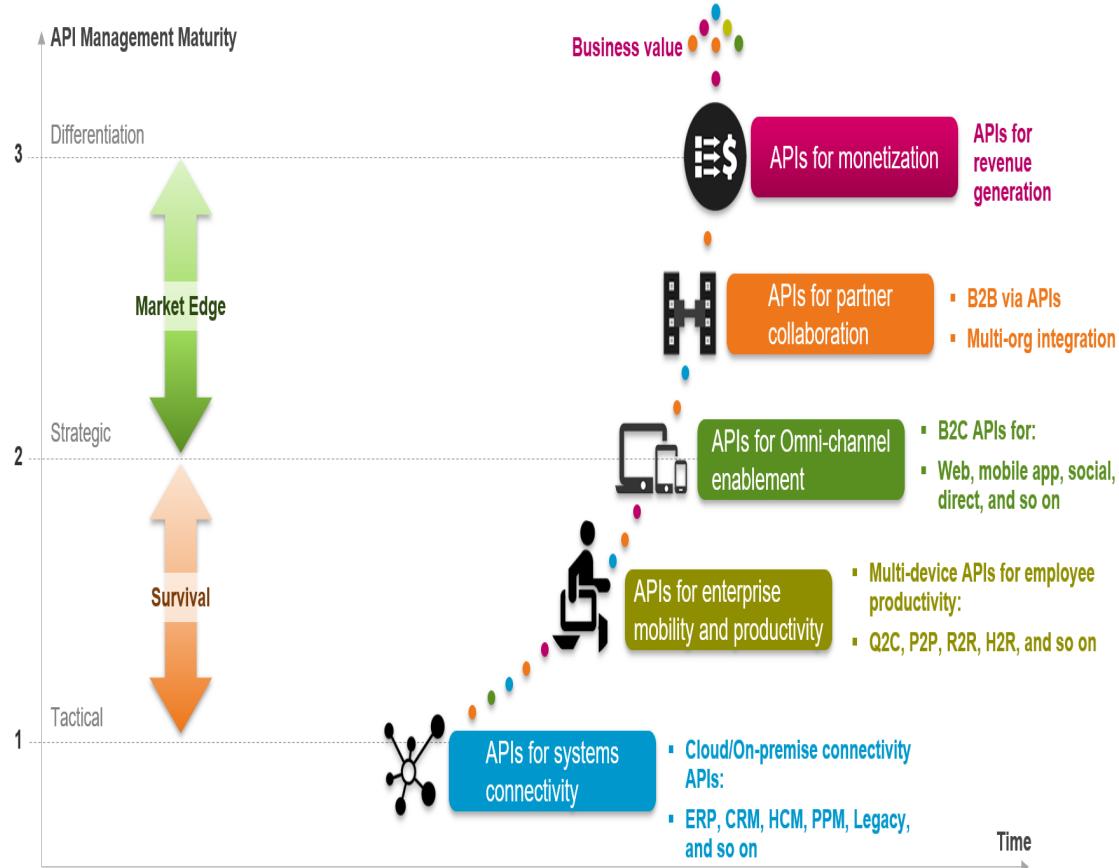


Figure 1.12: The API value chain

The value chain classifies APIs into five main groups. Each group is determined based on the business value it adds, which, in turn, also dictates maturity according to this business-led model:

API group	Description	API maturity	Business v
APIs for system	The most basic group	Level 1 – tactical	Access to co information

connectivity	<p>of APIs, as their aim is to provide access to core enterprise information assets, such as systems of record. Could be an on-premise system or SaaS applications.</p>		<p>the main business value. The business value can be realized through such access, which is much dependent on solutions but</p>
APIs for enterprise mobility	<p>APIs created in support of mobility solutions, meaning that they are not just about access to information, but rather they provide access to</p>	<p>Level 1 – tactical</p>	<p>This group of APIs is a more direct and measurable way to support the business, resulting in optimal business processes and effectiveness gained by all employees to interact with systems.</p>

	<p>business processes and other business capabilities.</p>		<p>different ways digital channels</p>
APIs for enterprise mobility and productivity	<p>APIs that enable a business to offer goods and services to customers via multiple digital channels. In other words, B2C. APIs in support of Internet of Things (IoT) solutions also fall within this group.</p>	<p>Level 2 – strategic</p>	<p>This group contains fundamental B2C digital interfaces that enables omnichannel strategies by providing information and functionality through multiple channels, from web, mobile, kiosks, and social media, to name a few. Now, because their business is evident and quantifiable, APIs that enable IoT also fall into this group. These APIs provide IoT</p>

			<p>provide for</p> <p>with access to enterprise infrastructure assets and functionality is worth noting. IoT is a much more complex topic and there is only one representative element of it.</p>
APIs for partner collaboration	<p>APIs that enable partner collaboration and</p> <p>Business to Business (B2B) by optimizing and simplifying business transactions.</p> <p>In other words, an API (and much more)</p>	Level 2 – strategic	<p>B2B transactions are complex top reasons, without integrations infrastructure required to do a major one. B2B and partner collaboration is hugely simplified. The cost of integration will also open up new ways for businesses to collaborate. The business collaboration considerability.</p>

	<p>much cheaper) alternative to traditional EDI-style integrations.</p>		<p>especially to organizations that don't deal with sales, but rather indirect ones, therefore relies on third parties to sell products and services.</p>
APIs for monetization	<p>APIs offered as commercial products in their own right. As such, their usage also entails a form of fee.</p> <p>The fee does not necessarily have to be a monetary one. As Mark O'Neill, a key</p>	<p>Level 3 – differentiation</p>	<p>By APIs becoming saleable products also become source of direct revenue for the business, whether actual products in the form of access to either information or business functions.</p> <p>Therefore, the value delivered by these products directly proportional to the success of the product itself.</p>

	<p>From, a key integration and API consultant from Gartner, said:</p> <p><i>"API monetization doesn't just mean charging for API calls."</i></p>		<p>Product innovation also means the business functions such as marketing, sales, and finance should play a role in making a successful product.</p>
	<p>According to Mark, organizations seeking to monetize APIs should first identify their monetization strategy and, from that, derive a charging or pricing model.</p> <p>Further details on</p>		<p>For this reason, although this APIs has the potential business value, realizing this potential also requires more mature discipline, and alignment with the business.</p> <p>It must be a well-driven initiative.</p>

how to
monetize
APIs are
provided in Ch
apter
3, Business-
Led API
Strategy.

Businesses that want to realize the full business benefits that APIs have to offer, for example, as part of their digital transformation initiatives, should first consider what would be their entry point and, based on that, determine a roadmap to get to the next levels. Details on how to define such a roadmap are described in Chapter 3, Business-Led API Strategy.

*There is another well-known and publicly available API maturity model, known as the **Richardson Maturity Model**. However, this model focuses more on the technical aspects of APIs, rather than the business and organizational aspects of an API management initiative. Therefore, both models can be complementary and can be used in conjunction to evaluate business, organizational, and technical aspects of APIs and their management.*

Further information on the Richardson Maturity Model is available at <https://martinfowler.com/articles/richardsonMaturityModel.html>

APIs as a driving force for many large acquisitions in the software industry

The value and potential that APIs bring to a business haven't gone unnoticed. Many of the largest software vendors worldwide have made considerable investments to strengthen their API management portfolios in a relatively short period of time. In less than three years, six major acquisitions have taken place:

1. TIBCO acquired Mashery from Intel, which was perhaps expected, as TIBCO, a well-known player in the integration space, did not really have a strong (or at least popular) API pure-play capability.
2. Red Hat acquired 3scale, which was expected to an extent, as the move was perceived as complementary to Red Hat's Fuse and OpenShift offering, the latter also a recent acquisition.
3. Next was the very surprising acquisition of Apigee by Google, which was considered by many as a sound and strategic move by Google to more rapidly penetrate the enterprise cloud software market.
4. More recent acquisitions started with Oracle acquiring the API-design pure-play Apiary, a move also

considered interesting and strategic, as Oracle had been investing, and continues to invest, heavily in strengthening its **Platform as a Service (PaaS)** offering.

5. The Salesforce.com acquisition of MuleSoft was also broadly expected, as both companies had enjoyed a strong partnership for a few years and the MuleSoft Anypoint offering is also seen as complementary to the Force.com platform.
6. Most recently (at least at the time this chapter was written), there was the highly unexpected acquisition of CA Technologies (also a leader in the API space) by Broadcom, which is traditionally a semi-conductor manufacturer.

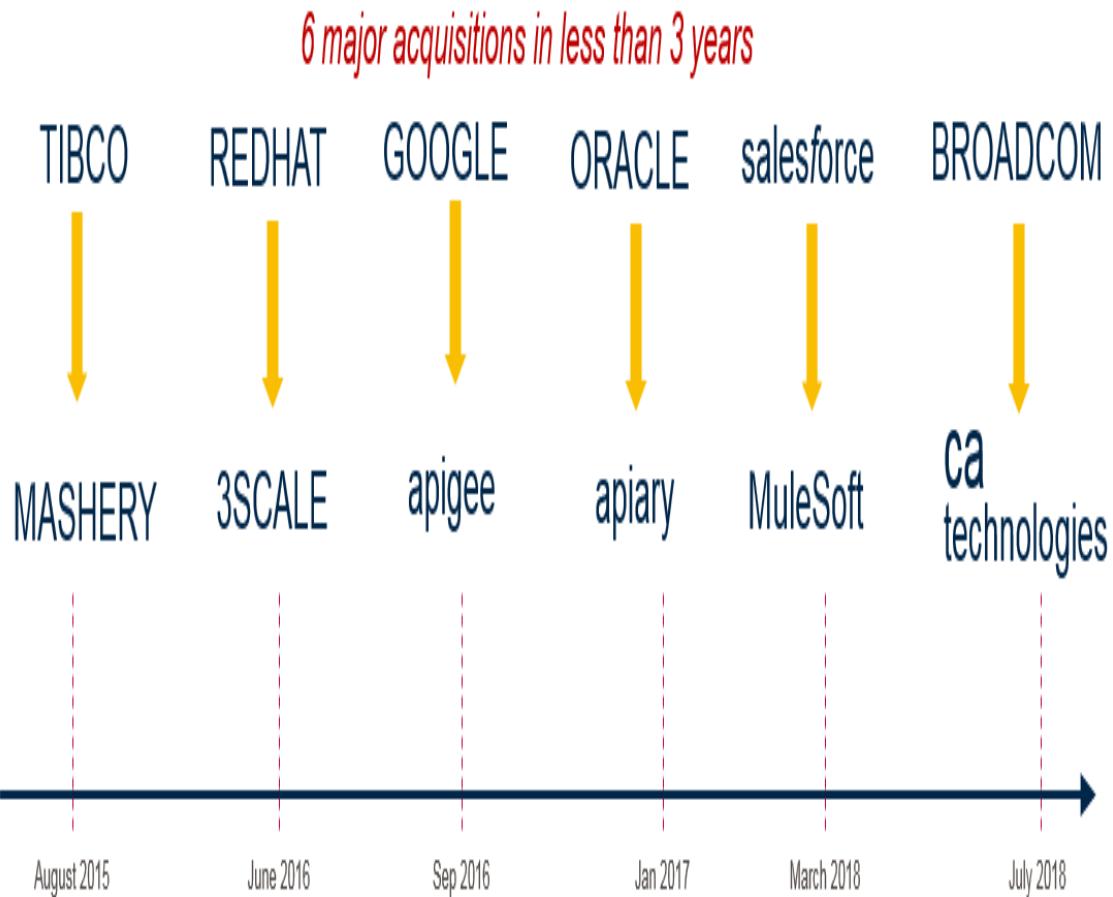


Figure 1.13: Recent acquisitions in the API market

So, what can be deduced from all of these acquisitions? First of all, of the six acquirers mentioned, three are actually major players in the enterprise cloud space. Therefore, their investment in the API space can be seen as a move to strengthen their PaaS portfolios, which is a multi-billion dollar market on its own. Furthermore, when it comes to cloud, APIs are considered the main means to get access to information and functionality electronically, so offering strong API management capability as part of an SaaS, PaaS, or even **Infrastructure as a Service (IaaS)** offering is a clear value add.

Secondly, the acquisitions made by TIBCO, Red Hat, and perhaps even Oracle, can be seen as an indication that the integration market is shifting and that more traditional integration capabilities (traditionally based on large-footprint integration middleware backboxes) are being superseded by API-led architectures, where the integration middleware is either very thin or non-existent (as is the case in Microservices Architectures, where event-driven interoperability is favored).

Lastly, although the acquisition by Broadcom was highly unexpected, the market is no stranger to such moves. The purchase is, in fact, comparable to the one made by Intel in 2013, when Mashery was acquired, in theory to strengthen Intel's play in the IoT. However, it's questionable whether the move paid off, as Intel soon after sold Mashery to TIBCO.

However, this last acquisition raises an important point: APIs being an enabler for the IoT. As devices and machines of all sorts, from wearables, to home appliances, vehicles and industrial machines, to name a few, all become smarter and more capable of storing and processing data, the need and demand to access information in real time can only increase. This means that APIs will also (if not already) be implemented to enable IoT. For companies such as Broadcom, and/or many others in the manufacturing/industrial space, this represents a huge opportunity, as they'll be able to expand their existing offerings to also offer digital services (for example, real-time monitoring and alerting, remote and real-time management of

infrastructure, predictive maintenance and analytics, to name a few).

Summary

This chapter delivered a comprehensive and business-oriented explanation on the value of APIs, and the reasons why they are a must in any digital strategy.

The chapter started by describing why and how digital disruptors are taking the industry by surprise, and the impact this is having on more established and traditional organizations, many of which are struggling to cope with the pace of change, and the level of innovations being introduced.

To this end, the chapter explained the true meaning of disruption and why understanding it is extremely important for successfully creating a digital strategy, and then embarking on a digital transformation journey.

In this same context, it was highlighted that gaining real-time access to an organization's enterprise information assets (many of which are locked in legacy systems) holds the key to success and, without this, a digital strategy's chances of success will be minimal.

The chapter continued by describing and positioning APIs as the means to deliver such access, and thus act as an enabler to digital strategies. It was described in great detail how APIs can

add value to a business, for example, by allowing the business to monetize information assets, comply with new regulations, and also enable innovation by simply providing access to business capabilities previously locked in old systems.

Subsequently, an API value chain was introduced, illustrating a business-centric API maturity model suitable for use as reference when embarking on an API implementation initiative.

The chapter concluded by describing how the software industry is reacting as some of the largest software vendors in the world make major acquisitions in the API space.

In the next chapter, a more technical point of view will be described, which explains how and why the technologies and platforms used to implement APIs have evolved from simple web proxies to third-generation API platforms.

The Evolution of API Platforms

The purpose of this chapter is to go beyond the business value of APIs (explained in detail in [Chapter 1, *The Business Value of APIs*](#)) and walk through the evolution of software architecture as a consequence of digital disruption and cloud adoption. The chapter describes in detail why and how different (middleware) technologies have evolved in order to cope with emerging requirements derived from cloud adoption and digital transformation, for example, the need to access information in real time via APIs, regardless of where data resides (for example, cloud and/or on-premise systems).

The journey of API platforms - from proxies to microgateways

As organizations continue to embrace cloud computing as the means to realize business benefits, for example, TCO reduction, business agility, and digital transformation, an inevitable side effect also takes place: information becomes more and more federated.

The rationale is simple and we will take a look at a typical on-premise system: **Enterprise Resource Planning (ERP)**. A typical ERP system encompasses not one, but several business capabilities (often referred to as modules, for example, finance, HR, SCM, and so on.) all supported by a single infrastructure (a monolith).

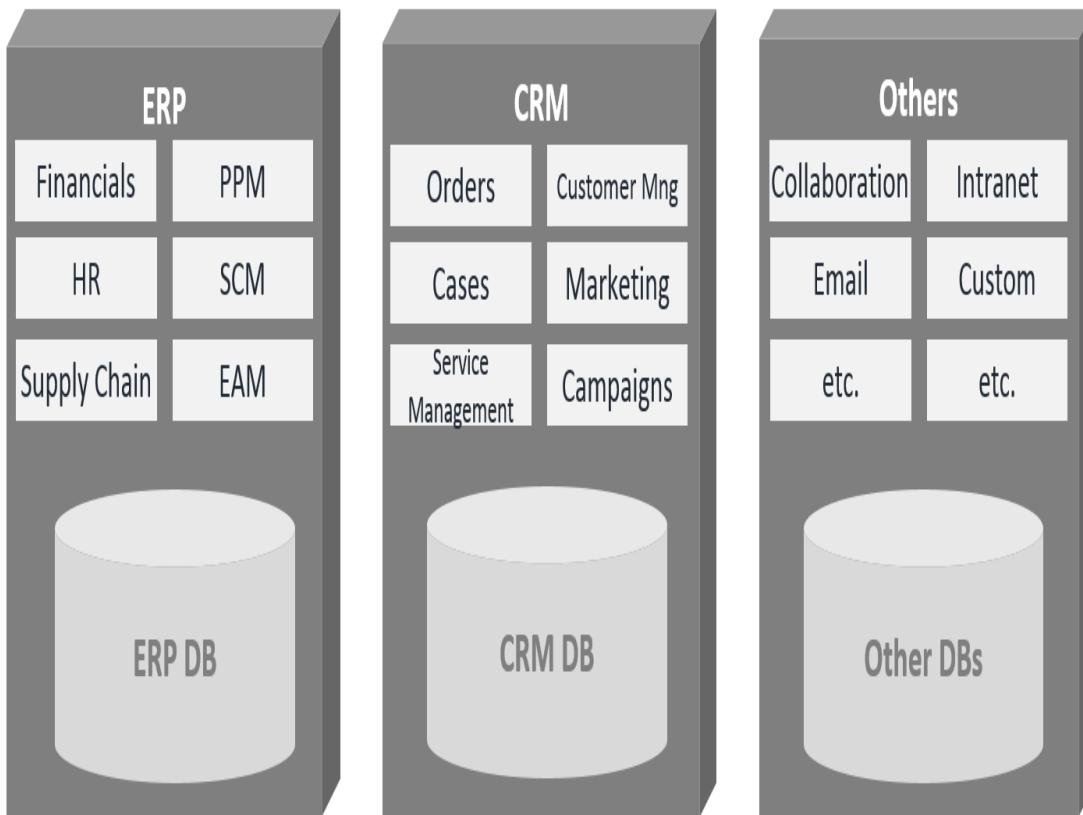


Figure 2.1: Monolithic systems

Now, because of this, all modules within the same monolith are integrated out of the box, mainly because they all share a single database. Therefore, this simplifies (at least a bit) the integration landscape. This also means, though, that if the common infrastructure is affected, all business capabilities will be too (all eggs in one basket). Customizing, extending, patching, and scaling a monolith, therefore, has to be done extremely carefully, as the entire system could be affected, impacting business operations heavily.

When it comes to the cloud (either **Software as a Service (SaaS)**, **Platform as a Service (PaaS)**, or **Infrastructure as a Service (IaaS)**) however, business and technical

capabilities don't have to reside in a single cloud application. In fact, in most cases they don't. Instead, capabilities are scattered across distinctive (smaller) cloud "services," all of which can be implemented individually.

*Please refer to the following document for the official **National Institute of Standards and Technology (NIST)** definition of cloud computing (SaaS, PaaS, and IaaS):*

<http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>

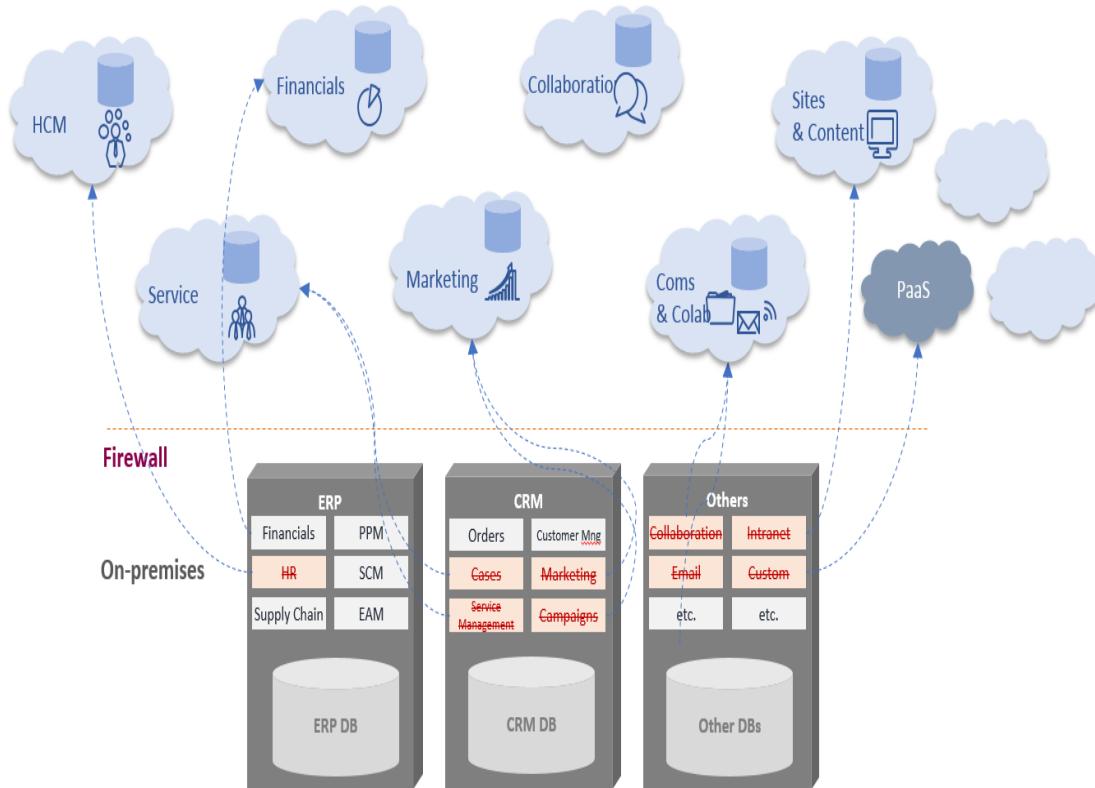


Figure 2.2: Capabilities scattered among cloud services

Given the practical and granular nature of cloud services, hundreds, if not thousands, of cloud vendors (especially in SaaS) have emerged, therefore giving organizations several options to choose from. This has (fortunately or unfortunately, depending from what angle you look at it) led to many organizations adopting **multi-vendor cloud strategies**, in many cases without actually even realizing it, as the adoption is

driven at a departmental/business unit level, and not as a corporate-wide IT initiative.

Recommended reading: The future isn't cloud. It's multi-cloud.

<http://www.networkworld.com/article/3165326/cloud-computing/the-future-isnt-cloud-its-multi-cloud.html>

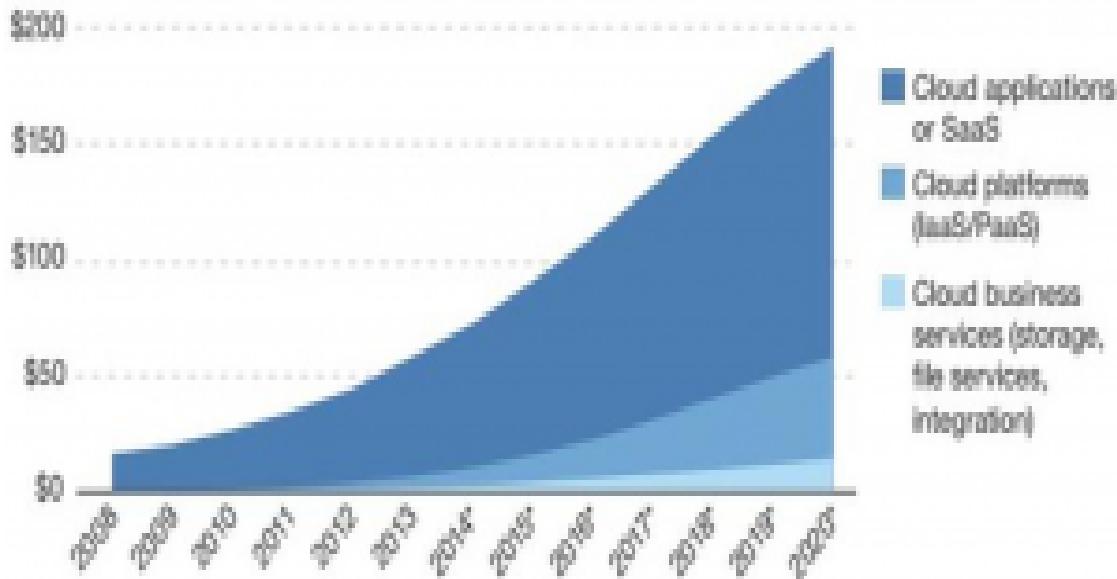


Figure 2.3: Global public cloud growth

Source: <https://www.zdnet.com/article/forrester-public-cloud-market-will-reach-191b-by-2020>

Unavoidably, information assets also become scattered (federated) across different cloud services. The more diverse and distinct an organization's cloud adoption is, the more federated the information becomes.

Moreover, in a highly competitive market, arguably dominated by digital disruptors (as mentioned in [Chapter 1, The Business Value of APIs](#)), such as Amazon, eBay, and Netflix, more traditional organizations are forced to also come up with innovative digital, customer-centric, and multichannel strategies in order to remain relevant and competitive. Needless to say, access to information (now federated) in a

standard, consistent, and secure way, across all digital channels, is a key requirement of any digital strategy.

Organizations that rush into creating multichannel strategies, without first defining a solution to provide access to key information assets, will most likely end up with lots of ad hoc solutions that will not only complicate the architectural landscape, but ultimately prevent the company from realizing the promised goals of the digital strategy.

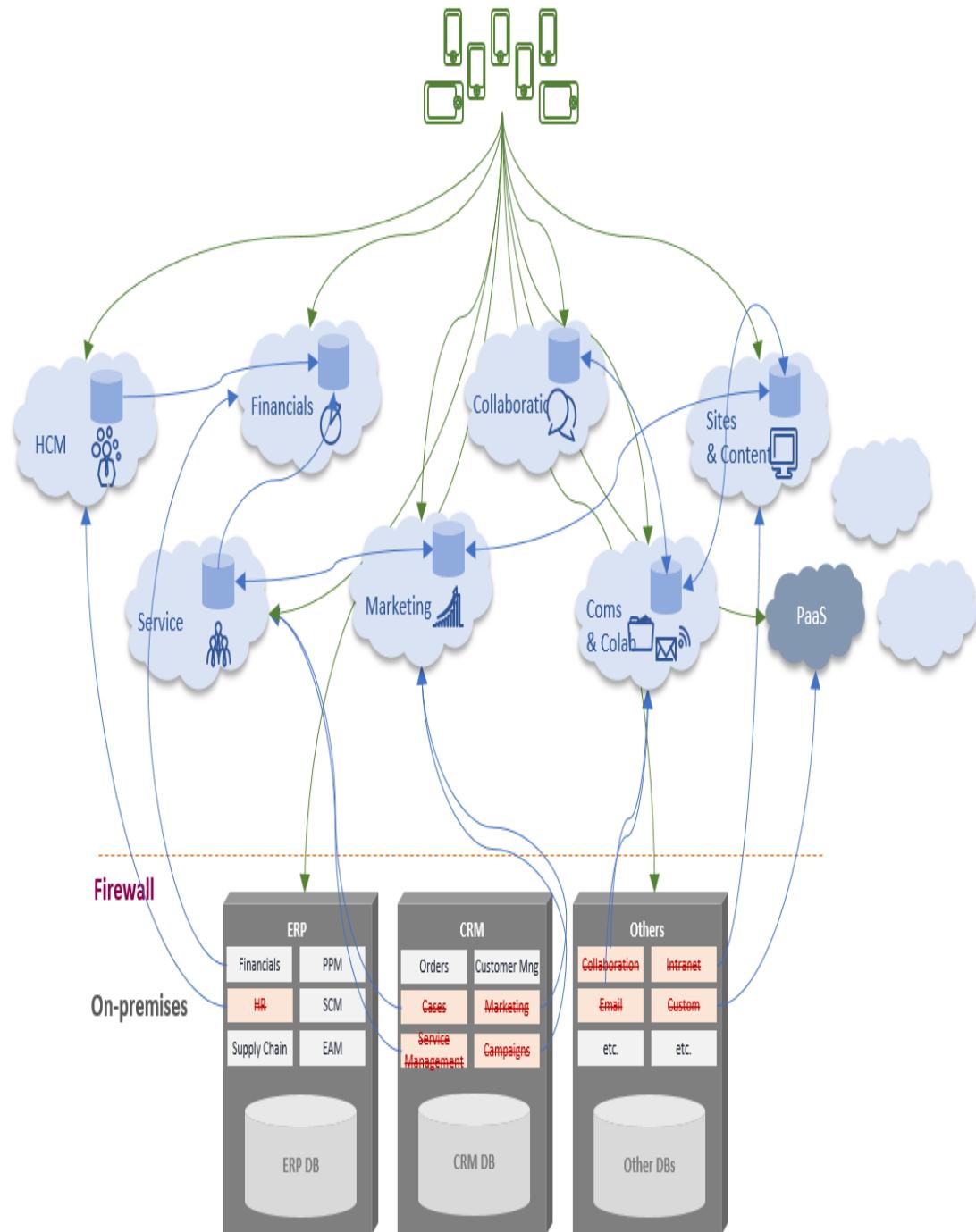


Figure 2.4: Accidental cloud architecture (cloud spaghetti)

In order to address this, a generally accepted approach is to implement a hybrid **Integration Platform as a Service (iPaaS)** solution, capable of providing access to information assets regardless of where they are. The iPaaS platform should

be capable of connecting to any cloud service and/or on-premise system, and delivering access to APIs.

The use of APIs as the means to deliver standard, secured, and real-time access to information enables multichannel applications to consume the assets as and when they need them.

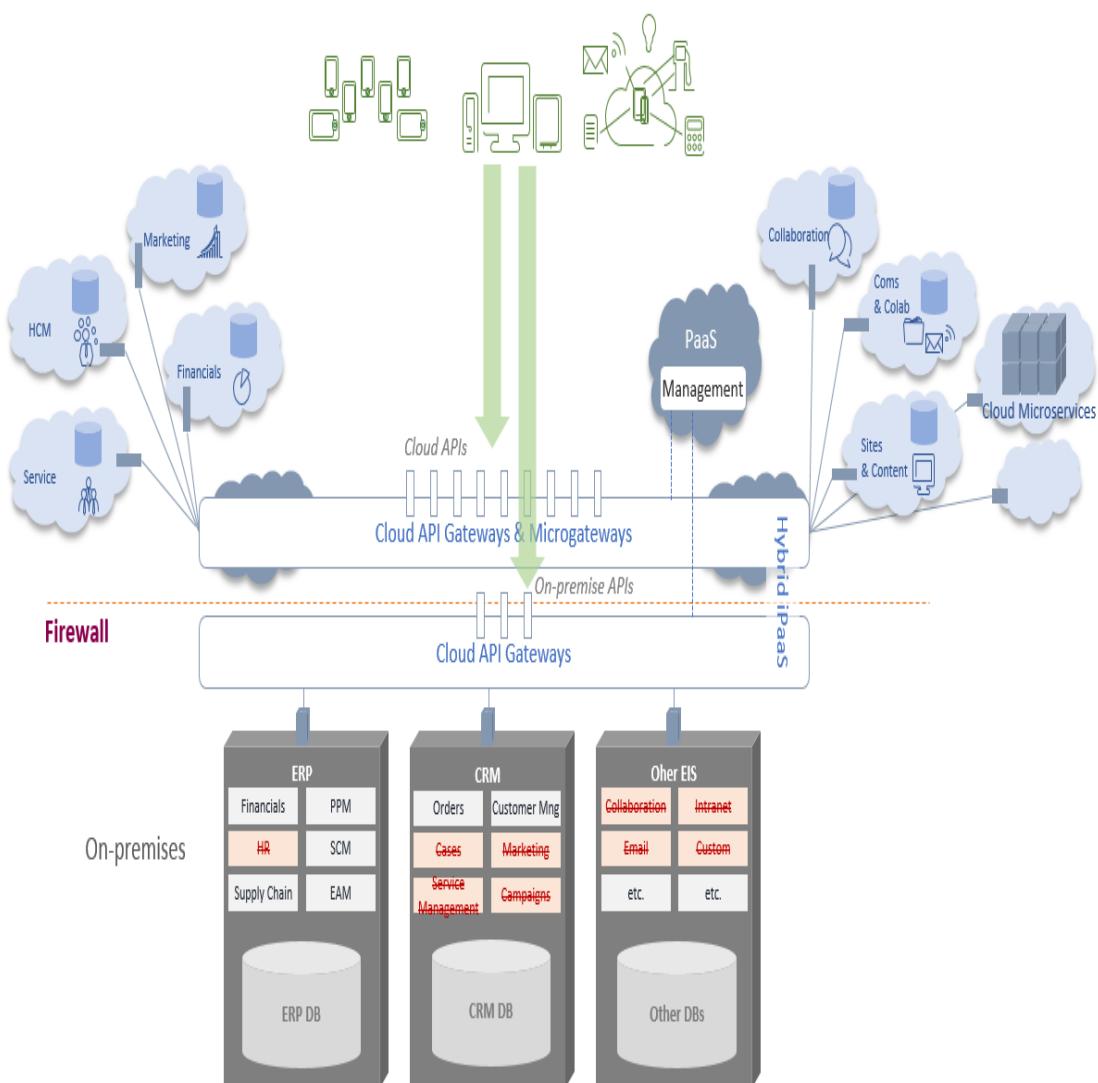


Figure 2.5: iPaaS solution with API management capabilities
Recommended reading: *iPaaS, what is it exactly?*

<http://www.soa4u.co.uk/2017/03/ipaas-what-is-it-exactly-is-it-on.html>

Although this may seem like the obvious answer, the truth is that unless the hybrid iPaaS solution delivers robust **API management** capabilities, it will struggle to address the aforementioned needs. An API has to be as close as possible to the source of information. This not being the case can cause unforeseen issues, such as latency and higher exposure to network problems, and even security threads, such as man-in-the-middle attacks. If information is federated among many different clouds and on-premise applications, so must the APIs be.

To put things into perspective, it is important to understand the main motivations leading to the evolution of (integration) middleware technologies into what this book refers to as third generation.

Generation zero

Remember when the first **Enterprise Service Bus (ESB)** came out? Although the term was first used in 2002, it wasn't until a few years later that their adoption and popularity began, eventually overtaking the proprietary-based **Enterprise Application Integration (EAI)** solutions.

Recommended reading: Enterprise service bus history. https://en.wikipedia.org/wiki/Enterprise_service_bus#History

One of the key reasons that ESBs became so popular is because of their relation to **service-oriented architectures (SOAs)** and the view that implementing an ESB was fundamental to realizing SOA.

The ability of ESBs to adopt open standards, such as **web service standards (WS-*)**, and act as integration hubs capable of connecting to multiple systems and exposing **Simple Object Access Protocol (SOAP)** web services, differentiated them from traditional **Enterprise Integration Architecture (EIA)** solutions.

For a full list of WS- you may refer to the following link:*
<http://servicetechspecs.com/ws>

Additionally, for a simple definition of SOAP web services, refer to:
<http://servicetechspecs.com/xml/soap2>

During this period, if a web service had to be accessed from outside the internal networks, typically **web proxies** would be implemented in **Demilitarized Zones (DMZs)**, to proxy the HTTP traffic to the ESB, and also implement transport security (HTTPS). Web proxies, however, offered very basic capabilities.

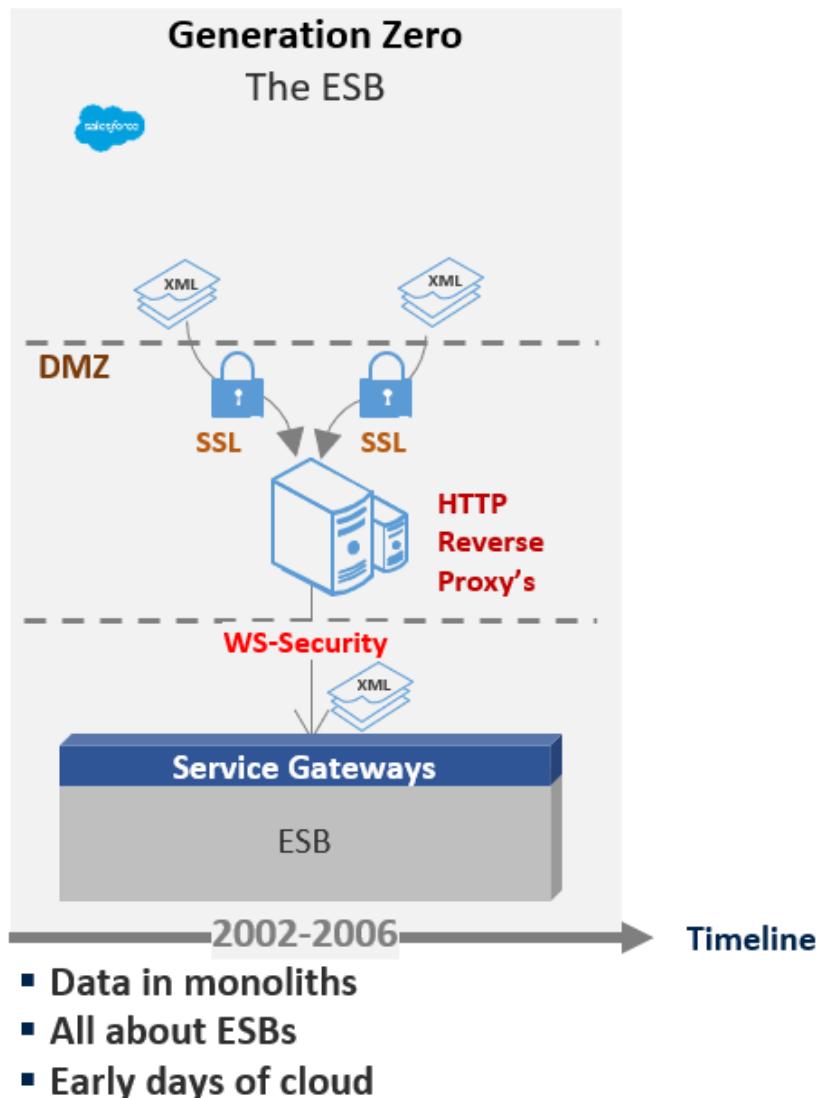


Figure 2.6: Generation zero – it all starts with ESBs

ESBs offered many capabilities, of which it is worth highlighting **basic security, message routing, data transformation, and protocol translation**, along with

adapters to connect to multiple backend system using different protocols (for example, SQLNET, HTTP, FTP, and SMTP). ESBs also allowed exposing functionality and access to information as standard SOAP web services. ESBs were able to receive HTTP/SOAP requests, transform the message payloads, perform message validations, and then route the calls to a given backend in the required protocol.

During this period, most ESB implementations were pretty straightforward. As the following diagram suggests, the amount of business logic implemented in an ESB was limited and constrained by the previously mentioned capabilities. The majority (if not all) of the business logic (for example, orchestration, content validation, business rules, and so on) resided in the client side or the backend system that the ESB was connecting to.

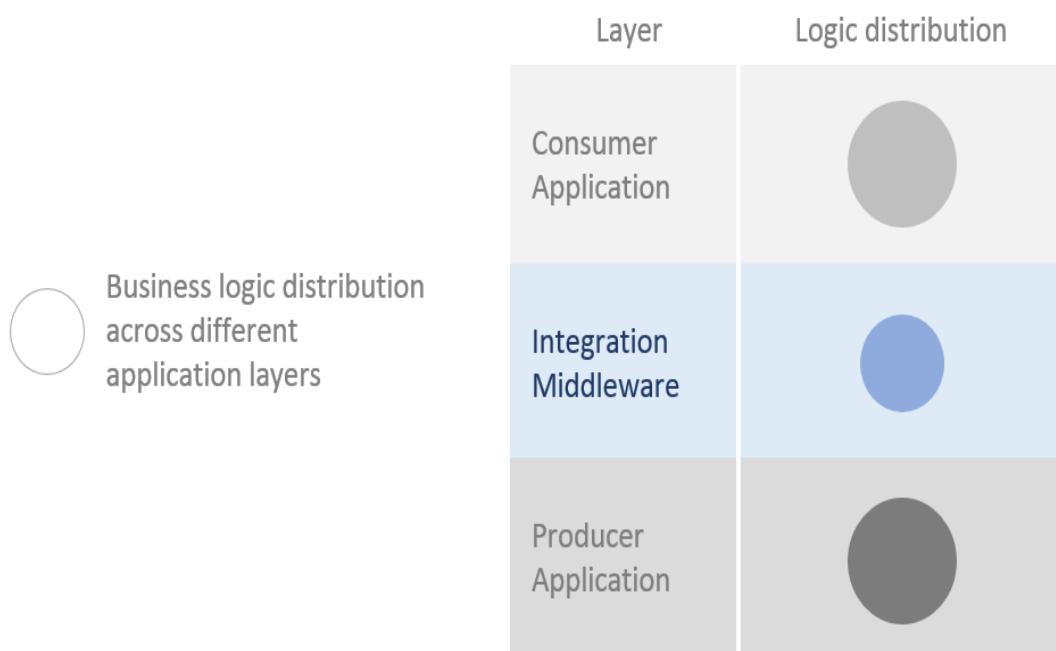


Figure 2.7: Logic distribution in generation zero

At a time when the industry lacked open standards for integration and the majority of products implemented proprietary protocols, ESBs were widely used.

First generation

As SOAs became more prevalent in enterprises and ESB technologies continued to mature, several new capabilities were also added to ESBs, mainly in support of SOAs. For example, the adoption of **Service Component Architecture (SCA)** as a standard and the introduction of gateways as a more robust capability, to securely expose web services to external networks.

For details on SCA refer to: <http://www.oasis-open.org/sca>

Gateways manifested themselves as either **XML accelerators** (running as black-box appliances) or add-ons to existing SOA/ESB infrastructure (commonly known as **service gateways**).

XML accelerators were ideal for DMZs because of their robust capabilities to secure SOAP web services and protect against external threads, such as the ones listed in the **OWASP Top Ten Project**. This made these appliances perfect as a form of first-line defense.

Recommended reading on the OWASP Top Ten Project: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Service gateways, on the other hand, were well-suited to securing services internally (second-line defense) and supported the implementation of standards such as WS-Security, WS-Trust, and WS-Policy.

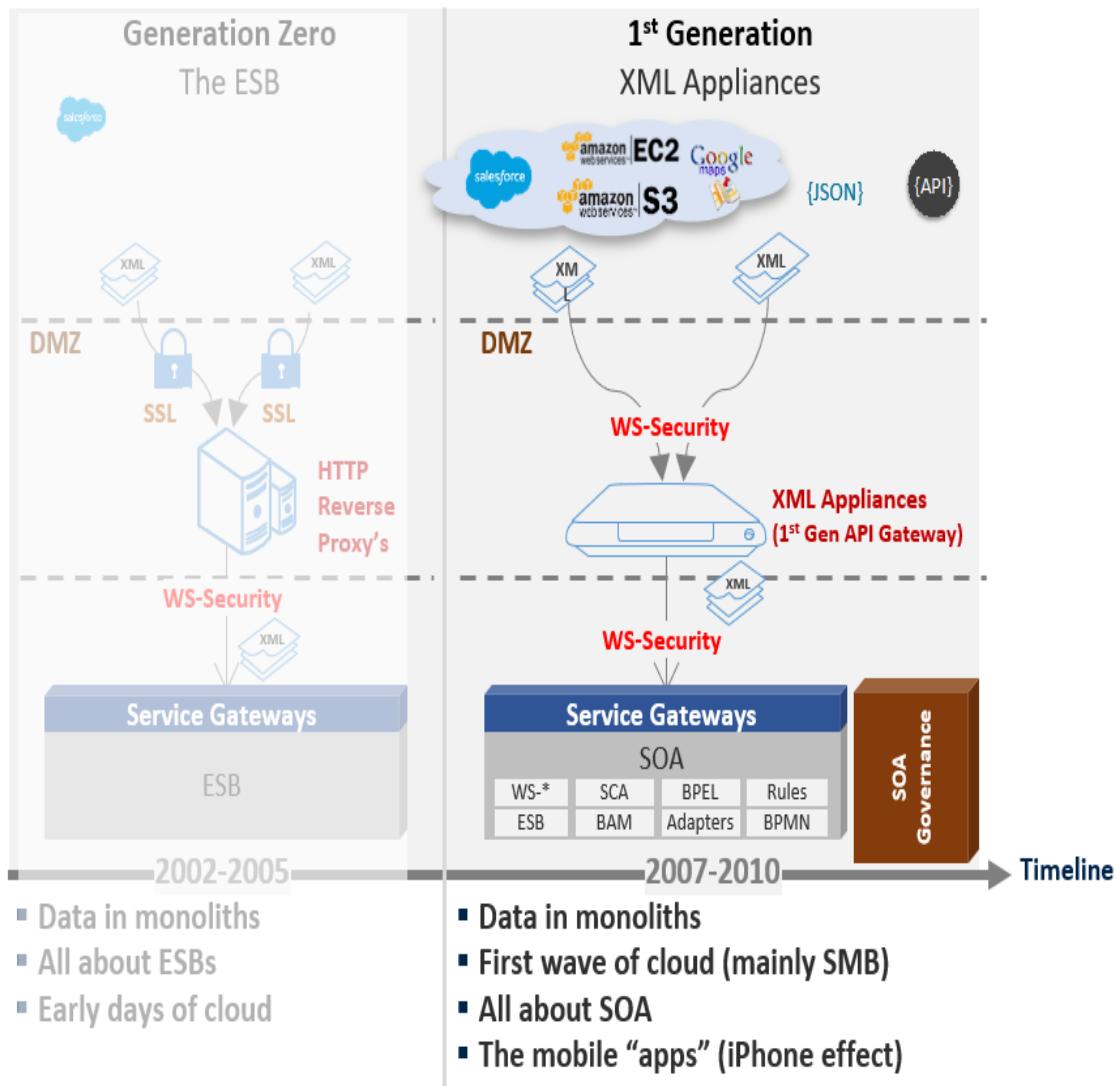


Figure 2.8: First-generation XML appliances

SCA introduced the concept of **composite applications**. In a nutshell, a composite application is a piece of software that assembled multiple web services together into a single deployment unit, in order to deliver a specific business

solution, which was also exposed as a web service. Composite applications ran in specific integration middleware that extended the ESB. Although SCA is a standard, in practice, each software vendor implemented its own flavor of it.

Although, as a concept, adopting composite applications seemed like a sound idea, in practice, the implementation of business logic, either as complex **business process execution language (BPEL)** orchestrations, as **human workflows**, or even as **business rules**, became common. It goes without saying that the footprint of underlying integration middleware platforms increased exponentially, in order to cope with the amount of processing logic introduced in the composite applications.

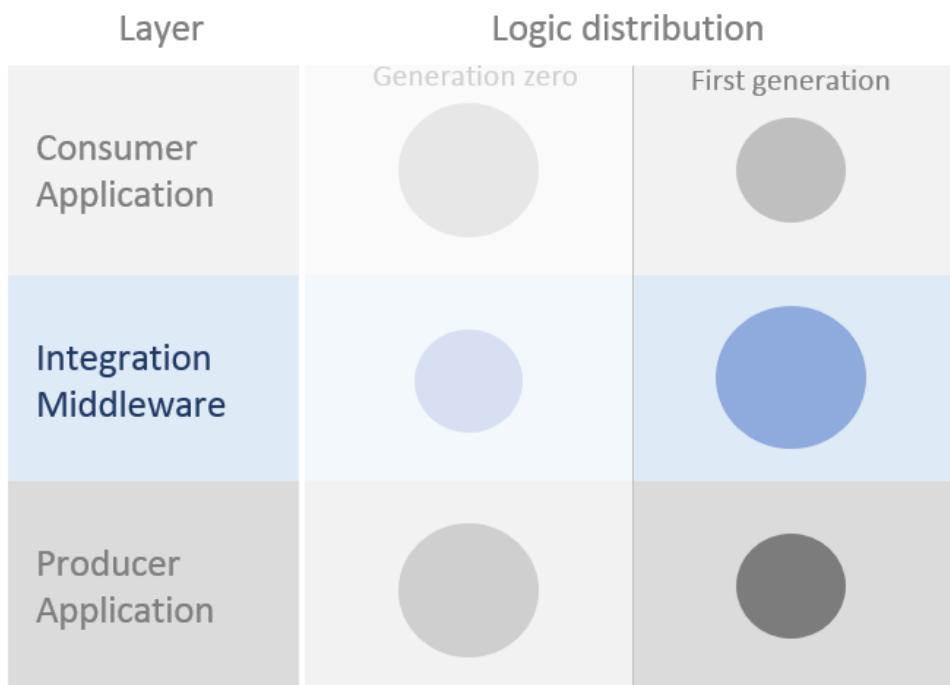


Figure 2.9: Logic distribution in first generation

As the number of web services increased, as well as the underlying technology footprints, so did the amount of people required to keep the initiative going. As the complexity and cost of SOA solutions continued to increase, **SOA governance** emerged as "the" discipline that could bring back control and maximize the chances of success, by aligning people, process, and tools toward commons goals, typically centered around key business benefits. Backed by specialized software, SOA governance became a big trend and thousands of organizations worldwide attempted its adoption.

For further information on SOA governance, concepts, and principles please refer to:

<http://www.soa4u.co.uk/2013/11/oracle-soa-governance-for-business.html>

Towards the end of the period, an inflection point occurred that completely changed the ball game in the entire IT industry. The first wave of cloud computing, combined with the launch of smart devices, such as the **iPod** and soon after the **iPhone**, completely disrupted the market and the IT industry. Thousands of mobile **apps** could now be easily found and installed via incorporated app stores, creating a huge marketplace, which is worth billions of dollars.

Now, because the mobile apps ran natively inside the mobile device, a new form of remote access to information and/or functionality available in backend systems was required. It had to be simple, lightweight, and secure given the constraints in terms of compute capacity of such devices.

Broadly based on the **Representational State Transfer (REST)** architectural style, a new flavor of APIs emerged.

Recommended reading: History of APIs.

<http://history.apievangelist.com/>

These APIs (referred to as either REST or web APIs) offered a much simpler and lightweight alternative to SOAP-based web services, especially when combined with JavaScript objects based on JSON to handle data payloads.

JavaScript Object Notation (JSON) is a *lightweight data-interchange format. It is easy for humans to read and write, and easy for machines to parse and generate. It is based on a subset of the JavaScript programming language. For further information, refer to:* <http://www.json.org/>

In no time, REST APIs became the main and most popular mechanism to implement backend code, and deliver remote access to information and functionality.

Second generation

As the number of smartphones rocketed, so did the number of mobile apps. Organizations of all sizes – many of which, at this point, were now rushing to have a mobile presence – had to quickly come up with solutions to deliver the so-called "APIs" and therefore give mobile apps access to information, either locked in backend systems and/or only accessible via SOAP web services.

It is worth noting that most organizations, having already made considerable investments in the adoption of traditional SOA solutions, understandably were (and many still are) keen to leverage their existing capabilities (not just in terms of technology but also in terms of people) in order to also satisfy these emerging requirements.

The reality was that, at the time, the vast majority of traditional SOA middleware, although very rich in capabilities to handle XML/SOAP-based payloads, lacked basic capabilities to handle REST/JSON services.

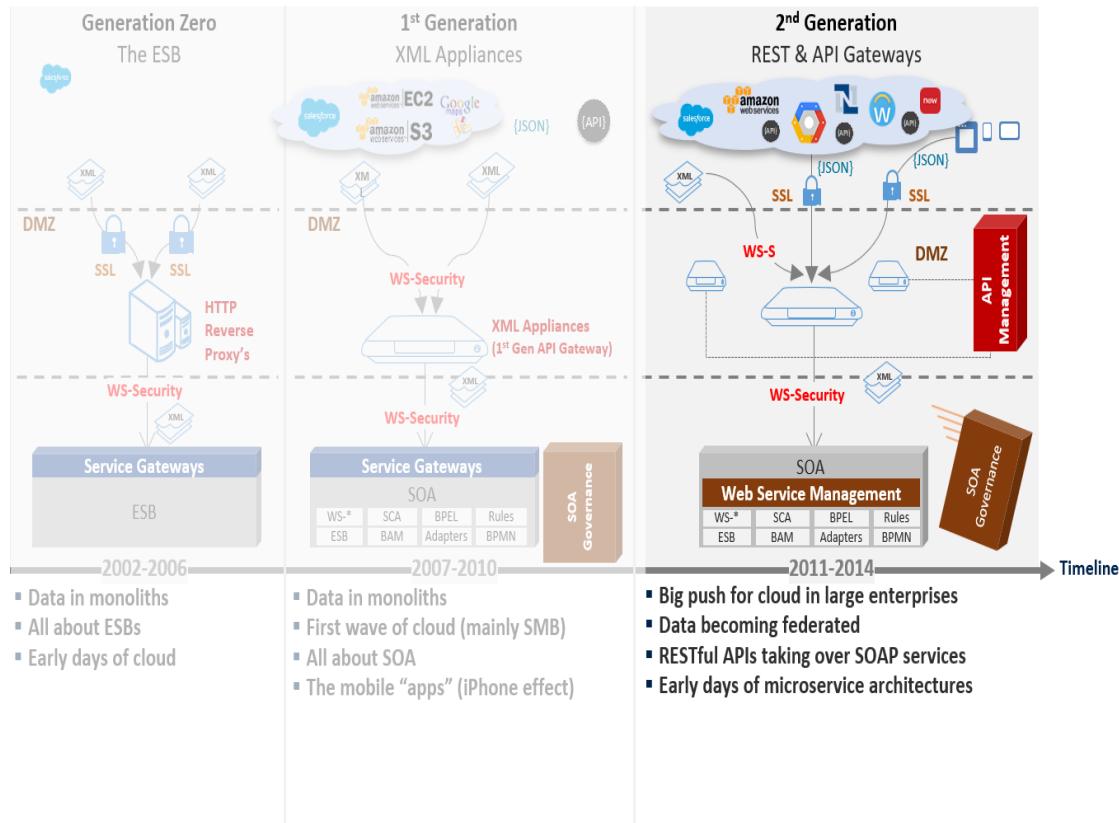


Figure 2.10: Second-generation API management is born

Another important difference that started to emerge was around governance. For mobile app developers, speed was the main factor. Their approach to governance (if any) was lightweight. Emphasis was given to adopting techniques to produce code quickly and encouraging developers to collaborate among themselves, as opposed to introducing heavy processes requiring a lot of policing in order to ensure that standards were adhered to.

SOA governance, on the other hand, backed by specialized (and expensive) software that was difficult to implement, didn't really stand to its promise, inevitably leading to criticism

industry-wide. At this point, SOA governance, both as a discipline and specialized software, was deemed a failure.

Recommended reading: SOA is Dead; Long Live Services.

<https://web.archive.org/web/20170823183641/http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html>

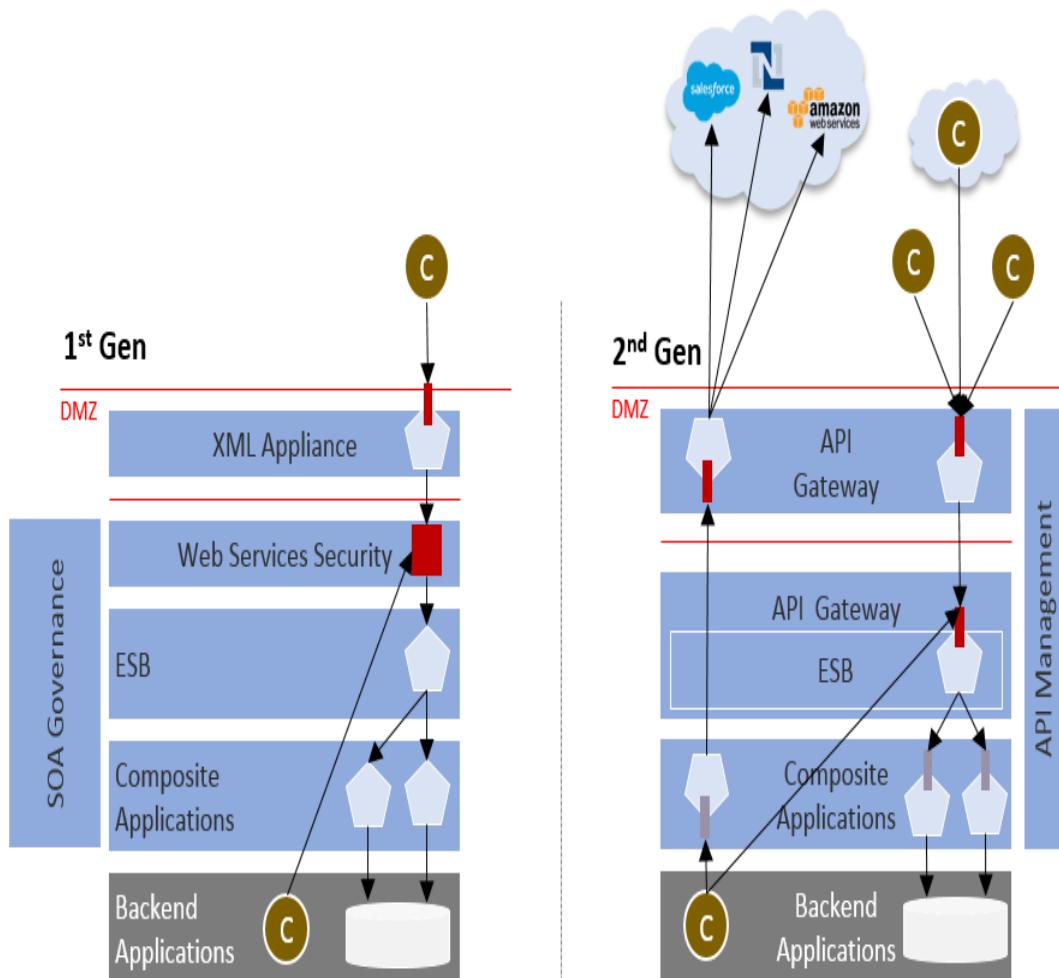
As the industry's interest in API-related capabilities increased, software vendors in response rapidly adapted/enhanced their traditional SOA stacks (for example, ESBs) to add RESTful/JSON processing capabilities.

Furthermore, as API management started to develop as a discipline, to manage APIs across their full life cycle, new technical capabilities were required in order to make this task a lot simpler. To this end, many SOA governance vendors also quickly adapted and/or simplified heavily their offerings, in order to make them suitable to manage APIs.

Some vendors went beyond adapting their SOA stacks to even changing their brand names to reflect this change of direction:

<https://sdtimes.com/akana-soa-software-changes-name-akana/>

The adaptation of first-generation ESBs, XML gateway appliances, and SOA governance tooling in support of API-specific capabilities, including their management, is referred to as **second-generation API platforms**. Now, because of this, second-generation API platforms can be easily identified, as API capabilities tend to be just add-ons on top of the vendor's existing ESB and/or XML gateway offering.



■ Web service policy enforcement point

■ Managed API: endpoint with policies applied

■ Unmanaged API: endpoint with no policies applied

Semi-decoupled service: stateless or statefull. Implements orchestration, transformation, rules, and other forms of business logic

C Consumers: any system, application or mobile device that consumes a web service or REST API

Diagram inspired from omesa.io

Figure 2.11: First- and second-generation API platforms' architectures compared

Another point worth highlighting is the definition of services as **semi-decoupled**. According to the diagram, a service is where business-logic-related capabilities, such as orchestration and business rules, are implemented. APIs, on the other hand, are the interface of such a service (which could be in any protocol, for example, SOAP or REST) and where policies, such

as authentication and authorization, are applied. In first and second generation, APIs and services are coupled as one thing and deployed into the same stack. An API and a service tend to be a single deployment unit.

For further information on the semi-decoupled service definition, refer to the Open Modern Software Architecture Project (OMESA):

omesa.io

At this point, the tendency to implement business logic across the different layers of the integration middleware continued. Multiple reasons can be blamed for this. Sometimes, it was because it was simpler to just use the integration layer as a sort of "hammer for all nails," and other times, it was because of lack of best practice and architecture governance. Note that during this period, the word "governance" was sort of prohibited, given the bad reputation that SOA governance had earned.

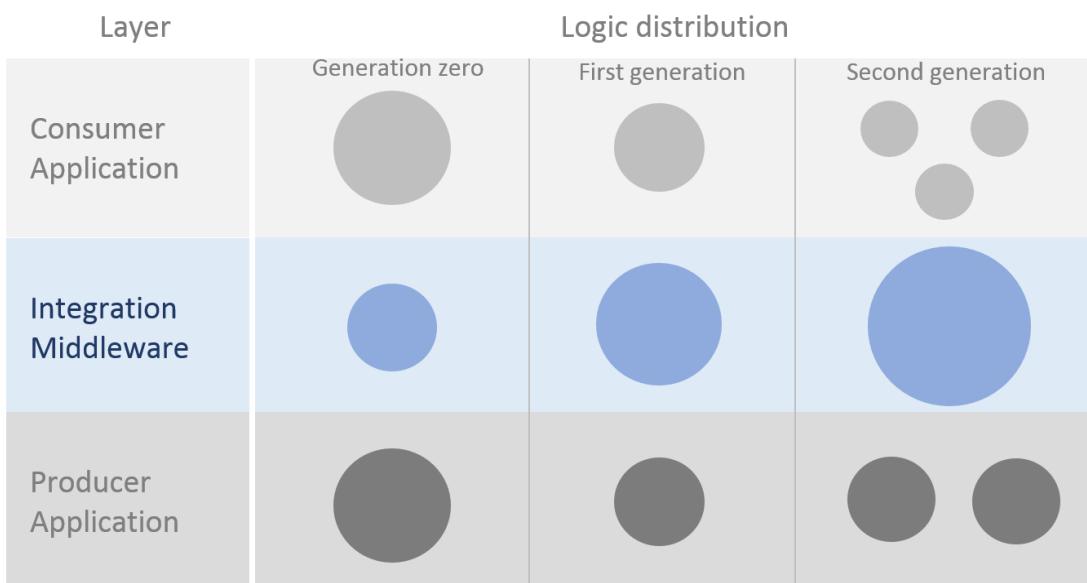


Figure 2.12: Logic distribution in second generation

This tendency of integration stacks becoming thicker and thicker was being heavily criticized by the then rapidly emerging communities of developers promoting **microservice architectures**. Challenges in horizontally scaling integration platforms, complex inter-dependencies when releasing software into production, a lack of flexibility when selecting technologies; and last but not least, a common practice of making the middleware fat by implementing business logic in the integrations were some of the most notable criticisms raised.

Recommended reading: Microservices, a definition of this new architectural term with emphasis on section Smart endpoints and dumb pipes:

<https://martinfowler.com/articles/microservices.html>

Application Services Governance

Towards the end of this period, Gartner came up with the concept of **Application Services Governance**. Gartner's view was that API management would eventually become part of SOA governance. The combination of the two is what Gartner named Application Services Governance.

Further details on Application Services Governance is available in the following link:

<https://www.akana.com/solutions/application-services-governance>

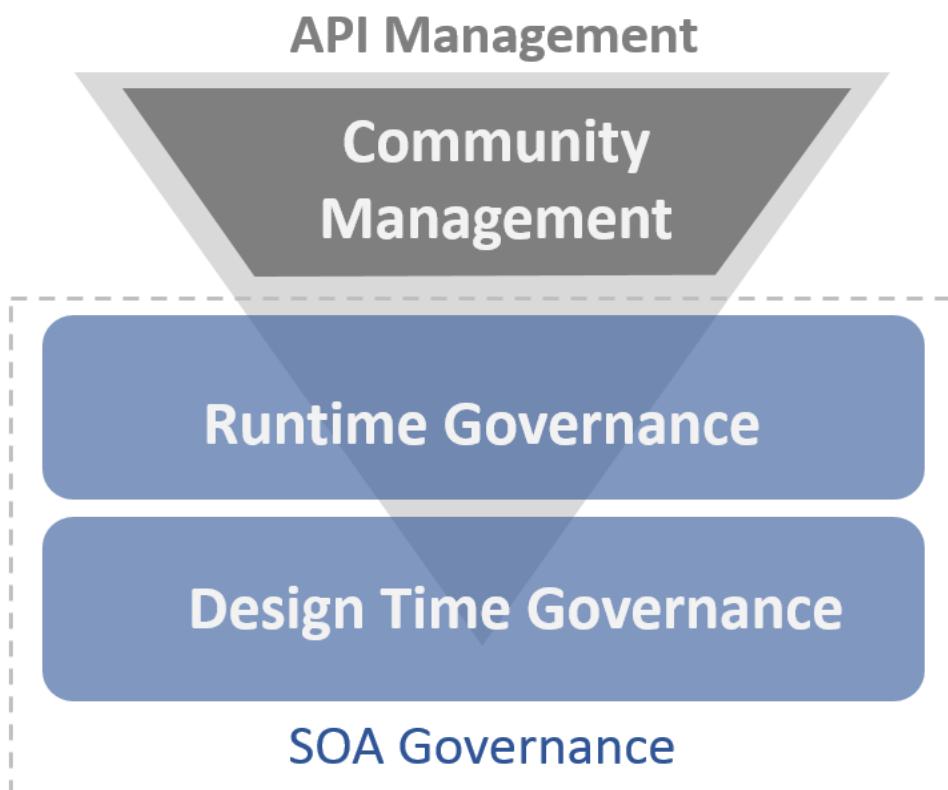


Figure 2.13: Gartner's Application Services Governance

In practice, instead of API management and SOA combining, the traditional way of realizing SOA was, as mentioned earlier, heavily challenged by the emerging communities promoting the microservices architectural style. These communities did not just put into question the use of traditional (monolithic) SOA stacks (for example, ESBs) but, broadly speaking, also regarded their use as a bad practice.

Third generation

What we see today, in the majority of organizations worldwide, is a big push to adopt cloud, achieve digital transformation, and also become customer-centric. Businesses are taking serious steps in order to achieve these goals. At this point, it starts to become evident that (monolithic) second-generation API platforms aren't suitable for, or capable of, delivering the capabilities required to achieve such goals.

To elaborate further, the following is an explanation of what these goals actually mean to IT and why/how they relate to API platforms.

Cloud adoption

Cloud adoption means moving on-premise applications into the cloud (IaaS, PaaS, or SaaS) or simply building applications directly into and for the cloud (a term known as cloud-native). As mentioned earlier, some of the drivers could be lowering operations costs, but also gaining more flexibility and agility. To this end, most organizations have taken (or are taking) a best-of-breed (multi-vendor) approach to cloud, as opposed to moving all of their applications to a single cloud provider.

Cloud adoption manifests itself in three ways:

- **Workload migration:** Lifting and shifting an on-premise workload (for example, databases, Java applications, or packaged applications) into an IaaS or PaaS cloud.
- **Cloud transformation:** Adopting one or many SaaS applications as a replacement for an on-premise one. It also means using cloud-native capabilities (typically PaaS) to extend the SaaS application when applicable. In this case, there is no lift-shift, but rather data migration and integration.
- **Cloud reengineering:** When a monolithic system is rewritten from scratch in the cloud using cloud-native

capabilities (typically in PaaS).

Digital transformation

In plain English, digital transformation means enabling the business to offer its products and services through as many digital channels as applicable (web, mobile apps, kiosks, partner online stores, bots, and so on). However, in order to do so, access to up-to-date information in real time (information which now happens to be federated across multiple cloud data centers and/or on-premise systems) becomes absolutely critical. Without this access, a digital strategy will simply not succeed.

For example, a basic requirement for organizations undergoing digital transformation is mobility. Mobility means many things, but for some organizations it could be just giving co-workers the ability to execute business processes while on the move – via multiple devices. For this to work, access to multiple systems of records via APIs must be in place.

Another key requirement that arises in digital transformations is the need to give businesses the agility and speed for new products and services, to be taken to market quicker and cheaper, but also the ability to fail fast, and fail cheaply, so new concepts can be tried without spending millions.

As Adrian Cockcroft from Netflix said, "Speed wins in the marketplace."

<https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/>

However, the majority of systems (especially monolithic ones) aren't suited to handling the load (or unpredictable peaks) that accessing information in real time demands. Also, changing these systems is complex, time-consuming, and risky.

This is where microservice architectures become so compelling and this is one of the reasons why they have become so popular. In short, they offer an approach to both software engineering and software architecture that enables the (end-to-end) implementation of business capabilities in a fully decoupled manner. This is not only in terms of the software development life cycle, but also in terms of technology, as each microservice is completely independent at runtime, and implements mechanisms to decouple itself from other systems and/or microservices that it may need to interact with. However, the proliferation of microservices also means that information becomes even more federated and is at a more granular level.

Recommended reading: Microservices and SOA.

<https://www.slideshare.net/capgemini/microservices-and-soa>

Customer-centricity

This means collecting, consolidating, and analyzing information about a customer's brand interactions/behavior, interests, purchasing patterns and history, personal details, and others, in order to create personalized, rich, and positive experiences for them. In turn, the expectation is that by delivering better and more tailored experiences, customer loyalty will increase and so will sales.

Although the concept is easy to understand, achieving it is a different story. This is because in the majority of organizations, customer information doesn't reside in one system but is scattered among several systems (many of which are legacy), which can be internal, external, or, often, belonging to business partners.

Common denominators

In order to provide the capabilities needed to achieve the aforementioned goals, a more sophisticated API platform (a third generation) is required. It must be one that:

- Allows the implementation of **APIs everywhere** (any cloud and/or on-premise), yet without introducing an operations nightmare and huge costs.
- Empowers communities of developers by letting them discover and subscribe to APIs via a **self-service developer portal**.
- Gives developers the tools they need to rapidly design and create APIs that are well-documented and easy to consume – **API first**.
- Gives information owners **full visibility and control** over their information, by letting them decide who by and how their information assets, exposed via APIs, are accessed.
- Delivers **strong security** to protect information assets against all major threats (for example, OWASP Top Ten).

- Is **lightweight, appliance-less/ESB-less** and suitable for microservice architectures.
- Is **elastic**, meaning that gateways can:
 - Scale in or out without manual intervention.
 - Integrate with registries to dynamically determine active service endpoints.
- Is **centrally managed**, regardless of the number of gateways, APIs, and their location.
- Enables **meaningful collection and use of statistics**, so operations data can be used to gain business insight and not just for monitoring and troubleshooting purposes.
- Is **consumption-based**, typically with no CPU-based licensing.

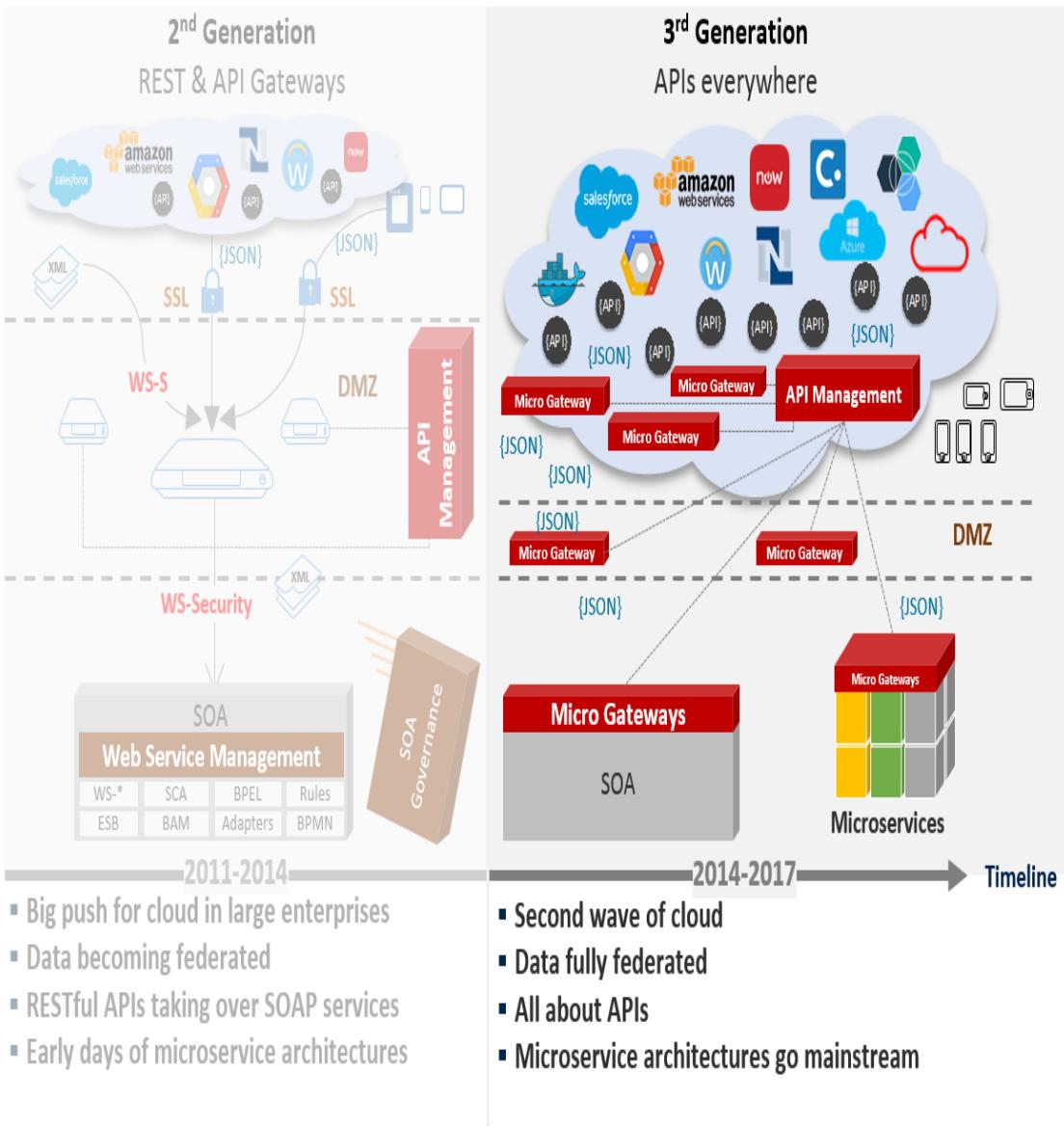


Figure 2.14: Third-generation APIs are everywhere

As the monoliths are broken down into smaller pieces and reimplemented as discrete cloud applications, either in SaaS or PaaS, the business logic and information contained in such monoliths also gets distributed. The tendency of integration middleware to become bigger and bigger seems to be reversing, almost like a big bubble that bursts into many smaller ones.

Recommended reading: An Ode to Middleware.

<http://www.openlegacy.com/blog/an-ode-to-middleware/>

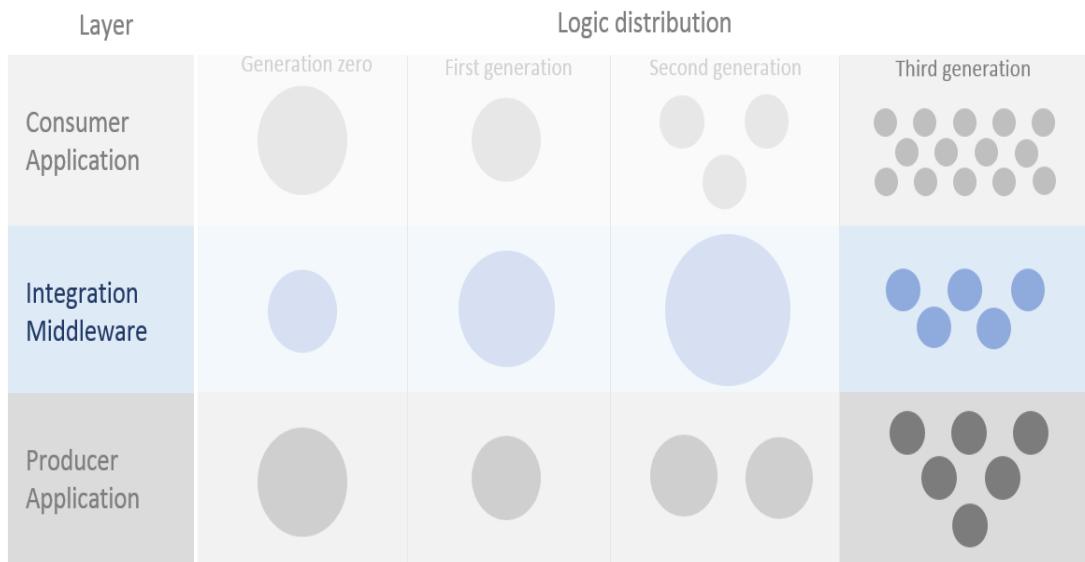


Figure 2.15: Logic distribution in third generation

Third-generation API platforms truly mark an inflection point for software architecture. Unlike their predecessors, because of the federated nature of such platforms, it is difficult to depict them in architectural layers. This is better appreciated by looking at the following diagram.

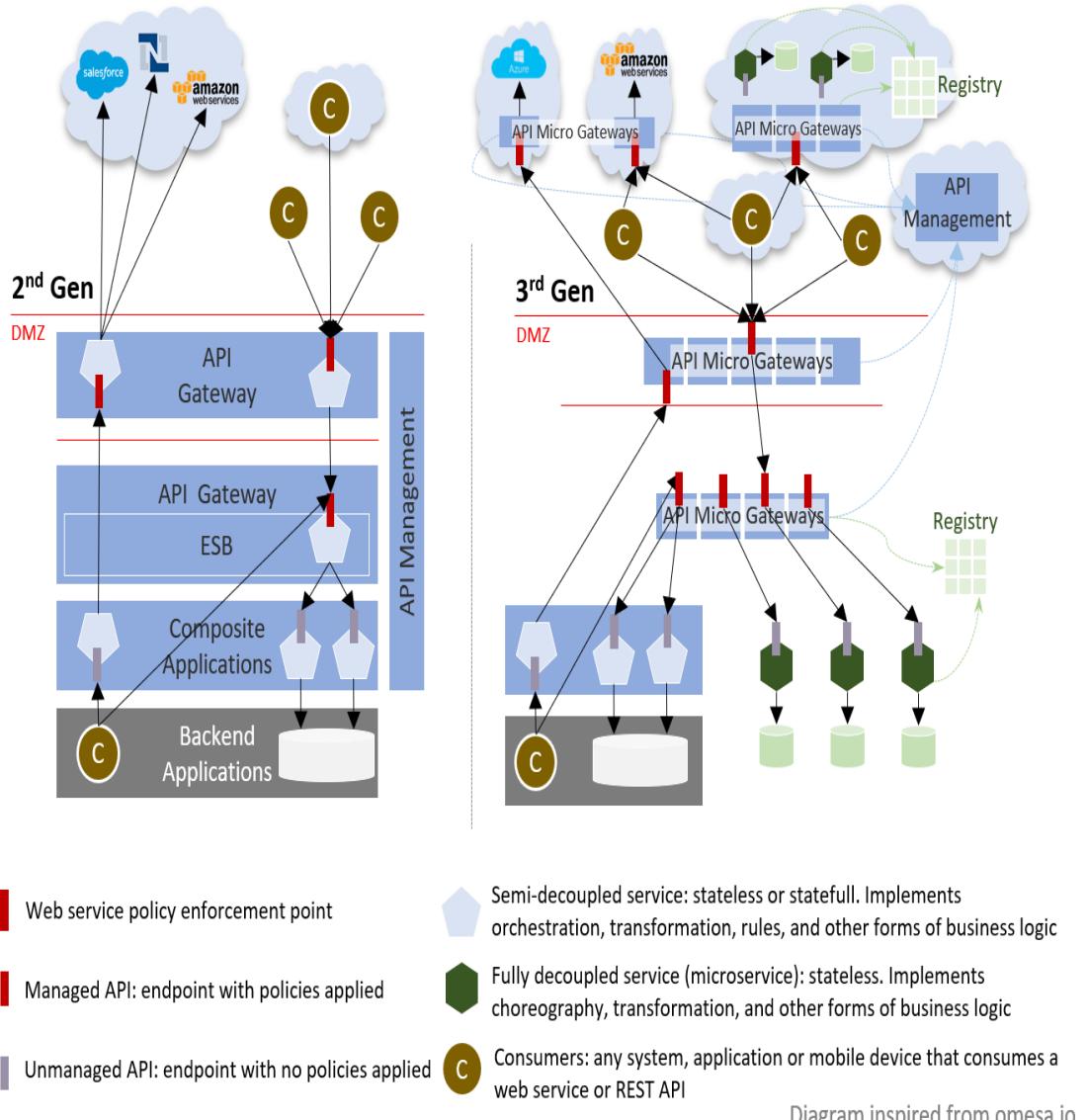


Figure 2.16: Second- and third-generation API platforms' architectures compared

Although further architectural details will be covered in subsequent chapters, there are some fundamental characteristics that set the third generation apart from previous generations:

- The management of APIs is fully decoupled from the service implementation itself; hence, by design, there is a separation of concerns.

At this point, it is very important to apprehend how to distinguish an API from a service, as although they complement each other, they are also distinct

A service is an independently deployable software unit (an application) that encapsulates business logic and can be accessed via a standard interface, for example, via a REST or SOAP endpoint. Services can be fully decoupled (a microservice) or semi-decoupled (implemented in a common integration stack).

A service endpoint(s), managed through an API platform is referred to as a managed API.

Therefore, a service endpoint that is not managed through an API platform is an unmanaged API.

To avoid confusion, this book refers to managed APIs as simply APIs.

- APIs, just like the information itself, are also federated. Therefore, APIs are implemented as close as possible to the source of information, regardless of where it resides (cloud and/or on-premise). Not doing so means that APIs could be exposed to latency and other network problems, including increased exposure to security threats.
- Even with APIs being federated, full control and visibility over who can access/when they can access the APIs is possible.
- APIs are discoverable via a developer portal. Through the portal, developers can search and subscribe to the APIs to which their role allows them to.
- The entire API platform operations are centralized and cloud-based. Therefore, this allows administrators to

deploy APIs to multiple locations from a central location and with little effort.

- APIs are managed via lightweight (meaning small footprint), independently deployable, and scalable API gateways that can run anywhere. Therefore, these are not ordinary gateways. They can handle extremely large volumes, as they run on highly scalable platforms that support asynchronous, non-blocking I/O threading models, for example, NGINX, Vertx, Netty, Grizzly, Node.js/Express, to name a few.

*A term that is becoming increasingly popular when referring to this specific type of gateways is **API microgateways**.*

In summary, API are:

- Non-monolithic, appliance-less, and ESB-less. They should be lightweight and very easy to implement (anywhere) – ideally using containers.
- Self-sufficient and should be responsible for retrieving APIs, policies, and even system patches from a central management unit.
- Stateless: no state should be managed for any transaction.
- Capable of rapidly scaling in and out dynamically, without manual intervention.

- Limited in their functionality by only delivering capabilities expected of a gateway, therefore preventing them from becoming fat, as experienced in second generation.

Summary

This chapter explained in detail how requirements derived from cloud adoption, digital transformation, and customer-centricity all demand a new set of capabilities that can't be simply satisfied by the adaptation of monolithic integration stacks, such as ESBs and XML appliances. To this end, the chapter walked through the evolution of API platforms, from generation zero to the third generation, in a chronological way, and also highlighted key events in the industry that notably contributed to this progression.

In summary, third-generation API platforms are all about delivering capabilities that facilitate the adoption of modern architectures, which in turn enables businesses to innovate and deliver new products, and offerings, quicker and cheaper, therefore helping them to remain relevant in a highly digitalized world.

For digital giants, such as Google, LinkedIn, or Amazon, this is old news. However, for the majority of organizations worldwide, the journey to cloud, digital transformation, and customer-centricity is just getting started. For such organizations, the content of this chapter will be useful, as it can, at the very least, serve as inspiration when defining new integration strategies.

The chapter also heavily emphasized the tendency of information to become more and more federated. This is a pattern that is only just getting started. With analysts consistently predicting that by 2020 the number of connected devices will reach up to 21 billion, the **Internet of Things (IoT)** has huge potential. However, devices also require and produce a vast amount of information.

For further information refer to Gartner Reveals Top Predictions for IT Organizations and Users in 2017 and Beyond:

<http://www.gartner.com/newsroom/id/3482117>

Without a doubt, businesses of all sorts will make use of the IoT to engage even closer with their customers, and completely change the way that customer interactions occur. A fourth-generation platform, one that goes beyond cloud and on-premise data centers, will be required so that access to billions of connected devices is provided but also managed.

The next chapter will deep dive into the architectural implications and the capabilities required in order to implement modern, third-generation API platforms enterprise-wide.

Business-Led API Strategy

This chapter builds on [Chapter 1, The Business Value of APIs](#), and [Chapter 2, The Evolution of API Platforms](#), by providing an approach for creating API strategies that are business-driven and have the main target of delivering direct or indirect business value.

I illustrate the activities that should be undertaken to successfully kick-start an API initiative. Next, the chapter explores in detail how business needs can be translated into drivers, and from them derive goals and objectives that aim toward the realization of business benefits. It also sets the scene for the rest of the book by presenting a framework in the form of a train map that illustrates all of the activities required in order to successfully deliver a business-led API strategy in the form of train stops and lines.

This chapter will be particularly useful for organizations that wish to kick-start an API initiative but don't know where to start, and thus are looking for sources of inspiration. Likewise, the chapter will also be useful to organizations with on-going API initiatives that wish to review the approach taken to ensure that such initiatives have greater chances of success.

Kick-starting a business-led API initiative

For an API initiative to be successful and long-lasting, it goes without saying that business benefits must be targeted and realized sooner rather than later. However, doing so is easier said than done and far too often, the adoption of not just APIs, but technology in general results in expensive IT or business endeavors that never get to see the light of day. Some obvious reasons for such a fate include:

- Solely IT-driven initiatives that aim to solve a technology problem and lack business context, requirements, and buy-in. They can't be justified against business metrics.
- Solely business-driven technology initiatives that lack adequate input from IT and thus can be very badly scoped, estimated, and planned.
- Initiatives that engage with end users way too late (or not at all). Even though the project might be successfully delivered, it fails to meet the end users' expectations (for example, an API that developers struggle to understand and use) and thus there is a huge amount of resistance to shifting to the new solution.

- Poor quality data. If the data offered by the API is either outdated or (worse) incorrect, consumers of the API will deem it as poor, regardless of how well designed or easy to use an API is.
- Lack of consensus, dependencies and/or stakeholder management, which can ultimately halt the entire initiative or dramatically delay it, causing a loss of momentum.
- Let's not forget the usual suspects: bad scope and project management, wrong methodology, and lack of testing, to name just a few.

API initiatives are, in fact, even more exposed to the aforesaid challenges, as APIs don't really have a business-user frontend that can be seen and touched; instead, the assets produced in an API initiative (for example, APIs themselves, but also their underpinning platform and developer portals) are technical in nature and the consumers (customers) of such assets will be application developers. For this reason, it is extremely important to have a clear understanding of what it actually means for an API initiative to be successful. For example, from a business standpoint, success would probably mean things like:

- New business applications with rich user interfaces that leverage the APIs created, and that expose information

and functionality that perhaps was inaccessible before (for example, locked in legacy systems).

- Decoupling of backend systems from user interfaces, therefore reducing the impact of change and enabling the modernization and/or upgrades of systems.
- More business transactions being processed quicker and with fewer technical issues.
- Existing and/or new partners are able to better do business-to-business transactions as a consequence of easy-to-use APIs.
- An increase in the partner ecosystems through partner APIs, resulting in an indirect increase in market share.
- The holy grail of APIs: information assets being monetized, thus resulting in revenue generation.

The truth is, unless it is well understood what success actually means for the business, the chances are that an API initiative (or any IT initiative, for that matter) will just fail to meet expectations.

To prevent this from happening, an API initiative should not be initiated without first understanding what is in it for the business; in other words, what is the business problem being solved, what business benefits will be delivered, and how will they be measured?

At this point, it's important to understand that APIs are the means to reaching a goal and not the goal itself. If an API solution is not driven by the business and is therefore not aimed at solving a business problem, business value can't be measured, and the chances are, the initiative will be perceived as just another expensive IT-led project and will likely be considered a failure. Alternatively, it will never actually start.

Therefore, before any work is spent on defining API reference architectures, or worse, selecting products to deliver API management, it is best to start thinking about:

- The **business drivers** for implementing an API solution. For example, should you enable some sort of omni-channel strategy, or perhaps just provide access to information locked in legacy systems or other information assets? Should you consider the monetization of information assets via APIs?
- If an API initiative is to be kick-started, what would be the **goals** and **objectives**? In other words, what are the broad ambitions and what can be realistically achieved as first steps?
- Just as importantly, what is the strategy to accomplish the set goals and objectives? In other words, how would the strategy be delivered? What would be the approach? How much effort and funding would be required? Who would need to be involved? How long would it take?
- How success will be measured; for example, what business **key performance indicators (KPIs)** can be used to track success?

The following is a more detailed explanation on how to overcome each of these points.

Defining the business drivers

It is not always clear how an API initiative (which from the outset seems quite technical) can be related to the business, let alone what/how business benefits can be realized and measured. Though the API value chain, described in [Chapter 1](#), *The Business Value of APIs*, helps by illustrating how different types of APIs within the chain incrementally deliver more business value from bottom (connectivity) to top (revenue generation), what it does not describe is how to determine the business requirements that ultimately drive the need for the APIs.

To do this, an understanding of the technology alone won't be enough; rather an understanding of the business domain, the business problem(s) being solved, and how the business will perceive and measure success become imperative.

The closer an API relates to the actual business problem being solved, the higher the perceived business value will be. For example, an API that provides backend connectivity to a key system of records will likely be used by many business applications. This is certainly important, as it enables access to key information assets perhaps not previously accessible. However, the perceived business value will likely be against the applications using the APIs and not the APIs themselves. This

is because in business terms, it is the application using the APIs that solves a business problem, which in turn delivers value that can be measured against specific KPIs. In this example, APIs will be considered an IT enabler.

KPIs are measurable values that show how effectively a business is performing against its goals within a specific time frame. KPIs exist for many areas of the business. For example, in sales, an obvious KPI can be monthly sales growth or monthly net new customers. In marketing, for example, a KPI could be increased monthly traffic in a digital channel (for example, the company's website). In project management, the KPI could be the percentage of projects completed on budget.

Therefore, in order to ensure that an API initiative is not seen as merely an IT enabler, but rather as a business-driven initiative, it is crucial to identify where and why APIs can be applied as the main means to satisfying business-driven requirements, especially those that can have clear KPIs associated with them. By spotting such requirements and deriving from them their business drivers, it will be a lot easier and more effective to articulate the business benefits behind APIs.

The following table illustrates a few examples whereby APIs are adopted as the main means to addressing different use cases. The table also indicates the potential business drivers behind the creation of APIs and the benefits that could be expected.

Business driver	Use case	API fitness	Business benefit

<p>Allow new business applications to be implemented quicker by facilitating access to key reference data.</p>	<p>A new application requires reference data from another system(s) on a periodic basis.</p>	<p>APIs created to deliver the access to reference data.</p>	<p>Benefits can be realized the same API is reused by many applications thus reducing the number of integration required to access the same reference data. Additionally abstraction through APIs can reduce the impact of change.</p> <p>Indirect benefits can be measured by calculating costs saved by not having to develop multiple applications.</p>
--	--	--	--

			to build multiple point-to-point integration or having to rework end-to-end solutions.
There is a need to speed up the realization of the digital strategy to ensure the business remains competitive and relevant in the digital marketplace.	The business can't wait until a legacy modernization strategy is delivered in order to digitalize key processes, products, and offerings. The business desires to start digitalization as soon as possible while the	APIs as the means to gain access to key information assets locked in the legacy systems and required by the digital solutions.	Benefits can be calculated indirectly as a result of improved KPIs, resulting from the digitalization itself. For example, customers now can access their account online (which was previously

	<p>modernization of legacy systems occurs in parallel.</p>		<p>paper-base and even create order online, resulting in increased customer satisfaction and sales.</p>
	<p>This could also be derived from an organization's marketing strategy towards being perceived as a modern and digital company.</p>	<p>APIs designed in such a way that they enable a smooth transition from legacy to the target architecture.</p>	
The quicker a SaaS application can be	<p>The business has decided to move some on-premise</p>	<p>The cost/effort of migration and integration</p>	<p>The complexity and cost of migrating</p>

<p>adopted, the sooner the total cost of ownership (TCO) can be reduced and other benefits realized.</p>	<p>applications to cloud SaaS applications. However, there are many concerns around data migration and especially integration; for example, how will integration be supported knowing that many systems will remain on-premise?</p>	<p>will be considerably lower if the SaaS application comes with well-documented and matured APIs. Therefore, SaaS applications that come with such APIs should be favored over those that don't.</p>	<p>data and integrating with the Sa application should be lower than traditional on-premise systems, so benefits can be calculated by comparison.</p>
<p>There is a desire to optimize business processes and thus gain</p>	<p>A new mobile application is being created that will allow staff to do certain business tasks</p>	<p>APIs created to access information assets and to implement common business logic</p>	<p>Any benefit resulting from introducing enterprise mobility could be</p>

<p>efficiencies as a direct result of introducing enterprise mobility.</p>	<p>while on the move; for example, sales reps being able to access orders and place orders via a mobile app.</p>	<p>(functionality) required by the mobile application.</p>	<p>indirectly linked to the APIs built in support of the mobile app for example, an increase in sales and customer satisfaction as a result of sales reps using the mobile app</p>
<p>The business realizes that the market is shifting more and more towards digital channels. Thus, the business sets an ambitious</p>	<p>As part of a new digital strategy, the organization wishes to start offering products and services through new digital channels and even beyond</p>	<p>APIs are required to consistently offer access to all functionality and information that is to be accessible through these new digital</p>	<p>Any benefit resulting from the adoption of the digital strategy could be indirectly linked to the APIs built in support of it. Now, because APIs are th</p>

<p>digital strategy to shift products and offerings to the digital marketplace.</p>	<p>traditional mobile apps, such as chatbots (for example, Facebook Messenger or WhatsApp), and virtual assistants (for example, Siri, Alexa, or Echo).</p>	<p>channels. Most likely, new APIs will be required in order to deliver a true omnichannel experience.</p>	<p>main and most important enabler for the strategy it could be said that without the use of APIs achieving such benefit could not have been possible.</p>
<p>The business feels that the time and cost to create new custom digital solutions is too high. There is a desire to find ways to reduce the</p>	<p>An organization wishes to reduce the cost of custom development by adopting commercial off-the-shelf (COTS) functionality, while</p>	<p>Publicly available APIs are adopted as the means to rapidly implement new backend functionality without having to build the solution from</p>	<p>The cost of developing backend functionality is reduced. Likewise, speed to market is increased by not having to build new functionality.</p>

<p>cost of custom development and decrease the time to market.</p>	<p>remaining in control of the user experience (the frontend/user interfaces).</p>	<p>scratch.</p>	<p>from scratch. Also, innovation perceived as increased, as new features that would've been otherwise near impossible implement are now incorporated to deliver better user experiences</p>
<p>The business realizes that the most valuable asset in the digital economy is information,</p>	<p>An organization wishes to offer certain products and services completely electronically</p>	<p>APIs are created as the main means to offer such products and services electronically.</p>	<p>APIs are the main means of enabling such new capabilities electronically. Benefits realized can</p>

and therefore there is a wish to monetize it.	and in the simplest possible way.	<p>Communities of developers can gain access to the API via a self-service developer portal. The portal provides the means to register, instructions on how to make use of different API features, and payment plans.</p>	<p>be measured directly as a result of adopting APIs.</p> <p>At this point APIs become a business product in their own right and business KPIs could even be defined against them such as the number of API calls per month, number of orders processed, and so on.</p>
The business has to	The business complies with	APIs act as the main	Prevention not just

<p>comply with new regulation.</p>	<p>regulation such as the General Data Protection Regulation (GDPR), or the second Payment Service Directive 2 (PSD2) introducing new requirements to organizations on how data and functionality is accessed.</p>	<p>means not just to access information assets and functionality, but also to introduce fine-grained controls over who and how (and even which) information assets (especially customer-related) are accessed.</p>	<p>substantial fines as a result of non-compliance but also the brand impact this may carry.</p>
------------------------------------	--	--	--

Because of the generic nature of the preceding mentioned drivers, the recommendation is not to use them as is, but rather

to use them as inspiration when identifying and creating drivers that are contextualized to an organization and also well founded. Failing to do this may result in lack of buy-in.

It's important to acknowledge that not all companies are Netflix or Amazon. The vast majority of organizations worldwide still rely on many legacy systems and outdated processes to run their businesses. As described in Chapter 1, The Business Value of APIs, businesses know that in order to remain relevant, not only do they have to modernize their IT landscape but most importantly, the way they do business. APIs are perhaps the most important business enabler to adopting digital strategies, therefore identifying relevant business drivers and use cases should really not be like finding needles in a haystack. The preceding table only aims to make this process even simpler.

Defining the goals and objectives

Once it becomes clearer what some of the main driving forces are behind the definition of an API strategy, the next step is to put together goals and objectives that can ensure the initiative remains on track and doesn't derail as it progresses.

The idea is that it should be routine to, at any point in time, take a step back and verify that goals and objectives are (or are on the right path to) being met, and if not, something somewhere could've gone wrong and a review of the initiative should take place in order to prevent further derail.

At this point, it's important to understand the difference between goals and objectives in this context. Whereas goals define primary outcomes that the initiative should focus on, they tend to be broad and thus difficult to measure. Objectives, on the other hand, are more specific steps targeting the delivery of the goals, and because they are more specific, they can be measured.

Looking at the previously mentioned business drivers, we can therefore say that the main goals of the API initiative is to deliver an API foundation capable of:

- Providing access to core information assets currently locked in the organization's many legacy systems via APIs.

- Introducing APIs as the in-support of a legacy modernization transition architecture to smoothly move from the old architecture to the target architecture.
- Enabling the adoption of SaaS or COTS by using APIs as the main means to migrate data and integrate with cloud applications.
- Adopting APIs to enable the creation of enriched user experiences beyond the traditional mobile apps into areas such as bots and virtual assistants.
- Delivering APIs that are discoverable and easy to use, thus reducing the time to market and complexity (hence costs).
- Allowing the use of public APIs to accelerate the speed of implementing new and innovative functionality.
- Monetizing the organization's information and functional assets by allowing products and services to be offered electronically as APIs.
- Helping to comply with regulation such as GDPR or PSD2, to name a couple, by introducing better controls and visibility over how and who can access customer-related information.

As it can be interpreted, the goals are very broad, though they do relate to previously mentioned business needs, and therefore in a real-world scenario, getting consensus from both

business and IT stakeholders should be feasible and in principle not a huge undertaking.

Objectives, on the other hand, have to be more specific and as opposed to goals, they must be measurable and bounded to a timeframe and thus can be trickier to define. For this, it is important to have enough context and understanding of what the business is trying to accomplish and what other initiatives have been kick-started in support of this. Then it is a matter of collecting enough data points that can help to define some realistic objectives that are specific enough that they can be measured. Ideally, such data should already be available if enough understanding was developed during the process of defining the business drivers.

Following are some example objectives, taking into account that many questions can't yet be answered.

The short-term objective (next eight to 12 weeks) for the API initiative is to deliver an API strategy that defines:

- The main business drivers, functional requirements, and use cases for APIs across all major business units.
- A catalog of APIs required to deliver the different requirements and use cases identified.
- A target API reference architecture including:
 - API architectural principles.

- Core building blocks of the solution.
 - API taxonomies.
 - API integration patterns.
 - API platform capabilities required.
-
- An API design-first software development life cycle for the creation of APIs.
 - An automated and continuous process for the delivery of APIs.
 - The technology of choice (vendor and/or open source) to be implemented in support of the reference architecture.
 - Delivers a proof of concept for an API based on real requirements and implemented using a sub-set of the technologies chosen.
 - A new operating model including how to transition from the current organization to the target operating model.
 - A business value-driven implementation roadmap illustrating:
 - What business benefits are expected and by when.

- What platform capabilities will be delivered and when.
- What APIs will be delivered and when.
- Key dependencies and when they should be delivered.

Notice from the above that all objectives have two main characteristics: first of all, they are all bound to a specific time frame (eight to 12 weeks). Secondly, they are specific enough so that proper actions and deliverables can be derived. This is important so that progress can be tracked.

Having said that, it can also be noticed that the objectives don't commit to delivering any specific business benefit. This is because at these early stages, there won't be enough information to commit to specifics. Rather, one of the objectives is in fact to come up with such an understanding and provide a clear view as to what business benefits will be delivered, by when, and what APIs have to be delivered to achieve this.

Defining the API strategy

A strategy defines the approach and steps to be taken in order to deliver the set objectives. In other words, whereas the objectives define the *what*, the strategy defines the *how*. The scope of the strategy should go well beyond technology and also cover architectural (for example, overarching architectural principles, reference architectures, and capability models), organizational (people, roles, and responsibilities) and process-related (software development life cycle, continuous delivery, and so on.) concerns. It should also aim to align with other initiatives and identify key dependencies that may alter the timelines and/or order of delivery.

As with everything, there are no silver bullets, so there are no exceptions here. However, steps can be taken in order to prevent common pitfalls from happening, most notably:

- Lack of business context, such as a proper understanding of functional needs and use cases. APIs should be seen by the business as products, meaning they serve a purpose, solve a problem, and deliver business value. Therefore, delivering an API without founded business context is risky and may result in failure from a business standpoint.

- Focusing too early on API vendor solutions, rather than identifying what technical capabilities are really required and which are more important.
- Introducing new technologies without first understanding what's already available.
- Lack of buy-in from important stakeholders by not getting them involved early and considering their feedback.
- Forgetting to solve the organizational problem and focusing just on technology.
- Setting wrong expectations by promising outcomes to the business without properly understanding scope and costs.

Although the list is by no means complete, it does provide a good indication of what symptoms can be expected should the creation of an API strategy de-rail.

In order to avoid such pitfalls, the following train map illustrates an approach to delivering an API strategy. The map illustrates the main phases of the strategy as train lines, each with a different color, and the activities within each phase as train stops, being the final destination for a given line as the main objective of the phase.

Just like a typical train journey, there isn't just one route (way) to get from point A to point B. Some journeys might require getting off at every single stop within a train line (for example, business case) and might even require

iterating several times through the same line as feedback is collected. Other journeys might be smoother and require fewer iterations. The point being, the journey should be iterative and not waterfall-like.

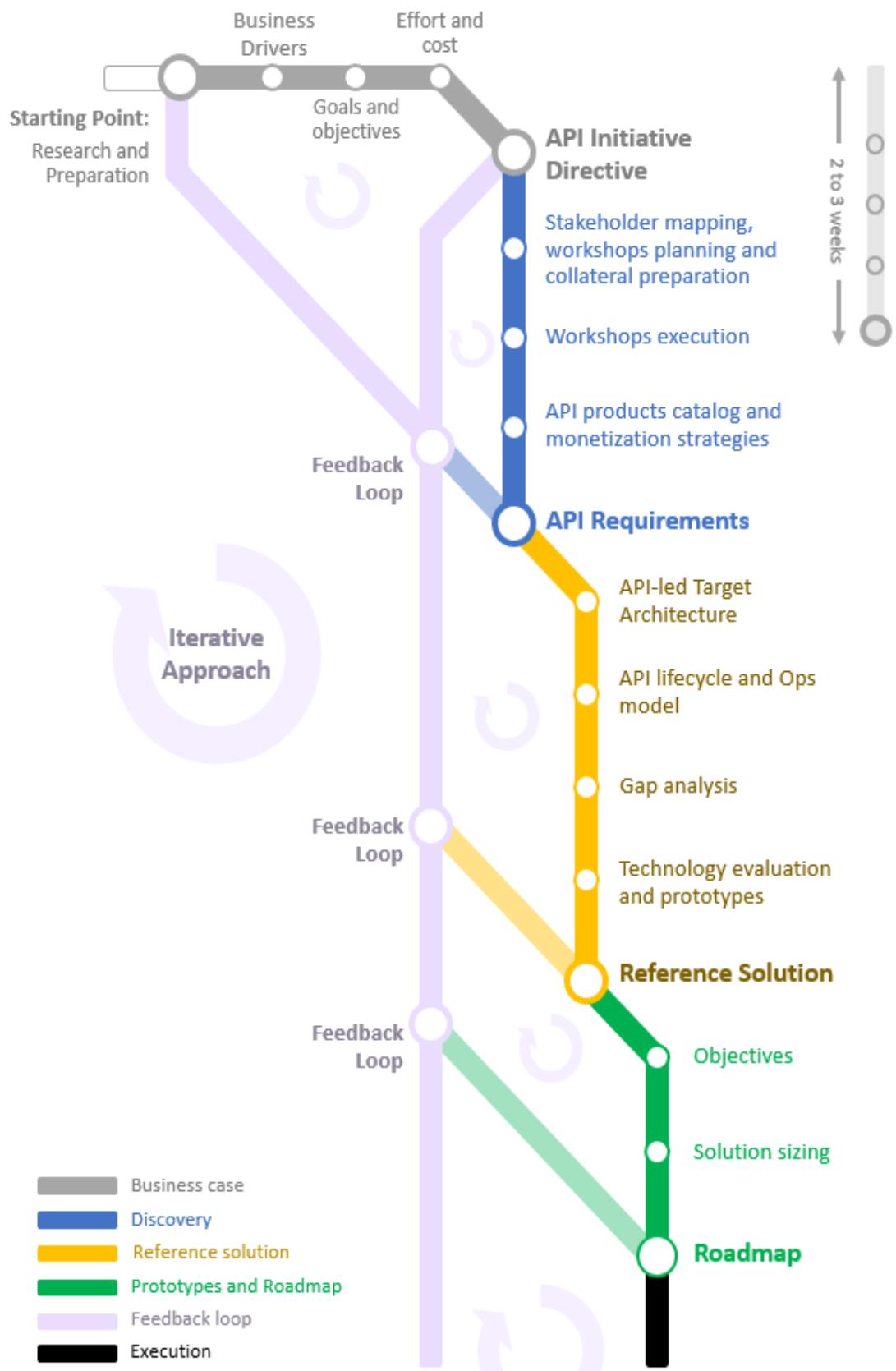


Figure 3.1: Business-led API strategy train map

As it can be seen, the train map aligns quite well to the objectives set, and assuming all stops are visited, the short-term objectives for the initiative should be accomplished in no more than 12 weeks. The exact elapsed time, however, will depend on how many iterations are executed per phase. Typically, one or two should do; however, this will depend heavily on the feedback collected along the way and how many loops will have to occur per phase in order to ensure that the feedback is reflected in the outcomes, which is extremely important to ensure stakeholders feel they too contributed and influenced the outcomes.

Note that in practice, the exact time frame will vary from organization to organization, depending on numerous factors (for example, objectives set, size, and culture of the organization, to name a few). As a rule of thumb, it is recommended that the elaboration of the strategy does not extend beyond 12 weeks, as otherwise it will be difficult to create momentum and ensure that all main stakeholders remain engaged.

The following is a brief description of each train line and its stops.

1. Business case

The objective of this train line is to collect enough business context, such as meaningful business drivers being identified, thus ensuring that goals and objectives that mean something for the business are defined. The final stop (outcome) of this line is to deliver a clear directive that articulates in business language what the API initiative aims accomplish (the goals

and objectives), how (the strategy), and what would be the effort of doing so.

Business drivers

Business drivers are the main motivations behind the implementation of APIs and the creation of an API strategy. They should, therefore, be derived from real business needs that when addressed, a positive impact on the KPIs of the business can be expected.

Goals and objectives

As previously described, goals define primary outcomes that the initiative should focus on. They tend to be broad and thus difficult to measure. Objectives, on the other hand, are more specific steps targeting the delivery of the goals. So, because they are more specific, they can be measured.

Effort and cost

No business will embark on an API initiative without first having an indication of effort and costs. As during early stages there won't be enough understanding of the medium term and long-term objectives (and scope), the recommendation is to deliver a view of short-term effort and costs to deliver a comprehensive strategy (as defined in the train map). One of the outcomes of the strategy will indeed be a view of

subsequent steps for the medium term and long term (including a view of effort and cost).

API initiative directive

The directive is nothing but the consolidation of the drivers, goals, objectives, and approach taken to deliver the strategy into a comprehensive pack that can be presented to the business.

In order to move to the next train line, the expectation is that the business will endorse the goals, objectives, and approach, and thus support is provided (funding and resources) to kick-start the creation of an API strategy.

Without completing the stops of this line, the chances are the initiative will lack business context and buy-in.

2. Discovery

As its name suggests, this train line has a core objective to gather information, and gain a proper understanding of the current technical and organizational landscape in relation to APIs. To this end, relevant stakeholders are identified and working sessions planned, and executed, with the aim to attain such knowledge. In doing so, however, an opportunity presents itself to share inspiring content on the topic of APIs but most importantly, sell the vision and motivations behind the API initiative. This, in turn, will help in getting support and buy-in

from the workshop participants, which is important if the initiative is to be successful.

Stakeholder mapping, workshops planning, and collateral preparation

Workshops should be as much about selling an idea (in this case selling the API initiative) as they are about gathering information, aligning to other initiatives, and identifying dependencies. Making sure that adequate and inspiring content is prepared ahead of the workshops is therefore key and so should be identifying and mapping key stakeholders, and other relevant participants.

Some of the key activities that should happen in the planning and preparation phase are as follows:

- Identify the right stakeholders and audiences according to the organizational scope of the initiative. Make sure relevant stakeholders from the different business units/departments and related initiatives (for example, digital or mobile) are identified, as this will be crucial in identifying dependencies, and getting consensus and alignment. Typically, this involves identifying stakeholders representing the core business, different functional domains, digital IT, traditional infrastructure, cloud, security, and even procurement.

- Don't just focus on management roles, but ensure that practitioners (for example, developers or engineers) are identified as well. The industry is moving more and more towards self-service and federated (as opposed to centralized) operating models, meaning that developers and platform engineers are becoming more influential and empowered. Getting them on board and getting their feedback is therefore equally crucial, perhaps even more so than management, as the latter will likely only get on board if developers and engineers do as well.
- Prepare a questionnaire suitable for different audiences from different backgrounds/domains and different levels of knowledge. Questions should be simple but aimed at gathering meaningful and relevant information that can help the initiative to move forward. As an example, you might want to find out what the main information assets are that other parts of the organization want access to (as these could be potentially good API candidates), or how far the organization is in the cloud adoption journey, how many data centers there are, where the main systems of record are located, and what the main security requirements to comply with are.
- Create meaningful and relevant content describing the "why" of the initiative. Don't just assume that everyone knows what an API is, let alone API management or the fact that some APIs can be monetized, effectively

becoming business products. Create content appropriate for all audiences and then tailor it (even if it has to be on the fly) accordingly. Key in this presentation will be a good articulation of business drivers, goals, and objectives, and some inspiring (not to say cool) content on APIs (for example, what are they? What's the fuss about?).

- During the workshops, some participants will likely highlight the fact that they've already built some APIs and thus implemented some technical capabilities, whereas other teams might just share requirements for new APIs and underlaying platforms. For the former, ensure information is collected around what capabilities the APIs offer and their business context, what API documentation that is available, consumers, usage data, sources of information, and external versus internal use.

For the latter, capture as many business requirements as it is possible to, as this will help to determine what capabilities the APIs need to offer and what technical capabilities might be required in order to deliver that. In both cases, always try to capture what business benefits are offered.

- Plan with enough time in advance, as people will likely be busy. Try and keep the size of the workshop as small as possible to avoid people getting intimidated and thus not giving enough feedback, or the other way around: a

chaos meeting that turns into a talking shop. In person, workshops should be favored over conference calls or video conferences; however, this may not always be viable (especially in global organizations), so be pragmatic and find the right balance.

- In terms of workshop duration, typically anything over two-three hours will lose the audience at some point. So, it's better to keep it short but interesting, but with enough time so that all points of the agenda are addressed.

Workshops execution

Once the planning and collateral preparation is done, the next step is to execute the workshops, collect as much inputs and feedback as it is possible to, then distribute all content gathered, thus ensuring everyone is on the same page and that nothing was missed.

In order to ensure that workshops are a success and positive noise is generated on the back of them, the following should be considered:

- Great content is important but not enough. As previously mentioned, getting key people together represents an invaluable opportunity to promote (sell) the API initiative. Therefore, make use of the opportunity accordingly by delivering the presentation

in a way that is inspiring and educational for all audiences. As this is easier said than done, consider alternatives such as engaging external thought leaders (if budget allows) to help deliver the workshops.

- Avoid the workshops becoming a monologue. Encourage interaction by asking some of the pre-arranged questions to the audience. Also use whiteboards and interactive techniques to make the workshops engaging and dynamic. As audiences are different, the same technique might not work for all audiences. For example, for IT audiences, whiteboarding an as-is architecture might be a good way to understand the current landscape and also identify existing APIs. With business/functional audiences, look for better understanding of business capabilities, and what functionality and information they offer, which could be accomplished by whiteboarding domain models.
- Once the workshop takes place, ensure that there is the means to interactively continue to engage with the audience. To this end, consider adopting tools like Slack or HipChat.
- Consider using modern documentation tools like Confluence or modern Wikis to document, and distribute, workshop notes and actions.

API catalog

The catalog is nothing but a well-structured document listing existing and/or desired APIs, which would have been captured during the workshops and/or other meetings. It's very important, though, to keep track as to where an existing and/or desired API comes from (meaning how was it identified), what the API does or is meant to do, who uses it and why, what potential business value it brings, and so on.

As described in [Chapter 1](#), *The Business Value of APIs*, not all APIs are the same. Some APIs might be great at connectivity, whereas others represent an opportunity to generate revenue.

Regardless of the type, capture enough metadata on all APIs identified, as this will not just follow up on the discussion, but also prioritize desired APIs based on factors such as business value and complexity. In the future, this document can be replaced by modern tooling (for example, an enterprise-wide API developer portal), but to start with, this approach should be enough.

API requirements

As the workshops are executed and more knowledge is gained on the current landscape in relation to APIs, it becomes possible to start creating a proper understanding, not just on the as-is, but on what the actual future needs are. This understanding can only be evolved as more and more feedback is collected through each iteration. The end result should be a comprehensive document listing the numerous API

requirements captured through the phase, including, but not limited to:

- Functional requirements:
 - Business capabilities that could be exposed as APIs and resources/operations that could potentially be offered.
 - Business context such as drivers, use cases that can be addressed by using APIs, and business benefits that could be realized.
 - APIs that are candidates for monetization and therefore will almost certainly require further analysis in terms of product packaging, monetization strategies/billing, marketing, and customer service.
- Non-functional requirements:
 - **Service-level agreements (SLAs)** and other availability requirements that might apply (for example, 99.98%).
 - Assuming some APIs are candidates for monetization, what API monetization strategies could be adopted (for example, pay per call, or freemium)? There is more on this in the next chapter.

- Volumes and throughput: high-throughput platforms will require different architectural decisions than low-throughput ones.
- Authentication and authorization requirements.
- Security requirements.

As not all requirements are the same and some will be more critical to the business and/or IT than others, it is highly recommended to conduct *MoSCoW analysis* of the requirements. This is crucial, as it can help to prioritize some requirements over others and even serve as the input for a **Minimum Viable Product (MVP)**.

More on the MoSCoW method on:

https://en.wikipedia.org/wiki/MoSCoW_method

More on MVPs:

https://en.wikipedia.org/wiki/Minimum_viable_product

3. Reference solution

This train line is all about determining what the future solution looks like, with the scope not limited to architecture and technology, but covering processes and organizational aspects, such as the API life cycle, roles, responsibilities, and the target operating model (for example, center of enablement versus center of excellence). Based on these outcomes, a comparative analysis of as-is versus to-be can take place, which in turns enables the evaluation of the different technology choices available in the market based on a well-defined criterion.

API-led target architecture

Building on top of all of the as-is knowledge/understanding gained during the workshops, this train stop translates the API initiative's goals and objectives into a set of assets that outline what the future API architecture looks like. It is not one asset per se, but rather a combination of core architectural assets that when combined, vital elements of the solution can be articulated. The core assets are the following:

- **Architecture principles:** The definition of the core principles that all API-related assets should be based upon. It's a critical deliverable and each principle should be justified and its implications explained.
- **Conceptual architecture:** Defines the core building blocks, core concepts, and API taxonomies that form the basis for the target API solution.
- **API implementation patterns:** Design patterns define common and reusable solutions that can be adopted to address different requirements. In the context of APIs, this is critical, as an API solution consists of multiple elements and depending on the style of architecture adopted (for example, microservices architectures), the way to implement APIs to solve different problems can vary significantly. API implementation patterns help by defining standard ways to solve multiple problems based on a given criteria.

- **Technical capability model:** Derived technical capabilities needed in order to satisfy the identified technical requirements. All identified capabilities will be described and should map to all components of the conceptual architecture.

The target architecture must be technology- and vendor-agnostic, as the assets produced within will, in fact, be used to evaluate the different options available in the market and identify those that best fit the requirements.

The creation of a target architecture will be covered in more detail in [Chapter 4, API-Led Architectures](#).

API life cycle and operational model

This train stop covers the process and organizational aspects of the target API solution including:

- **API life cycle:** The definition of an API design and development approach focused on ensuring that the APIs delivered are not just fit for purpose, but also usable and well documented. The cycle also defines the roles and responsibilities, as well as tools involved throughout the cycle.
- **Continuous Integration and Continuous Deployment (CICD) framework:** Related to the API life cycle, the CICD framework focuses on defining what tools to use and how they will be implemented, in order to automate and streamline the management of source

code, unit and regression testing APIs, and also their release process across environments.

- **Target operating model:** Defines the modus operandi of the API solution, not in terms of technology, but in terms of organization (roles and responsibilities) and related processes. It should also define how API capabilities will/should be offered to the rest of organization as an enterprise service.

As the outcomes of this deliverable will most likely have a direct impact on people (for example, new or changed roles and responsibilities) it can very easily become political if stakeholders feel that way.

Gap analysis

A key outcome of the reference solution is a comparative analysis of the current technological and organizational landscape against the API-led target architecture, and API life cycle and operational model defined in previous stops.

The analysis is a key outcome, as it will determine what gaps should be addressed and thus require the evaluating of different technologies and/or vendor products. Furthermore, the activities that are to be executed in order to address the gaps will constitute the basis for creating new objectives and defining an implementation roadmap.

Technology evaluation and prototypes

Making the right technology choices, especially when it comes to evaluating different vendor options, can be very tricky and can often result in controversy. To avoid such a critical task being overshadowed by polemics, the advice is to create well-defined criteria weighted in accordance to what matters most from a business value standpoint, but also covering the following:

- **Reference solution fitness:** Ensures that the gaps identified in the analysis can all be addressed either out of the box, or by extending the baseline technology and/or vendor product with additional capabilities. As an example, if a given vendor product fills 70% of the gaps identified, but addressing the remaining 30% may result in a very difficult undertaking (for example, because of lack of extensibility options), then perhaps favoring an option that has less of a fit (for example, addresses 50-60% of the gaps) is the right thing to do, provided it comes with very rich extensibility features (in other words, a much more flexible solution).
- **Developer experience:** Evaluates the usability of a technology and/or vendor product against features that boost the productivity of API developers and API consumers alike. For example, some vendor options might come with rich self-service capabilities and thus

will likely score higher over those that don't deliver such features.

- **Operational experience:** Similar to developer experience, operational experience is adopting a technology and/or vendor product that makes it easier for platform teams to operate an API platform. This is about evaluating capabilities around managing the platform (for example, rich logging, runtime analytics, and even troubleshooting capabilities).
- **Infrastructure footprint and deployment models:** Even when a product/technology is a great fit in terms of reference solution and developer/operational experience, there are no guarantees that it will be easy to implement, or scale, or even if it can be deployed in hybrid (multi-cloud and on-premise) topologies. In order to ensure that the technologies and/or vendor products evaluated can satisfy modern API requirements, criteria can be derived from the third-generation API platform concept described in detail in [Chapter 2, *The Evolution of API Platforms*](#).
- **Licensing model and costs:** Last but not least, careful consideration should be made around how different API vendors license their products and what model best suits your organization. As an example, whereas some vendors license their API products based on the number of API calls, others may do so based on

the number of CPU cores used to run the API gateways. Each approach has pros and cons but generally speaking, CPU-based models tend to be more restrictive, inflexible, and expensive. This, of course, highly depends on the requirements and size of implementation.

Also, bear in mind that some vendors might offer open-source flavors of their products; however, these offerings typically come with many restrictions and will require a license fee in order to extend the capabilities.

As the evaluation process matures and more understanding is gained from the different options considered, implementing a prototype API based on two, or perhaps three, of the preferred options will help to further refine the selection. This is highly recommended, as the insight that will be acquired from experimenting with the different technologies and/or vendor products considered will make the evaluation process less theoretical and instead, more practical and accurate.

The prototype API should be based on a realistic need (for example, one of the APIs identified during the discovery phase) and it should ideally be low in complexity but high in business value. That way, the API can be delivered relatively quickly and the benefits it delivers can too be explained.

Note that the majority of vendors do provide trial versions of their products, which can be used to implement an API prototype.

As an additional step, the API vendors could be invited to showcase how they would approach the implementation of the

API prototype and also answer some of the questions in the criteria. This will require further preparation and planning.

In [Chapter 8](#), *API Products' Target Operating Model*, it is explained in detail how to create the evaluation criteria based on what is discussed at this train stop.

4. Roadmap

This train line is all about defining a new set of objectives and an implementation roadmap to deliver them. The roadmap should illustrate what activities are to be executed, when/in what order, and what/when business benefits are delivered, so expectations can be set for the business and further funding justified.

Objectives

These are a new set of objectives aimed at delivering not just the reference solution, but also the APIs identified throughout the journey. Objectives should be grouped in such a way that they can be delivered in multiple milestones, ideally not too far apart from each other, so business benefits can be demonstrated periodically. As an example, the first set of objectives could be to stand up an API platform foundation so that a second set of objectives, the delivery of the first APIs, can subsequently take place.

Solution sizing

As new objectives are defined, it should be possible to define finer-grained tasks required to deliver them. Such tasks when qualified/quantified in terms of complexity and effort can be used to calculate costs. This is important, as without an indication of effort and costs, it will not be possible to go back to the business and ask for more budget.

5. Feedback loops

Feedback loops ensure that inputs from different stakeholders are captured throughout all train lines and their stops. The idea being that by having feedback regularly retro-fitted into the strategy outcomes, consensus from different stakeholders will be easier, as they too will have contributed towards the final outcomes. The importance of this last point cannot be overstated, as many strategies fail not because they lack thought and substance, but because they lack buy-in from key stakeholders.

6. Execution

As the name suggests, the execution is the actual implementation of the strategy. It is at this point that the initiative moves away from being a relatively small (and even perhaps under the radar) undertaking, into a more substantial, well-funded, but also more visible activity. At this point, the business will expect results to be delivered and accountability to be clearly defined.

Note that as technology is moving so fast, the strategy does not end once the reference solution and all APIs have been implemented. In fact, the strategy should be revisited on a periodic basis (once or twice a year preferably) to ensure that newly arising requirements can be satisfied and also to ensure that the technology choices made are kept relatively up to date.

Summary

This chapter described in a great degree of detail how to define an API strategy driven by business needs, and with business goals and objectives in mind. To this end, the chapter started by providing guidance on how to map the different APIs of the value chain (described in [Chapter 1, *The Business Value of APIs*](#)) to typical business drivers and sample use cases that can be seen in the majority of organizations.

The chapter then continued by illustrating what would constitute a good set of **goals** and **objectives** suitable for underpinning an API initiative.

Subsequent to this, the main highlight of the chapter, the API strategy train map, was introduced as the means to illustrate the different activities and phases that should take place in order to deliver a strategy that results in the actual implementation of an API reference solution, including the delivery of APIs discovered throughout the journey. The chapter concluded by describing each of the train lines and their stops.

The next chapter will focus on **API-led architectures**, including what they are and how a reference solution can be based on them.

API-Led Architectures

This chapter elaborates on the core concepts and capabilities required to implement API-led architectures. From architectural building blocks to listing out individual capabilities, this chapter delivers a comprehensive and practical explanation as to why and how API-led architectures can be implemented to deliver business benefits. The content within this chapter serves as a strong foundation to help you define a target architecture, as explained in the previous chapter.

The API-led architecture described in this chapter takes inspiration from [OMESA.IO](https://omesa.io).

What is API-led?

As APIs continue to open doors to information and functionality, they move well beyond being merely technical interfaces to becoming central actors in new digital business models. However, for such business models to be effective and successful, technical teams are confronted with the unique challenge of API-enabling systems that weren't necessarily built to provide real-time access. Because of this, there are several implementations beyond the obvious that must be catered for in order to create APIs that are easy to use and can truly scale to handle the sort of volumes expected from real-time systems.

API-led is therefore an architectural approach that puts APIs at the epicenter of communications between applications and the business capabilities they need to access, in order to consistently deliver seamless functionality across all digital channels. The following diagram, for example, illustrates how an **API exposure** capability, such as an **API gateway**, can provide access to APIs that can be consumed to leverage different business capabilities.

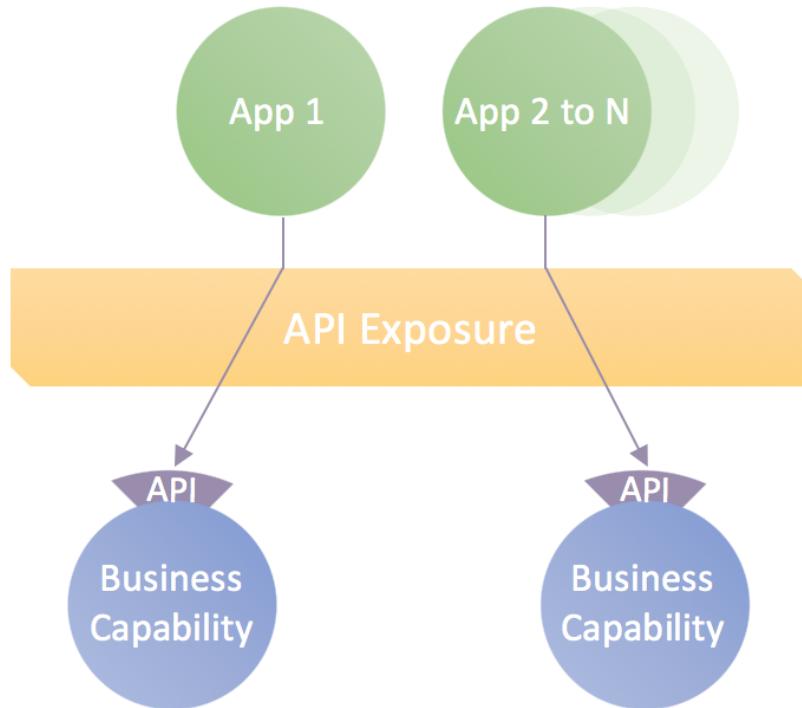


Figure 4.1: API exposure

Furthermore, because APIs can directly and/or indirectly drive revenue generation, the department that delivers them can evolve from being a cost center to potentially becoming a profit center. In other words, an API becomes a first-class business citizen, as opposed to a necessary evil, which is often how integration is perceived by businesses. Long story short, APIs become products in their own right and thus require the same level of design thinking, ongoing attention, and evolution as other business products.

Architecting API-led

In [Chapter 1](#), *The Business Value of APIs*, doors were used as a metaphor to articulate the role of APIs in delivering access to enterprise information assets and functionality, or in business terms, **business capabilities**. However, just like doors, which come in different types, materials, and sizes, often determined by what sort of access they provide, APIs too can be classified in different types.

For example, some APIs might be built with a specific use case in mind and in support of a specific application. Because of this, such APIs can be quite specialized and tailored for the purpose they were built to serve. In other words, they are single purpose and are not suitable for reuse outside the context they were built for.

*A common term used to refer to these types of (single-purpose) APIs is **Experience APIs**, mainly because of their role in enabling applications that humans directly interact with (for example, mobile apps, web apps, and so on). However, not all applications that require specialized-purpose APIs have to interact with humans. For example, in Industry 2.0, APIs may be built in support of modern industrial lines, or in farming and agriculture, drones are being used to scan soil conditions across large areas of land, and APIs are used to obtain and send data in real time.*

This is why this book favors a more generic term: single-purpose APIs.

Other APIs, however, might be built specifically with reuse in mind. Such APIs will be more generic in nature and won't be tied to a particular use case. Therefore, these APIs are multi-

purpose, meaning they can be used in a variety of scenarios and thus should be able to serve many applications. The following diagram illustrates that APIs that don't provide access to tailored business capabilities can be consumed by many applications to address different use cases.

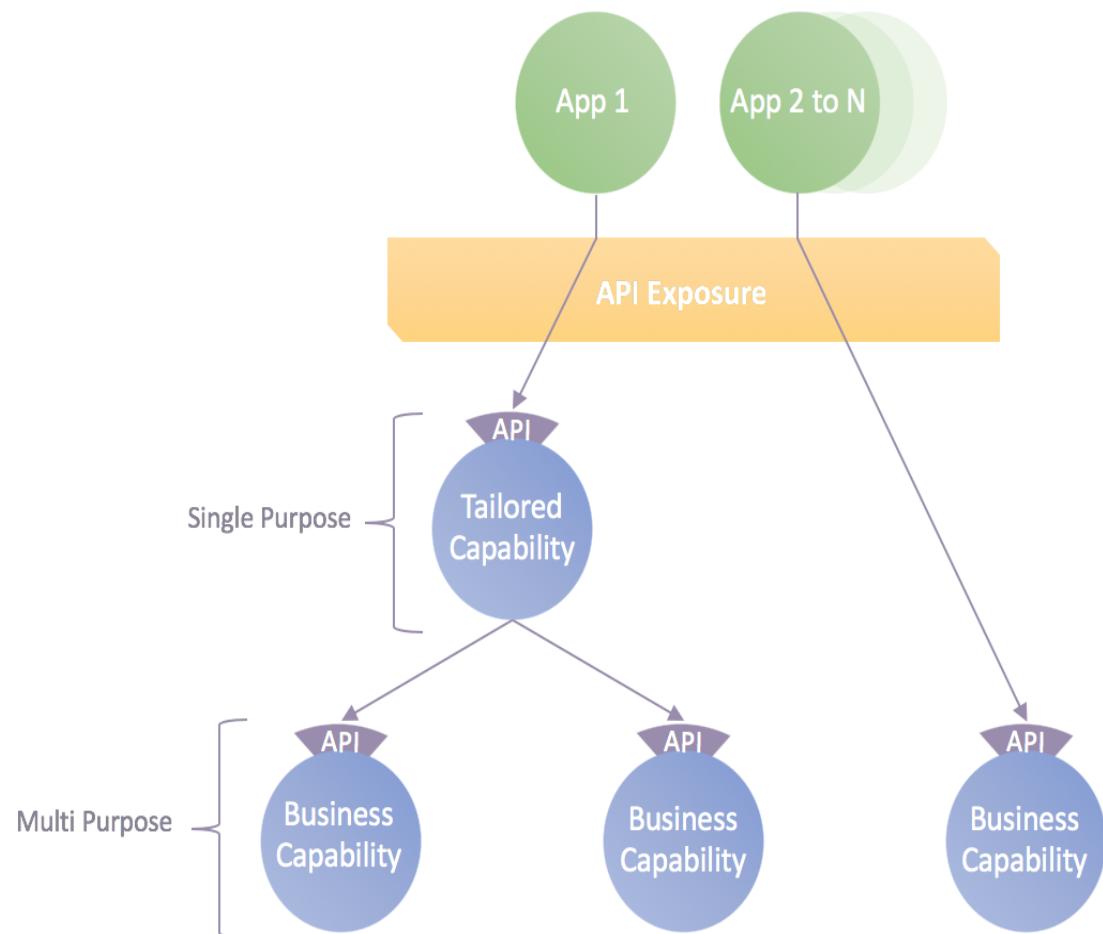


Figure 4.2: API-led communication

Just understanding the different types of APIs isn't enough to architect API-led solutions. A robust architecture must always be reflective of a solid understanding of the business domain in question and the capabilities required in order to address it. In

the context of enterprise-grade APIs, at the most basic level, API-led architecture should address things like:

- Securing APIs from unauthorized access and major security threats.
- Ensuring that consuming applications can always find the right API endpoint.
- Throttling and/or limiting the number of calls made to an API to ensure continuous availability.
- Supporting capabilities such as API design, testing, continuous integration, life cycle management, monitoring, and operations, to name a few.
- Error handling and preventing error propagation across the stack.
- Real-time monitoring of APIs with rich analytics and insight.
- An approach for implementing scalable and flexible business capabilities, for example, in support of **microservices architectures**.

In the subsequent sections, a top-down approach for elaborating an API-led architecture is presented. We start by first defining conceptually the core building blocks of the architecture and describing their purpose. We then continue by defining in more detail the individual capabilities required by

each block in order to accomplish its purpose. Finally, the section illustrates how all the capabilities hang together cohesively as a single, but modular, platform.

Conceptual architecture view

The conceptual API-led view defines the main building blocks of the architecture, along with their purpose and responsibility. As in *Figure 4.3*, the conceptual API-led architecture consists of four main building blocks, plus the **consuming applications**. Horizontal blocks represent core runtime capabilities, without which implementing APIs would not be possible. Vertical blocks represent important supporting capabilities geared towards life cycle support, management, operations, and analytics.

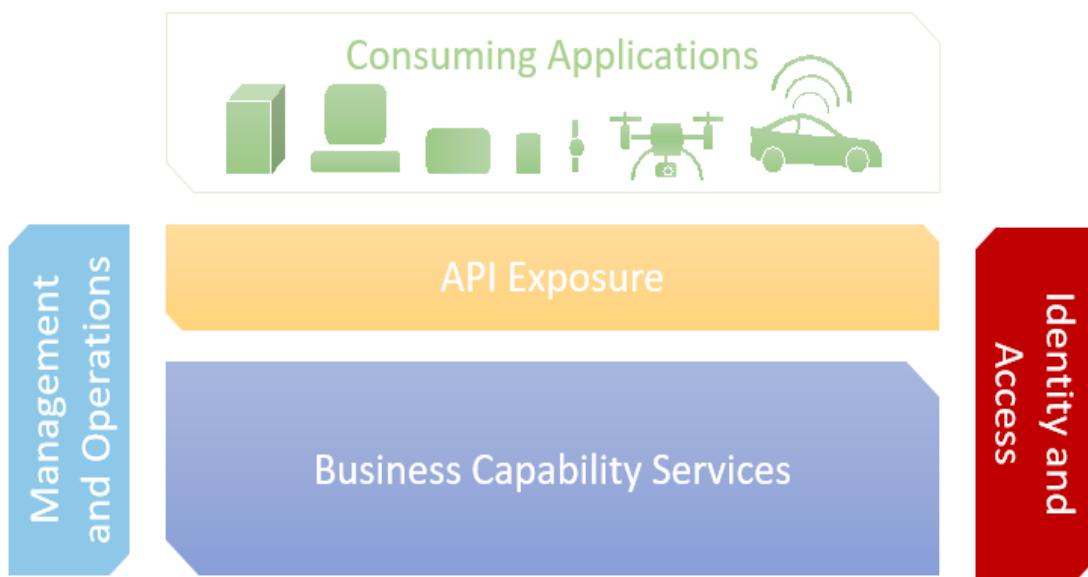


Figure 4.3: Conceptual API-led architecture

Consuming applications are considered as any computer program capable of calling and making use of an API. In real-world terms, these may vary from traditional commercial off-

the-shelf applications (for example, commerce systems) to web applications, mobile applications, wearable devices, and even more sophisticated things such as drones and smart cars.

The **API exposure** building block, as its names implies, is responsible for securely and reliably exposing access to API endpoints.

Services are units of software that deliver a well-defined and bounded functionality. Such functionality is referred to as a **business capability** because it means something to the business and the function the service delivers can be mapped to a business process. Therefore, in order to deliver such functionality, a service must be capable of implementing business logic, data transformation and validation, business rules, and orchestrations and/or choreographies, to name a few. Services expose their functionality via API endpoints that are not accessed directly but mediated via the API exposure layer.

The **management and operations** block consists of capabilities in aid of the end-to-end life cycle management of APIs, including, but not limited to, API design and mocking, policy implementation, deployment, promotion, runtime operations and analytics, and deprecation and retirement.

Developer-centric capabilities such as API pages, a developer portal for API discoverability and subscription, and application keys management also form part of this building block. This

building blocks acts as an aid when monetizing APIs, as it's responsible for collecting important metrics that might be required when billing APIs based on usage.

Lastly, the **identity and access** block refers to capabilities in support of user, roles, and access management features. For example, from a life cycle and operations perspective, this block enables different users (API product owners, administrators, designers, and developers) to seamlessly logon to the API management console and/or API developer portal using existing enterprise credentials. It also restricts access to different areas depending on the user role. From an API exposure perspective, this block aids authentication and authorization policies by, for example, enabling tokens (for example, OAuth 2.0, OpenID, and/or even **Security Assertion Markup Language (SAML)**) to be generated and enforced at runtime.

Technical capability view

This view extends the conceptual architecture by defining the main technical features expected of each building block.

Note that the business capability block is not to be confused with the technical capability model. The latter refers to technical features that are expected of each building block. The former refers to services that offer access to business functionality equally implemented using platform features.

Defining technical capabilities and segmenting them by building blocks is very important because it enables:

- The visualization of the responsibilities expected of each block, or in other words, clear understanding what a building block should and should not do.
- Evaluating/comparing technology choices, for example, by comparing how a specific vendor and/or product satisfies the capability view and/or a part of it.
- Conducting gap analysis by identifying what capabilities might or might not be available within an existing technology landscape.
- Defining patterns on how certain capabilities can be used in conjunction, in order to address a common problem.

In the following sections, the individual capabilities within each building block are defined.

Management and operations

The API management and operations block delivers critical capabilities in support of the full API life cycle, including runtime operations and analytics. This capability is typically delivered in the form of a central web application(s) from which the following functionality is accessible:

- Self-service onboarding.
- API discovery and subscription.
- API design and mocking.
- API creation and configuration of API exposure policies.
- API documentation and content management.
- API web pages, developer portals, and marketplaces.
- API deployment and publishing.
- API versioning, deprecation, and retirement.
- API exposure infrastructure configuration and management.
- Real-time monitoring and analytics of APIs and API exposure infrastructure.
- Management APIs and control planes.

- API monetization and billing.

This block, together with API exposure (returned to later in this chapter), conforms to what vendors typically refer to as API management platforms.

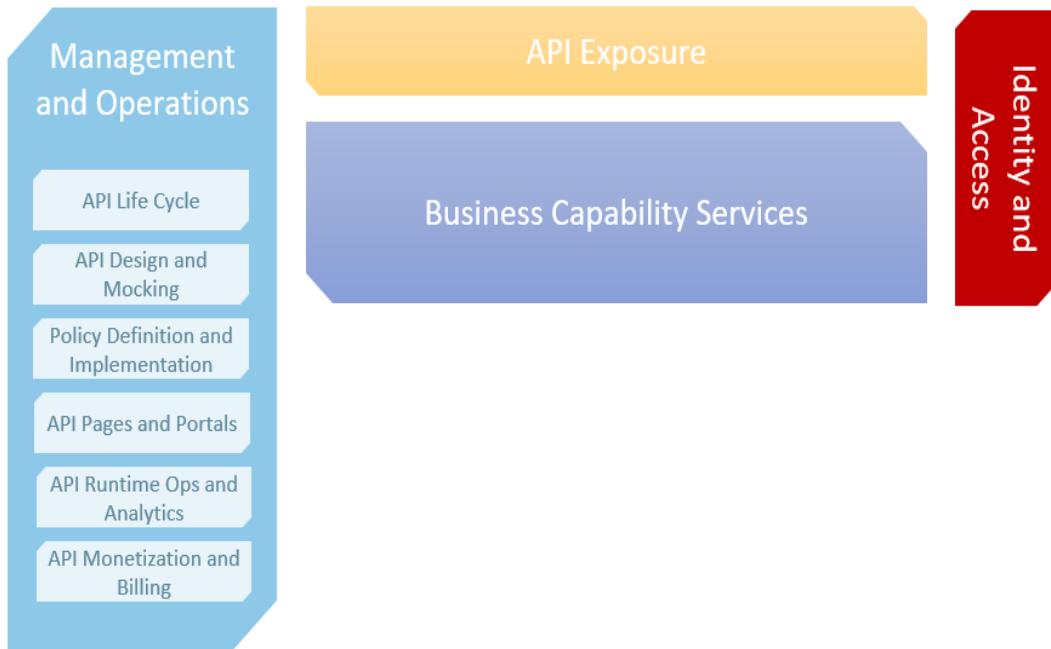


Figure 4.4: Management and operations

The capabilities expected of this block are described in the following sections.

API life cycle

The section describes capabilities required exclusively in support of the life cycle of individual APIs.

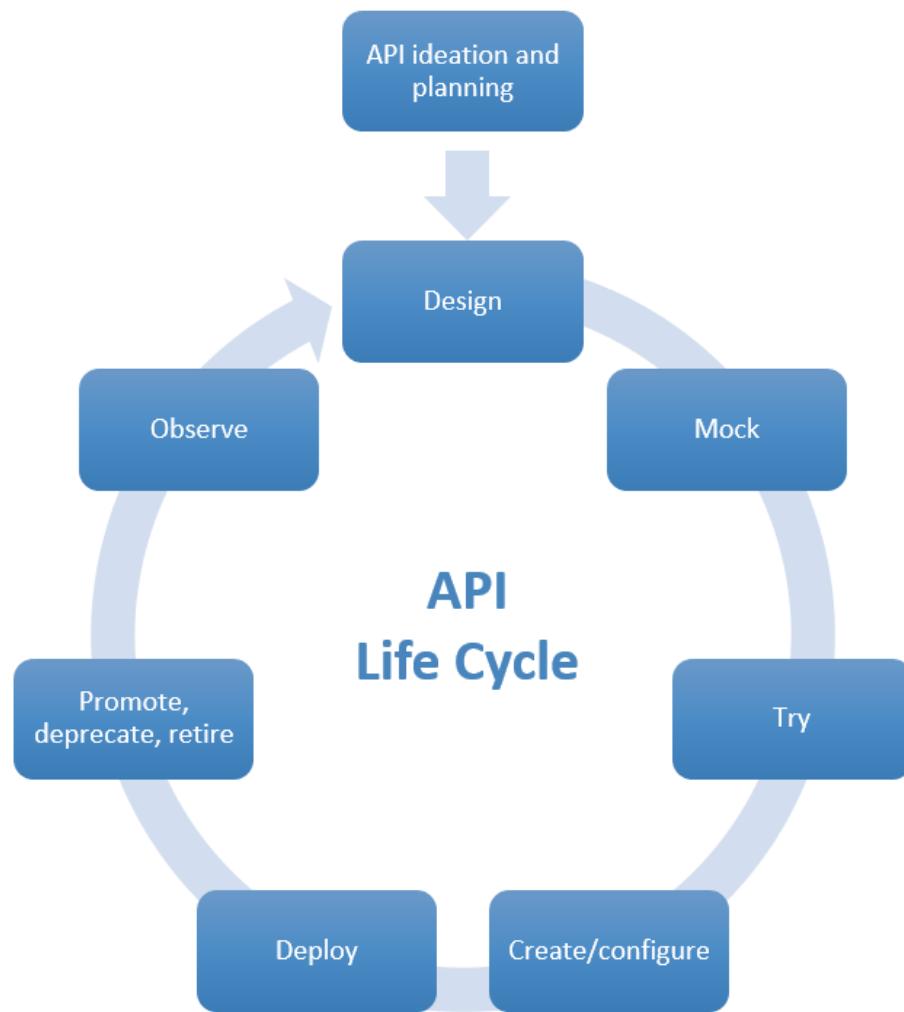


Figure 4.5: API life cycle

This typically involves having the ability to:

- Identify new APIs and subsequent planning.
- Design and mock APIs.
- Create and configure API exposure policies, including policies related to access and monetization.
- Create and publish API web pages either standalone, as part of a developer portal, or using a public marketplace.
- Deploy APIs to the API exposure infrastructure, which typically involves several API gateways.
- Manage API versions, including deprecating and retiring older versions.

API design and mocking

Although part of the API life cycle sub-block, this capability can also exist on its own and therefore it has been separated to give it more emphasis and a detailed overview. A robust API design capability should enable API designers and developers to take part in what is known as the **API design-first cycle**, as illustrated in *Figure 4.6*.

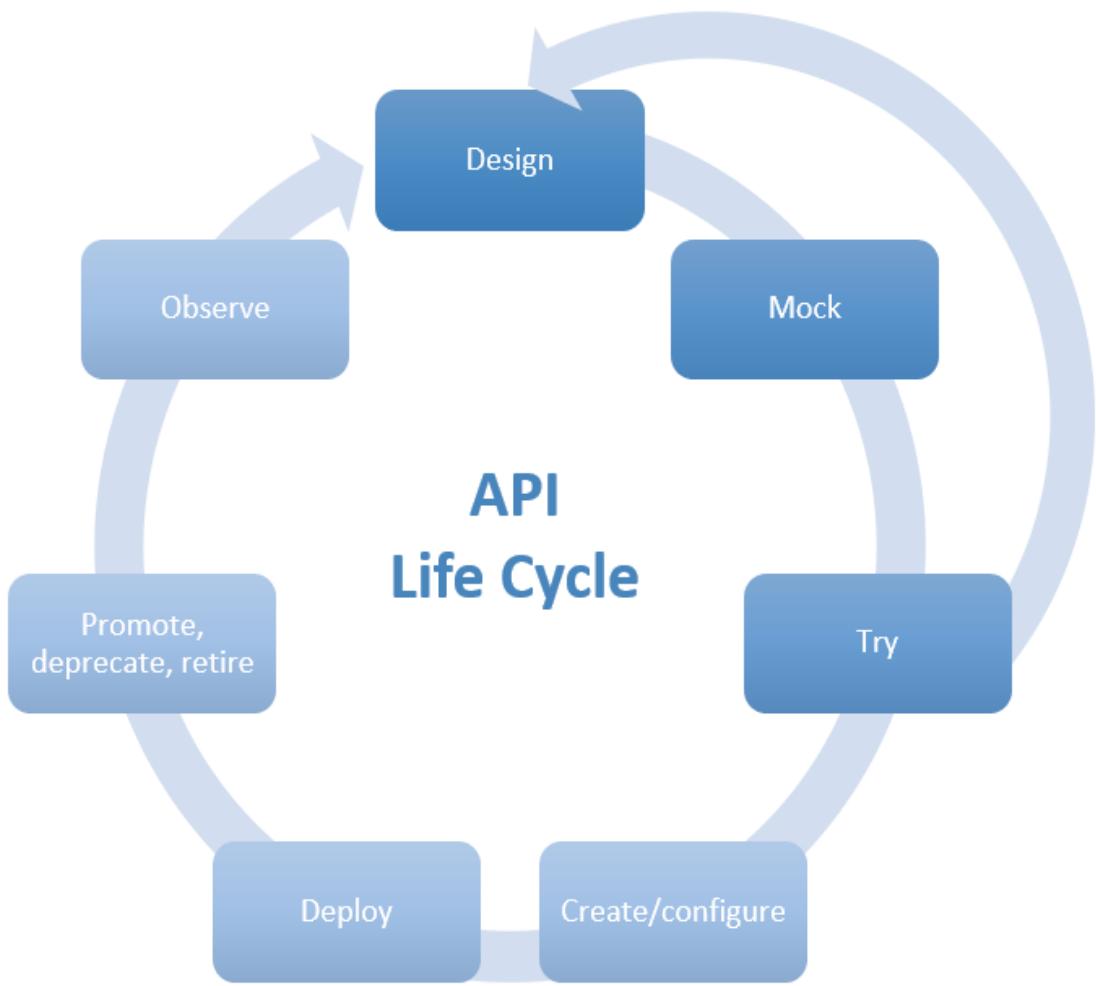


Figure 4.6: API design-first cycle

The idea behind this concept is to not just allow API consumers and producers to work on client and service implementations in parallel, but even more importantly, to enable both parties to engage in feedback loops as early as possible in order to iteratively evolve the design of the API. This is as opposed to having to wait until an API has been implemented (typically involving the creation of a service) only to find out that the API didn't fully meet needs, or it was too difficult to use. Instead, an API design-first cycle enables API designers to quickly design

and mock API resources so consumers of the API can quickly try it out and feedback can be provided. This process continues until it is felt that the API satisfies consumer needs.

*There are tools on the market dedicated to the API design-first cycle, the most popular ones being [Apiary.io](#) and [SwaggerHub](#) ([swagger.io](#)); however, the latter only supports the **OpenAPI Specification (OAS)**, whereas Apiary offers OAS in addition to API Blueprint.*

The expectation is that this capability will offer a self-service functionality that will allow us to:

- Quickly design APIs with any of the main specifications available, such as OAS, API Blueprint, **RESTful API Modeling Language (RAML)**, or even GraphQL-based APIs.
- Automatically generate API mocks directly from the API specification without having to write any code. The idea is that the API mock, its specification, and any relevant documentation can be shared with the API's consumers for feedback.
- Monitor the API mock usage, including successful and failed tests.
- Carry out API specification compliance checks (typically referred to as **stylesheets**) in order to automatically verify consistency and compliance in the way APIs are being designed. Ideally, it should be possible to create custom stylesheets based on one's specific needs and desires.

Further functionality includes:

- A templating engine, so the process of designing an API can be aided with pre-created sample API specifications.
- An API catalogue, so available designs can be found and reused if desired.
- Integration with version control systems (for example, **GitHub**), so the API specification and associated documentation can be version controlled and managed just like another piece of code.
- A rich CLI, so API specifications can also be found and tried via command lines.
- Scaffolding of server and client code (in multiple languages) generated directly from the API specification. This feature is important, as it can seriously speed up the process of consuming and implementing APIs and associated services.
- A runtime API specification validation tool, so it's possible to verify the compliance of a running service against its specification.

As you can see, the API design-first cycle does imply a rich set of additional capabilities that are not always offered out of the box within a vendor's API management offer. Therefore, it is important to fully understand this cycle in order to identify any potential gaps in the product and determine how such gaps can be complemented by, for example, adopting some of the aforementioned API design-first tools.

Policy definition and implementation

API policies are a crucial piece of capability that enable the change of behavior of an API through configuration. Policies are collections of statements that are executed sequentially on the request or response of an API.

The API exposure block defines several types of policies that can be used to address different types of requirements. For example, policies can be applied to protect against common security threats, to implement authentication and authorization, to route and load balance incoming calls to multiple backend services, and even to enforce monetization plans.

This capability in particular refers to having the ability to apply to individual APIs, through a central management capability, policies that are to be enforced in the API exposure infrastructure (typically an API gateway).

API pages, developer portal, and marketplaces

Once an API reaches the point where it can be made available for use, be it in beta or general availability, it is empirical for the success of the API that comprehensive and rich documentation is also made available. Equally important is to ensure that the **developer experience (DX)** is also considered, thus ensuring that APIs are easy to find, understand, subscribe to, and use (regardless of whether the API is external or internal).

The one thing that most API practitioners agree upon is that an API is only as good as its documentation. Therefore, an API should not be considered done until documentation is also made available.

However, what does comprehensive and rich documentation mean and how can a good DX be achieved?

A common misconception is that an API specification (for example, OAS or API Blueprint) is enough and also caters for the documentation aspects of an API, whereas certainly the API specification should be part of its documentation, but by no means is it enough.

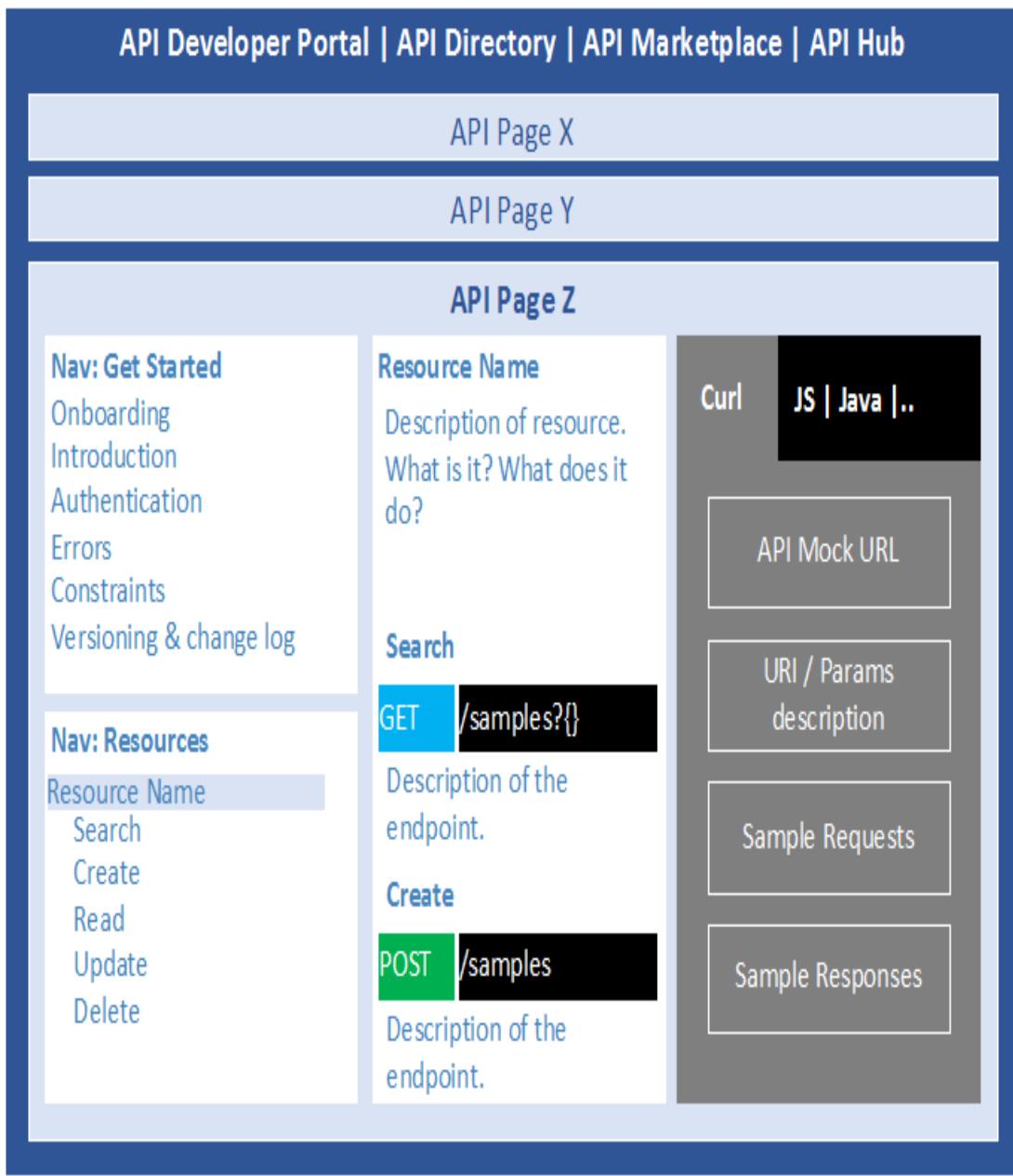


Figure 4.7: API channels and documentation

As depicted in *Figure 4.7*, there are several aspects that this capability should be able to address, with the aim of delivering comprehensive and rich API documentation that delivers great DX and makes it very easy for consumers to find, enroll, and start using APIs.

To elaborate further on the features that are expected within this capability:

- First of all, it should be possible for APIs to be published in a variety of channels (also offered as part of this capability), mainly, but not limited to:
 - An organization's **API developer portal**, where all APIs belonging to the organization are listed, and internal or external developers can find, subscribe, and use the APIs.

Note that vendors tend to offer API developer portal capabilities within their standard API management offers. Some vendors charge for this capability as an add-on, while others just include it in their offering.
 - An **API directory**, where APIs are listed along with key metadata and documentation, but, with a link to the API page in the organization's developer portal.

programmableweb.com and apilist.fun are both good examples of API directories.

- A public **marketplace**, where APIs belonging not just to a single organization, but multiple organizations, are also listed. A marketplace differs from a directory as from here it is also possible to subscribe and use APIs, as opposed to API directories, which only offer limited information.

rapidapi.com is probably the best example of a good API marketplace. However, it has become common for API

management vendors to also offer their own API marketplaces, for example market.mashape.com.

- Other channels where it should be possible to publish the APIs to are the so-called **API hubs**. An API hub is typically a form of self-service integration platform that (in principle) enables technofunctional developers to create integration flows orchestrating one or multiple public APIs published in the hub.

ifttt.com and elastic.io are both good examples of API hubs.

- Creation of **API pages**. An API page delivers comprehensive and detailed documentation describing all the features of the API, including, but not limited to:
 - A getting started section describing things such as:
 - The process of onboarding to start using the API. This may involve a registration process, which may include steps to capture the user details, selecting the monetization scheme and payment options when/if applicable, and obtaining user credentials and an application key.
 - A comprehensive overview of the API's functionality and different features available.
 - Details on how the API handles authentication and authorization, and other aspects of security.

As authentication and authorization tends to be a complicated topic, it's extremely important to make the documentation around this topic as simple as possible.

- How the API handles errors and what error codes can be expected.
- Description of any constraints.
- A section dedicated to versioning, describing things such as current, future, and previous versions (basically a change log) and how switching between versions is to be handled, especially from a consumer standpoint.
- A comprehensive description of all available resources (or operations in the case of GraphQL), their parameters, and HTTP verbs supported, including plenty of interactive sample request/response payloads.

Interactive means that the samples can be tried directly from the API page itself.

- Details of the API mock (for example, the URL) and ideally an embedded online console could be used to try out different API calls directly from the API page.
- Just like in the case of API design, an API page should also offer capabilities to scaffold server and client code

in multiple languages, directly from the API specification.

- Any other additional pages, such as information on the payment options, subscriptions, and terms and conditions, to name a few.

API runtime operations and analytics

The capabilities in this block serve two main objectives.

1. This first one is to ensure that the "lights are always on" by providing capabilities that aid the real-time monitoring of APIs and their underlying runtime infrastructure (for example, API gateways). Ideally, this should include the ability to configure SLAs.
2. The second one relates to real-time analytics. The focus is to provide comprehensive data visualizations and reports around API usage patterns and DX, transaction throughput, success/failure rates, response times, and/or other interesting information that enables architects and designers to make design decisions on how to improve an API. This same information can also be used when considering if certain APIs are candidates for retirement and decommissioning.

Note that the vast majority of commercial API management solutions do offer broad capabilities to monitor, visualize, and analyze APIs, and their underlying infrastructure. However, such offers won't typically cover the independent runtimes (for the service infrastructure) and additional tooling, such as the popular prometheus.io, datadoghq.com, and elastic.co or the equivalent, may be required.

API monetization and billing

The monetization of APIs is perhaps the most important promise of the so-called API economy. However, before it is possible to charge on the functionality offered by APIs, it is important to understand what API monetization actually means.

It's important to acknowledge that API monetization means far more than just charging for API calls. A better and more commonly accepted definition is that API monetization refers to having the ability to drive revenue through the use of APIs. This definition differs from the former as it is much broader and in addition to charging for API calls, other forms of revenue generation can be defined, as illustrated in *Figure 4.8*:

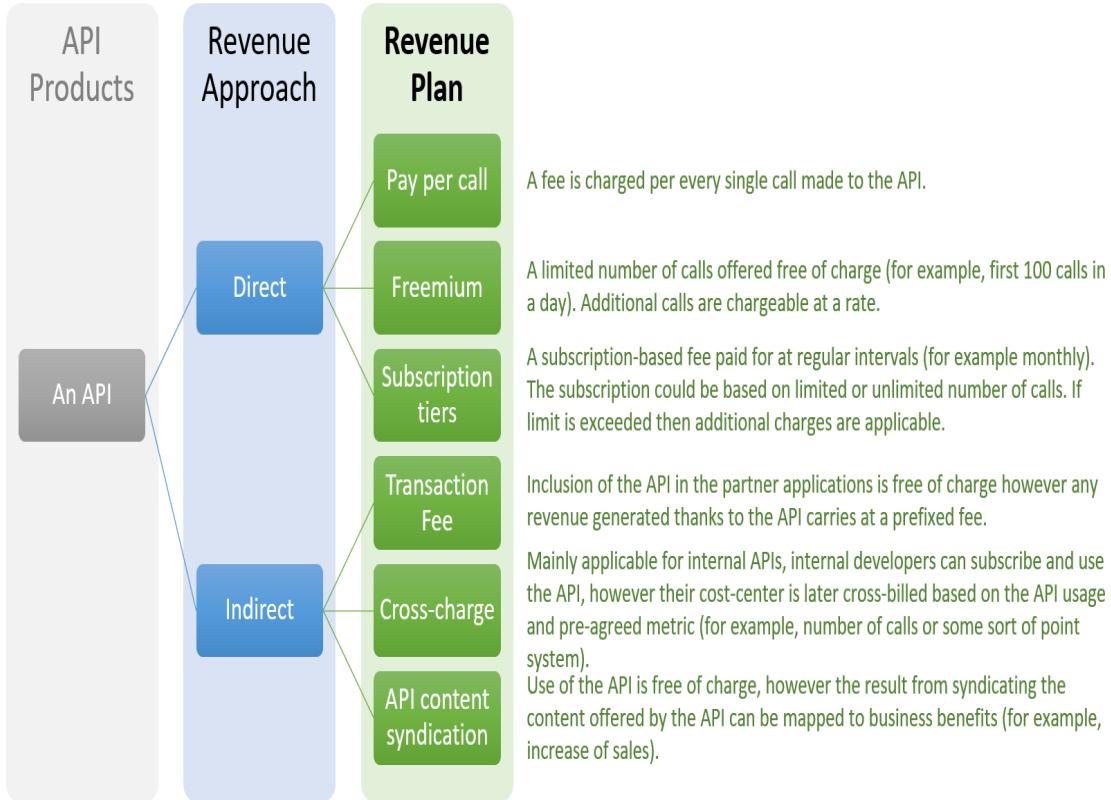


Figure 4.8: API monetization plans

Having said that, monetization won't happen on its own and therefore a set of capabilities is required, most notably:

- **API plans:** The ability to define different monetization plans based on direct or indirect schemes, such as the ones mentioned earlier (for example, pay per call, freemium, and so on), or custom ones.
- **API monetization policies:** The ability to implement and enforce a monetization plan to any given API or group of APIs (this capability is also explained in the subsequent chapter).
- **API billing:** The ability to bill based on the implemented plan. When the scheme is based on a

direct monetization plan, then this capability should also offer either basic finance capabilities, such as the ability to bill and invoice based on the scheme selected, or a pre-built integration with any of the popular billing systems.

API exposure

As stated earlier, this block is responsible for securely and reliably exposing access to API endpoints. The typical means to deliver the set of capabilities within the block is by implementing an API gateway.

An API gateway is a runtime component that handles incoming requests, which are then mediated (also known as routed) to the individual services that are responsible for delivering the business capability that a consuming application is after.

During the mediation process, different actions (commonly referred to as policies) may be executed in order to perform common tasks that don't really belong to the business capability. Examples include, verifying the identity of a caller and the rights to call the API, redacting the payloads, throttling traffic, and/or limiting the number of calls based on different rules, or split-joining calls to multiple services, to name a few.

In [Chapter 2](#), *The Evolution of API Platforms*, it was explained in detail how modern API gateways have evolved from traditional (and heavy) XML appliances and/or ESBs into more lightweight and microservice-oriented API microgateways, which are suitable to be executed in highly scalable and containerized environments such as Kubernetes.

In the future, it is likely that even API gateways will be offered as a serverless capability of some sort.

Serverless computing is a cloud-computing execution model in which the cloud provider acts as the server, dynamically managing the allocation of machine resources. Applications built based on the serverless computing model only concern themselves with the code required to deliver the specific functionality expected of the application, and not on how or where the code itself is executed at runtime. For further information, please refer to https://en.wikipedia.org/wiki/Serverless_computing.

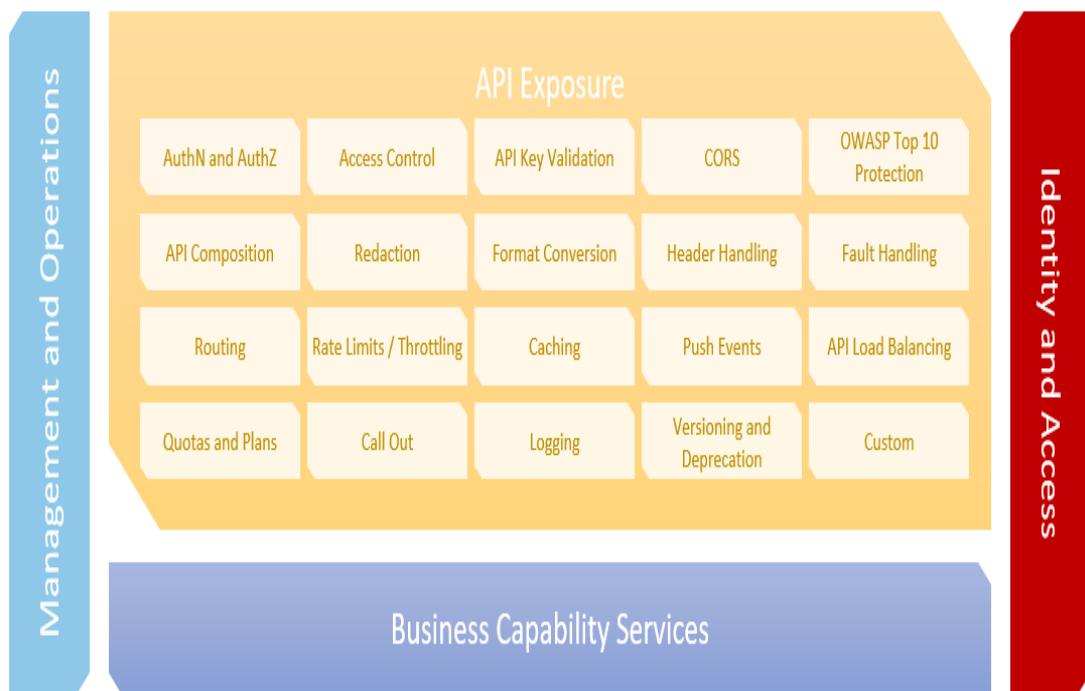


Figure 4.9: API exposure building block

As illustrated, the capabilities expected of the API exposure block can be summarized as follows.

Authentication (AuthN) and authorization (AuthZ)

This refers to having the ability to enforce authentication and authorization policies against API endpoints. Authentication refers to having the ability to verify a caller's identity based on the credentials (typically just username/password) supplied as part of a call. For example, the simplest type of authentication is HTTP basic authentication, where credentials are provided by the caller in the HTTP header and validated against a **Lightweight Directory Access Protocol (LDAP)** server. More complex authentication capabilities can be based, for example, on client certificates and/or tokens as opposed to a username/password.

Authorization, however, refers to having the ability to verify that a (valid) caller has adequate rights to access a given resource (for example, a specific API endpoint). Given that authentication precedes authorization, major protocols, most notably OAuth 2.0, define flows as how to handle the authentication and subsequent authorization enforcement for a given caller. OAuth is a good example but not the only one; other protocols, such as OpenID and SAML, should also be supported by this capability.

Note that authorization related patterns will be covered in the next chapter.

Access control

Just like **access control lists (ACLs)** in networking, this capability refers to having the ability to define access rules at transport level, for example, defining an access rule that only allows calls to an API that come from a specific IP range, or the opposite: allowing all callers but those coming from a specific range.

This capability can be useful, for example, in scenarios where it's required to restrict access to an API based on a given region (for example, only calls originating from IPs belonging to EU countries are allowed). Because there are ways to get around IP restrictions, this capability should not be used as the only means to control access to an API.

API key validation

This refers to having the ability to validate the presence of a valid application API key in API calls (for example, as part of the HTTP header or URI). Application keys are commonly issued via self-service capabilities in the API developer portal for registered consuming applications.

This capability can be extremely useful when wanting to understand who the consumers of a given API call are. This, in turn, is critical in change control as it, for example, enables us to gain a better understanding of who the impacted audience for a given API change may be.

CORS

User agents commonly apply same-origin restrictions when making requests to different URLs. These restrictions can prevent client-side applications (for example, an AngularJS application running on a browser) from calling API endpoints from domains different to the one used to access the application itself.

Cross-Origin Resource Sharing (CORS) is a W3C standard for allowing user agents (for example, a browser) to enable different-origin requests to take place in a secure way, therefore allowing user agents to securely get by restrictions.

Further information can be found at <https://www.w3.org/TR/cors/>.

This capability refers to having the ability to apply cross-domain checks to, for example, enable CORS on certain API endpoints.

OWASP Top 10 protection

The **Open Web Application Security Project (OWASP) Top 10** delivers an industry-recognized awareness document for the most common web application security threats.

As APIs make use of web protocols (most notably HTTP and HTTPS), they too are exposed to such threats. Therefore, capabilities that help to prevent such threats are important, especially for APIs that are exposed to the general public via the internet.

API composition

API composition refers to the ability of an API exposure block to perform split joins against one or many downstream service endpoints. The idea is that instead of an API consumer having to make multiple API requests, for example, to perform multiple queries, a single request payload can be split into multiple downstream calls. The responses can then be joined into a single payload before the response is sent back to the consumer.

This capability can be very useful, for example, when it is required to optimize the number of API calls made by a mobile device, as network bandwidth is often a premium resource, or when the number of API calls required would end in too much chattiness between the API consumer and API exposure, therefore affecting the overall user experience.

API composition differs from orchestration in that there is no (or should not be any) business logic implemented in the composition, for example, if/then/switch conditionals, for/while loops, or complex data transformations. If business processing logic is required, then this should be implemented as a service in the business capability block.

Redaction

Redaction refers to having the ability of removing, masking, and/or limiting the presence of fields within request/response payloads and/or headers. Such a capability can be useful; for example, when it is required to completely remove sensitive data from payloads (for example, credit card data) and thus comply with regulation such as GDPR in the EU.

Format conversion

This refers to having the ability to convert payloads from one format to another over the same transport. For example, from XML/SOAP over HTTPS to JSON over HTTPS and vice versa.

Note that in certain scenarios, such as in the case of the **Internet of Things (IoT)**, some specialized API gateways may also support, in addition to payload conversions, transport conversions. For example, converting from **Message Queuing Telemetry Transport (MQTT)** over TCP (a very popular transport in IoT) to JSON over HTTPS.

This capability can be useful, for example, when it is required to repurpose existing APIs built in less popular protocols (for example, SOAP) as rewriting or refactoring them (for example, to support REST) may not be an option. It can also be useful to expand the audience of an API so it can be used, for example, in support of IoT.

When implementing format conversations, be aware that not all APIs will convert well. For example, when converting SOAP/WSDL-based web services into REST, not all SOAP operations may convert into fully compliant REST endpoints, and this is simply because the way SOAP operations and REST resources are modeled is fundamentally different.

Header handling

Header handling refers to being able to propagate, add, rename and/or remove transport headers. This capability can be useful when there is a requirement to propagate or modify transport headers in support of downstream service invocations.

Fault handling

As the API exposure block handles incoming calls and enforces policies such as authentication/authorization, API key validation, and routing, errors might occur, which will result in exceptions being thrown back to the caller.

This capability is about being able to handle faults, such as adequate and standard error codes, and ensuring messages can be sent back to the callers, as opposed to just relying on the underlying infrastructure to do this, which typically is the default position.

Routing

Routing refers to the ability of mediating HTTP(s) calls received on a specific resource (for example, `/myapi`) to different downstream service endpoints, ideally based on different conditions. The conditions, for example, could be static, meaning a one-to-one mapping of an URI to a corresponding service endpoint, or dynamic-based information, such as application keys, headers, and even originating geography. API gateways are known for implementing such capabilities. More mature gateways provide rich options to define how mediation should occur.

Rate limits

This allows for the implementation of hard limits in the number of calls that can be made to a given API resource. Limits could be applied to specific applications, users, or just overall without distinction. This capability is very useful to protect against unexpected peaks (for example, by setting a limit to the maximum number that the system can handle) or even to protect against denial of service attacks.

Throttling

This capability offers the ability to limit or regulate the maximum throughput that can be handled at a given timeframe. Being able to statically and/or dynamically control the throughput (for example, 100 transactions per second) ensures that downstream services won't have to handle a larger number of requests than they physically can. It also helps to prevent a given API from overusing the capacity of the underlying infrastructure, thus degrading the performance of all APIs.

Caching

In simple terms, caching refers to being able to cache API response data. In the most basic scenario, this is the ability to cache the response of a downstream service call. For example, when a second (similar) call occurs, the downstream service won't have to be invoked as the response is already cached.

This known as a Response Cache.

This capability can be extremely useful when preventing a large number of similar calls translating to an even larger number of downstream service calls. Caching response data means not only faster response times, but also avoiding unnecessary downstream service calls. However, conditions will be required in order to prevent consumers from receiving outdated/irrelevant data.

Push notification

Typically, the communication between an API consumer and the API originates from the former and occurs in one direction. In this approach, when the most recent data (for example, changes made to a record) is required, it is the API consumer's responsibility to poll for changes (meaning constantly calling an API). Not only does this result in chatty interactions, but it is also highly inefficient from a transport use perspective. It can also result in poor user experience (for example, users having to refresh application screens so that data updates).

An approach to get around such limitations is to adopt a capability that enables bi-directional communication, such that the API itself can initiate a request to the API consumer(s). This would not only allow API consumers to be notified when certain events occur at the server side (for example, a change in a customer record) but would prevent the constant polling of an API.

The most common way to implement this capability is via **Webhooks**. Webhooks enable API consumers to subscribe to specific API events, for example, changes in data for a specific resource, and during the subscription process (which is typically just an API POST request), API consumers provide a

call-back URL that is subsequently used by the server to push the events.

Webhooks will be explained in greater detail in chapter 5, API-Led Architecture Patterns.

API load balancing

In traditional API gateway implementations, there tends to be a load balancer situated in between the gateway and the service endpoints. The problem with this architecture is that unless the load balancer itself is capable of automatically detecting added and/or removed service endpoints (for example, during a service scale up or down), customizations are required in the load balancer infrastructure in order for it to be dynamically configured for such rapid and sudden changes.

Instead, this capability refers to the ability of an API gateway to also act as a client-based load balancer, thus removing the need for a load balancer in between. The consequence of this is that the API gateway should have the ability to introspect a service registry in order to dynamically determine the active endpoints of a service at any given point in time. Therefore, when a request does come in, the gateway itself can load-balance to all active endpoints.

The following article authored by Phil Wilkins explains in a great level of detail how this capability could be implemented:

<https://paascommunity.com/2018/03/25/registries-use-cases-for-api-management-and-microservices-by-phil-wilkins/>

Quotas and plans

A capability in support of API monetization, quotes, and plans refers to having the ability to define and enforce quotas and/or plans to a given API, against a well-defined audience. For example, user Luis is subscribed to the Gold Plan of the Orders API. The Gold Plan defines a quota of 10k API calls per day. Exceeding that quota may result in additional charges and/or simply a blockage of the service for the rest of the day.

Versioning and deprecation

This refers to having the ability to define and run multiple concurrent versions of an API. This is particularly important as APIs will evolve over time, and having the ability to manage concurrent versions of an API will enable API consumers to incrementally switch to newer versions of an API, so older versions can be deprecated and ultimately retired.

Custom policies

This is the ability to implement custom policies either via scripting languages, such as JavaScript or Groovy, or by means of an SDK. Regardless of the approach, this capability is about enabling the creation, deployment, and enforcement of bespoke policies.

Business capability services

As previously mentioned, business capabilities are delivered in the form of services, which in themselves are units of software that encompass a well-defined and bounded functionality.

In domain-driven design, such functionality is referred to as bounded context. The following book by Eric Evans describes this in a great deal of detail: <https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/>

A service can (and should) therefore implement business logic in support of things such as data transformation and validation, service orchestrations (in short, a single service that implements a process flow that involves calling other services to either enrich or validate payloads), or service choreography (communication with other services via events), to name a few.

Services should not be accessed directly but mediated via the API exposure layer, as this one extra layer of abstraction allows for common policies to be applied (for example, authentication, throttling, and so on), thus preventing the need to replicate policy functionality across services.

The following diagram illustrates the technical capabilities expected of the business capability services block:

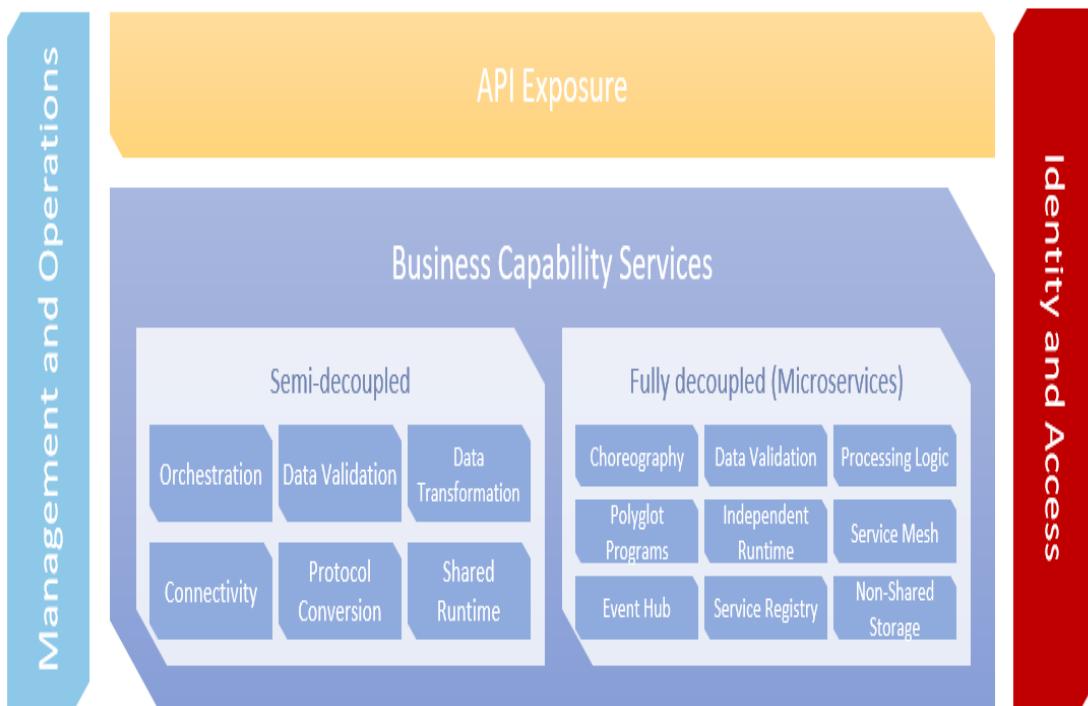


Figure 4.10: Business capability services building block

As you can see, the capabilities are sub-divided into two main groups of capabilities: semi-decoupled and fully decoupled services.

Semi-decoupled services

These are services that don't deliver enough isolation because they either share a common runtime, share common metadata, or simply are tightly coupled to each other (meaning that a service binds to another without any sort of runtime decoupling). Because these services typically share a common runtime, they can't really scale independently of each other and faults have the potential to propagate across the system.

The capabilities expected of this sub-block typically include:

Orchestration

Service orchestration, a well-known capability in traditional service-oriented architectures, typically refers to the ability of a service to coordinate multiple calls to other services in order to deliver a pre-defined process. As the orchestration itself happens within another service, such functionality is also accessible via well-defined interfaces (for example, via a REST endpoint).

Data validation

As the name suggests, data validation refers to the ability to validate data that is structural (for example, compliance with pre-defined schemas), semantic (for example, consistent use of terms), or functional (for example, verifying that payloads conform to business rules).

Note that this capability does not imply having to implement business rules in the form of a rules engine; however, this may be an additional capability that's needed depending on the type of validation required.

Data transformation

This refers to the process of converting data payloads from one structure (for example, a JSON schema) to another. Data transformation is typically combined with an orchestration capability in order to transform payloads resulting from the orchestration itself into a common data object, which is then sent back to the consumer of the service.

Connectivity

A core capability within any service infrastructure, connectivity refers to the ability of a service to connect to multiple backends (for example, databases, ERP systems, messaging systems, and others), using a variety of protocols. For example, it should be possible to establish connectivity with sources such as Oracle databases, NoSQL databases such as MongoDB or Cassandra, SOAP endpoints, REST endpoints, Kafka Brokers, and JMS topics/queues, to name a few.

Protocol conversion

This is the ability to convert between the underlying protocols used to transport data, for example, converting from HTTPS to SQLNET or vice versa. This capability is commonly used along with connectivity, as in many cases this implies converting between protocols.

Shared runtime

Shared runtime refers to allowing (but not restricting) one or more services to share a common runtime. For example, this could be two Spring services running on a single application server, or multiple containers running as a single POD in the case of Kubernetes infrastructure.

Fully decoupled services

Otherwise known as microservices, these are highly autonomous and self-contained services built in compliance with the microservices architecture patterns and that deliver a well-defined and bounded business capability. Therefore, they offer a much higher degree of isolation, flexibility, and scalability. For example, faults typically won't (or shouldn't) propagate outside of a service-bounded context. Also, these services can scale independently of each other, as they don't share a common runtime.

Each service is responsible for its own data and only interacts outside its bounded context via events. A service mesh (described subsequently) is typically implemented as a means to isolate faults within a bounded context, so services can bind to one another but still achieve runtime decoupling. By definition, these services can be implemented in a variety of programming languages and typically run independent runtimes (later described).

The capabilities expected of this sub-block are therefore as follows:

Choreography

This refers to a style of interaction whereby services interact without the need for a central coordinator (also known as orchestration engine). Instead, participants of a choreographed process consume and react to events that are relevant and also produce events other participants (services) can be interested in. The natural implication is that each service must be capable of consuming and producing events in addition to being smart, as it becomes a service's responsibility to execute any business logic as a result of a given event.

As with choreographies, a service doesn't bind directly to another. Instead, all interactions are event-driven and asynchronous, and this capability is typically supported with the use of an Event Hub as the core infrastructure where events are published and consumed from.

Data validation

This is similar to the previous description; however, in this case, if the execution of business rules is required, it must either be self-contained within the logic of the service itself or as a sidecar.

Sidecar is a pattern typically adopted in container orchestration environments (for example, Kubernetes clusters) whereby a container is attached to another (like a sidecar attached to a motorbike) in support of specific functionality. The following article describes this pattern in great detail: <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>.

Processing logic

In microservices architectures, a service should be self-sufficient and self-contained. In architectural terms, this means that all logic that belongs to a given bounded context should sit within a service boundary itself. In practical terms, it means that the service implements all of the business logic that's within the scope of its own context.

This capability aligns well to the generally accepted notion that in microservices architectures, it is the endpoints that should be smart and not the transports used to get access to them (the pipes).

Polyglot programming

This is a key principle in microservices architectures to enable services to be developed in a variety of languages and technologies. This capability therefore refers to having the ability to write and execute programs written in a variety of languages, for example, Java, JavaScript, or Python, without any major restrictions. In other words, it allows the best tool for the job to be adopted.

*Although in theory the idea is to be able to run programs written in any language, in practice, the industry seems to be favoring the use of some languages, especially in combination with the use of microservices frameworks as aids. For example, **Spring Boot** in the case of Java, **Node.js/Express** in the case of server-side JavaScript, and **Flask** in the case of Python.*

Independent runtime

In microservices architectures, services should be fully decoupled, not just in terms of specific business functionality (bounded context) but also with regard to where they run. For example, if all services share a single monolithic runtime (for example, an ESB), then in order to scale a single service, the entire monolith has to be scaled. Likewise, if the common runtime (the ESB) goes down, then all services are equally affected.

This problem becomes even more obvious when services from different bounded contexts share the common runtime. In order to separate concerns and avoid putting all eggs in one basket, each service must therefore run in its own runtime, thus allowing services to be independently:

- Scaled or shrunk
- Deployed or undeployed
- Restarted or stopped
- Monitored

Kubernetes.io is an open-source container orchestration platform originally developed by Google, and is fast becoming the technology of choice for running fully decoupled services.

Service mesh

Having services that run independently of each other is undoubtably an important and fundamental capability within a Microservices Architecture. However, having an all-encompassing and single executable unit in support of a service bounded by a context can also be limiting and in many cases undesirable, as it couples different aspects of a service that may have to undergo change and deployments at a different pace, or might even have different scalability requirements, as is the case between reads and writes, where reads often have a much more dependent scalability need.

For this and many other scenarios, a service within a single-bounded context might have to be broken down into smaller (executable) pieces that bind directly to each other to collectively deliver the bounded-context functionality. For example, a subscriber agent responsible for consuming events from an Event Hub might not implement the event processing logic. Instead, it might bind to another service to dispatch the event for further processing. To this end, a bounded context may have many physical services that together form the bounded-context *logical* service.

However, a service that directly binds to another service is also hard-coupling to it. This introduces notable downsides, as it

not only becomes difficult to evolve the software (as all inter-dependencies need to be considered), but it also introduces more risk to the solution, as faults can easily be propagated across services, thus bringing down the entire functionality of the context.

In order to overcome the preceding issues and others, a mechanism to decouple inter-service communication is needed, one that introduces a reliable, secure, fast, and scalable inter-service communication channel (the transport), but that can also prevent faults from being propagated. This smart inter-service communication infrastructure is referred to as a service mesh.

A service mesh should deliver the following capabilities:

- A reliable, efficient, secured, and fast transport infrastructure that ensures that services can always reach one another.
- Dynamically discover new services and their active instances through the use of a service registry (later described).
- Dynamically route and load-balance requests to all or specific active instances of a service (especially as instances are added or removed on the fly). For example, it might be desired to incrementally introduce a new version of a service by only routing a certain percentage of its traffic to a newly deployed version.

- Prevent faults from being propagated by implementing bulkheads and circuit breakers so service instances that are underperforming and/or malfunctioning can be isolated, thus preventing faults from spreading to all services in the bounded context.

Note that a service mesh is typically delivered as part of an independent runtime and a service registry (described subsequently).

A popular choice for implementing a service mesh in Kubernetes infrastructures is [Istio.io](#). Another popular choice for a Java-based service mesh is Hystrix, which was originally developed by Netflix but also open sourced. Do note, however, that this is an evolving market with several options available, and many others being created by the day.

Event Hub

As previously described, fully decoupled services interact via asynchronous events (choreographies). An Event Hub delivers the infrastructure capabilities required to publish and subscribe to events. Such a capability ensures that events can be reliably sent and received by one or many subscribers.

An Event Hub should offer the following capabilities:

- **Event store:** The ability to persist events, permanently if desired, as a series of immutable logs organized over time.
- **Reliable messaging:** Guarantees that a message has been successfully dispatched from the source to all targets.
- **Publish and subscribe to events:** This capability refers to the general availability of client libraries (in multiple programming languages) that can be adopted and bundled within an application (for example, a microservice) to publish and subscribe events using the Event Hub.
- **Connectors:** The ability to natively connect multiple sources or targets, in a variety of protocols (for example,

SQLNET, JMS, or AMQP, to name a few), directly into the Event Hub. This capability is typically used, for example, when implementing a **Change Data Capture** pattern, whereby it is possible to detect and propagate to multiple target changes in a source database.

- **Event processing:** The ability to introspect in real time the inflow of data, using predefined queries in order to detect patterns in the data that can be acted upon.

Apache Kafka, originally developed by LinkedIn and now also open sourced, is by far the most popular technology used to implement an Event Hub.

Service registry

This is a form of key/pair storage typically used in fully decoupled service infrastructures to store runtime (for example, active service endpoints and status) and configuration metadata (for example, environment variables and other application properties).

There are many use cases for such a capability. However, in the context of fully decoupled services, a registry can be used by a service to obtain configuration metadata during startup and to register its runtime metadata (for example, HTTP endpoints) once it is up and running. The registry can also be used by other infrastructure components (for example, API gateways or a service mesh) to dynamically determine the status of a service and dynamically route requests to active and healthy service endpoints.

*Note that service registries are **not** to be confused with UDDI registries. Although there is a level of similarity, the former can be considered legacy and never really enjoyed the popularity or level of adoption as modern registries, which are simpler and more straightforward to adopt.*

There are some of the capabilities expected of a service registry:

- **Resource registration:** Refers to the ability to register a service/API endpoint in the service registry by calling a registry's management REST API.

- **Resource discovery:** Refers to the ability to dynamically discover what services are registered in the registry, including their status and active endpoints. This capability is critical to implementing application-based client load balancing.
- **K/V storage:** A key-value store is collections of arbitrary name/value pairs (entries) that can be accessed at runtime by API proxies or load balancers.
- **K/V replication:** The ability to replicate the key-value store across different instances of a service registry. This is specifically important not just for highly available deployments, but also deployments whereby service endpoints exist in multiple geographies and thus a registry must be deployed in multiple data centers.
- **Resource health-check:** The ability to allow services to provide continuous status checks and the ability for service consumers to query the status of services and their endpoints.
- **Secret vault:** The capability to securely store and access secrets. A secret is anything that requires tightly controlled access, such as API keys, passwords, and certificates, among others. The vault provides a unified interface to any secret, while providing tight access control and recording a detailed audit log. Note that a secret vault can be (and typically is) considered a capability in its own right.

*Service registries have become very common as a fundamental and internal component of modern infrastructures. For example, Apache Kafka uses **Apache Zookeeper** as its internal registry. Kubernetes makes use of **ETCD** as its internal registry. There are many popular open source registries (such as the ones earlier mentioned) and commercial options too, such as HashiCorp's **Consul**.*

Non-shared storage

In microservice architectures, by definition, each service owns its data. The implication of this is that each service therefore requires some form of persistence. This form of storage, typically a database or an in-memory cache, is effectively part of the service infrastructure and only directly accessed by the service itself, meaning that it is inaccessible by outside consumers. As is the case in programming languages, there are also several types of persistence options, therefore this capability should also be polyglot, meaning that it should be possible to, for example, adopt:

- A relational database (for example, **Oracle**, **MySQL**, and **PostgreSQL**) when it is desired to persist structured data and/or provide support for transactional workloads.
- A document store (for example, **MongoDB**, **Couchbase**, and even **Elasticsearch**) when the desire is to persist and access semi-structured data as documents, for example, in JSON format.
- An in-memory cache (for example, **Redis**, **Memcached**, and **Hazelcast**) when the need is to serve data faster directly from memory and without having to hit the database.

- A graph database (for example, **Neo4j** and **Neptune**) when the need is to store and access data as a collection of nodes (an entity, for example, a person, business, or a thing) and edges (connections and relationships between nodes).
- A data lake (for example, based on **Hadoop**) when the data accessed is unstructured and/or exists in vast volumes.

Note that many other forms of persistency exist. However, the ones mentioned here provide a summary of the most common ones.

Identity and access

This block refers to capabilities that aid the creation, management, auditing, and verification of user identities, their roles, and the associated user rights. In the context of the API-led architectures, the block ensures that only genuine, authenticated, and authorized users can interact with any of the capabilities available in all blocks.

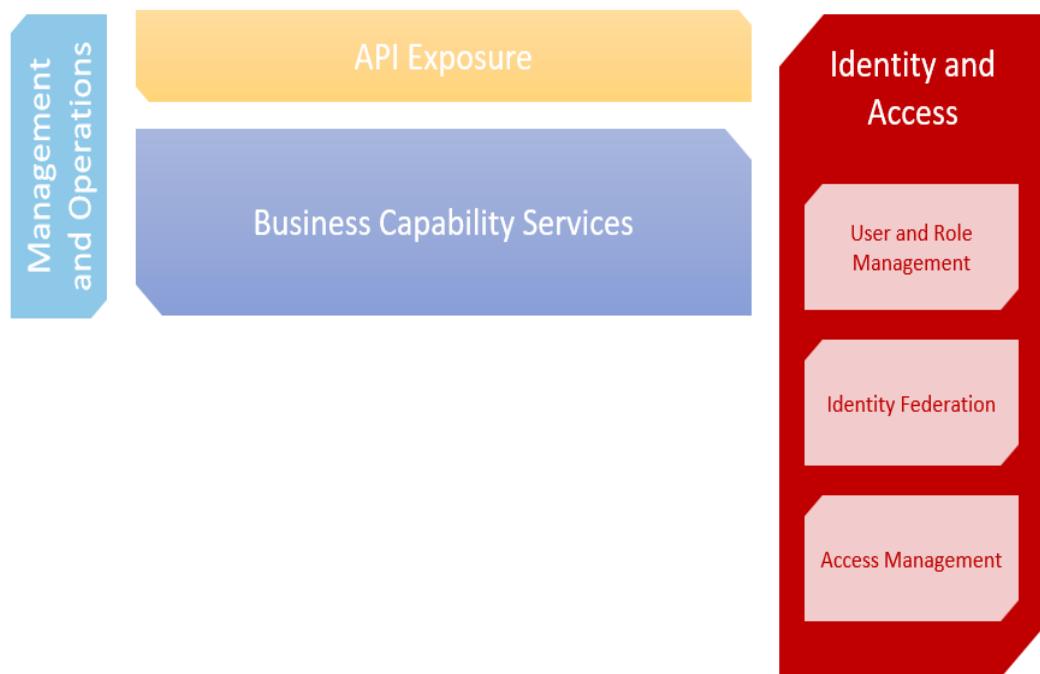


Figure 4.11: Identity and access

The main capabilities that are offered by the block are:

Users and roles management

This refers to having the ability to create, change, and manage user accounts and their respective roles and responsibilities. In some cases, this capability also requires users to be synchronized from external corporate directories.

Identity federation

When the requirement is to allow seamless access to users whose accounts reside in an outside security domain to that of the application (meaning that synchronizing identities using LDAP, for example, is not an option), other protocols come into play, most notably SAML, which allows trust to be established with an external identity provider. Once the trust is configured (in short through a process of exchanging security certificates), then it is possible for users whose accounts reside in the identity provider to also be granted access to the main application in question.

Access management

In the context of API-led architectures, the main access management capability required is having the ability to issue, revoke, validate, and introspect user tokens (for example, based on JSON Web Tokens) and adopt open standards, such as OAuth 2.0 (including all of its grants) and OpenID. For this to be possible, then this capability must offer an authorization server that takes in the different authorization flows.

Summary

This chapter started by delivering a comprehensive overview of what API-led architectures actually are, including a detailed explanation of the fact that there isn't one, but at least two main types of API: single-purpose, focused on delivering tailored functionality in support of a specific and well-known digital experience, and multi-purpose, a more generic functionality aimed at solving many use cases (most of which are probably not known at the time the API is conceived).

The chapter then continued to explain that APIs are like doors but to information and functionality, and that these doors (the APIs) are accessible through a set of API exposure capabilities, responsible for providing fast, secure, and reliable access to such assets.

It was later explained, however, that the heavy lifting doesn't (and shouldn't) happen in the API exposure block, and that any business functionality offered by the API should be implemented as a distinct and discrete service that can be easily associated with a known business capability. It was explained that services also come in different flavors, and that depending on their implementation style, they can be semi-decoupled, meaning that services share a common application infrastructure and can directly bind to each other, or fully

decoupled (also known as microservices), where the services don't share an application infrastructure, own their data, and use events as the main means to interact with other services.

At this point, it was also explained that other management and operations capabilities in aid of a full API life cycle were required, in terms of design-time capabilities, for example, being able to adopt an API design-first approach or deliver rich API pages that offer a great DX. The chapter also highlighted the need for runtime analytics, monitoring, and identity and access management capabilities.

The chapter then continued by providing a thorough explanation of all of the different technical capabilities expected of each block.

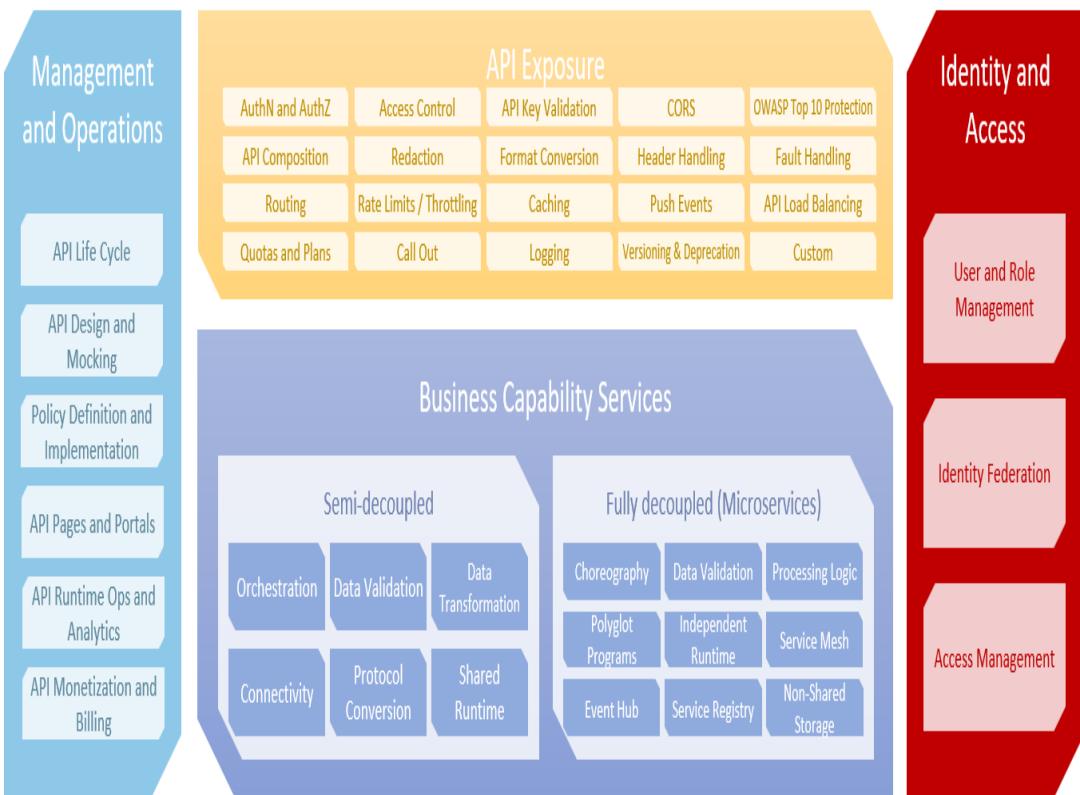


Figure 4.12: API-led technical capability model

The next chapter will put all of these technical capabilities into perspective by illustrating how common API implementation patterns combine some of these capabilities in order to address common challenges.

API-Led Architecture Patterns

The objective of this chapter is to elaborate on how the different capabilities of an API-led architecture, as described in the previous chapter, can consistently address requirements. Throughout the chapter, you will learn how these different capabilities can be combined and applied in order to realize different API-led patterns, each of which will have its own context, benefits, and subject area.

Patterns in the context of APIs

Design patterns, in a nutshell, define common solutions to address re-occurring problems.

For a more elaborate definition of design patterns, refer to the following link:

https://en.wikipedia.org/wiki/Software_design_pattern

In the context of API-led architectures, patterns deliver common approaches to implement and combine technical capabilities (the majority of which were defined in [Chapter 4, API-Led Architectures](#)) with the objective of addressing common requirements.

An example is the need to access information in real time. In a world that's becoming more digitized by the day, the need to access information at any time, from any location and device, can only increase. However, many challenges may arise when addressing this one requirement. For example:

- Due to the sensitivity of the data being accessed, does it require strong security controls? Alternatively, is it just public information? What about the **General Data Protection Regulation (GDPR)** and **Payment Card Industry (PCI)** compliance?

- Is the information accessed in batches (for example, the application will download the data and use it offline), or could it be on-demand and bidirectional?
- Where exactly does the information live (for example, cloud or on-premise) and where is it accessed from? Could there be latency issues?
- The type of backend system that holds the information: is it a legacy system, a **Software as a Service (SaaS)** application, or a custom database? Can it handle the load?
- The expected throughput and peaks: can all solution components, such as API gateways and services, be easily scaled?
- What about monetizing on APIs based on calls?

We can conclude that patterns can be applied as a vehicle to deliver proven, consistent, and repeatable solutions that address our needs. Nevertheless, given the nature of each point, it can also be deduced that there won't be a single pattern that can solve all problems. Instead, different patterns can be applied/combined to address requirements of different natures. For example, how an API handles authentication and authorization is more of a security concern than how an API gateway can be applied in order to route access to multiple backend endpoints, which is more of a mediation concern.

The following diagram illustrates the four main types of patterns identified in this book as being key in API-led architectures:

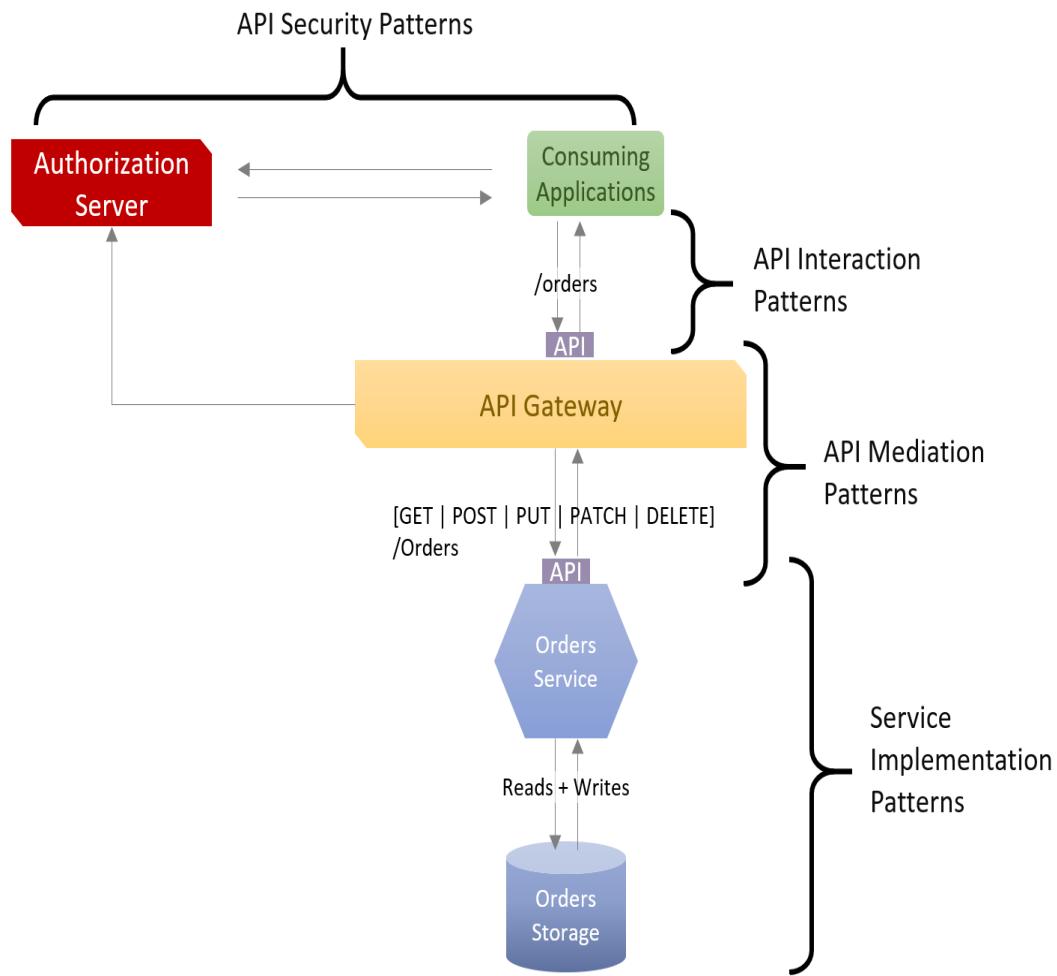


Figure 5.1: API pattern types

As you can see from the diagram, there are four main types of patterns:

- **API security patterns:** They encompass solutions that address common authentication, authorization, and other security-related requirements. An example is a bearer-of-key pattern.

- **API interaction patterns:** Their main focus is to define common approaches for how an API consumer interacts with an API and its endpoint. For example, most REST APIs adopt limits and offsets to allow consumers to paginate through API content, a pattern typically referred to as API pagination.
- **API mediation patterns:** They are centered on the API exposure and they encompass common routing patterns; for example, by using an API gateway.
- **Service implementation patterns:** They focus on the business capability services block and go a level deeper to illustrate common approaches that combine service capabilities in order to address important non-functional requirements.

API-led architecture patterns described

The patterns in this section aim to address common requirements/challenges that arise when adopting API-led architectures. Although they are not specific or tied to any API architectural style (for example, **Representational State Transfer (REST)**, **Graph Query Language (GraphQL)**, or **Google Remote Procedure Calls (gRPC)**), some may be better suited to a specific style, and this is indicated accordingly. Also note that [Chapter 6, Modern API Architectural Styles](#), offers a comprehensive overview and comparison of the trendiest API architectural styles (at the time the book was written).

Each pattern listed is described as follows:

- **Name:** A meaningful and self-explanatory (as much as possible) name for the pattern.
- **Problem statement:** Provides context on the challenge/situation that the pattern aims to address.
- **Solution:** Describes the pattern through a meaningful illustration and a text narrative.

- **Drawbacks:** Elaborates on any potential drawbacks of the pattern.
- **Tags:** Which types of pattern apply (for example, security, interaction, mediation, or implementation).
- **Applicability:** An indication of what API architectural style (for example, REST, GraphQL, or gRPC) best suits the pattern.

The following table summarizes the patterns that will be described in subsequent sections in more detail.

Pattern name	Main pattern type	Brief description
API resource routing	Mediation	Uses an API gateway to route calls based on unique resource identifiers (URIs) .
API content-based routing	Mediation	Uses an API gateway to route calls based on the

		content of a call (for example, HTTP header or message body) instead of just the URI.
Payload pagination	Interaction	Implements limits and offsets parameters in an API to allow pagination on a large set of records.
Create, Read, Update, and Delete (CRUD) API service	Implementation	A service, proxied by an API gateway, that implements CRUD operations against a single database.
Command Query Responsibility Segregation	Implementation	Implements reads and writes for a given entity as separate services, each with a

(CQRS) API service		datastore. Uses an API gateway to present the service externally as a single API.
API aggregator	Implementation	Performs operations (for example, queries) against multiple services with a single HTTP request/response call.
API orchestration service	Implementation	Implements a process flow as a series of activities all executed and coordinated by a single process runtime.
API microgateway	Meditation	A lightweight API gateway implemented as the

		entry point to services in an independent runtime, such as Kubernetes.
Sidecar API gateway	Mediation	Implements an API gateway as a container attached to a service in an independent runtime, such as Kubernetes.

Webhook	Interaction	Asynchronous APIs that allow API consumers to be notified (called back) when a change of state takes place in a given record or set of records.
API geo-routing	Mediation	Routes API calls to the nearest API gateway based

		on where they originate.
API firewall	Security	Implements a web application firewall (WAF) as the first line of defense before calls even reach the API gateway.
API basic authentication	Security	Implements HTTP basic authentication as a policy in the API gateway.
API bearer of token	Security	Implements an authorization flow based on OAuth 2.0 and using tokens.
API bearer of obscure token	Security	Similar to the API bearer of token but with an additional step to obscure and de-obscure tokens.

API resource routing

Problem statement:

As the number of services increases, so does the number of HTTP endpoints exposed to access them. From an API consumer standpoint, this adds complexity not only because keeping track of all endpoints becomes difficult, but also because any changes made to a URL will result in changes on the consumer side. Furthermore, common requirements, such as authentication/authorization, throttling, and rate limiting, have to be implemented separately by each service, which isn't optimal and may lead to inconsistencies and redundancy.

Solution:

Implementing an API gateway as the only entry point to all services means that API consumers only have to be aware of one URL domain. In this way, it becomes the API gateway's responsibility to route the traffic to the corresponding service endpoints and also enforce any applied policies.

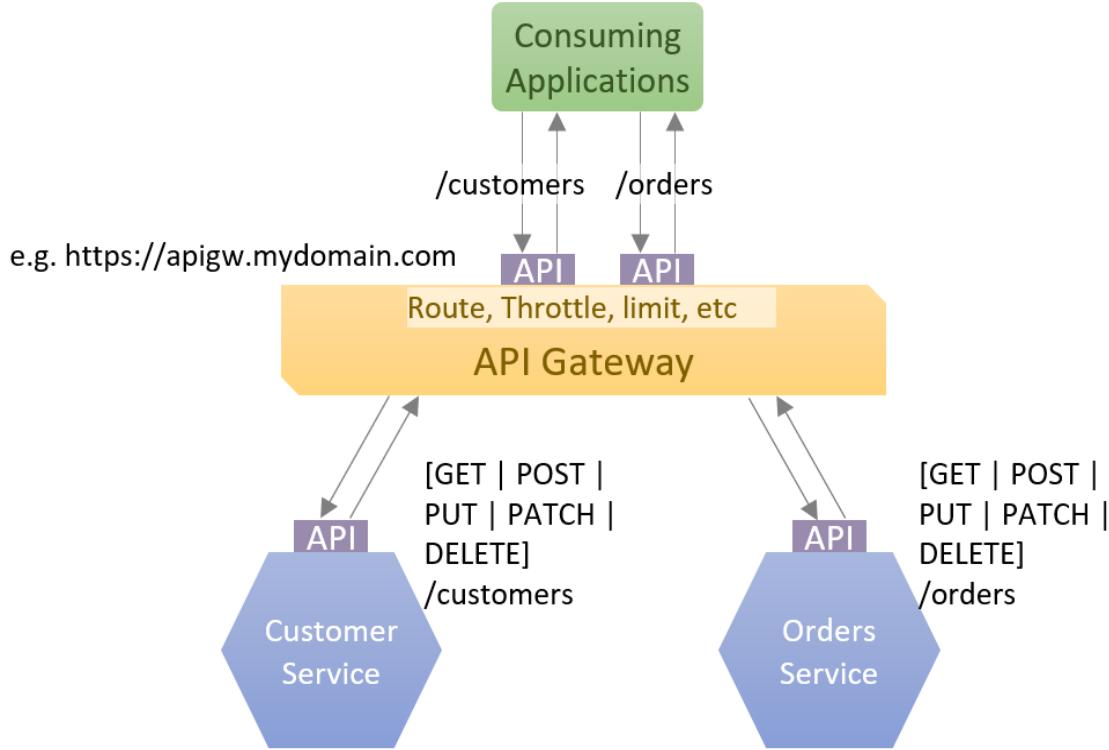


Figure 5.2: The API resource routing pattern

The preceding diagram illustrates how an API gateway routes the inbound HTTP(s) traffic (in any HTTP verb) corresponding the resource `/customers` to the customer service and `/orders` to the orders service. The diagram also indicates that in order do this, a routing policy is enforced in the API gateway. Other policies, such as throttling and/or rate limiting, could also be applied alongside the routing policy.

Drawbacks:

A common criticism of this pattern is the fact that an additional middle tier has to be introduced, which may lead to additional complexity and costs.

Tags: Mediation (main) and implementation (supporting).

Applicability:

This pattern is suited to REST APIs but can also be applied to other API architectural styles (for example, GraphQL or gRPC) depending on the capabilities offered by the API gateway.

For example, a GraphQL service is also accessible via an HTTP(s) endpoint, but there is no concept of URIs in GraphQL. Its one endpoint has access to all operations. So, unless the API gateway is able to interpret GraphQL payloads, it won't be able to perform operation-level routing (which is straightforward in REST based on HTTP verbs).

In the case of gRPC, it's a similar situation. Unless the API gateway supports protocol buffers (Google's open-source approach for defining interfaces and serialized payloads), it won't be able to route traffic to gRPC-based APIs.

API content-based routing

Problem statement:

In some circumstances, and often for historical reasons, there might be multiple datastores for the same entity. For example, it is not uncommon for an organization to have multiple **Customer Relationship Management (CRM)** systems.

Although there might be plans to consolidate them into a single master version, until such consolidation takes place (which can take time), all the systems may have to be accessed when obtaining certain customers' records.

Another scenario is when database sharding is applied in order to distribute the load across multiple database instances. This technique is typically applied when the overall number of records stored is huge and a single instance struggles to cope with the load. Instead, records are spread across multiple instances.

In both scenarios, there is a common challenge: how to determine which data store to access at runtime for a given record.

Solution:

A solution is to implement multiple services, one per unique datastore, and adopt an API gateway as the only entry point to all services. You could then configure the API gateway to route calls to the corresponding service based on a key obtained either from the HTTP header or the payload.

The API gateway needs to be configured to dynamically determine at runtime which service to route to based on the key (meaning that some form of routing table will be required). For example, customer IDs starting from one to 100,000 are routed to service-x, and IDs from 101,000 to 200,000 are routed to service-y, and so on. The routing could also be based on the format of a key depending on the backend system where a record lives.

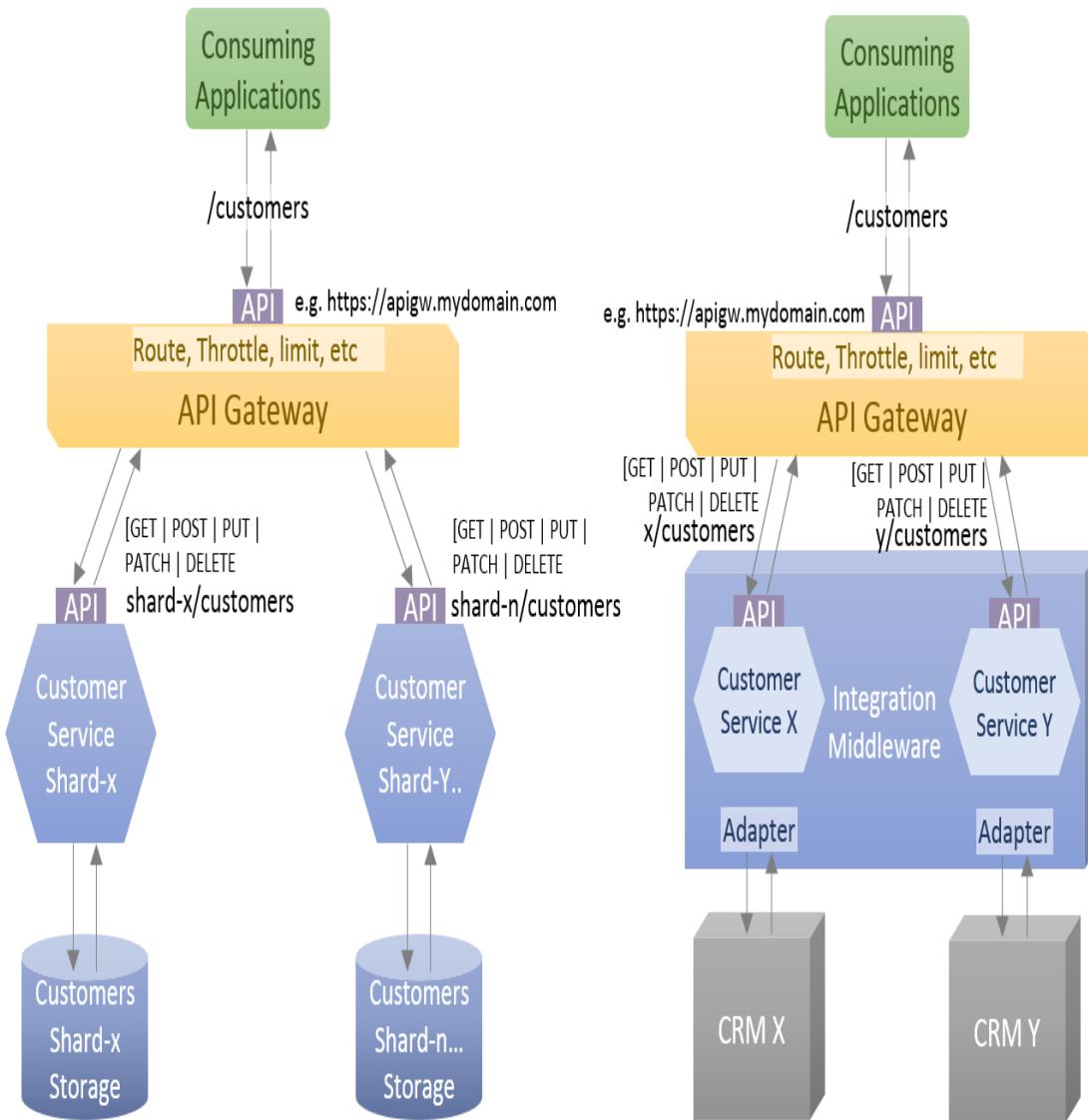


Figure 5.3: The content-based routing pattern

In the preceding diagram, an API gateway is exposing a single `/customers` resource for multiple customer services, each with a different data store.

Drawbacks:

Unless an API consumer sets a HTTP header value up front (which should happen in the API specification), inspecting the

body of a message on each call can result in slower response times.

Tags: Mediation (main) and implementation (supporting).

Applicability:

This pattern can apply to any API architectural style, though it can be very useful in GraphQL and gRPC, given that both make use of a single HTTP endpoint and routing must be done based on the content of a message (either HTTP headers or the message body).

Payload pagination

Problem statement:

A consuming application needs to consume large payloads of data on a pre-defined schedule (for example, daily batches or weekly batches). Although this is typically addressed via traditional batch-based integrations (for example, ETL or just a file transfer), in this case, the information is only accessible through an API.

Solution:

You can create an API that allows payload collections to be retrieved when the plural verb of the resource is accessed. For example, `/orders` should return the orders collection and `/customers` the customer collection. However, in order to prevent unbounded responses that could result not just in timeouts but also a degraded performance, limits, and offsets (typically as URL parameters) should be applied as the means to paginate through the responses, allowing API consumers to make as many API calls as required to download all the records needed.

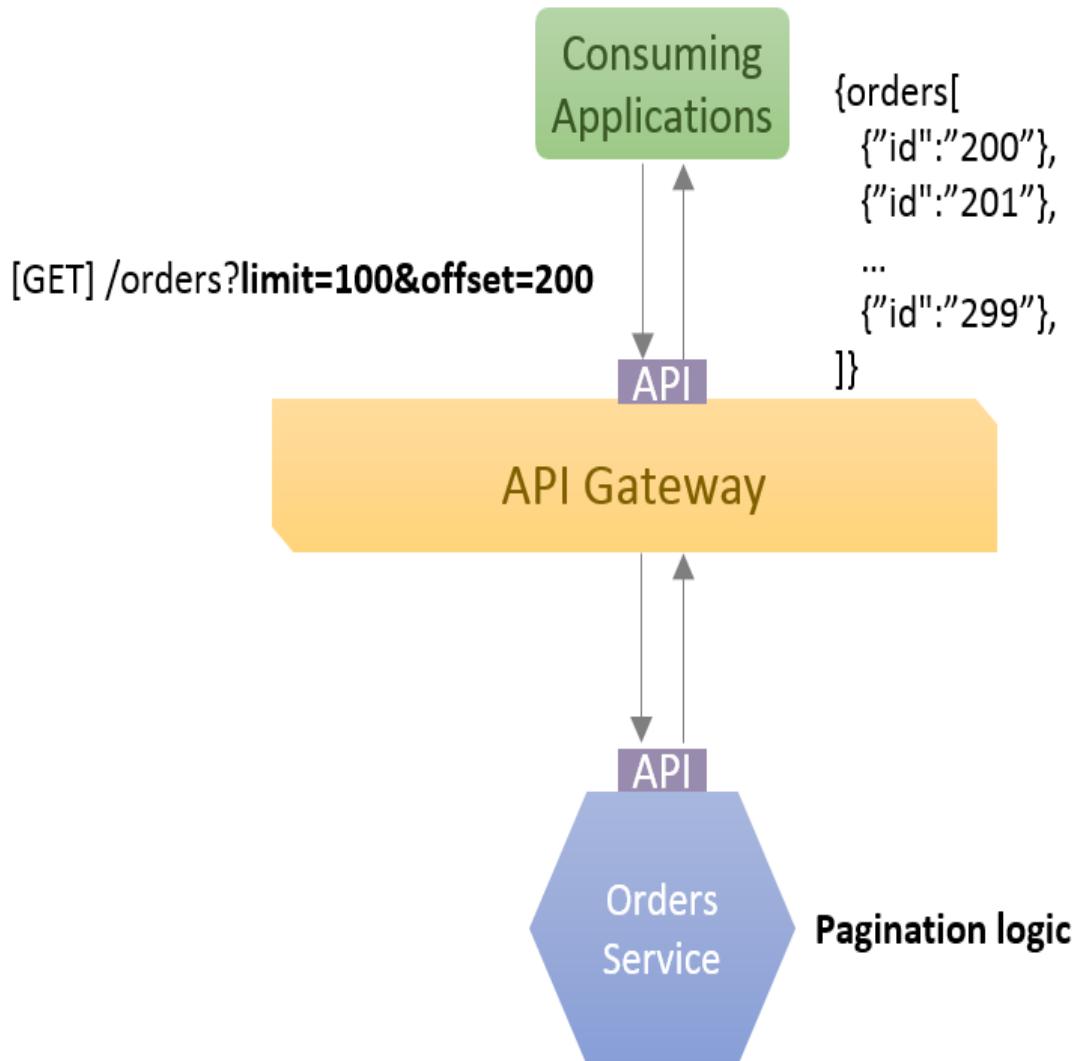


Figure 5.4: An API call in process

The preceding diagram illustrates how an API call made to resource `/orders`, with the HTTP parameters `limit=100` and `offset=200`, results in a response containing an orders JSON collection starting from record 200 (because of the offset value) up to 299 (because of the limit value).

Limits and offsets could also be combined with other parameters. For example, the parameter `change_since` could also be added to only obtain records changed from a given date.

Drawbacks:

The main drawback of this pattern is that it encourages request/response APIs as the means to deliver batch integrations. This can be highly inefficient, especially if the number of records is so large that it requires the API consumer to iterate through hundreds or thousands of API calls.

Another common challenge of this pattern is determining what constitutes an optimal limit. This can be challenging given that records may vary in size.

Tags: Interaction (main) and implementation (supporting).

Applicability:

Although the illustration is based on REST, the pagination pattern can be applied in any API architectural style (for example, GraphQL or gRPC).

CRUD API service

Problem statement:

A mobile, web, or device application has a requirement to perform **create, read, update, and delete (CRUD)** operations against a specific resource (for example, orders or customers). As the functionality is accessed from multiple channels, applications, and devices, a web API represents the best way to expose such a functionality.

Solution:

A CRUD API service pattern consists of an API gateway acting as a proxy (pass through) to a single service that implements all CRUD operations against a single database (relational or non-relational). Because a single service performs all operations, it can be relatively straightforward and simple to implement.

However, to avoid implementing capabilities such as authorization, throttling, and rate limiting directly in the service, an API gateway is used instead.

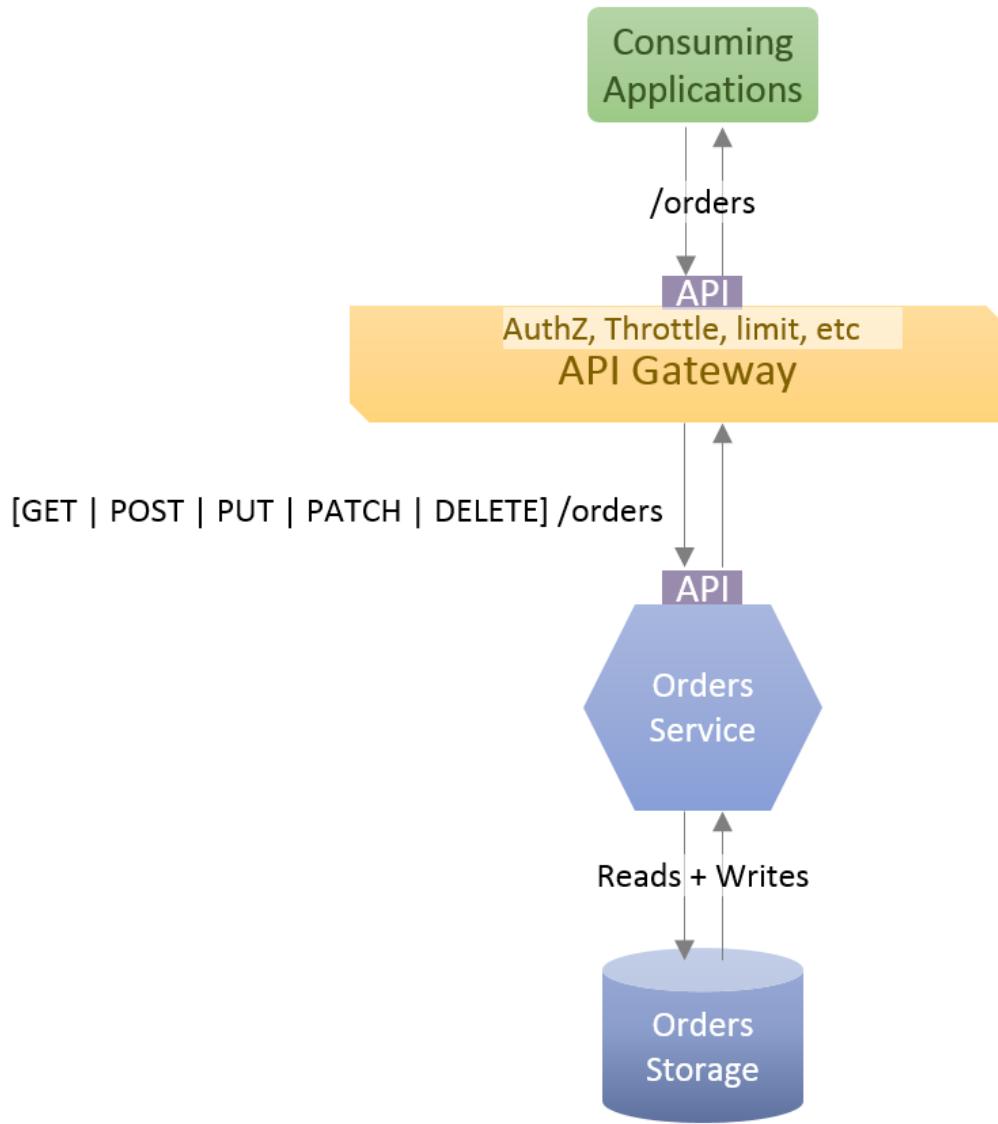


Figure 5.5: The CRUD service API pattern

The preceding diagram illustrates the pattern by showing how an API gateway routes calls made to the `/orders` resource (in all HTTP verbs) directly to the orders service. This basically means that the orders service is fully responsible for the support of all CRUD operations.

Drawbacks:

There are some known limitations to this pattern. Some well-known ones are:

- As volumes increase (especially during call peaks), the chances of concurrent write operations happening against the same resource will also increase. This may lead to undesirable behaviors, such as transaction locking causing write operations to fail or performance degradations as the number of queries increases.
- As APIs are consumed to satisfy different user journey requirements, the way data needs to be represented will likely change from use case to use case. Addressing such needs in this pattern is not straightforward as a single model to access the storage is used.
- Additional data integration may be required in order to synchronize changes made to the storage with other systems. This is true for changes made to other systems that may have to be reflected in the service storage.

Tags: Implementation (main) and mediation (supporting).

Applicability:

Although the illustration is based on REST, the same patterns can be applied in any API architectural style (for example, GraphQL or gRPC).

CQRS API service

Problem statement:

The main problem statement is similar to that of the CRUD API service pattern. However, in this instance, some or all of the following challenges might be true:

- Expected throughput is high and will continue to grow over time.
- The vast majority of calls are expected to be reads (for example, searches).
- Changes to the API resource(s) in context (for example, new records or updates) must be propagated to other systems.
- Different data representations are required in order to address the multiple API consumer needs.

Solution:

Adopting a **Command Query Responsibility Segregation (CQRS)** pattern helps, as instead of having a common service and storage supporting traditional CRUD operations, query

and upsert (updates or creates) responsibilities are split (segregated) into different services, each with its own storage.

Furthermore, an API gateway should also be implemented as a resource router, thus preventing API consumers from having to deal with different URLs depending on the action being performed (for example, read, create, or update).

*This CQRS pattern, as it is known today, was first introduced by Greg Young and was inspired by Bertrand Meyer's **command-query separation principle**. Since its introduction, the pattern has gained a lot of popularity and several resources can be found online describing its many flavors. The following link is a good source describing Young's original thinking behind the pattern:*

https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf

This pattern is typically combined with **event sourcing**, as it ensures that all changes made to a resource state are stored as a sequence of immutable events in an event log. This allows the reconstruction of the resource's latest state by just playing back the events. This pattern is very useful not just because it enables different systems to consume resource state changes as a series of events in the log via an Event Hub capability, but also because it allows for multiple query or command models to be implemented separately, thus satisfying the need to support different data representations that are easier to scale.

Note that event sourcing is not individually covered in this chapter. However, the following link describes it well: <https://microservices.io/patterns/data/event-sourcing.html>

It's also important to highlight that sourcing **commands** (messages that instruct an action to take place) and **events** (messages that reflect a change of state, for example, that an

action has taken place) means that the read storage won't be immediately updated as part of the transaction (for example, as in the CRUD service). Such a delay in storage reflecting the latest state of a resource is referred to as **eventual consistency**, and its implications should be factored into the overall system design. For example, a read call may not reflect the latest state of a record.

More on eventual consistency can be found by following this link:

https://en.wikipedia.org/wiki/Eventual_consistency

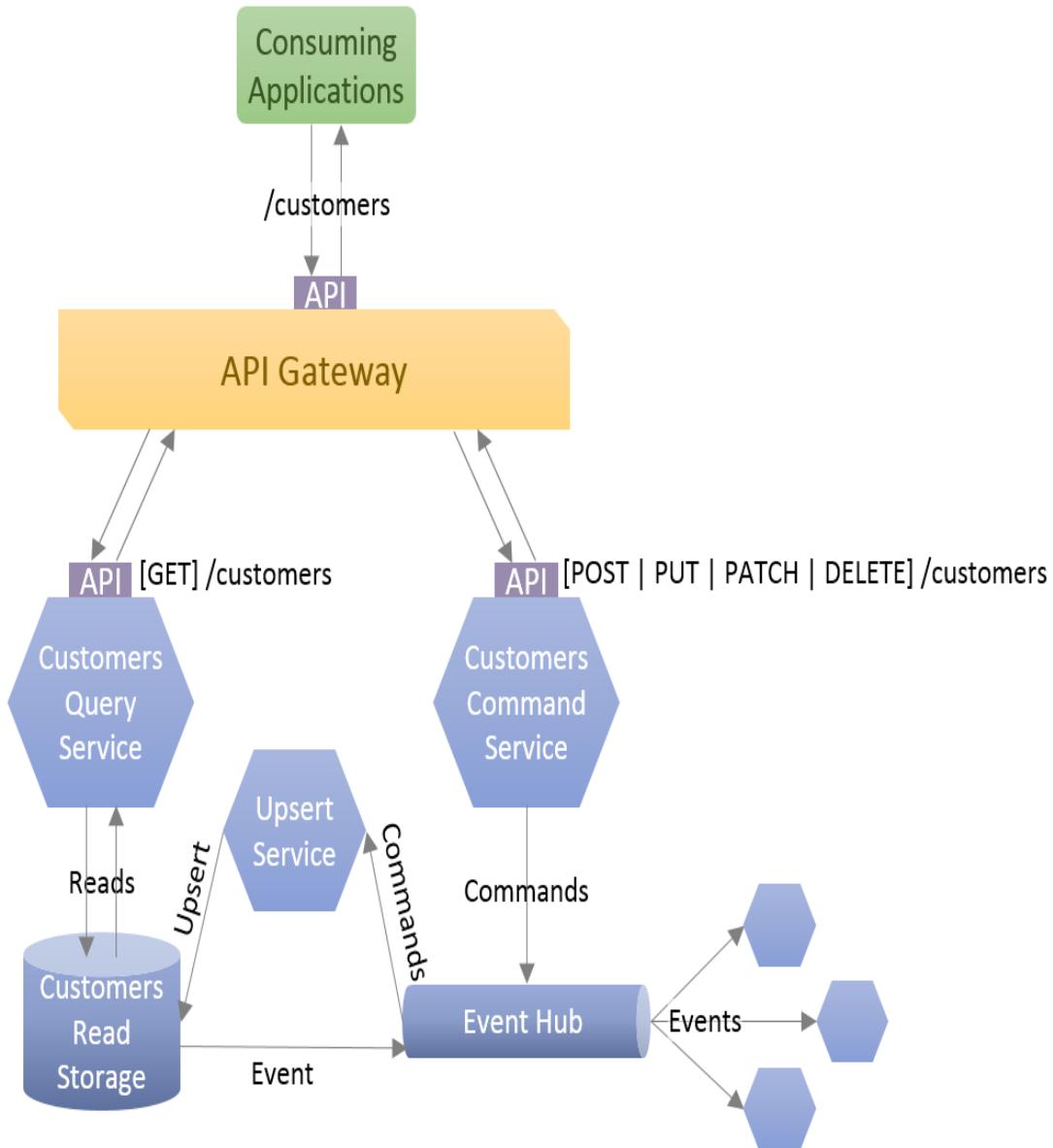


Figure 5.6: The CQRS service API pattern

The preceding diagram illustrates the pattern first and foremost by showing how an API gateway implements resourcing routing to route read calls to the customer's query service and upsert calls to the customer's command service.

The diagram also shows the customer's query operations performed against an orders read-only storage and the

customer's command operations persisted in an Event Hub capability. They are then picked up by an upsert service responsible for upserting (create, updates, and logical deletes) the read storage. Once an upsert action takes place, an event is generated that can be consumed by other services interested in any changes of state in customer records.

Lastly, as the query and command data models can be different, the upsert service may also have to deal with semantics and data transformations.

Drawbacks:

Although the pattern clearly has some notable advantages, there are also some disadvantages, such as:

- Increased complexity of implementation, especially when compared with traditional CRUD services.
- Multiple flavors of the same pattern add to the complexity.

Tags: Implementation (main) and mediation (supporting).

Applicability:

Although the example is based on REST, it can also be applied to other API architectural styles (for example, GraphQL or gRPC) depending on the capabilities offered by the API

gateway (content-based routing may be required instead to route based on an action defined in the message body or header value).

API aggregator

Problem statement:

A consuming application (for example, a mobile or JavaScript browser app) has little choice but to make multiple calls to different APIs in support of a single user journey. Take, for example, an eCommerce mobile app. When a user opens the app and the main page loads, typically a summary of the customer details, the latest orders placed, and even the loyalty status level and points are displayed all on the same page. However, the information displayed won't come from the same source and thus multiple APIs have to be called in order to collect all the information required and subsequently present it in the desired format.

This approach can result in inefficiencies, such as increased complexity in the client-side code, over-utilization of network resources, and even poor user experience as the application is more exposed to latency issues.

Solution:

Instead of having a client application making several calls to multiple APIs, an API aggregator does this on behalf of the consumer on the server-side. Furthermore, as the requirements

will likely differ from case to case, the aggregator should be either very flexible, thus allowing the client to define exactly what data they want and from where, or a single-purpose service should be built per use case to satisfy the client's needs.

There are a few considerations to be made when adopting this pattern:

- The aggregator must accept as input all information required so it can construct and make all subsequent calls.
- The aggregator must understand all data structures from all APIs that it interacts with.
- The aggregator must be able to transform the response payloads so they can be sent back to the caller as a uniform payload tailored for the consumer.
- The aggregator should be stateless and fully decoupled so it is also suitable for addressing high-volume requirements.
- An API gateway acting as a service proxy must implement common policies (for example, authorization, throttling, and rate limiting).

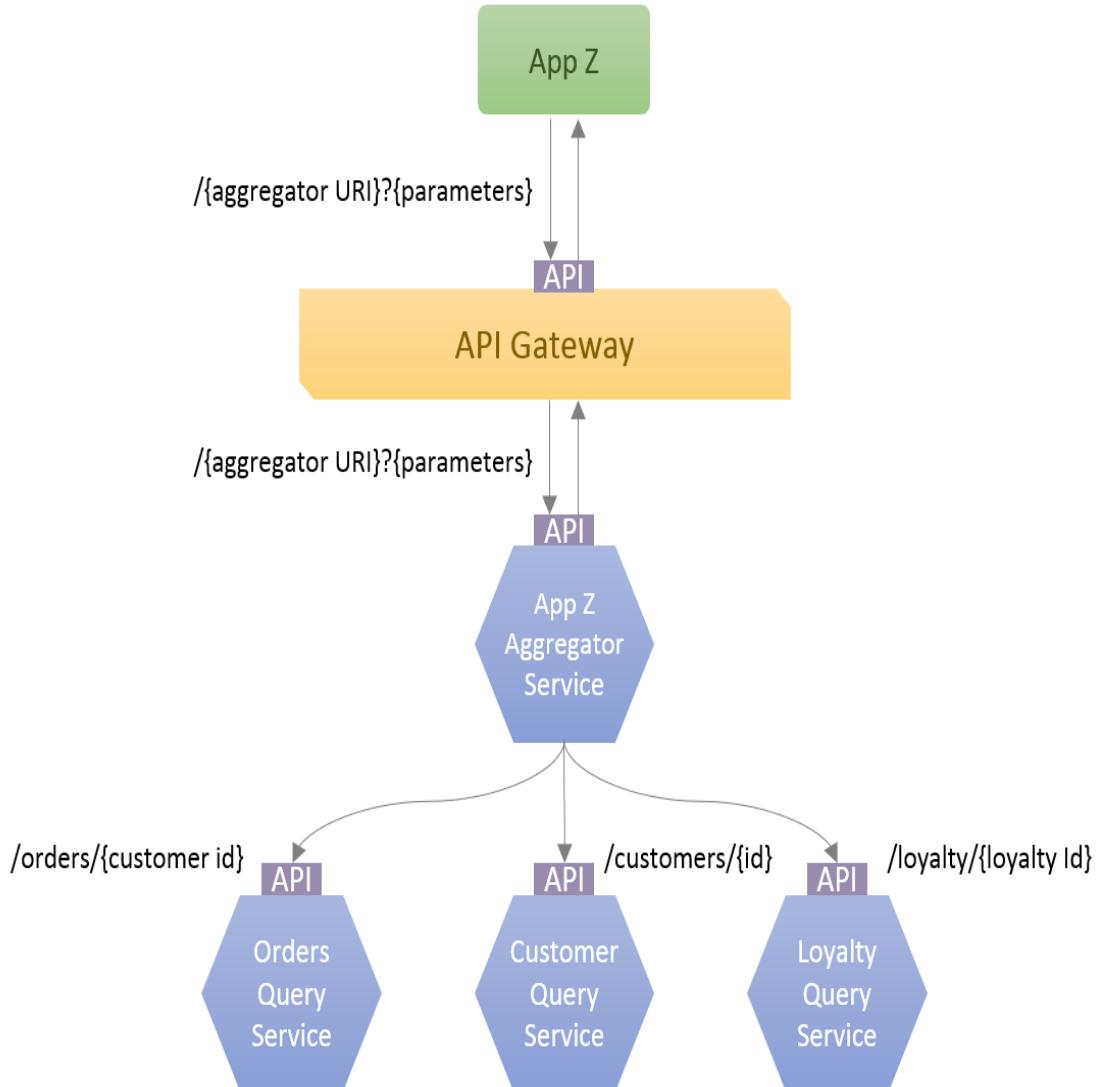


Figure 5.7: The API aggregator pattern

The preceding diagram illustrates an API gateway acting as a resource router based on the aggregator's unique URI. The input required by the aggregator should either come as a parameter or as a payload in the body depending on the HTTP verb. The aggregator is then responsible for making subsequent calls in order to collect, transform, and send back a uniform payload as expected by the consuming application.

This pattern fits quite well with the GraphQL architectural style because its entire focus is to be client-driven and flexible. A comparison of REST and

GraphQL is covered in Chapter 6, Modern API Architectural Styles. However, the following video is a good overview of GraphQL and how it compares to REST:

<https://tinyurl.com/restvsgraphql>

Drawbacks:

The number of API endpoints may increase considerably if the REST architectural style is adopted to deliver this pattern. The reason is that REST requires unique URIs for each resource. Considering that reuse of these endpoints will be limited, as they are very tailored to specific use cases, new endpoints have to be created in support of new requirements.

Tags: Implementation (main) and mediation (supporting).

Applicability:

This pattern is better addressed with GraphQL for the reasons mentioned, although technically speaking, the pattern can also be realized with other API architectural styles.

API orchestration service

Problem statement:

A consuming application has a requirement to implement a piece of functionality expressed in the form of a well-defined business process. As the same functionality is needed in not one, but multiple consuming applications, implementing the same functionality multiple times on each application would be impractical, time-consuming, and costly to deliver and support.

Furthermore, as part of the business process, multiple validations and checks against other system's interfaces are needed. The process must be implemented in a specific order and comply with the business rules as specified by the business in the process requirements.

Solution:

The solution is to adopt a **process engine** as the means to design and implement the business process **orchestration** in accordance with the business requirements. You can then expose the process orchestration as an API so it can be accessed from multiple consuming applications. An API gateway can also be implemented as a resource router into the process API.

Furthermore, as process engines have been around for a while, especially those based on the **business process orchestration language (BPEL)** and/or the **business process model and notation (BPMN) v2.0**, the chances are that some of these can be reused and exposed as APIs. However, in order to do so, it may require converting from older protocols, such as SOAP into REST or even GraphQL, for example.

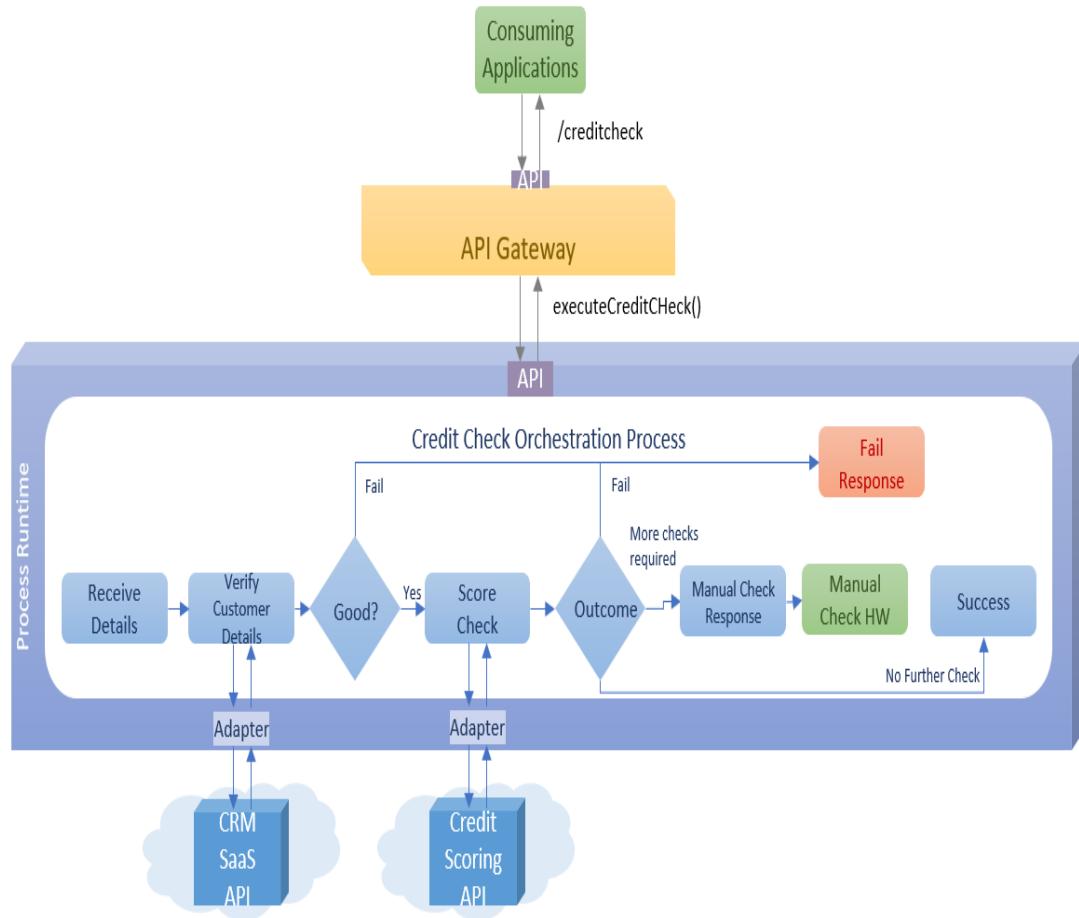


Figure 5.8: The API orchestration service pattern

The preceding diagram illustrates consuming applications accessing a process orchestration responsible for executing a

credit check. The API gateway, in addition to performing resource routing, also performs a protocol conversion as the process itself is exposed through an endpoint.

The illustration also shows the different process activities and validations, including activities that require interactions with external interfaces from other systems.

Drawbacks:

The implementation of a business process, as illustrated, requires, in the majority of cases, that a process runtime is capable of maintaining state, which may limit an engine's ability to scale and handle high throughputs.

Furthermore, both stateful services and orchestrations are highly discouraged in microservices architectures in favor of stateless services that adopt choreographies and event-driven architectures as the means to interact with and accomplish a process.

The following article offers a good perspective on how choreographies compare to orchestrations in the context of microservices architectures:

<http://www.soa4u.co.uk/2018/02/is-bpm-dead-long-live-microservices.html>

Tags: Implementation (main) and mediation (supporting).

Applicability:

This pattern can be applied to any API architectural style supported by the API gateway and the process runtime.

API microgateway

Problem statement:

An independent runtime has been implemented as the main runtime for fully decoupled services (microservices). Adopting an API gateway as a resource router into the runtime can be inefficient because typical independent runtimes, such as Kubernetes, make use of an ingress load balancer as an entry point to internal services.

While the approach mentioned would work, it also implies a two-layer API gateway architecture, adding complexity and additional compute costs. The outer gateway acts as a policy enforcement point and router into the ingress, and the ingress itself acts as an API load balancer into service endpoints inside the runtime.

Solution:

A better solution would be for the ingress load balancer to also act as an API gateway. This is referred to as an **API microgateway**. This not only means that the API gateway itself would fit more natively into an independent runtime's architecture, but that it too could directly leverage other runtime capabilities, such as a service mesh (see the previous

chapter for an explanation of this capability). Furthermore, it would simplify the solution as there would be one instead of two layers to troubleshoot and maintain.

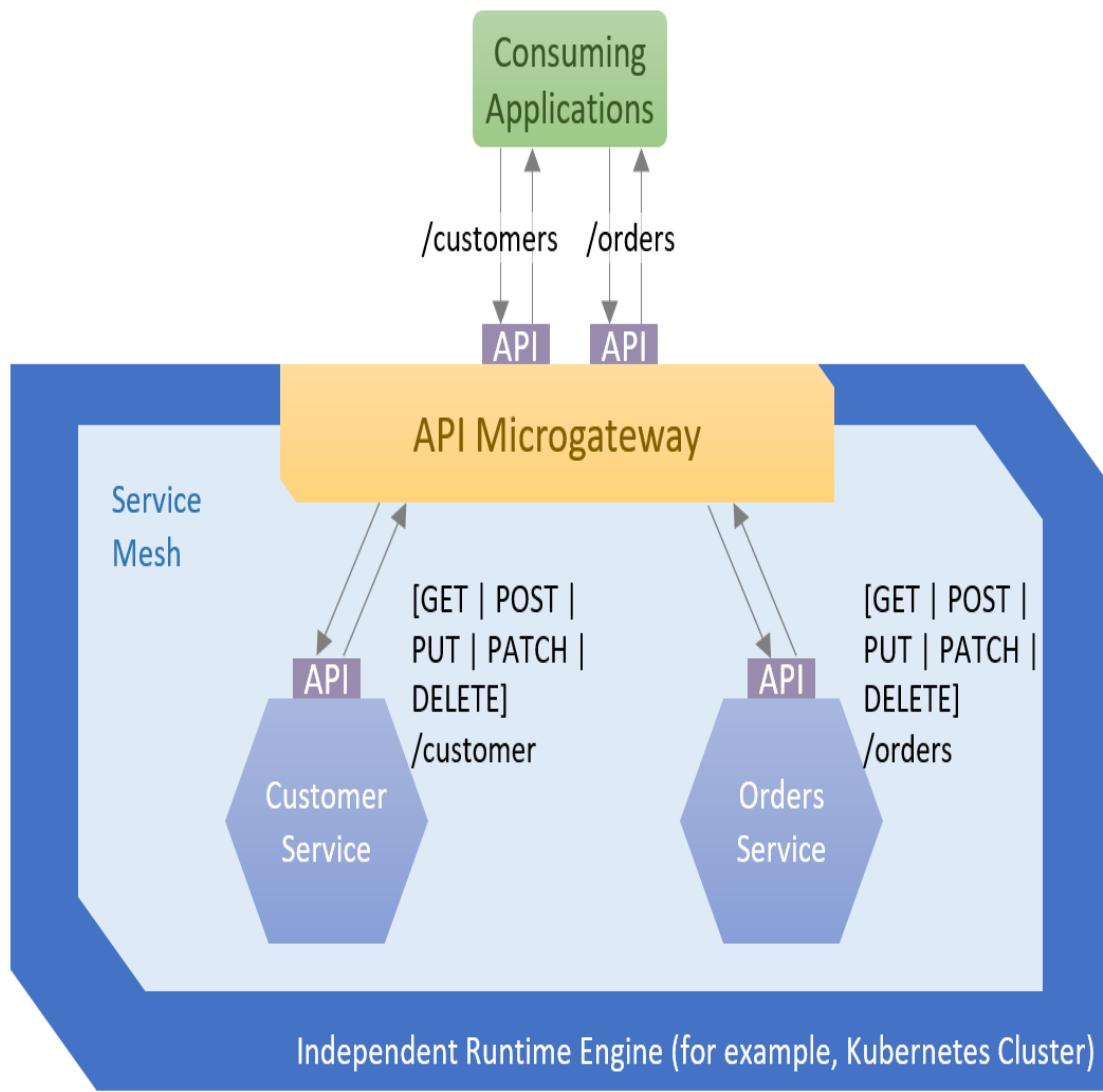


Figure 5.9: The API microgateway pattern

The preceding diagram illustrates an ingress load balancer acting as an API microgateway. It shows the API gateway implementing a resource routing pattern that also natively leverages a service mesh capability of the runtime.

Drawbacks:

An important drawback of this pattern is that the majority of API vendors (at least at the time of writing) don't yet have API microgateways as part of their offering (though this is changing rapidly).

Tags: Mediation (main) and implementation (supporting).

Applicability:

Although the example is based on REST, it can also be applied to other API architectural styles (for example, REST, GraphQL, and gRPC) depending on the capabilities offered by the microgateway.

Sidecar API gateway

Problem statement:

In certain scenarios, it may not be desirable to have a single API microgateway mediating access to all service endpoints within an independent runtime (for example, Kubernetes clusters). Some of the reasons could be:

- Different services require fundamentally different gateway setups and therefore sharing a single one becomes highly impractical and restrictive.
- Separation of concerns: if an issue occurs in the shared microgateway infrastructure then all services are impacted.
- Freedom of choice: each sidecar gateway can effectively be from a different vendor.

Solution:

Instead of implementing an API microgateway as a replacement for an ingress load balancer, let the latter act as an API load balancer and resource router, and adopt a sidecar pattern for the API gateway capability.

The sidecar pattern is described by Microsoft in the following article:

<https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>

The benefits of adopting this pattern are that each service runtime can configure the API gateway in the best way. Furthermore, if required, the technology and/or vendor used to implement the API gateway can vary from service to service.

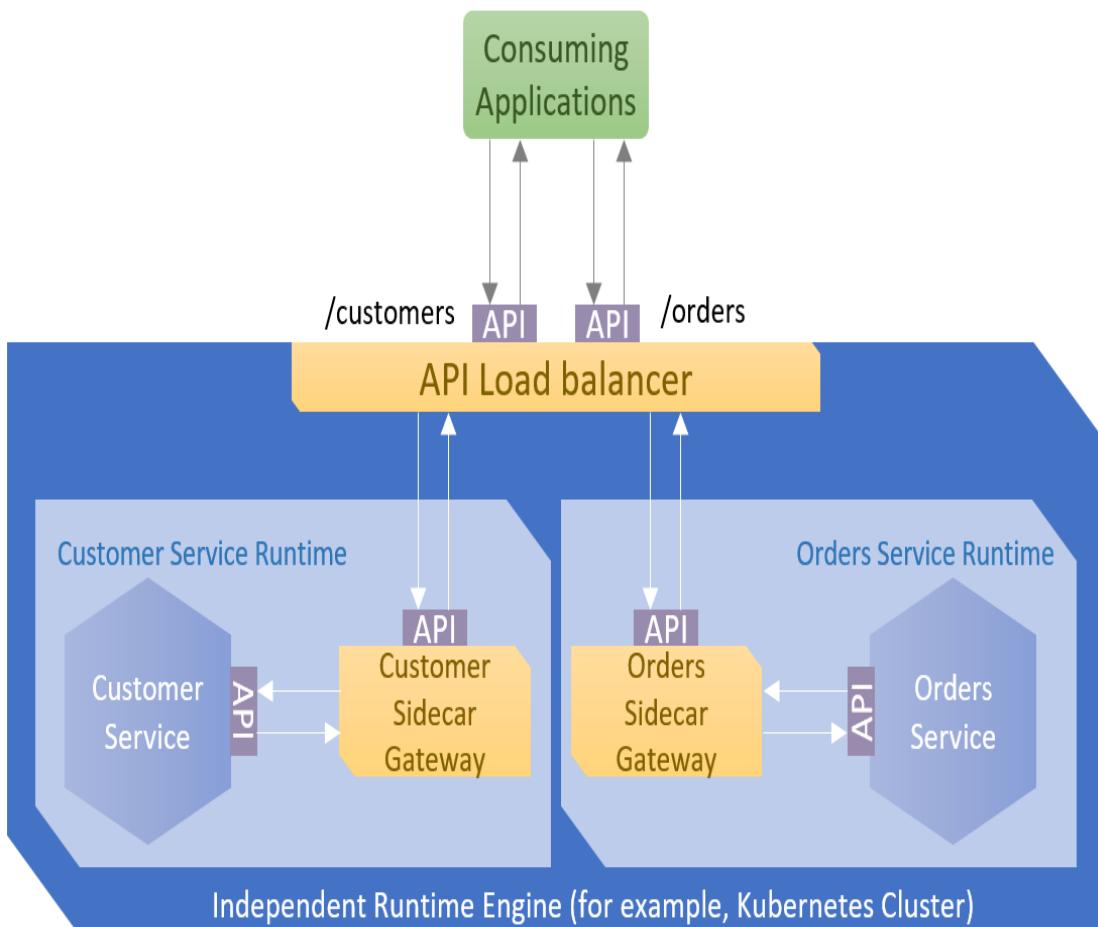


Figure 5.10: The sidecar API gateway pattern

The preceding diagram illustrates an ingress acting as an API load balancer and resource router into each service endpoint. The entry point for the service is not the service endpoint itself but rather a sidecar API gateway. The sidecar can then perform

any of the capabilities offered by the API gateway in addition to routing traffic to the service endpoint.

Drawbacks:

Although the pattern offers some notable benefits, as the number of gateways increases, so will the complexity of the solution, especially if the technologies and/or vendors used for the API gateway are very diverse. It can also result in added compute (as gateways also run on the service runtimes) and license costs (depending on the API gateway used and the license model if not open-sourced).

Tags: Mediation (main) and implementation (supporting).

Applicability:

Although the example is based on REST, it can also be applied on other API architectural styles (for example, GraphQL or gRPC) depending on the capabilities offered by both the API load balancer and the sidecar gateways.

Webhook

Problem statement:

A consuming application desires to be informed of any change of state on a specific record or records; for example, updating or adding customers in a CRM SaaS, currency exchange rates from a foreign exchange (forex) application, or even new posts on a user's blog.

The majority of APIs only support these types of requirements by having the consuming application constantly poll for changes. This means that the consuming application has to make frequent API calls to find out any changes of state in a desired resource. Not only is this highly inefficient, as calls may result in empty payloads when there haven't been any updates, but it could also affect the user experience, especially when the behavior is reflected back to the user interface (meaning a user has to refresh a page to get the latest changes).

Solution:

Instead of having to constantly poll for changes, create a subscription endpoint against a specific resource so consuming applications can register their interest to be informed on any change of state (an event) by providing a call-back endpoint. At

this point, it becomes the API's responsibility to send back any change of state by posting the updates to the registered endpoint.

In order to facilitate this functionality, a service must have the capability of not only storing all registered call-back endpoints, but also keeping track of which events each subscriber is subscribed to, so when the event is detected, a call-back can be triggered.

The following is an example of how to implement this functionality on the server-side.

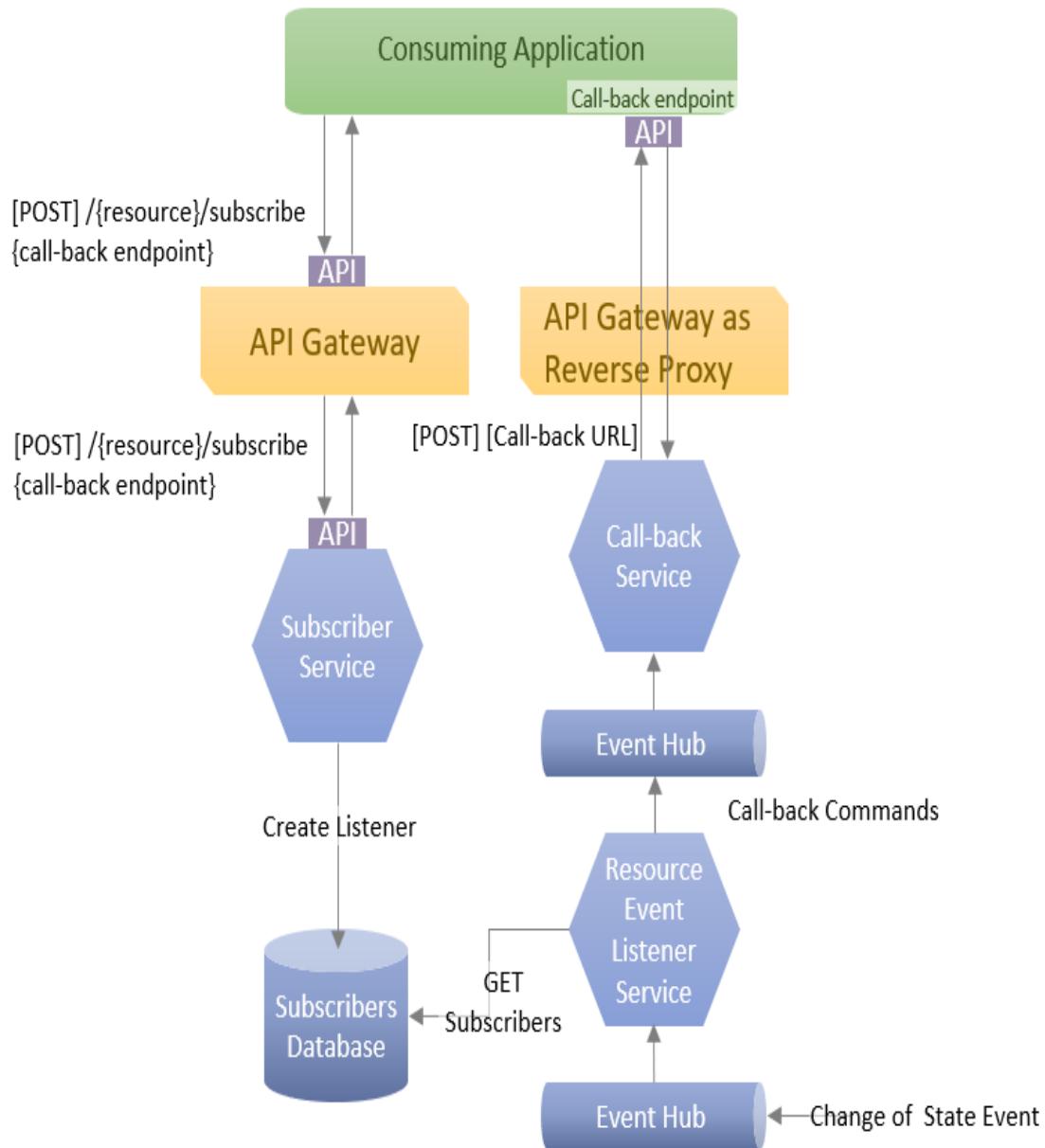


Figure 5.11: The webhook pattern

The preceding diagram illustrates a consuming application subscribing for changes in a resource by making a POST call (with the call-back URL in the body) to a resource subscription API endpoint (for example, `/resource/subscribe`) exposed in an API gateway. Once the API gateway receives the call, it routes the request to the subscription service, which then adds the

subscriber details to a database. The database is then queried by an event listener service as subscribers are matched against particular processed events. The event listener service then creates call-back commands and publishes them in an Event Hub so a call-back service can then execute all API calls (and retries if necessary) via the API gateway that is now acting as a reverse proxy.

*Note that a variation of this pattern could be implemented using **web sockets**. In such a variation, when the API consumer makes a subscription call, a web socket could be established to asynchronously communicate back to the consumer any events that may take place. In this variation, there is no need for a call-back API; however, further considerations should be made about handling the sockets, especially when dealing with scenarios where a socket is broken. For more information on web sockets, refer to the following link:*

<https://en.wikipedia.org/wiki/WebSocket>

Drawbacks:

This is a complicated pattern to implement given its many moving pieces. This is especially true for exception handling of call-back retries, which can be complicated given that the API is not in control of whether or not a consuming application endpoint is up and running.

Tags: Interaction (main); mediation and implementation (supporting).

Applicability:

Although the illustration is based on REST, GraphQL also provides native support for Webhooks in its specification via the subscription's operation. In the case of gRPC, a similar

interaction can be accomplished with gRPC bidirectional streaming, although not in exactly the same way.

*Note that Webhooks are officially supported in **OAS 3.0**, which is described in [Chapter 6](#), Modern API Architectural Styles.*

API geo-routing

Problem statement:

An API has global reach with consuming applications spread all over the world. In order to prevent latency issues and other unforeseen issues that may occur due to distance (for example, a consuming application from Asia calling an API located in North America), API gateways and other service infrastructure have been deployed in multiple regions across the world as needed (for example, in support of data sovereignty requirements). The challenge remains, however, of deciding the most effective way for consuming applications to determine what localized URL they should be using.

Multiple options exist but they are deemed sub-optimal, for example, using different sub-domains for each API gateway in each region and letting the consuming application determine the nearest gateway based on application logic.

Solution:

The solution is to adopt a DNS and traffic management service (for example, Amazon Route 53) with global reach that will offer the ability to configure intelligent responses against specific DNS queries. This will allow a master domain to

resolve to other predefined sub-domains based on location or availability, as an example.

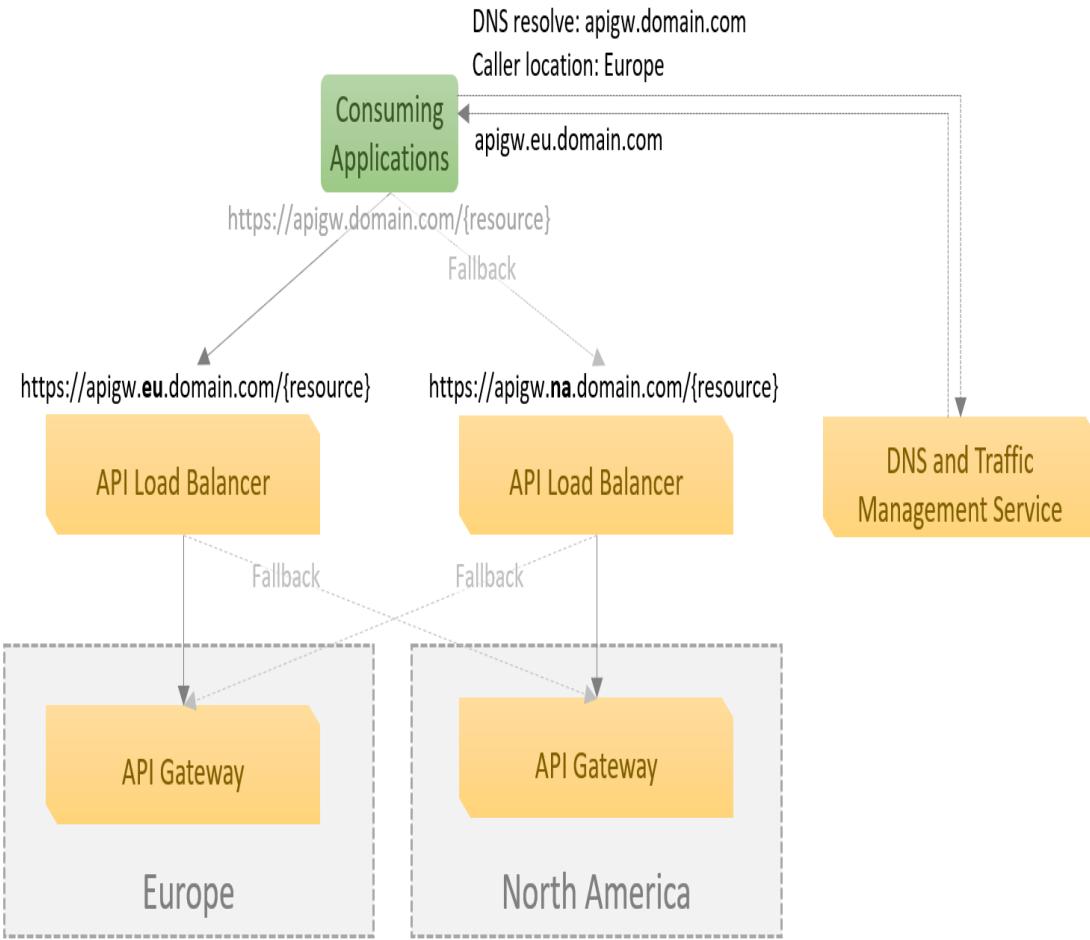


Figure 5.12: The API geo-routing pattern

The preceding diagram illustrates a client application wishing to call an API resource through the URL `https://apigw.domain.com`. However, when making the call, the `apigw.domain.com` domain resolves against `apigw.eu.domain.com`, as that's the localized sub-domain for the region the call originates from. The diagram also indicates that `apigw.na.domain.com` is configured as a fallback in case there is an issue with the European Union region.

Furthermore, the diagram also illustrates each sub-domain resolving against a region's load balancer instead of to the API gateways directly. This is to allow an active-active multi-region configuration, so if one region is overloaded with traffic, it can offload some of it to another region.

Drawbacks:

In multi-cloud vendor implementations, the solution isn't straightforward as different cloud vendors offer their own DNS and traffic management services. Likewise, if implementing in private networks, further considerations may be required, such as establishing a hybrid architecture.

Tags: Mediation.

Applicability:

This pattern is applicable to any API architectural style.

API firewall

Problem statement:

As APIs become the main entry point for all sorts of internal and external applications to access information and functionality, they too become the center of focus for a multitude of malicious attacks. Although API gateways do provide a level of protection against malicious threats, they don't offer protection to the extent that an API can be considered fully secured against all major attacks, such as the ones listed in the **Open Web Application Security Project (OWASP) Top 10** project or the SANS Institute's **Common Weakness Enumeration (CWE) Top 25**.

Further information on the OWASP Top 10 and the SANS CWE 25 can be found at the following links:

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

<https://www.sans.org/top25-software-errors>

Solution:

Instead of solely relying on an API gateway for threat protection, implement a level-seven application firewall as an API firewall on all endpoints exposed by the API gateway. The API firewall should always act as the first line of defense against malicious threats and it should be transparent to API consumers. Furthermore, it should be configured to always

forward traffic to the corresponding API gateways regardless of the resource being accessed (basically just acting as a proxy and not a resource router).

When/if possible, the firewall should also be configured as a load balancer, thus avoiding another application layer.

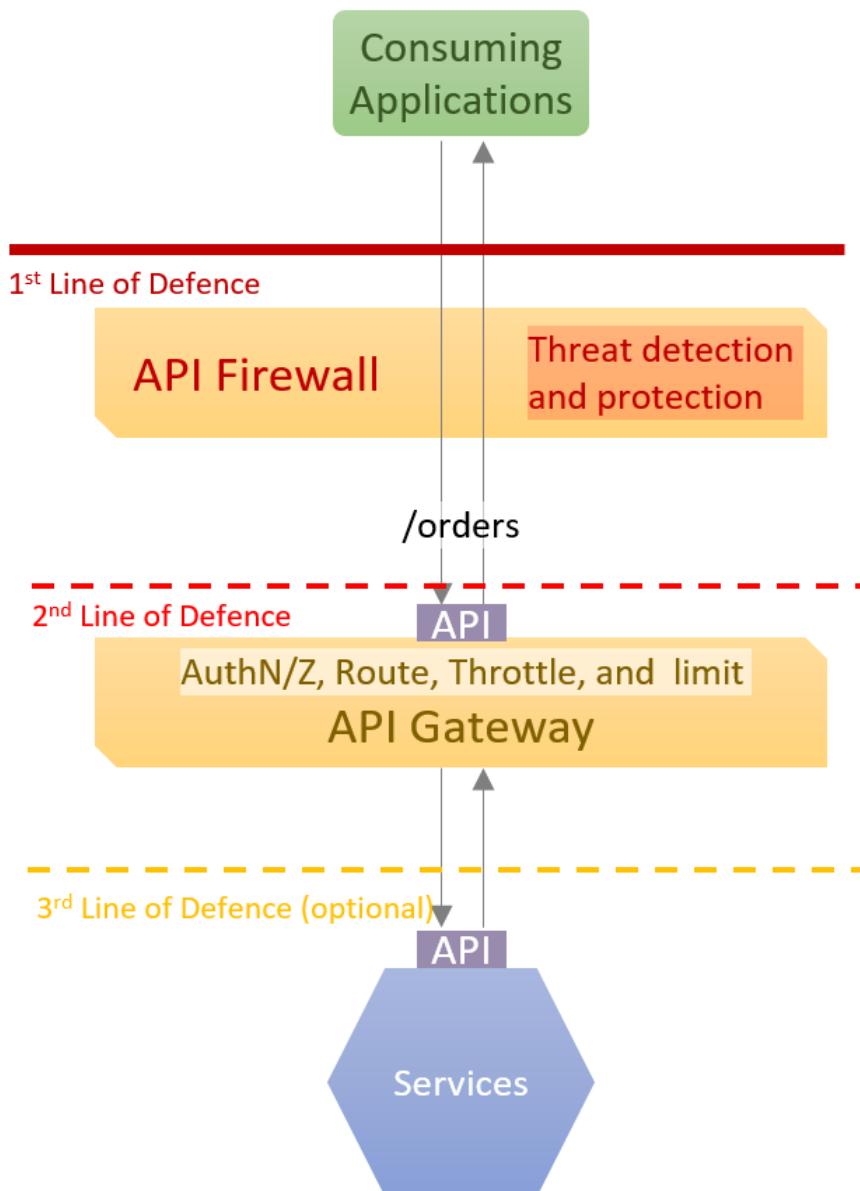


Figure 5.13: The API firewall pattern

The preceding diagram illustrates a consuming application accessing the `/orders` resource and an API firewall acting as the first line of defense. Once the firewall inspects the call and deems it valid, it then forwards the call to the API gateway, which then acts as a second line of defense. Alternatively, the service layer can also act as a third line of defense should it be necessary to add an additional layer of security.

Drawbacks:

There are no major drawbacks apart from the additional complexity implied by adding an additional level of security.

Tags: Security (main) and mediation (supporting).

Applicability:

The majority of application firewalls will support the REST architectural style out of the box, given that it reflects the way the majority of HTTP-based web applications work. Support for other styles, such as GraphQL and/or gRPC, will be highly dependent on the capabilities of the application firewall, and limitations may exist.

API basic authentication

Problem statement:

An internal API, meaning an API that is only accessible from within a corporate **wide area network (WAN)**, needs to verify that the application users exist and have valid credentials in the corporate directory (for example, MS Active Directory and/or any other LDAP server). This means that the application is trusted.

Solution:

Assuming that HTTPS is used for transport encryption between the consuming application and the API gateway, a straightforward solution is to implement HTTP basic authentication at the API gateway level.

To this end, a consuming application must include the user's credentials in the HTTP header as follows:

```
Authorization: Basic <base64(username:password)>
```

Note that the username and password must be joined by a colon and then encoded in base64.

The API gateway, once it receives the request, should take the credentials from the header, decode them, and perform a validation against the corporate service typically using the **lightweight directory access protocol (LDAP)**.

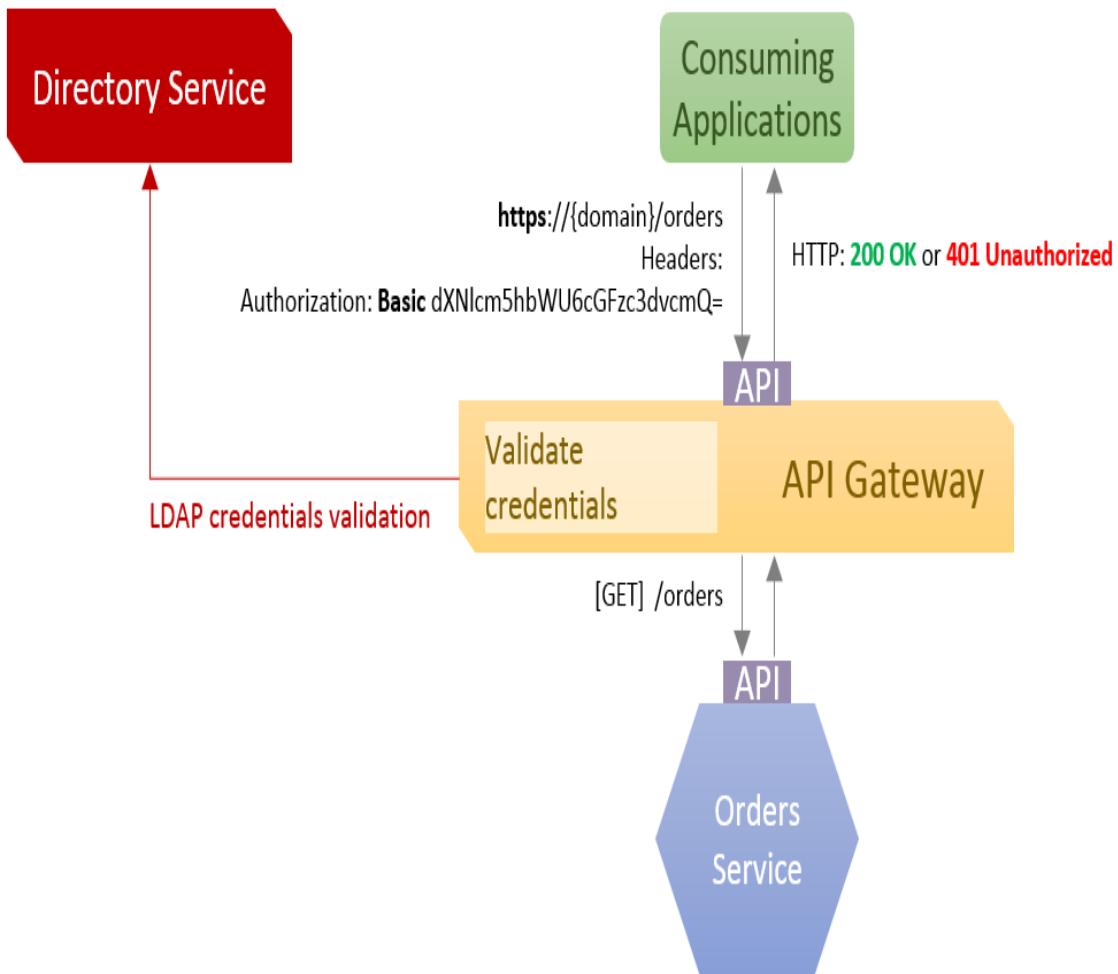


Figure 5.14: The API basic authentication pattern

The preceding diagram illustrates a consuming application calling the `/orders` resource. As part of the call, the user credentials are included in the HTTP authorization header encoded in base64. Subsequently, the API gateway validates the credentials against the corporate directory service (for example, MS Active Directory) using LDAP. Once a user is

validated, then a call is routed to its corresponding backend service. Otherwise, an **HTTP 401 unauthorized** message is sent back to the consuming application indicating that the validation failed.

Drawbacks:

This authentication pattern is not completely safe for the following reasons:

- Even if HTTPS is applied to encrypt all transport communications, as credentials are simply encoded using base64, if either the consuming application and/or the API gateway get compromised, the attacker can easily get hold of them.

Note that even when both consuming applications and API gateways reside within a corporate WAN (thus they are inaccessible from external networks), this pattern is still exposed to internal threats, which are often the source of the majority of attacks. Please refer to the following article for more information:

<https://dzone.com/articles/internal-or-insider-threats-are-far-more-dangerous>

- As the credentials have to be included in every single API call, the pattern is prone to brute force attacks, where an attacker just tries different usernames and passwords until the right combination is successful.

More info on this attack can be found at the following link:

https://en.wikipedia.org/wiki/Brute-force_attack

- If the consuming application runs in a web browser, then the credentials are cached at minimum temporarily during

the authentication window, thus exposing the user to **Cross-Site Request Forgery (CSRF)**.

More on CSRF can be found at the following link:

[https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))

Tags: Security (main) and mediation (supporting).

Applicability:

This pattern applies to any HTTP and/or HTTP/2-based API, assuming the API gateway supports HTTP basic authentication.

API bearer of token

Problem statement:

For compliance, regulatory, or security reasons, a consuming application must not be able to access a HTTP resource without the explicit consent of the resource owner.

The application itself may or may not be trusted, and therefore there must be a mechanism that allows:

- Users to be authenticated and authorized, if required, without the consuming application ever getting exposed to the credentials (in case the application is not trusted).
- Access tokens to be issued so they can be used by a consuming application to access a protected resource exposed by the API gateway.
- An API gateway to fully trust the tokens by having the ability to verify that the tokens are genuine and issued by a trusted server to the consuming application making the call.

Solution:

The solution is to implement a bearer token pattern based on an open standard, such as **OAuth 2.0**, thus allowing different authorization flows to be implemented depending on the requirements at hand.

Note that OAuth 2.0, including the main concepts and grants, will be covered in subsequent chapters. In addition, further details on OAuth 2.0 can be found at the following link:

<https://oauth.net/2/>

At minimum, the solution will require:

- An **authorization server** responsible for authenticating and authorizing users against valid applications and, if successful, issuing tokens.
- An API gateway acting as a **resource server** to enforce that only callers bearing a valid token can access a given resource (for example, `/orders`).
- If necessary, the implementation of **mutual transport layer security (mutual TLS)** so only calls originating from a trusted consuming application are processed.

More details on mutual authentication can be found at the following link:

https://en.wikipedia.org/wiki/Mutual_authentication

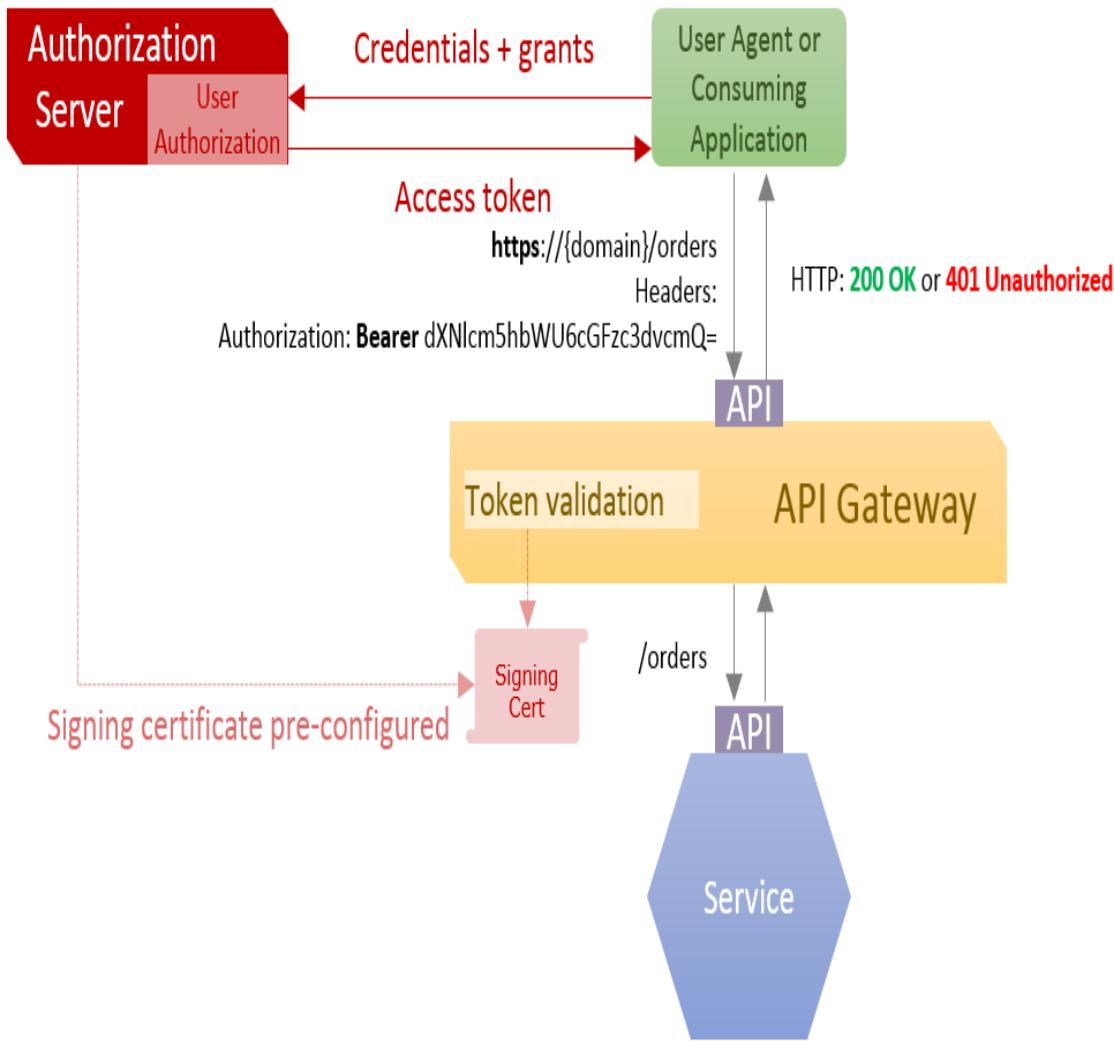


Figure 5.15: The API bearer of token pattern

The preceding diagram illustrates a user agent (for example, a browser) and/or a consuming application authenticating/authorizing against an authorization server.

Whether it's the user agent and/or the application itself interacting with the authorization server depends on the authorization flow being followed.

Provided the user credentials are successfully validated by the authorization server and, if applicable, assuming the user grants the application access to the resource during the authorization process, a token is generated, typically based on

the **JSON Web Token (JWT)** format and sent back to the client application.

More on JWTs can be found at the following link: <https://jwt.io>

The client application then tries to access the resource by providing the token in the HTTP header but indicating that it's a **bearer** token. At this point, the API gateway validates the authenticity of the token by verifying its signature against the authorization server's **signing certificate** (which should have been previously obtained and configured in the API gateway) and if this is valid, optionally the API gateway can also check that the user has rights to access the specific resource (for example, by looking at the scopes within the token). If this is successful, then access to the resource is allowed.

Drawbacks:

As the authorization flow may differ depending on the requirements, the pattern isn't completely safe if a consuming application isn't trusted and the credentials flow through the application. Although this can be prevented by adopting a more complex flow, doing so would require more effort and understanding of protocols such as OAuth 2.0. Furthermore, this flow will typically require further applications infrastructure (for example, an authorization server).

Tags: Security (main) and mediation (supporting).

Applicability:

This pattern applies to any HTTP and/or HTTP/2-based API, assuming the API gateway supports the OAuth 2.0 protocol.

API bearer of obscure token

Problem statement:

This pattern has similar requirements to the bearer of token, with the difference being that in this case, a consuming application can't ever be trusted and thus will never be exposed to the user credentials or any details of the user that may be available in the token.

Solution:

A solution is to implement the OAuth 2.0 **authorization code** grant, but extend it with the use of obscure tokens so the consuming application never gets exposed to any details of the token. The way this works is that instead of the authorization server issuing a standard JWT, once the consuming application obtains the authorization code, the server will issue an obscured token, for example, a random string, that has no meaning to the application.

Once the API gateway receives the obscure token, it sends it to the authorization server, which then verifies whether the obscured token is valid and if so, sends the access token back to the API gateway for further verification.

In addition, in order to ensure this flow is completely safe, mutual TLS should be implemented between the consuming application and the authorization server, the consuming application and the API gateway, and finally, between the API gateway and the authorization server.

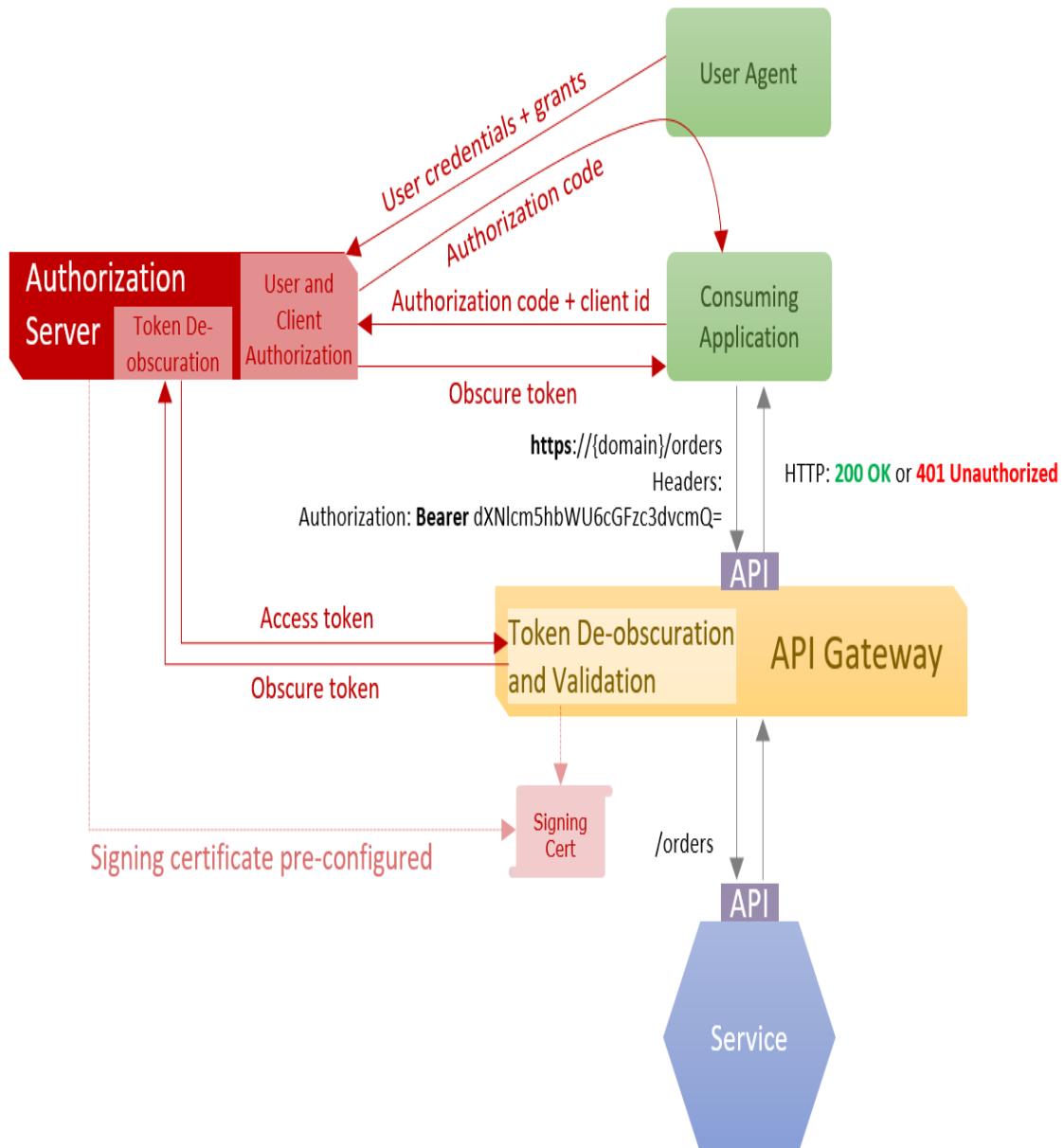


Figure 5.16: The API bearer of obscure token pattern

The preceding diagram illustrates an OAuth 2.0 authorization grant flow but with two variations: firstly, there is no use of client secrets (as the application isn't trusted) and secondly, obscure tokens are generated in the server instead of JWTs.

Drawbacks:

This pattern can be very complex to implement, so unless the requirement dictates a very high level of safety, simpler but still secure alternative flows should be considered.

Tags: Security (main) and mediation (supporting).

Applicability:

This pattern applies to any HTTP and/or HTTP/2-based API, assuming both the API gateway and authorization server support not just the OAuth 2.0, but also obscure tokens.

Summary

Throughout the different sections of this chapter, many of the technical capabilities described in [Chapter 4, API-Led Architectures](#), were brought to life. The chapter started by describing what it means to adopt design patterns in the context of APIs and also the different types of patterns that should be considered in end-to-end API architectures.

From simple mediation patterns, such as the resource router, which suggests the use of an API gateway to route API calls to backend services based on the URIs, to more complex patterns, such as Webhooks, which make use of several service capabilities in order to enable asynchronous communication originating from the API provider to the consuming application, the chapter delivered a thorough elaboration of many design patterns applicable in the context of API architectures.

The next chapter will deliver a detailed overview and comparison of the API architectural styles briefly touched upon throughout this chapter.

Modern API Architectural Styles

This chapter complements [Chapter 5, *API-Led Architecture Patterns*](#), by discussing in more detail the different modern API architectural styles, such as **Representational State Transfer (REST)**, **Graph Query Language (GraphQL)**, and **Google Remote Procedure Calls (gRPC)**, that were briefly mentioned in the patterns illustrated. This chapter accomplishes this by first delivering a point of view on the application interface protocols and standards that have led to modern APIs. Next, the chapter will elaborate on their unique characteristics, their pros and cons, and when to (or not to) use each of them.

A brief history of interfaces

As mentioned in the *Preface*, the use of programming interfaces as the means for one application (or application module) to interact with another is by no means new. In fact, the first use of the term **API** can be traced back to the 1968 publication *Data Structures and Techniques for Remote Computer Graphics* by I. W. Cotton and F. S. Greatorex, Jr. The paper described an approach to achieving hardware abstraction, modularity, and reusability by implementing hardware-agnostic APIs that can be remotely accessed from multiple display devices.

For more information on this publication, please refer to the following link:

<https://www.computer.org/cSDL/proceedings/afips/1968/5072/00/50720533-abs.html>

However, in spite of the obvious similarities with the contemporary use of interfaces, it is difficult to tell whether such early use of the term had any influence on how we employ APIs today. Regardless of this, what can be concluded without question is that the notion of interfaces as the means of achieving some form of abstraction has been present since the early days of computer science. Recognizing this is extremely important as it's clear evidence that APIs have and will continue to evolve. This will be the case even more so as faster, more efficient, and more compact hardware infrastructure,

application communication protocols, and architecture paradigms are introduced to the industry.

This is one of the main reasons that this book has, to the extent possible, decoupled itself from just one API architectural style (for example, REST) and rather it focuses on the bigger picture, such as approaches, concepts, technical capabilities, and patterns that may also apply in the future, beyond APIs as we know them today.

The rise of RPC

Modern APIs are broadly considered to be an evolution of the **Remote Procedure Call (RPC)** protocol. Although research for the protocol started in the 1970s, it wasn't until the late 1980s, when Sun Microsystems (now part of Oracle Corporation) first released its UNIX-based RPC implementation, that the protocol gained wide popularity and adoption.

Please refer to the following article for further details: https://en.wikipedia.org/wiki/Remote_procedure_call#History_and_origins

The **Open Network Computing (ONC)** RPC, also referred to as Sun's RPC, had many of the characteristics expected of modern APIs:

- It made use of the **External Data Representation (XDR)** standard to define an interface that both server and client applications should comply with.
- It made use of XDR as the means to serialize and deserialize data over the wire (basically request and response messages).
- It made use of the still-widespread **Transmission Control Protocol (TCP)** or **User Datagram Protocol (UDP)** as transport.

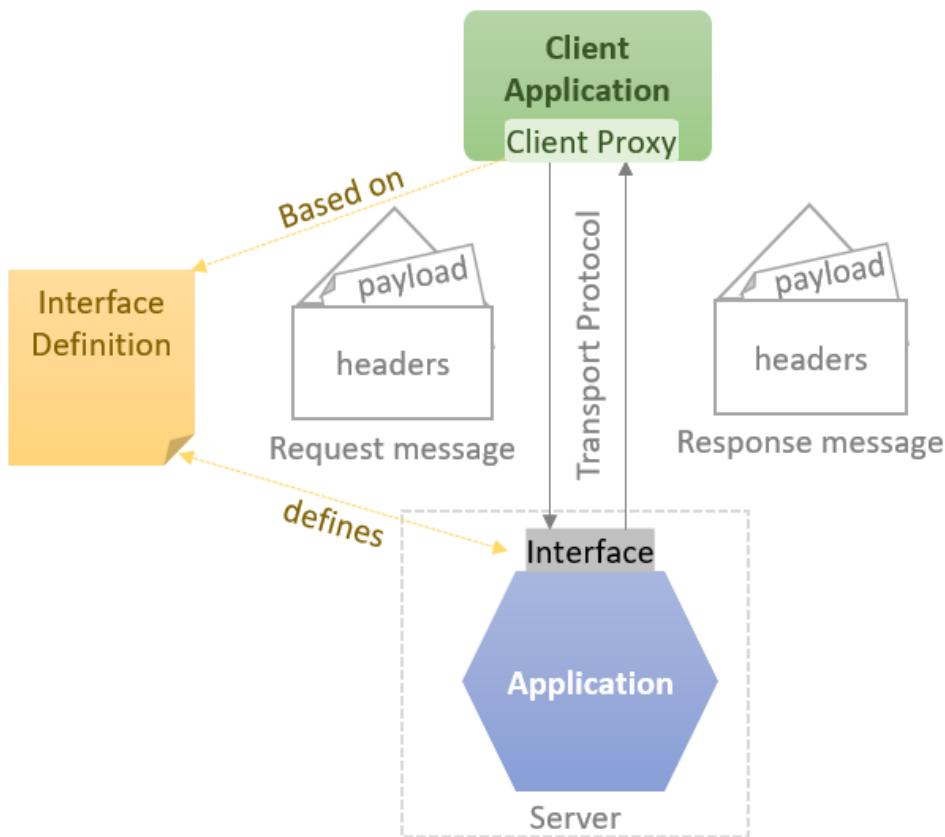


Figure 6.1: Modern APIs as perceived today

As illustrated in preceding diagram, modern APIs, such as RPC, also consist of a client application (the consumer of the API), a server (the producer of the API) and an interface definition document detailing, in technical terms, all the details of an interface, such as the actions/methods/resources supported and data inputs and outputs.

However, ONC RPC wasn't the only popular RPC implementation. The following diagram illustrates in chronological order the interface protocols and standards that followed and still apply today.

As can be seen, within five years of its inception, the **Open Software Foundation (OSF)**, a non-profit organization originally consisting of Apollo Computer, Groupe Bull, Digital Equipment Corporation, Hewlett-Packard, IBM, Nixdorf Computer, and Siemens AG, (referred to as the *Gang of Seven*), released its own implementation of RPC called the **Distributed Computing Environment (DCE) RPC**.

*Note that in February 1996, the OSF merged with X/Open to become **The Open Group**.*

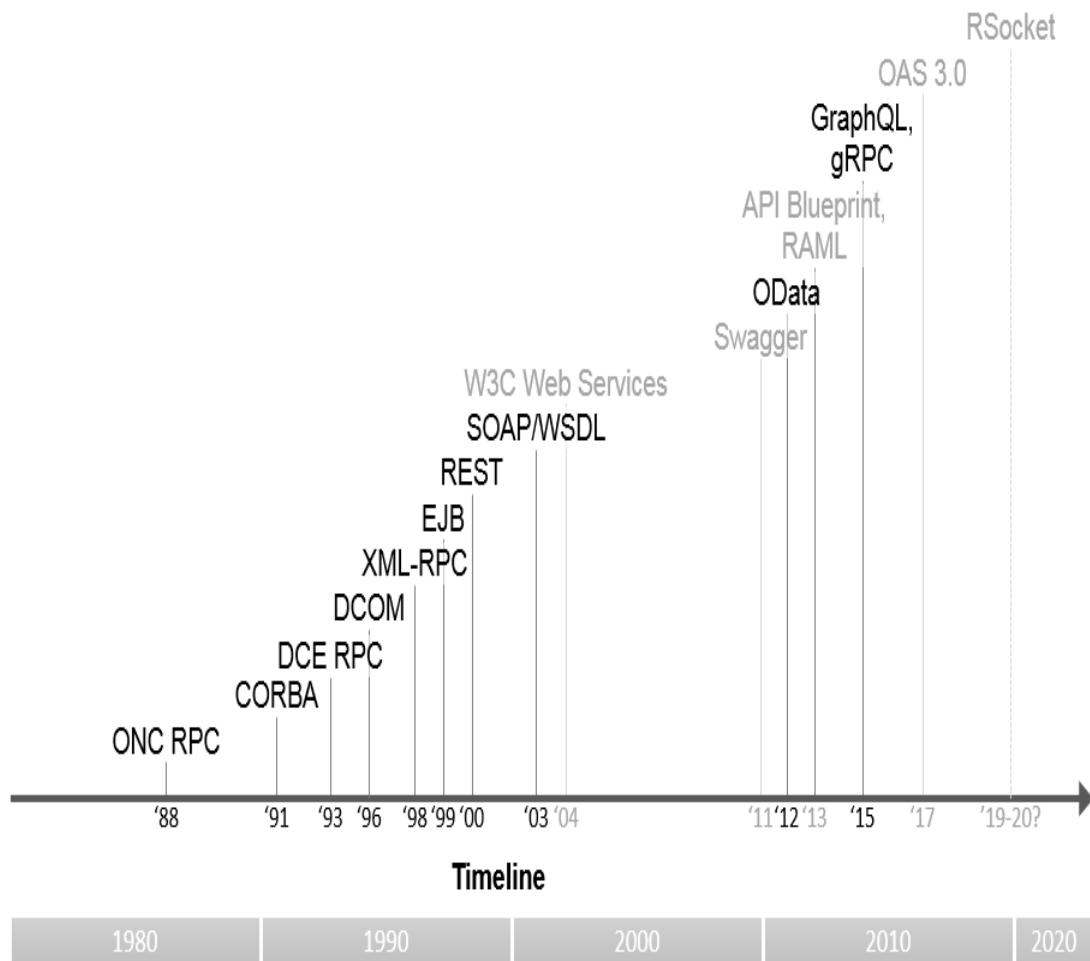


Figure 6.2: Application interfaces protocols and standards evolving through time

Although, at a conceptual level, DCE RPC shared many of the characteristics of ONC RPC, in actual practice there were many notable differences. For example, the former made use of an XDR standard as the means to define public interfaces and also to serialize and deserialize data over the network. The latter made use of its own **interface definition language (IDL)** and a **network data representation (NDR)** specification to serialize data into octet streams (a process known as **marshaling**) and then deserialize it (a process known as **demarshaling**).

Another difference was that DCE RPC did not limit the number of parameters on a call; in ONC RPC, on the other hand, calls were limited to one input and one output parameter.

All of the interface protocols and standards are described in detail in the following article:

<https://www.soa4u.co.uk/2019/02/a-brief-look-at-evolution-of-interface.html>

RPC and object-oriented programming

One important characteristic that both DCE and ONC RPC protocols had in common, though, was the fact that neither were based on the object-oriented programming model, which was increasingly gaining popularity in the early 1990s.

The first RPC-based protocol natively supporting the object-oriented programming model was the **Common Object Request Broker Architecture (CORBA)**, created by the **Object Management Group (OMG)** consortium. CORBA, just like other RPC systems, followed a client-server architecture, with the major difference being that instead of serializing/deserializing flat data over the network, it adopted the **Internet InterORB Protocol (IIOP)** as a messaging protocol to transfer objects over a TCP/IP transport.

The **Distributed Component Object Model (DCOM)** was perhaps the second RPC-based protocol supporting the object-oriented programming model. DCOM was created by and remains the property of Microsoft. However of interest is the fact that DCOM made heavy use of **Microsoft RPC (MSRPC)**, which is nothing but a modified version of DCE

RPC but was originally created to support a client/server model in the Windows NT operating system.

*It's worth noting that with the popularity of these proprietary protocols also came the emergence of a new set of tools in support of **enterprise application integration (EAI)**. EAI tools made use of these protocols to enable the integration of different applications that could not otherwise talk to each other. Refer to [Chapter 2](#), *The Evolution of API Platforms*, for a comprehensive overview of how EAI evolved into the modern platforms we see today.*

A third and very popular implementation of RPC for the object-oriented programming model was the Java **Remote Method Invocation (RMI)**, which arguably is an RPC system but for the **Java Virtual Machine (JVM)**. In a nutshell, Java RMI allows for an object running in one JVM to invoke methods running in another JVM. The most popular use of the Java RMI is in **Enterprise Java Beans (EJB)** for the **Java Enterprise Edition (JEE)** platform (now Jakarta EE).

XML to the rescue

Some of the challenges faced by the RPC alternatives available in the early 1990s were due to the fact that many made use of proprietary standards. This either restricted the use of the technology or required the acquisition of a commercial license to do so.

Needless to say, this was far from ideal as it meant that interoperability of systems heavily depended on the vendor of a system and what protocols/standards it opted to use.

This was all about to change with the introduction of the **Extensible Markup Language (XML)** as an open standard by the **World Wide Web Consortium (W3C)** in 1996. XML enabled data to be formatted in a document that was not only readable by both humans and machines, but in an open (text-based) format that did not require licenses or permissions for its use, meaning it was implementation-neutral and supported by open standards. This was a game changer in the industry as it meant that any protocol and/or standard that was based on XML would face far less interoperability challenges and thus experience increased adoption rates.

Not long after XML became a standard, an XML-based RPC implementation was released by Microsoft. It was called RPC-

XML. Because RPC-XML not only made XML the message format, but also adopted **Hypertext Transfer Protocol (HTTP)** as the transport protocol, it quickly became a popular choice when implementing client-server applications over the network and even over the public Internet.

Within four years, RPC-XML evolved into becoming the **Simple Object Access Protocol (SOAP)**, which, along with another XML-based standard called the **Web Service Description Language (WSDL)**, which was used as the means to describe interfaces based on SOAP, became the foundation for **Web Services** within the official W3C recommendation **Web Services Architecture** published in 2004.

*Web Services was perhaps the most important and influential concept behind the still-popular **Service Oriented Architectures (SOA)** architectural paradigm.*

SOAP/WSDL and associated standards and tools, most notably the **Enterprise Service Bus (ESB)**, which became a sort of silver bullet (perhaps because vendors sold it as such), prevailed in the industry for over a decade as the main means to deliver (or expose) Web Services and enable SOA. However, as with everything, this dominance wasn't meant to last forever.

Towards early 2010, with the rise of mobile applications and cloud computing, the industry felt constrained by the capabilities offered by ESBs but also by using Web Services as a standard. This was especially true when it came to satisfy

emerging requirements around mobility, the cloud, and even the **Internet of Things (IoT)**. Both the use of ESBs and XML-based protocols felt heavy, so lighter-weight alternatives were favored.

These challenges, along with a description of how and why the tooling also evolved into modern API platforms, are further described in Chapter 2, The Evolution of API Platforms.

It was at this point that the industry started to rapidly shift towards adopting an alternative approach called **REST**. REST was perceived by many to be lighter (no need for SOAP envelopes or even XML) and simpler (with the use of **unique resource identifiers (URIs)**, HTTP verbs, and hyperlinks, just like the web in general) to implement over SOAP/WSDL-based Web Services exposed using ESBs.

Moreover, because REST was fully built on the still-popular HTTP v1.1 protocol, its adoption was straightforward, especially for those familiar with web standards.

REST, along with other modern API architectural styles, will be described in detail in the subsequent sections of the chapter.

Latest trends

It is 15 years since W3C's recommended Web Services Architecture was released, and what we see today (as of 2019) is a prevalence of REST-based APIs, at least according to the following Google trends analysis, which compares the popularity (in terms of Google keyword searches) of the terms "GraphQL," "REST API," "OData," "WSDL," and "gRPC" in the last 10 years:

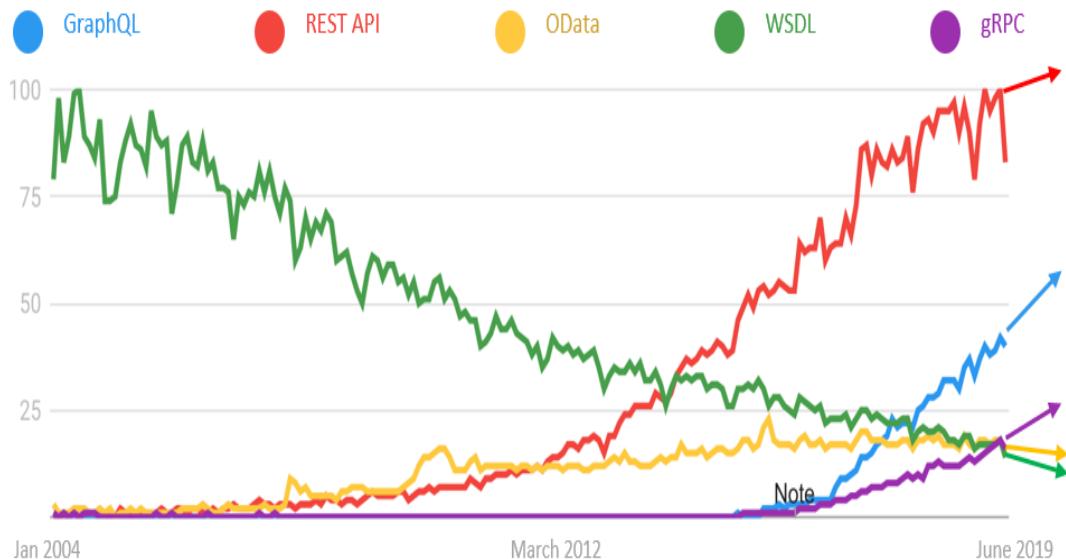


Figure 6.3: API trends for March 2019

To recreate the Google trend diagram, open the following URL in any browser. Note that the dates in the URL can be adjusted as they are in the format yyyy-mm-dd:

<https://trends.google.com/trends/explore?date=2004-01-01%202019-06-01&q=GraphQL,REST%20API,OData,WSDL,gRPC>

The analysis suggests that REST APIs have followed almost exactly the opposite trend as WSDL, having now reached the same level of popularity as WSDL in early 2004. But just like WSDL, there is some evidence that the popularity has started to decline.

The decline (or slowing down) of REST is perhaps more evident when looking at the exponential growth in popularity that GraphQL experienced not long after Facebook made it public in 2015. Moreover, the GraphQL trend appears to be even more aggressive than that of REST, to the point that if such a trend continues, GraphQL could match or even surpass REST (in terms of popularity) within the next three years or even less.

Then there is gRPC, Google's own implementation of the RPC protocol, which was also open-sourced in 2015. Although gRPC is not experiencing the same level of hype as GraphQL, the trend analysis suggests that it is indeed rapidly increasing in popularity.

What does this trend analysis really tell us?

Understanding market trends is important as it provides guidance as to where the industry is going, which, in turn, helps with making important decisions. At the end of the day, there is always an underlying reason for why certain technologies increase in popularity and others decline. It can't all be based on hype. For example, GraphQL claims to have a developer-friendly user interface and promises to deliver a far more flexible API whereby developers can more easily define exactly what data to fetch from an API without the nuances of REST.

gRPC, on the other hand, offers a robust transport protocol that leverages HTTP/2 and thus supports features such as streaming and is very efficient for service-to-service communications (which is why Google created it in the first place).

In spite of this, however, architectural decisions should never be based on hypes or trends. Architects and developers should develop a deeper understanding of the viable options available to them, and based on facts and common sense, determine what option best fits the need.

The following sections therefore focus on providing a deeper insight about the three most trendy API architectural styles at the time this book was written. These are REST, GraphQL, and gRPC.

Each of the options will be covered in a good degree of detail, but most importantly, they will be covered from a practical point of view. You will get a brief history of their inception, their architectural anatomy, and their pros and cons based on different dimensions.

REST

REST is an **architectural style** for building web APIs. REST is not a standard per se and is perhaps better defined as a set of constraints (six to be exact) that you should adhere to when defining and/or implementing REST-based APIs.

REST was first introduced by Roy Fielding in his 2000 PhD dissertation titled *Architectural Styles and the Design of Network-based Software Architectures* at the University of California Irvine.

Although the first (or at least the first publicly known) REST API was launched by eBay in the same year as Fielding's dissertation (refer to <https://thehistoryoftheweb.com/ebay-apis-connected-web>), the adoption of REST and its main alternative (SOAP/WSDL Web Services) really only gained traction toward the end of 2004 when Flickr first launched its public REST API, shortly followed by Facebook and Twitter.

Of relevance is the fact that while working on REST, Fielding was in parallel working on version 1.1 of HTTP. This is most likely the reason that REST relies so heavily on HTTP URIs and HTTP verbs, which will both be explained subsequently.

Fielding's dissertation is publicly available at the following URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Architecture

As mentioned, REST is a resource-centric architectural style that makes use of URIs and HTTP verbs (for example, GET, PUT, POST, DELETE, and PATCH) in order to perform actions against a given (unique) resource. This is in contrast to RPC-based protocols, such as SOAP, gRPC, and even GraphQL, where the action is explicit in the payload of the message. For example, a HTTP GET request against a `http(s)://<server>/customers` URI implies listing all customers available in the `/customers` resource.

Doing a HTTP POST call against the same resource means creating a new customer record, details of which should be provided as part of the HTTP payload.

The following is a process view representation of REST extracted from Fielding's dissertation.

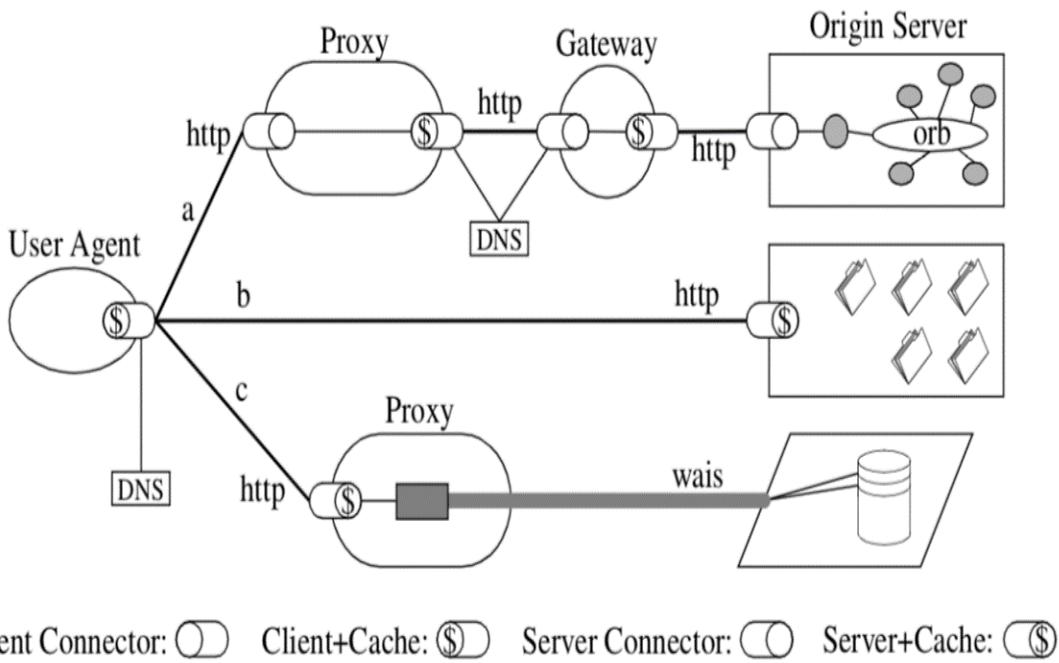


Figure 6.4: A process view of REST-based architectures

The preceding diagram is an extract from section 5.3.1 Process View from the following link:

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

The diagram illustrates a user agent (for example, a browser) that implements a client cache and accesses three resources through different interactions. The first one is against a **proxy** (this could be an Internet proxy or a load balancer, for example), which in turn interfaces with a **gateway** (for example, an API gateway) that also acts as an additional cache layer. The gateway, in turn, interfaces with the **Origin Server** (the resource ultimately being accessed), which exposes a uniform interface, which abstracts the underlying implementation.

The second interaction is directly against the Origin Server (basically against the resource directly without any middle tiers), while the third interaction is against a **proxy** that is

capable of translating the HTTP calls into the protocol Z39.50 adopted by **Wide Area Information Servers (WAIS)**.

It is critical (if not mandatory) for you to fully understand the **six constraints** mentioned earlier, as they form the basis of the architectural style illustrated.

What follows is a summary of each of the constraints as they appeared in the dissertation, along with an explanation describing the practicalities of each.

Client-server

"Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the internet-scale requirement of multiple organizational domains."

In practical terms, this means separating a client from servers. In this separation, a client (for example, a mobile app or a JavaScript client application) should not be concerned with how data is stored in the server-side, and likewise, the server should not care about user interface needs.

Basically, all the client needs to be concerned with is the API it is calling; the servers do the rest. In achieving this, the client and servers can then evolve independently. This is an important benefit and one we see reflected today in how fast user interfaces evolve when in actual practice, the servers

accessed via APIs may have not dramatically changed or not changed at all.

Stateless

"Communication must be stateless in nature such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client."

This constraint effectively means that an API should handle a single call as a unique one and therefore all of the contextual information required to process the call (for example, the authorization token, resource identifier, or search filters) should be contained within the call itself. This is a good example of how HTTP principles are reflected in REST.

There are substantial benefits to this approach. First of all, scaling the server becomes a lot easier, as the implementation of the API, ideally a fully decoupled service (microservice), can scale horizontally and on demand without having to worry about session sharing. This also simplifies tasks such as monitoring and troubleshooting because all contextual information is in the same call and there is no need to look for historical transactions to get a further understanding of what happened.

Cache

"In order to improve network efficiency, we add cache constraints to form the client-cache-stateless-server style. Cache constraints require that the data within a response to a request be implicitly or explicitly labelled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests."

Perhaps another good example of how much REST is heavily influenced by core HTTP principles is that just like browsers are able to cache the **Hypertext Markup Language (HTML)** of web pages already visited, REST also supports the concept of client cache through this constraint. In practical terms, this means that client applications are allowed (or are supposed to) cache API responses, which are marked as cacheable in the HTTP (cache-control) header. This constraint can result in considerable performance gains, as not every user action has to translate into an API call. Common API responses that don't often change could be cached by the client.

Uniform interface

*"The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, **hypermedia as the engine of application state (HATEOAS)**."*

This constraint sets REST apart from the RPC-inspired API approaches and introduces four very important concepts:

1. **Identification of resources:** In practice, this means making use of HTTP URIs in order to uniquely identify and access uniform resources (for example, <https://mydomain.com/api/orders>). The use of the term **uniform** is deliberate, as it indicates that resource interfaces should be standardized and should not reflect the

underlying implementation of the resource. In other words, a uniform interface abstracts and decouples a client from the server implementation (for example, a microservice), thus also allowing a server implementation to evolve without changing an interface.

2. Manipulation of resources through representations:

This means making use of HTTP verbs, such as POST, PUT, PATCH, or even DELETE, to create, update/amend, and even delete resources. It also implies that a client application should hold enough information that such manipulation can occur.

3. Self-descriptive messages:

This means that every message generated either by a client or server should have enough information for any processing to occur.

4. HATEOAS:

This is one of the pillars of REST APIs and yet another example of the influence of HTTP and the web on the architectural style. Hypermedia is an extension of the **hypertext** concept (the H in HTTP), which basically means cross-referencing associated sections of text and graphics within a digitally displayed text, for example, a HTML page rendered in a browser using **hyperlinks**. Hypermedia, however, expands the concept by also encompassing other media types such as video and audio. Putting hypermedia into the context of HATEOAS, it means that payloads contained within API responses should, when applicable, also include the

URIs of related resources. The main benefit of HATEOAS is that it allows client applications accessing a resource to also access (or be aware of) related resources without prior knowledge of their existence. Simply put, this is like browsing a web page and navigating through the site by just clicking on different links. HATEOAS enables a similar experience but in the context of APIs.

Layered system

"In order to further improve behavior for internet-scale requirements the layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors."

This constraint introduces the notion of **abstraction** in the REST architectural style. Briefly discussed earlier, it allows, for example, a client to be unaware of the implementation details of a server resource (which could well be a legacy system) by accessing it via uniform interfaces.

Putting the concept into practice, the uniform interface could be, for example, an API exposed via an API gateway, whereas the implementation details reside within a service that the API gateway routes to.

However, the concept goes beyond that, as the APIs in the API gateway and in the service could both be REST-based, meaning there could be multiple layers of abstraction in the end-to-end architecture.

Code-on-demand

"REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST."

Perhaps one of the lesser-known constraints in REST, code-on-demand talks about accessing a resource that when parsed on the client-side (for example, a browser) actually contains code that can be executed by the client itself. This means that a server resource is also able to delegate some of the processing logic to the client. In fact, many APIs today make use of this constraint. An example is the Google Maps API, which allows browsers to display a map and interact with it.

Interface definition

Although the REST architectural style doesn't explicitly provide an **interface definition language (IDL)**, in the past 10 years or so many languages have emerged as options for defining REST-based APIs. Of all the options, the **Open API Specification (OAS)** is by far the most popular one. Nonetheless, the following section contains a summary of the most popular open alternatives.

A full list of all IDLs for REST can be found at the following URL:

https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages#List_of_RESTful_API_DLs

OAS



Figure 6.5: The Open API Initiative logo

Originally known as the **Swagger Specification**, as it was conceived as part of an open project with the same name, in 2016, OAS became a separate project called the **OpenAPI Initiative** (openapis.org) before being given its current name.

The project is community-driven and sponsored by the **Linux Foundation** (linuxfoundation.org). This means that active members of the community can contribute to the evolution of the specification. Since it became open, several well-known organizations, such as Google, Microsoft, Oracle, and IBM, have become members.

In terms of the specification itself, OAS can be written using the **JavaScript Object Notation** ([JSON.org](https://json.org)) or **YAML Ain't Markup Language (YAML)** (yaml.org).

OAS 3.0 is the latest version (at the time of writing) and introduces some notable differences when compared with its popular predecessor, 2.0:

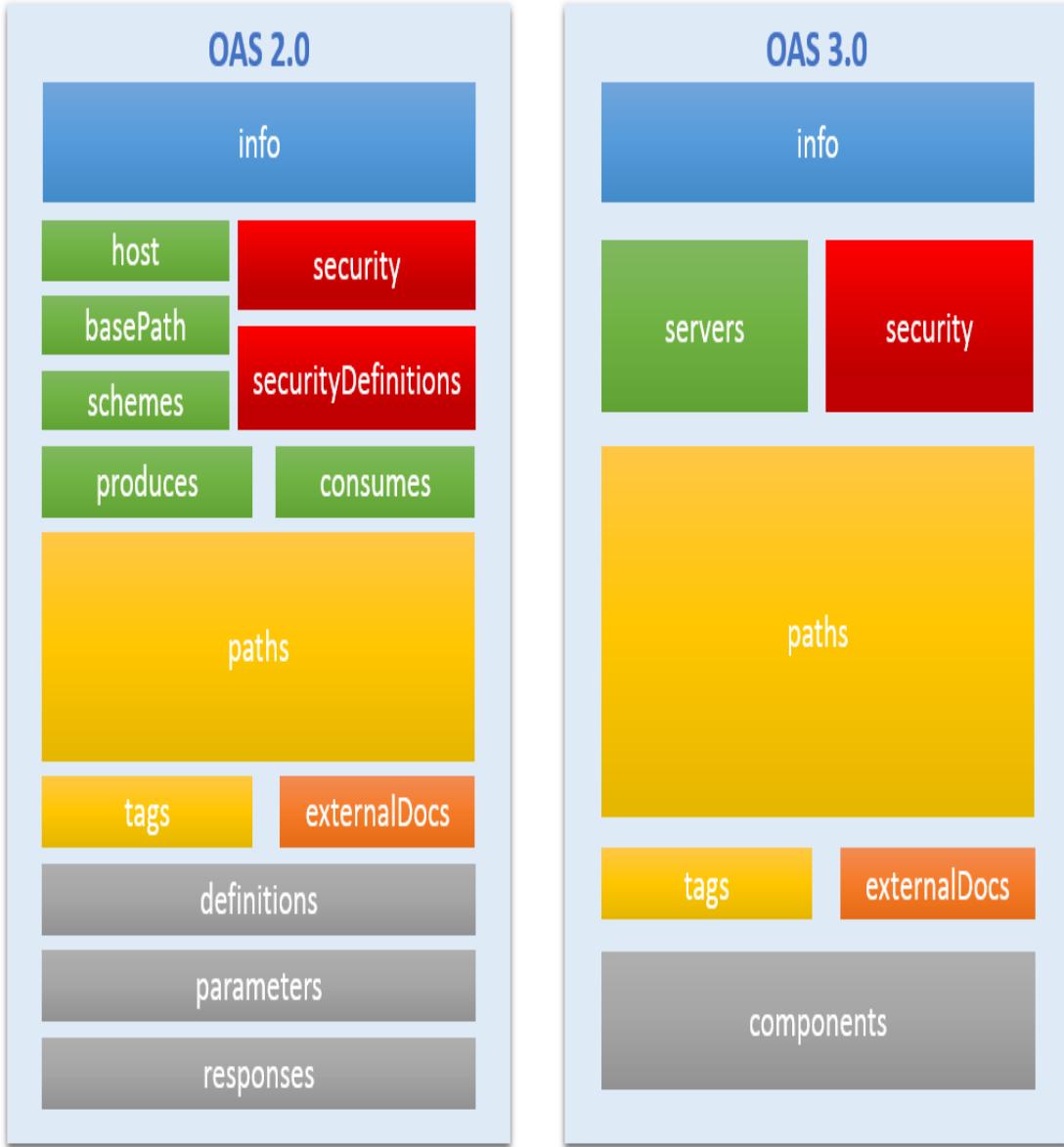


Figure 6.6: OAS 2.0 versus 3.0

OAS 3.0 aims to simplify the structure and also increase reusability of components, such as parameters, headers, examples, and security schemes.

To view and compare actual examples of OAS 2.0 and 3.0 for a simple API, you can check out the following GitHub resources:

<https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore.yaml>

<https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v2.0/yaml/petstore-simple.yaml>

Lastly, it is worth mentioning that the tooling ecosystem for OAS is quite broad (the broadest of all options) with support from the vast majority (if not all) API vendors and open source tools on the market.

More details on OAS, including the full specification and tutorials, can be found at the following link: <https://github.com/OAI/OpenAPI-Specification>

API Blueprint



Figure 6.7: The API Blueprint logo

Created to be a much simpler and user-centric alternative to the then Swagger and **Web Application Description Language (WADL)**, an XML-based language for defining REST APIs that wasn't very popular, API Blueprint is a markdown-based language for defining APIs. The specification was created by the founders of **Apiary** (`apiary.io` is now part of Oracle) in 2013 as an open-source project.

What sets API Blueprint apart from OAS and other alternatives is the fact that a single blueprint document can incorporate not just the technical specification of an API (for example, resources, verbs, paths, parameters, and payloads) but also its functional documentation. Furthermore, as the markdown markup language is very easy to learn and read even without an editor, API blueprints are very compelling for more functional audiences that wish to take part in the definition of APIs and their documentation.

From a usability standpoint, API Blueprint has a fairly large ecosystem of open-source tools available for creating, editing, visualizing, and even testing APIs defined in this language. Several API vendors also support API Blueprint.

More details on API Blueprint can be found at:

<https://apiblueprint.org>

For API Blueprint examples, check the following URL:

<https://apiblueprint.org/documentation/examples>

RAML



Figure 6.8: The RAML logo

The **RESTful API modeling language (RAML)** is a YAML-based language originally created by MuleSoft in 2013 as an open-source project and as another alternative to the then Swagger and WADL. Unlike OAS and API Blueprint, RAML claims to be unique, as APIs defined with it don't have to obey all of the REST constraints.

In practice, however, the last statement is ambiguous, as even OAS and/or API Blueprint can be used to define APIs that don't strictly follow all REST constraints. For example, it is possible to define the URI /getCustomers. Although such practice goes against the uniform interface REST principle, both languages would still allow it.

A good feature that RAML has over OAS 2.0 and API Blueprint is the ability to reuse external fragments (for example, reference another file that defines a JSON object), although this was introduced in OAS 3.0.

RAML does benefit from a healthy community and a good ecosystem of tools around it, although the language tends to be heavily associated with MuleSoft software.

For more information on RAML, please refer to: <https://raml.org/>

Transport and payloads

Strictly speaking, REST does not enforce any specific transport protocol or data format. However, in actual practice the majority (if not all) REST APIs are implemented using HTTP(S) as the transport, with JSON as the content type.

However, there are many APIs out there that also make use of XML (in some cases in addition to JSON) and other content types.

Usage flow

To show a common usage flow from the perspective of a frontend developer, the following sequence diagrams illustrate the typical steps followed when consuming a REST API. The steps include building the client code based on a REST IDL (for example, OAS) in order to support a given functionality within the frontend app itself, which in the example is a search for customers based on a string and then based on the search results (for example, JSON collection) to obtain further details for a specific customer.

Note that the steps assume that:

- There is a REST API specification (for example, in OAS 3.0 this had been previously defined).
- A service based on the API specification is up and running.
- An API gateway has also been deployed.

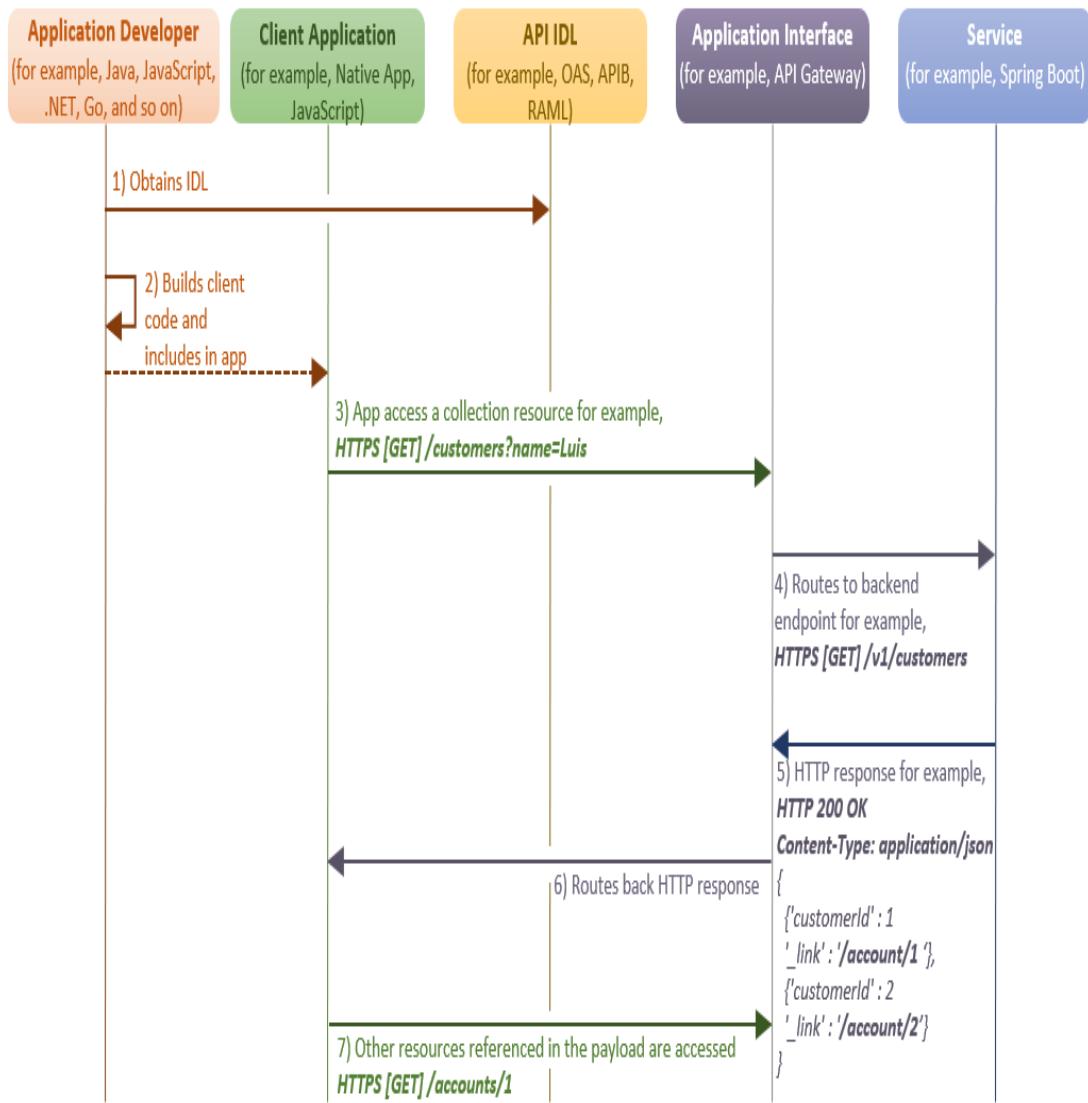


Figure 6.9: The REST API usage flow

The steps are as follows:

1. The frontend application developer obtains the interface definition for the API (for example, an OAS 3.0 document).
2. The developer creates the client-side code that consumes the different resources exposed by the API. This could be done manually (meaning the developer

has to interpret the IDL to then handcraft the code) or preferably using a utility to aid with automatically generating client code directly with an API IDL.

Note that there are many utilities out there that auto-generate client-side code from a REST API IDL. However, some popular examples are SwaggerHub and the aforementioned Apiary, both of which not only aid the design of the IDL itself, but also the generation of client and even server code.

3. An application, making use of the client-side code built by the developer, is able to consume a REST resource typically exposed through an API gateway. Usually, the first call involves calling a resource that returns a collection (for example, the result of a search for customers named "Luis").
4. Once the API gateway receives a valid HTTP request, it routes the call to the respective backend service endpoint that is implementing the resource.
5. Assuming the request is valid, the backend service processes the request and sends back a `HTTP 200` status code (meaning the request was successfully processed), including any payload in the respective format as specified in the HTTP header `Content-Type` (for example, JSON). Furthermore, the response payload may include a HATEOAS link to any related resource(s) (for example, specific URIs for individual customers found and included in the collection).

6. The API gateway routes back the HTTP response to the client application.
7. The client application makes additional calls as required to access the referenced resources.

For best practices on REST design, refer to the following eBook by Todd Fredrich from eCollege.com:

<https://www.restapitutorial.com/resources.html>

*For a reference maturity model for REST APIs, refer to the **Richardson Maturity Model**:*

<https://restfulapi.net/richardson-maturity-model/>

For bad practices to avoid and how to do so, refer to this article:

<http://www.soa4u.co.uk/2018/10/the-se7en-deadly-sins-of-api-design.html>

GraphQL

The **Graph Query Language (GraphQL)** was created by **Facebook** around 2012 by engineers Lee Byron, Dan Schafer, and Nick Schrock. It was open-sourced three years later, in 2015.

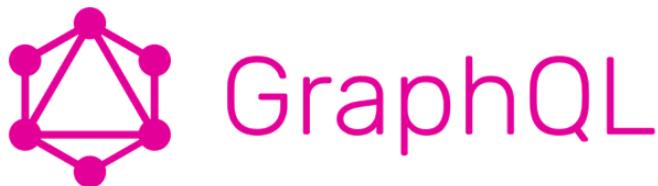


Figure 6.10: The GraphQL logo

The GraphQL project is accessible through its official website:

<https://graphql.org/>

As described by Byron himself in one of his articles, GraphQL came at a time when Facebook's iOS and Android mobile apps were just thin wrappers (mobile WebView) of Facebook's web application. As the number of users grew, so did the complexity of the mobile WebView and thus it started suffering from poor performance and it constantly crashed.

To overcome this challenge, Facebook started transitioning its apps into natively implemented views (as opposed to just web wrappers). However, this raised the need for a mechanism other than HTML to fetch data. In other words, there was a need for APIs.

It was at this point that Facebook started evaluating different API architectural options, including the implementation of a RESTful server. However, Facebook developers concluded that REST didn't satisfy its app data access needs, as there were considerable gaps between the data it wanted to access and the API calls required in order to do so. Furthermore, a considerable amount of code was required at both client- and server-side in order to prepare the data as required by the app and then for the app itself to parse it.

This frustration inspired a few developers at Facebook to create an alternative approach to access data in a far more flexible and dynamic way. Most importantly, the focus was the perspective of app designers and developers. This project ultimately became GraphQL.

The name GraphQL comes from the fact that Facebook's developers didn't think of data as resource URLs or joint tables, but rather as a graph of objects that could be queried within application models.

The following article describes in great detail the original motivations for GraphQL as a means to overcome the limitations of REST:

<https://code.facebook.com/posts/1691455094417024>

Architecture

In a nutshell, a GraphQL API consists of a service that runs a **GraphQL server**. The server, in turn, implements the different **types**, as defined in the **service schema**, and the **resolvers** responsible for executing the functionality of the **operation types**.

In addition, most GraphQL servers can expose the **GraphiQL** (pronounced graphical) web client, which is an interactive in-browser **interface development environment (IDE)** for constructing GraphQL operations.

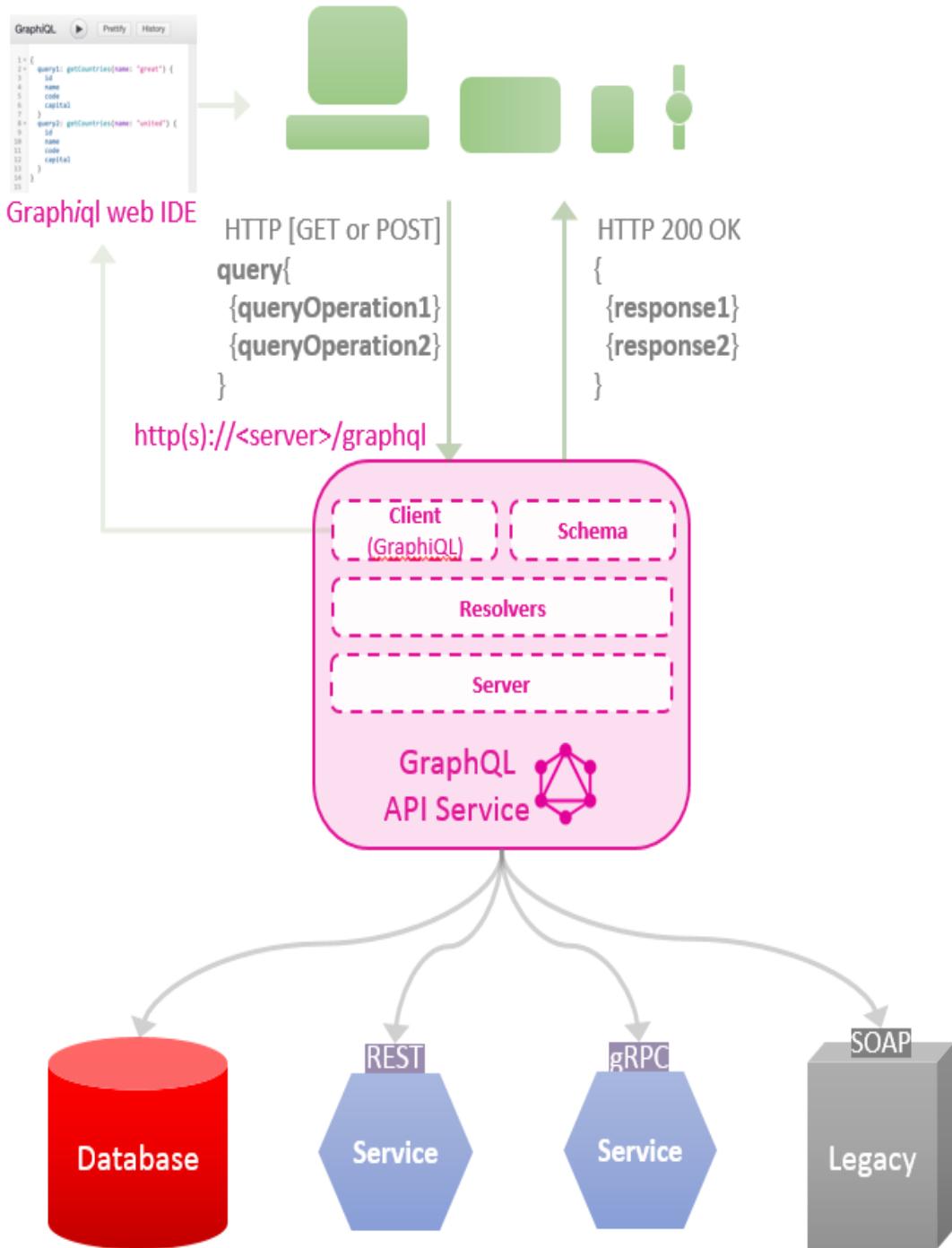


Figure 6.11: The GraphQL architecture

The preceding diagram illustrates consuming applications (for example, desktop, tablet, mobile, and even wearable apps) performing multiple queries against a GraphQL service. The diagram also shows how the service may connect to multiple

backends (for example, a database, REST APIs, gRPC APIs, or even a legacy system via a SOAP interface).

Other important architectural characteristics of GraphQL that can be deduced from the diagram are as follows:

- A single URL is used to access all operations offered by the GraphQL service.
- Operations are expressed within the HTTP payload. There is no use of URIs.
- Multiple operations (for example, multiple queries) can be performed in a single HTTP call.
- GraphQL supports two HTTP verbs: GET (in which case the payload is passed as GET parameters) and POST (in which case the payload is passed as part of the HTTP payload).

It is important to note that in its nature, GraphQL is more similar to RPC-based protocols, such as SOAP and gRPC, than to REST. In fact, it could be said that GraphQL is, at the very least, RPC-inspired given the number of similarities.

To see how GraphQL works, look at the following example created in graphqlhub.com (a public GraphQL API):

GraphQLHub

Operation

Queries

Service

Responses

```

1 query ($qKey1: String!,  

2   $qKey2: String!,  

3   $qKey3: String!,  

4   $count: Int) {  

5     twitter {  

6       q1: search(q: $qKey1, count: $count) {  

7         user {  

8           screen_name  

9             followers_count  

10          }  

11          retweet_count  

12          text  

13        }  

14        q2: search(q: $qKey2, count: $count) {  

15          user {  

16            screen_name  

17          }  

18          text  

19          id  

20          created_at  

21          retweet_count  

22        }  

23        q3: search(q: $qKey3, count: $count) {  

24          id  

25          created_at  

26          text  

27          retweet_count  

28        }  

29      }  

30    }
  
```

QUERY VARIABLES

```

1 {  

2   "qKey1": "GraphQL",  

3   "qKey2": "REST",  

4   "qKey3": "gRPC",  

5   "count": 1
6 }
  
```

Variables

Figure 6.12: A GraphQL sample using a GraphiQL client

Notice that on the left-hand side the operation to be performed on the GraphQL service, in this case a query, is defined within the payload itself. This is because in GraphQL, all operations share only a single endpoint (typically `<server>/graphql`) as

opposed to REST, wherein each resource is a URI. Furthermore, and as shown, multiple individual queries, each with different parameters and fields, can be made in a single request. The response from the service, which is shown on the right-hand side, corresponds exactly to the queries made.

This sample can be accessed from the following URL:

<https://tinyurl.com/graphql-sample>

Architectural principles

GraphQL is **not** a programming language but rather a language for defining GraphQL operations, such as queries, mutations, or subscriptions against a GraphQL server. Furthermore, GraphQL was designed with the following principles in mind:

Hierarchical

Queries are hierarchies of data definitions, shaped just how data is expected to be returned.

View-centric

GraphQL is unapologetically driven by the requirements of views and the frontend engineers that write them. GraphQL started with their way of thinking and their requirements, before building the language and runtime necessary to enable that.

Strongly-typed

A GraphQL server defines a specific **type system**. GraphQL operations (for example, queries) are executed within this context.

Client-driven

Through the type systems, a server publishes an interface for clients to construct operations against. Then it is the responsibility of the clients to define how exactly to implement an operation, including defining exactly what payload to fetch in the case of queries.

Introspective

The type system itself is queryable. Tools are built around this capability, such as the GraphQL client.

Version-free

From its inception, the GraphQL specification has taken a strong stand against implementing versioning at service level; instead, it provides tools and mechanisms for continuous evolution of the service and thus provides the means to support backwards and future compatibility.

Miscellaneous

GraphQL is intentionally silent on important architectural considerations, most notably caching, authorization, and pagination. This is the case because they are considered to be outside of the scope of the specification itself, even though a variety of solutions may already exist.

More information on the GraphQL specification can be found at the following link: <https://graphql.github.io/graphql-spec>

Interface definition

Interfaces are defined using the GraphQL **schema definition language (SDL)**, which is part of the specification itself.

Unlike REST, there isn't any other standard for defining GraphQL interfaces. Even though GraphQL services can be implemented in a wide variety of programming languages, such as JavaScript, Java, Scala, Python, Go, Ruby, and PHP, they all comply with the GraphQL specification and type system.

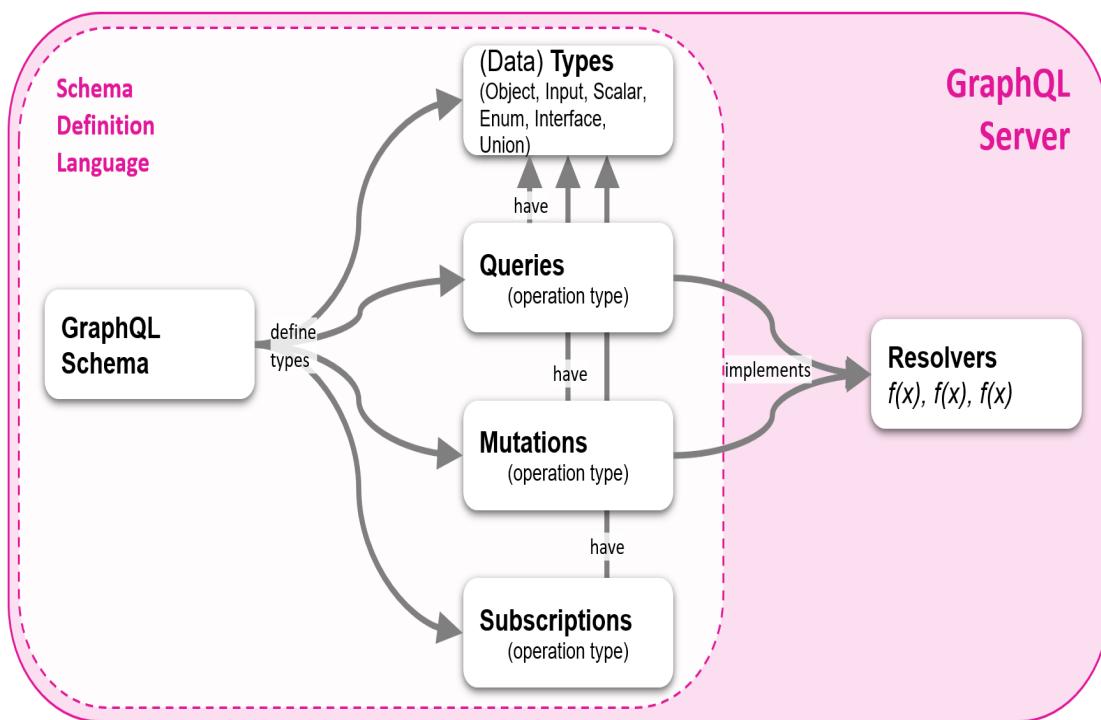


Figure 6.13: SDL defining types

As the preceding figure shows, the GraphQL SDL mainly defines types and their relationship. Although a full description

of all types available is not included in this chapter, what follows is a summary with the two main categories.

Types that define operations

Operations are types that enable a GraphQL service to expose any given functionality that is to be offered by a GraphQL service.

Think of them as public functions within a class.

There are three operation types supported in GraphQL:

Query

This type is used to define operations that fetch data and thus are read only.

Mutation

This type is used to define operations that create and/or manipulate data, such as create, update, and delete. A mutation may also respond with a data fetch.

Subscription

GraphQL's implementation of web events (for example, Webhooks) allows clients that subscribe to a given event to then asynchronously receive updates.

For more information, refer to the section on operations on GitHub:

<https://graphql.github.io/graphql-spec/June2018/#sec-Validation.Operations>

Types that define data

Data in GraphQL can be defined using the following types:

Object

This type represents any kind of object that can be **fetched** from a GraphQL service through a **query** operation, including the fields within the object and/or relationships to other objects. Furthermore, fields within an object can have zero or many **arguments**.

Input

Similar to the object type, this type is used to define objects to be used as **inputs** in a **mutation** operation.

Scalar

This type defines the type of data within the fields of an object or input type. There are five default scalar types:

1. **Int**: A 32-bit integer.
2. **Float**: A double precision floating-point value.
3. **String**: A UTF-8 character sequence.
4. **Boolean**: True or false.

5. **ID**: Represents a unique identifier and is serialized as a string.

In addition, GraphQL allows for custom types to be defined within a given schema; for example, a scalar of type date could be defined that would serialize as an integer timestamp.

Enumeration

Referred to as **enums**, they represent a special type of scalar that is restricted to a particular list of values.

Interface

This is an abstract type that predefines a set of fields that an object type must include when it implements the interface.

Union

The unions type allows different object types to be combined as an abstract object.

Refer to the following link for more information and examples of GraphQL types:

<https://graphql.org/learn/schema/>

Transport and payloads

GraphQL is typically served through a single HTTP(s) endpoint with JSON as the content format (for both request and response payloads). Though, strictly speaking, the specification does not impose this.

Usage flow

To show a common usage flow from the perspective of a frontend developer, the following sequence diagrams illustrate the typical steps followed when consuming a GraphQL API. The steps include using the GraphiQL web IDE to inspect the API and try out different queries, and then incorporate an already-tested query within a frontend application.

In the example, you can see a consuming application that wishes to find all countries that contain within their names the text "great" and then convert their currencies to US dollars. This is achieved through a single GraphQL HTTP [POST] request that contains two distinct queries, as will be subsequently described.

Note that the steps assume that a GraphQL service was previously designed and deployed, and also that the backend APIs consumed from the GraphQL service already exist.

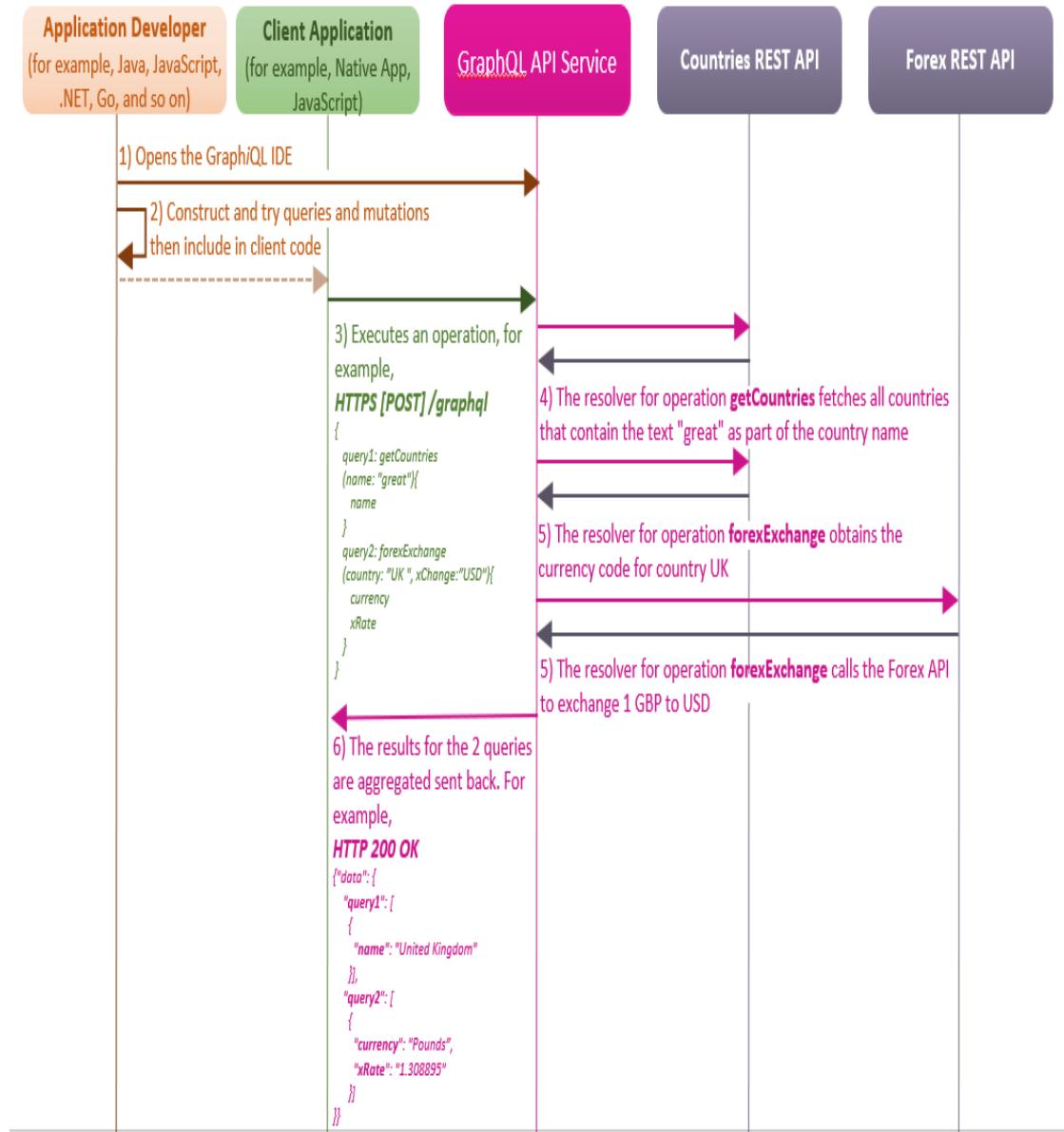


Figure 6.14: The GraphQL API usage flow

The steps are as follows:

1. A frontend application developer accesses the **GraphiQL IDE** web client using a browser. Using the IDE, the developer tries out different operations (in the example of a type query) until the right one is defined. In the example, two queries are defined: one for

obtaining all countries that contain the text "great" and another to obtain the forex exchange rate for the UK's currency into US dollars. Once this is done, the JSON generated in the IDE representing the operations can be extracted and incorporated into the bit of the **frontend application** code responsible for implementing the functionality that requires the information delivered by the queries.

2. The **frontend application** makes a single `HTTP(s) [POST]` request against the `/graphql` endpoint for the service containing both queries as a JSON payload.
3. The **GraphQL service** implements the **resolver** corresponding to the operation **getCountries**. In the example, the resolver fetches the required data against a REST API (for example, `HTTP [GET]` against `/countries?name=great`).

Note that in the example a REST API is used but, in practice, it could be any data source.

4. In parallel, the GraphQL service implements the resolver corresponding to the `forexExchange` operation. In the example, the resolver fetches the required data against two distinct REST APIs: first against the REST Countries API to obtain the country's currency code (for example, `HTTP [GET]` against `/countries/{country code e.g. uk}/currency`), and then against a forex exchange API to

obtain the exchange rate (for example, `HTTP [GET]` against `/exchange?from=GBP&to=USD&amount=1`).

5. The results of both queries are aggregated into a single JSON payload containing exactly the fields requested. This is then sent back to the caller.

gRPC

gRPC is a modern RPC protocol that runs on top of **HTTP/2**. It was created by **Google** and open-sourced in 2015 as a **Cloud Native Computing Foundation (CNCF)** project.

The gRPC project is accessible through the official website:

<https://grpc.io>



Figure 6.15: The gRPC logo

gRPC has its origins in a proprietary protocol called **Stubby**, a multi-language RPC framework also developed by Google around 2001 with the aim of addressing scaling and communication challenges when building loosely coupled distributed systems.

Stubby, at its core, consisted of an RPC layer capable of handling Internet-scale volumes (up to the billions of requests per second). When open-sourced in 2015 under the name gRPC, the protocol maintained many of its original characteristics, with the difference that it now embraced emerging and complementary open standards such as HTTP/2.

As the protocol is HTTP/2-based, it benefits from the following improvements:

- A **binary protocol**, thus making the transport far more compact and efficient.
- **Header compression**, thus reducing transmission overheads.
- **Multiplexed requests over a single TCP connection**, thus making communications much faster and more efficient.

For more information on HTTP/2, refer to the official FAQ page:

<https://http2.github.io/faq>

- **Bidirectional streaming**, allowing clients and servers to communicate in both directions.
- In-built **flow control**, thus preventing slow receivers from being overwhelmed by faster senders.
- A **strongly typed** interface definition language.
- A **single compiler** (protoc) to generate a client and servers in multiple languages.

*Perhaps because of these features and its high-scaling capabilities, gRPC is particularly popular in **microservice architectures** as the means to enable service-to-service communications in high-throughput environments. For additional information on gRPC and answers to many other questions, refer to the official FAQ page: <https://grpc.io/faq/>. For more information on the motivations and design principles behind gRPC, please refer to the following URL:*

<https://grpc.io/blog/principles>

Architecture

Just like any other RPC protocol, gRPC consists of a server that specifies methods that can be remotely invoked by clients. To achieve this, gRPC introduces the notion of **gRPC client stubs** and **gRPC servers**.

An important feature of gRPC, and probably one of the main reasons for its popularity in microservice architectures, is its support for the following four interaction styles:

1. **Unary**: A traditional synchronous request/response interaction.
2. **Server streaming**: A client sends a request to the server and the server in response sends back a message stream. The client reads the message stream until there are no more messages.

Note that gRPC guarantees message ordering within an individual RPC call.

3. **Client streaming**: The client writes a sequence of messages and streams them to the server. Once the client finishes writing messages, it waits for the server to return a response.

Likewise, in client streaming gRPC guarantees message ordering within an individual RPC call.

4. **Bidirectional streaming:** Both the client and servers communicate in both directions and completely asynchronously using read and write streams.

For more information on basic gRPC concepts, please refer to the following link:

<https://grpc.io/docs/guides/concepts.html>

The following diagram illustrates the gRPC architecture, including how gRPC stubs can be implemented by services and/or consuming applications to interact with gRPC servers that are also implemented by services.

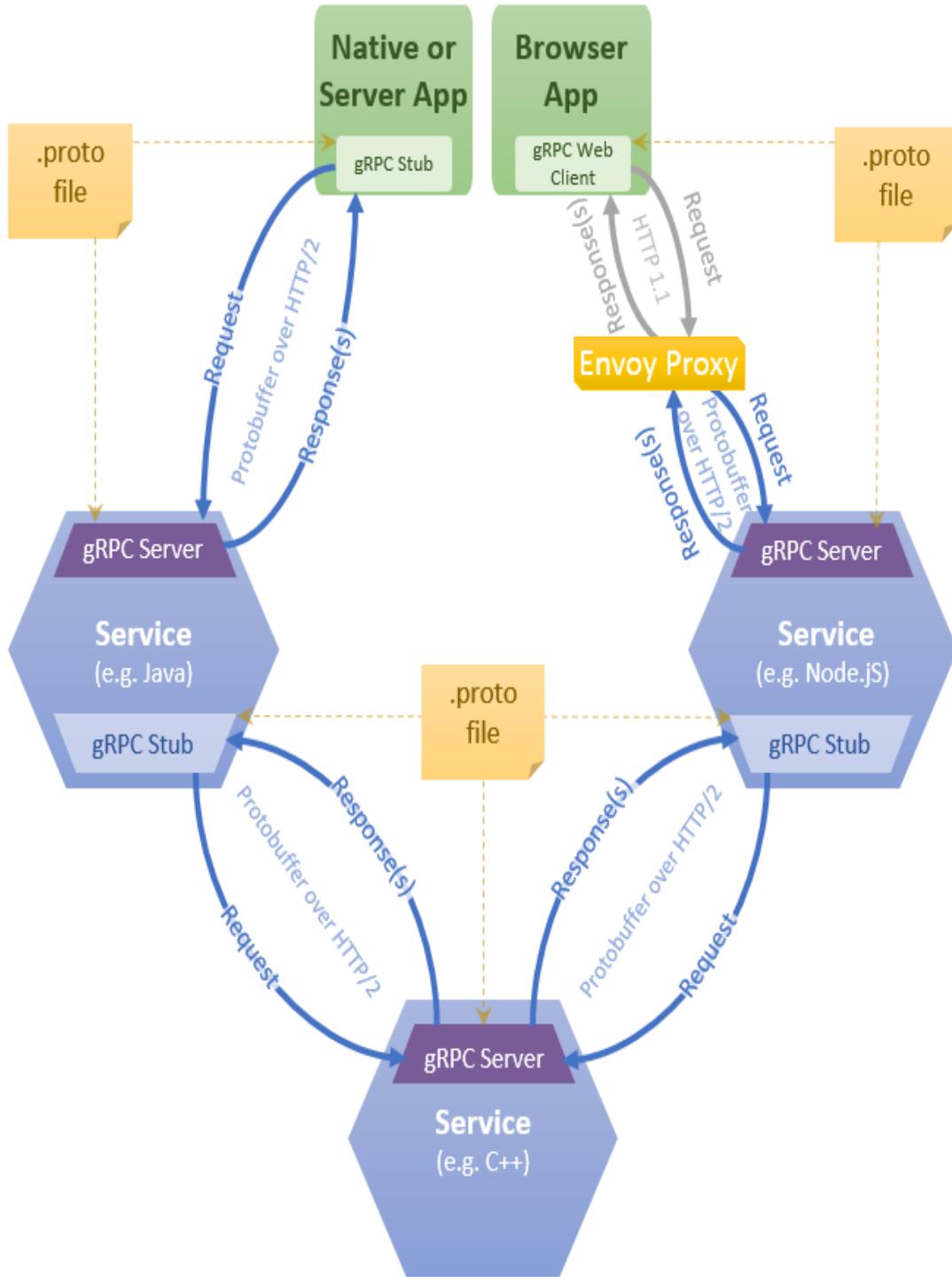


Figure 6.16: The gRPC architecture

The preceding diagram illustrates three gRPC services, two of which are actually wrappers of a common service and therefore also implement a gRPC stub. The reason for this is to illustrate

a scenario wherein a service is tailored for a client. In practice, this may not be required.

The diagram also shows that whereas native applications can implement gRPC stubs, browser applications can't and require the use of the gRPC web client and an Envoy proxy as HTTP 1.1 to HTTP/2 bridge (this is described in more detail in the usage flow).

The gRPC architecture has the following characteristics:

- **HTTP/2** is used as transport in gRPC.
- On top of HTTP/2, gRPC makes use of **protocol buffers** to define the service interface and as a payload format.
- gRPC servers and stubs can be generated directly from the `.proto` file in a variety of programming languages (using the `protoc` command line).
- gRPC servers and stubs can be implemented by services calling other services.
- gRPC stubs can be implemented by native frontend applications (for example, Android apps) and server-side applications (for example, traditional MVC apps) calling gRPC servers.
- Browser-based applications can't implement gRPC stubs; instead, the **gRPC-Web** JavaScript client library

has to be adopted, which lets browser clients access a gRPC server via an Envoy proxy acting as a HTTP 1.1 <> HTTP/2 bridge.

For more information on gRPC-Web, refer to the following link:

<https://github.com/grpc/grpc-web>

Interface definition, transport, and payload

As previously mentioned, gRPC makes use of protocol buffers, which are Google's mature and open-source mechanism for serializing and de-serializing structured data.

In gRPC, protocol buffers are used to:

- Define a service interface (basically an IDL) and the structure of messages.
- Serialize and deserialize the message payloads.
- Generate source code, in multiple languages, for both gRPC servers and stubs.
- Manage message versioning to ensure compatibility of servers and stubs.

In gRPC, interfaces are defined using a `.proto` file, which is a text file that defines the data structures that are to be serialized and de-serialized using protocol buffers, and also the methods supported.

Once the `.proto` file is created, then the protocol buffer compiler, `protoc`, can be used to generate data access classes in a variety of programming languages.

Note that although protocol buffers have been around for a while and multiple versions exist, the latest version (at least at the time this book was written) is version three, referred to as **proto3**.

For more details on protocol buffers, refer to the official developer guide:

<https://developers.google.com/protocol-buffers/docs/overview>

Usage flow

Given that the implementation steps in gRPC vary depending on whether a consuming application is browser-based or not, the former has been assumed in order to be consistent with the REST and GraphQL examples provided earlier, both of which depict usage flows based on browser-based client apps.

The example illustrates a common usage flow from the perspective of a frontend developer wishing to generate a gRPC client that can consume a gRPC service from within a JavaScript browser-based application.

The steps assume that:

- A `.proto` file has been previously created by a service designer and is accessible for use.
- A gRPC server was previously generated from the same `.proto` file and is already up and running.
- An **Envoy proxy** has already been implemented and thus is also up and running.
- The client makes **unary** (request/response) calls.

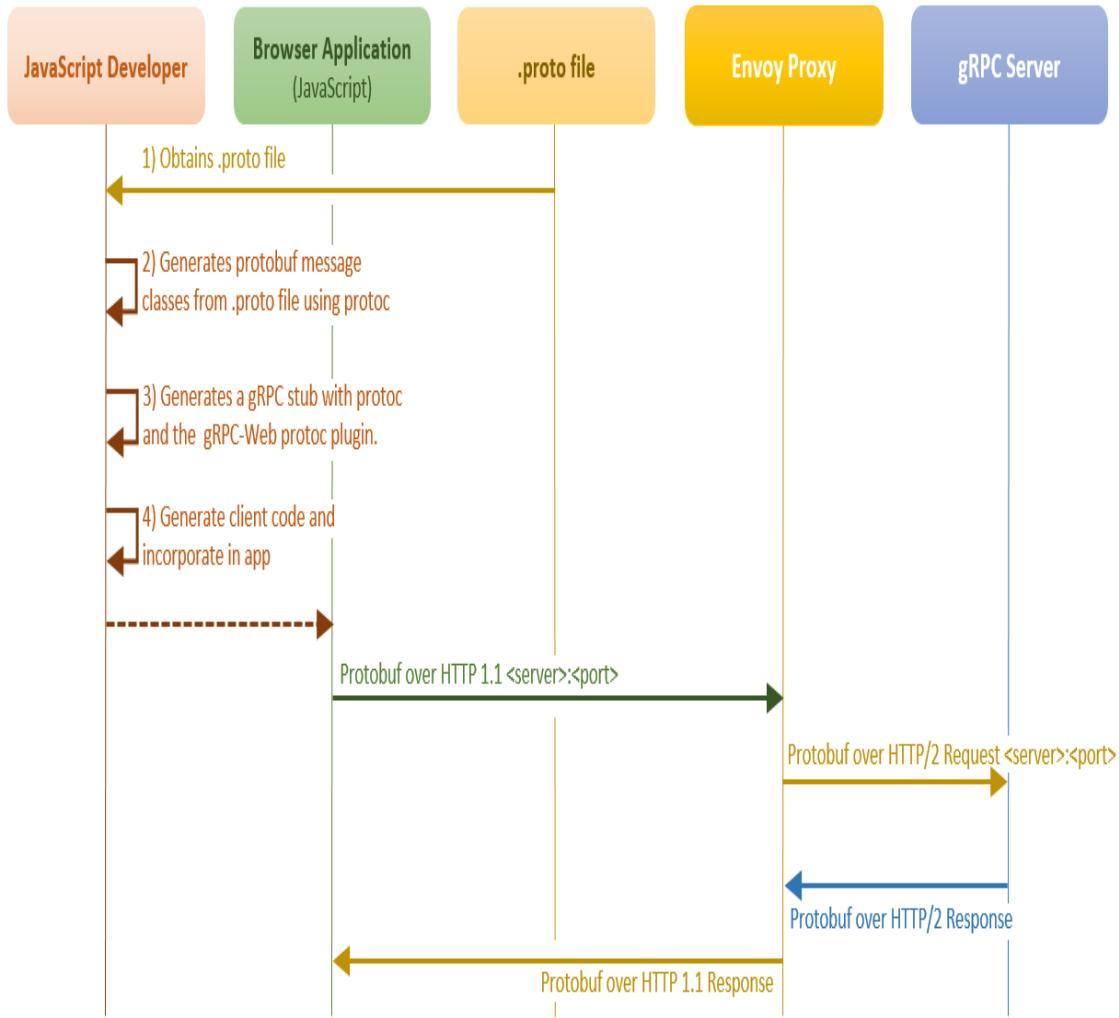


Figure 6.17: The gRPC usage flow

The steps are as follows:

1. A frontend application developer obtains the `.proto` file containing the data structures and methods supported by the gRPC server (the API).
 2. The developer generates protocol buffer **message classes** for their language of choice using the `protoc` command line.

3. The developer then generates the **gRPC-Web client stub** using the `protoc` command line with the **gRPC-Web protoc plugin**.
4. The developer then writes the **client code** that is to be incorporated in the app itself.
5. The client application makes a **HTTP 1.1** request against the Envoy proxy URL, which contains as payload the protocol buffer message. The message itself contains information about what method is being invoked at the server-side, as well as any other required data.
6. The Envoy proxy converts the request from **HTTP 1.1** to a fully compliant **HTTP/2** protocol buffer request that is then forwarded to the gRPC server.
7. The gRPC server receives the request and processes it, and then sends a **HTTP/2 response**.
8. The Envoy proxy converts the HTTP/2 response into a HTTP 1.1 response and sends it back to the client app.

For detailed steps on how to implement a gRPC server and an Envoy proxy, please refer to the following link:

<https://github.com/grpc/grpc-web/blob/master/net/grpc/gateway/examples/echo/tutorial.md>

Comparing the options

Now that I've introduced to a good degree REST, GraphQL, and gRPC, a more objective comparison of the three can be made.

As there are no silver bullets; each of the options will be evaluated against multiple criteria, aiming to cover key aspects of the API life cycle and architecture, such as developer experience from the perspective of both frontend and backend developers, common patterns, fitness, authentication and authorization, caching, and versioning.

The scoring is simple: just based on pluses (+) and minuses (-). Two pluses are the maximum score (++), indicating that the option delivers the best possible answer to a given criterion. The worst score is two minuses (--), indicating the opposite. When there aren't any obvious pluses or minuses, the tilde (~) is used to mark a neutral answer.

Developer experience: frontend developer

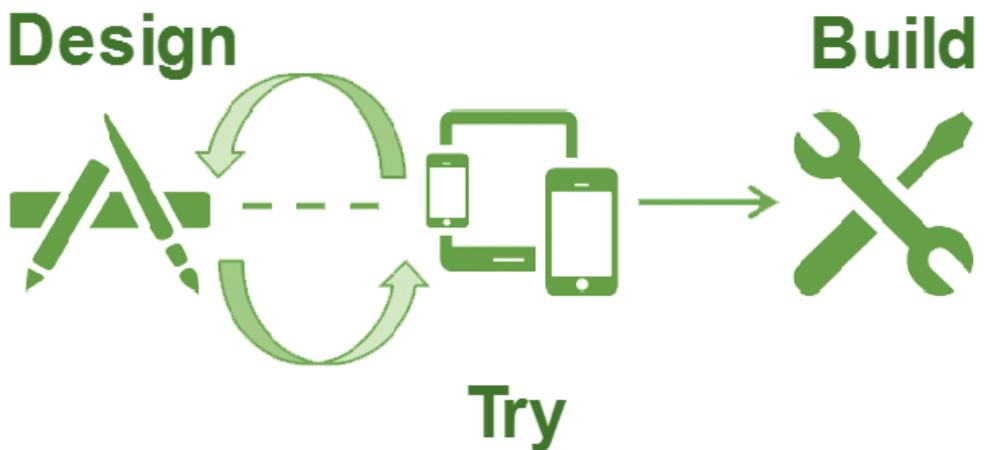


Figure 6.18: Developer experience

The criterion evaluates how easy or complex it is for a frontend developer to understand what the API does and how to use it, try it out if possible, and then write client application code to start using it within an application. A great experience means that the developer can almost intuitively (without having to read tons of documents) start trying out the API, meaning that the API itself should also deliver some form of playground capability to enable this.

This criterion also means that it should not take too long for a developer to start building client application code that uses the API. In some cases, there is even tooling to auto-generate client code, making it even easier to start using the API.

A bad experience means that the developer has to spend a considerable amount of time reading documentation to just be able to try out the API. In the worst case, there even isn't a facility for the developer to try the API, meaning that actual client code has to be written to try the API.

REST	GraphQL	gRPC
<p>-</p> <p>The API usability experience highly depends on how well documented an API is and the features that are available, along with the documentation (for example, the playground to try out the API). If the API is not properly documented, it can take a while to start using it. The scoring is driven by the fact that whether or not an API is well documented depends entirely on the producer of the API.</p> <p>Furthermore, HATEOAS can lead to very chatty APIs with many HTTP calls, thus requiring multiple calls to fetch all the needed data.</p>	<p>++</p> <p>The GraphQL IDE makes it extremely easy to interactively understand and construct queries and mutations against a GraphQL service. Even if a developer doesn't know the data structures or methods supported by the API, the editor helps with introspecting the schema and aids in the creation of queries and mutations.</p>	<p>-</p> <p>Before a developer can even start trying out a gRPC API, they have to first generate client stubs to use in client code. For browser-based applications, this is even worse, as the gRPC-web plugin has to be used in order to generate stubs that can be used in JavaScript client apps. When compared with REST and GraphQL, it is clear that gRPC is behind in this area.</p> <p>Note that new features, such as Server Reflection, are being added, which can help in this space. However, in general, it is still early days.</p>

Developer experience: backend developer

This criterion evaluates how easy or complex it is for a backend developer to design an API and then build server code that implements it. A great experience means that the developer can very rapidly design an API using rich tooling that aids throughout the design process, meaning that faster and higher quality designs can be produced. Furthermore, it is possible to auto-generate server code and even automate tests to ensure that the API design and server implementation match.

A bad experience means that the developer has to spend a considerable amount of time writing code from scratch in order to implement an API and it may take several steps. This indicates that there isn't tooling (or the tooling is poor) to verify the API design versus server code compliance, thus increasing the chance of defects.

REST	GraphQL	gRPC
++	-	~
A vast amount of tooling is available to aid in the design of the REST API, including the ability	When it comes to API design and testing, the tooling for GraphQL is still maturing. So, in	This is similar to GraphQL in the sense that the

<p>to mock APIs, create playgrounds, auto-generate server code, and also test against API designs. However, because there are so many standards to define REST APIs, there also are inconsistencies market-wise concerning what standard to use and why.</p>	<p>order for a developer to design an API code, it has to be written. The same is true for creating an API mock, which isn't ideal in the early stages of design when it is desirable to try out different options and quickly get feedback from multiple users of the API.</p>	<p>tooling is still maturing; however, in the case of gRPC the <code>protoc</code> command line makes it easier to generate servers that comply with the interface design by default.</p>
--	---	---

API gateway

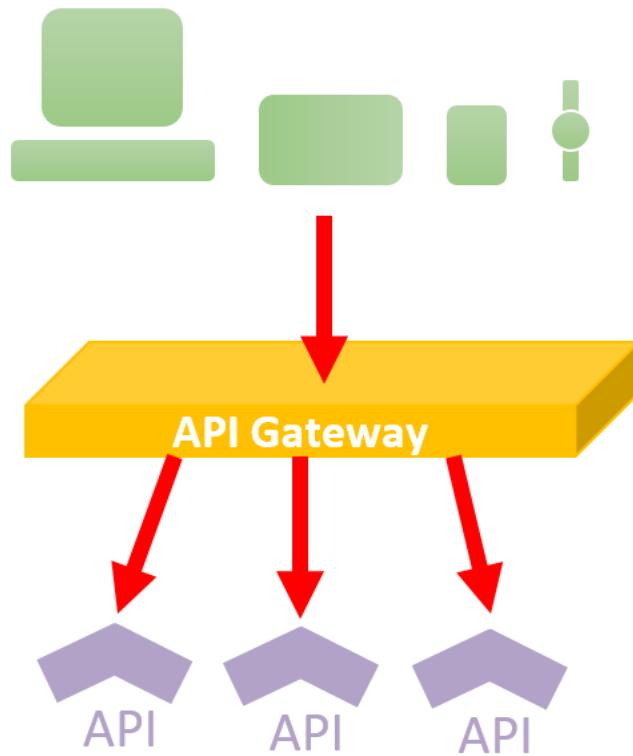


Figure 6.19: The API gateway

This criterion evaluates the feasibility of leveraging an API gateway as the means to expose, mediate, and manage APIs. This criterion is particularly important as most organizations today have already made investments into API management products that typically come with API gateways.

Therefore, those options that can fully leverage existing API gateway infrastructure will get the higher score over other options that require new infrastructure or technologies in order to make use of them.

REST	GraphQL	gRPC
++ API gateways, by default, offer capabilities in support of REST APIs (for example, OAuth, API keys, throttling, and security).	- The majority of API gateways available today don't provide native support to GraphQL. However, because GraphQL is also accessed via a HTTP endpoint (though there is just one endpoint for all methods), at least some capabilities of the API gateway can be leveraged.	-- In spite of the fact that gRPC uses HTTP/2 as its transport, many API gateways still don't support it. Even if they do support HTTP/2, at the time of writing, very few (if not none) provide support for protocol buffers and the different interaction styles supported by gRPC.

API composition

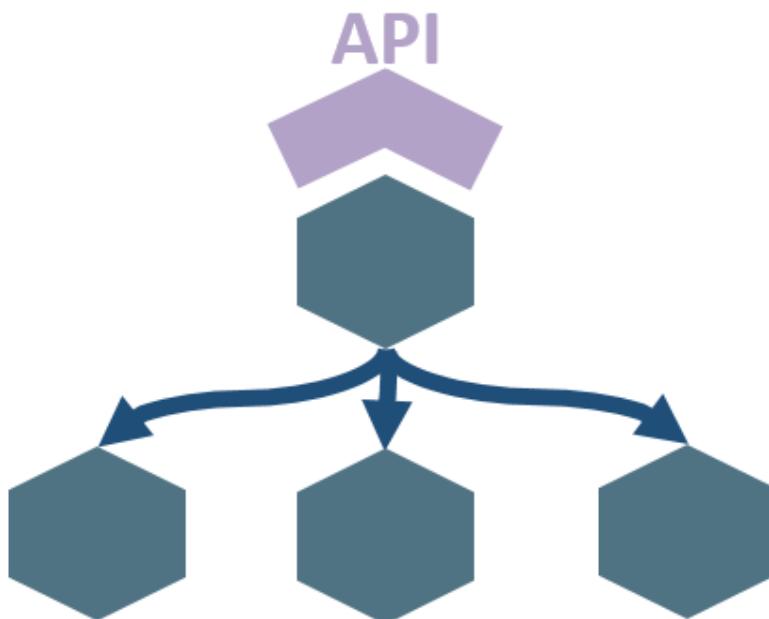


Figure 6.20: The API composition pattern

This criterion evaluates how suitable and if so, to what extent, a given option lends itself to the adoption of an API composition pattern (refer to the previous chapter for more details). Options that almost natively can support API composition have the higher score. Options that require more considerations or

adaptations, or would be more complex to implement, have the lower score.

REST	GraphQL	gRPC
-- As REST is entirely resource-based, it is not natural to define endpoints that combine data structures coming from multiple resources.	++ GraphQL is perfectly suited for API composition, almost as if it was created with this objective in mind. This is because in GraphQL, each field within a query can in fact be fetched (in parallel) from multiple sources. Most importantly, this is part of the standard behavior of GraphQL and not a customization or adaptation.	~ Being an RPC-based protocol, gRPC doesn't really impose any restrictions about how a method is to be implemented.

Authentication/authorization

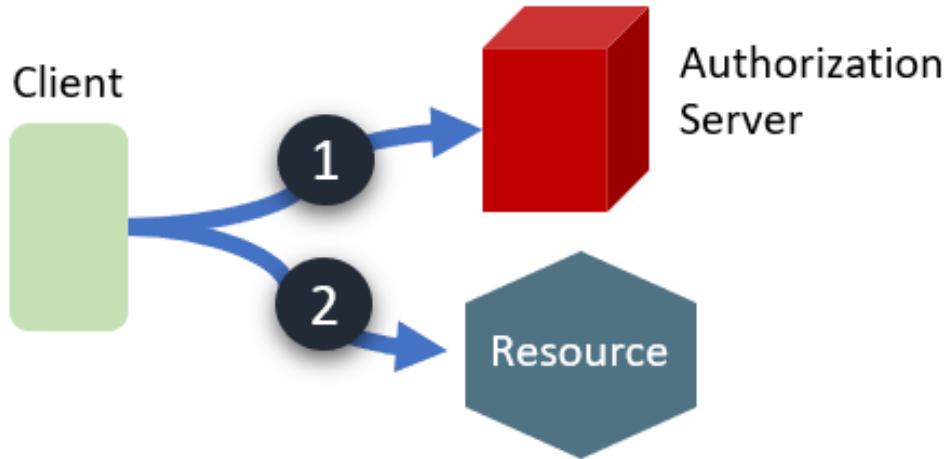


Figure 6.21: Authentication and authorization

In the world of APIs, the most common mechanism to implement the authentication and authorization of APIs is by adopting open standards such as **OAuth 2.0** and **OpenID Connect**. Options that can adopt the previously mentioned standards almost natively by leveraging existing tooling or frameworks get the highest score. Options that require either additional tooling, considerations, or custom code get the lowest score.

REST	GraphQL	gRPC
++	-	--

<p>Standards such as OAuth and OpenID are well aligned with REST and can be easily implemented with existing REST tooling. In fact, the vast majority of API gateways support these standards out of the box.</p>	<p>Because GraphQL is accessed through a HTTP endpoint, standards such as OAuth and OpenID can be adopted because all operations can be accessed by a single URI, but custom authorization modules are typically required within the GraphQL service implementation itself.</p>	<p>OAuth and OpenID can both be implemented in gRPC; however, custom code is required, thus adding to the complexity.</p>
---	---	---

Caching

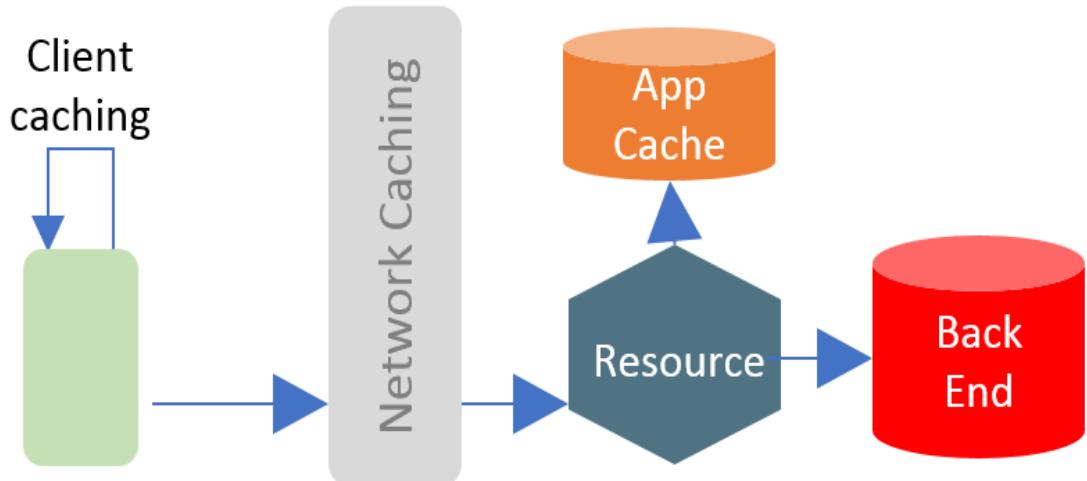


Figure 6.22: Caching

Given the volumes that Internet-based solutions may have to handle, the ability to implement caching, either at server-side or client-side, can dramatically improve the performance of an application. Options that can adopt broadly available tools and frameworks get the highest score. Options that require either additional or custom tooling, considerations, or custom code get the lowest score.

REST	GraphQL	gRPC
++ REST being so aligned with HTTP 1.1 means that it benefits from all the	~ Caching is down to the implementation of the code for both the server	- Just like in GraphQL, caching in gRPC is down to the implementation of

caching support that comes with network appliances and even web browsers. The majority of API gateways also support response caching.	and the client. However, tooling is evolving rapidly and there are many implementations today that support server- and client-side caching out of the box.	the code for both the server and the client. However, when compared with GraphQL, there isn't the same number of implementations that support caching.
---	--	--

Versioning

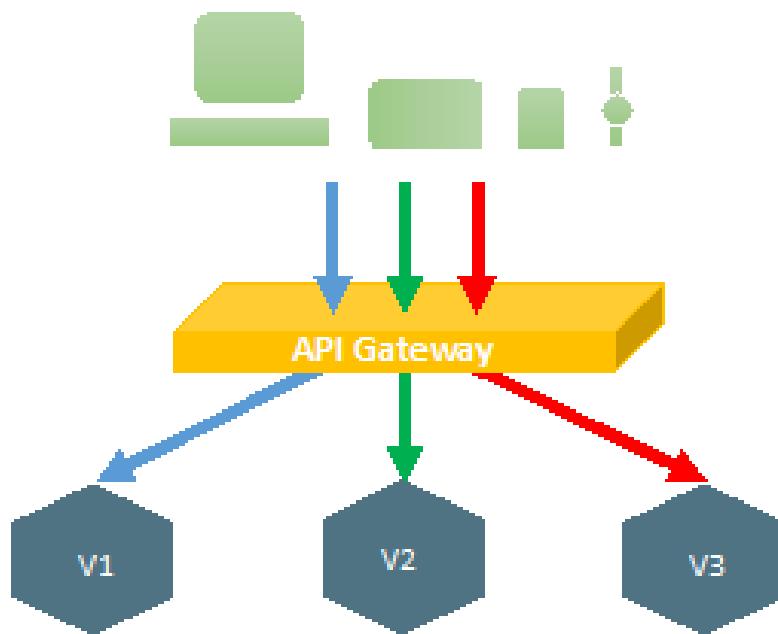


Figure 6.23: Versioning

Being able to handle different versions of an API while maintaining backward and forward compatibility can be challenging for both frontend and backend developers. This is important as an API, just like any other software application, can (and should) evolve either in support of new features or simply just in response to bug fixes.

However, some versioning strategies can make this already-complicated task even more complex. So, this criterion looks at how the different options provide out-of-the-box support for versioning. Options that are ambiguous about how versions should be handled, thus leaving it for the developer to decide what version strategy to adopt (meaning lots of inconsistencies and debate industry-wide on the best way to do this), get the lowest score.

Options that deliver clear guidelines and tooling support to handle versions and also forward/backward compatibility will get the highest score.

REST	GraphQL	gRPC
-	++	~
There is a lot of industry-wide debate about how	Best practices are clear and publicly available in the	Protocol buffers by design support

<p>versions should be handled in REST. Some favor URI-based versioning, others header-based versioning, and others no versions at all. So, ultimately, it is down to the developer what strategy to adopt.</p>	<p>official GraphQL site. Versioning should be avoided and tooling is available to support forward and backward compatibility without having to run in parallel multiple versions of the service.</p>	<p>backward compatibility. Furthermore, as servers and clients are generated from the <code>.proto</code> file, different versions of the file could be maintained, though this could lead to confusion if many versions exist.</p>
--	---	---

Asynchronous communication

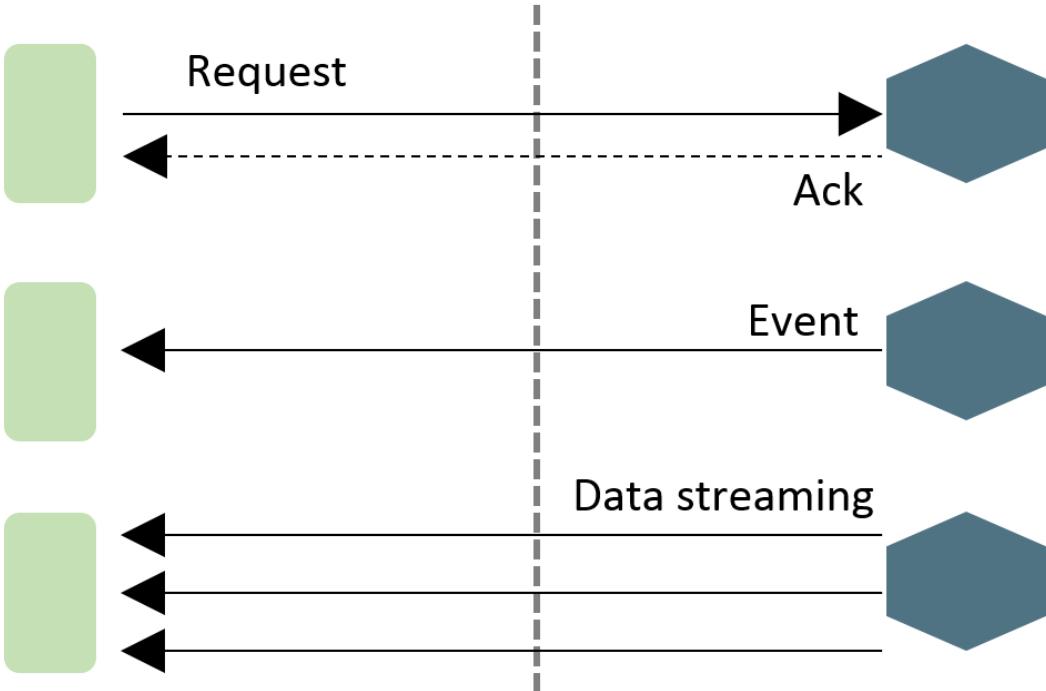


Figure 6.24: Asynchronous communication

Support for asynchronous communication in APIs, beyond the traditional request/response interaction style, already is a hot topic. This is because forcing consumers to constantly call an API to obtain changes and/or notifications of any sort is not just impractical, but forces unnecessary uses of application and network resources. Therefore, this criterion evaluates the feasibility of the different options for implementing asynchronous communication in addition to common request/response methods.

REST	GraphQL	gRPC
- REST does not explicitly support asynchronous communication; however, a popular approach to support it is by implementing Webhooks, which allows clients to subscribe to a given event. The client is then notified whenever the event takes place.	+\$ GraphQL supports asynchronous communication natively via subscriptions, which is another operation just like queries and mutations.	++ By design, gRPC is fully bidirectional, not just supporting events, but even supporting message streaming generated by the server to the client or the client to the sever completely asynchronously.

Comparison totals

The following diagram illustrates the totals derived from the comparison.



Figure 6.25: Comparison totals

As you can see, there are still several advantages in adopting REST. From the availability of tooling for API design and implementation, and out-of-the-box support in the majority (if not all) of API products, to the amount of information available online to learn and implement it, REST is a safe option when it comes to deciding what architectural style to adopt for a given API. However, there are also downsides to using REST, the main one being the fact that by their own nature, REST APIs

can be very chatty, especially if HATEOAS is overused (or wrongly used). When this is the case, additional work and complexity for frontend developers can be experienced.

On the flip side, GraphQL is rapidly catching up with REST in terms of the tooling available and support in API products, but it is still not there yet. Therefore, adopting GraphQL at this point in time (or at the time of writing) may result in having to compromise on capabilities, customize existing tools, or even introduce new tools that do support this technology. The fact remains, however, that when it comes to usability experience, GraphQL truly is the gem. This is not just because of the fact that tools such as GraphiQL make it a lot easier to consume APIs, but also because of the architecture, which by design was meant to give back control to the consumers of the API.

gRPC, on the other hand, is not great when it comes to usability in the context of browser-based applications, which is the reason why the scoring was the lowest. This does not mean that gRPC is bad; it just means that for the use cases detailed in the criteria, it lags behind the other two. For example, gRPC is extremely popular and widely adopted as the means to deliver service-to-service communication, as well as asynchronous communication. Because of this, gRPC could well be used in conjunction with REST and/or GraphQL, so from this point of view, it would be a great complement to an overall solution architecture.

Summary

This chapter delivered a comprehensive overview of the main API architectural styles dominating the industry at the moment. The chapter started by providing context on the evolution of APIs and then deep-dived into the three most popular options at present.

The chapter then provided an opinionative comparison based on usability, tool ecosystems, and key features expected of APIs. Based on this, a summary was provided explaining why and when each of the options evaluated might be a good fit or not.

The next chapter will focus on the API development life cycle and organizations. It will talk about what a good process for designing and implementing APIs looks like, as well as the roles and responsibilities required in order to create an organization suitable for delivering APIs as business products.

API Life Cycle

This chapter moves away from the technical and architectural aspects of APIs to cover essential processes and methods required throughout the entire **API life cycle**. The chapter starts by describing the overall development life cycle, not just for APIs but for related activities and assets, which includes the API design-first, service, and consuming application life cycles. It then continues by describing each of the cycles in detail.

This chapter will aid organizations and/or individuals looking to implement the full API life cycle from scratch, or just looking for sources of inspiration to optimize and/or refine existing ones.

The full API development life cycle

In [Chapter 4](#), *API-Led Architectures*, the topic of the API life cycle was briefly described; however, this was more from a capability standpoint rather than as a process. The full API life cycle consists of a series of recurring steps (iterations) that, when executed properly and cohesively, should result in APIs that are fit for purpose, well documented, and implemented according to their specification. Most importantly, the life cycle should result in assets that deliver customer value, continue to do so over time, and are, therefore, being continuously improved.

The full API life cycle, however, is not just one cycle but a chain of related cycles, as will be subsequently described.

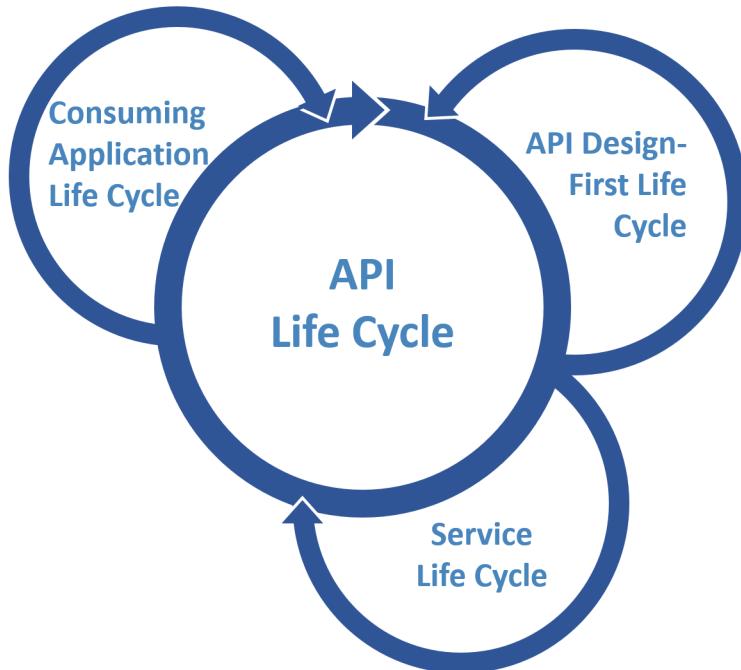


Figure 7.1: The overall API life cycle

At the very least, the life cycle consists of four interconnected cycles:

- **API life cycle:** This is the main flow from which all activities are derived. It is the one that initiates the chain of events that ultimately results in the API being designed and delivered, but it also triggers related iterations around the API design cycle, service implementation, and even consuming applications.
- **API design-first life cycle:** This related cycle focuses exclusively on the design of APIs and establishes a process by which API consumers get exposed early in the process to the API design artifacts, such as an API specification or API mocks and stubs, with the idea of

collecting feedback as soon as possible in order to prevent reworking later on.

- **Service life cycle:** Once an API design is completed, the related business capabilities must be implemented in the form of **business services**, as described in [Chapter 4, API-Led Architectures](#). This cycle, therefore, describes the different steps typically carried out when implementing services derived from an API design-first approach.
- **Consuming application life cycle:** This cycle defines the steps that can be carried out to implement consuming application code, while ideally also leveraging capabilities available in the platform.

The following is a detailed description of each cycle.

API life cycle

This cycle is the main or core cycle that triggers the chain of activities that ideally should never end (as long as the product is successful).

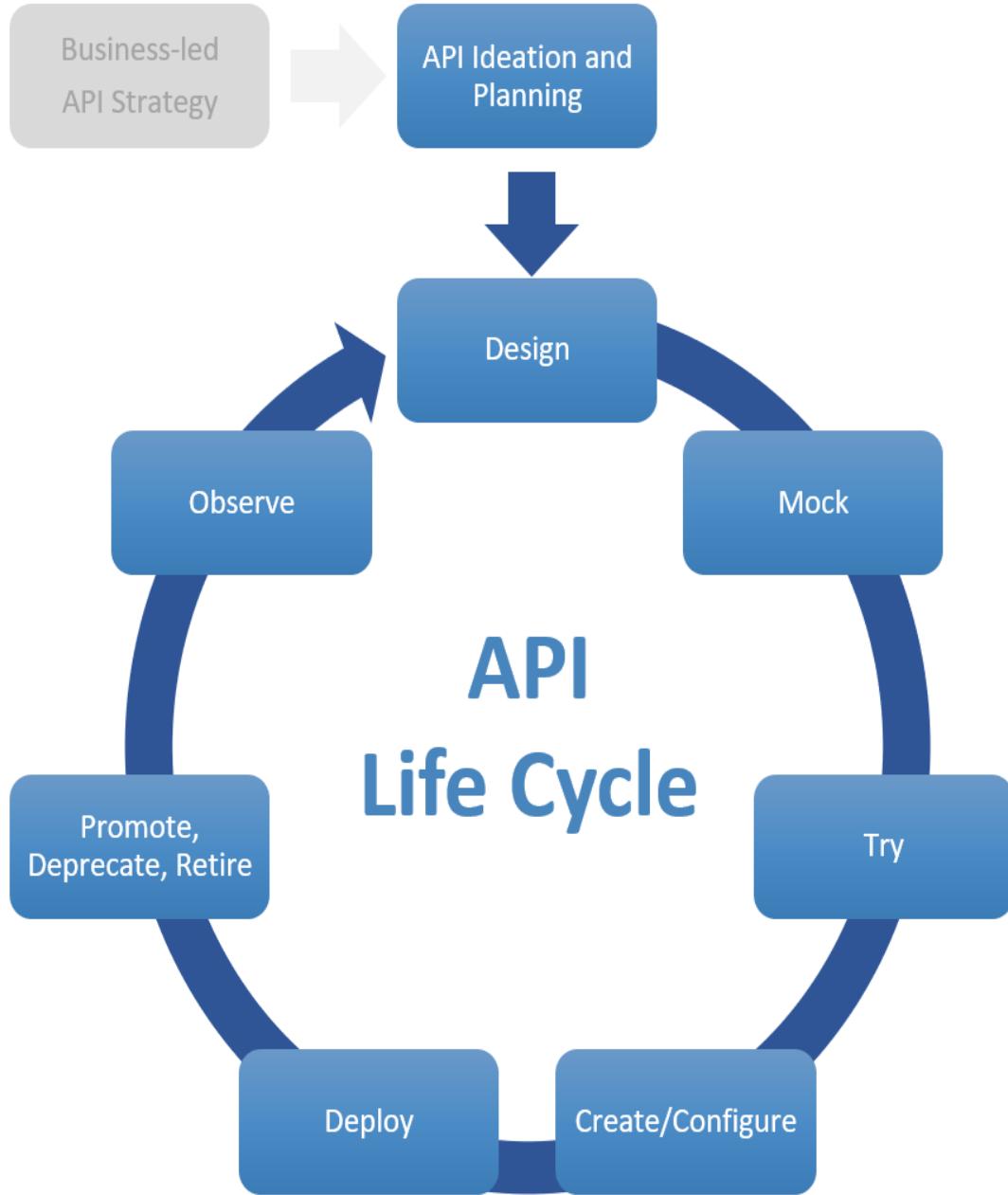


Figure 7.2: The core API life cycle

Note that the implementation of a full API life cycle should always be preceded by the creation of an API strategy that doesn't just bring business context, but also defines clear goals and objectives about why APIs are being delivered in the first place. This is important as it brings relevance to the process and intrinsic justification to each of the steps.

The creation of an API strategy is covered in detail in Chapter 3, Business-Led API Strategy.

The steps of the API life cycle are as follows:

API ideation and planning

Great ideas are not conceived by magic; they are typically the result of a process aimed exclusively at identifying, qualifying, and selecting the best ones for creating the **product backlog** and, subsequently, development.

Ideation is a creative process whereby new and innovative ideas are generated and captured. It typically involves sessions where brainstorming, sketching, and even quick prototyping (as is the case with hackathons) takes place. During the creative process, good ideas are shortlisted and should, in principle, become candidates for implementation, at which point planning takes place.

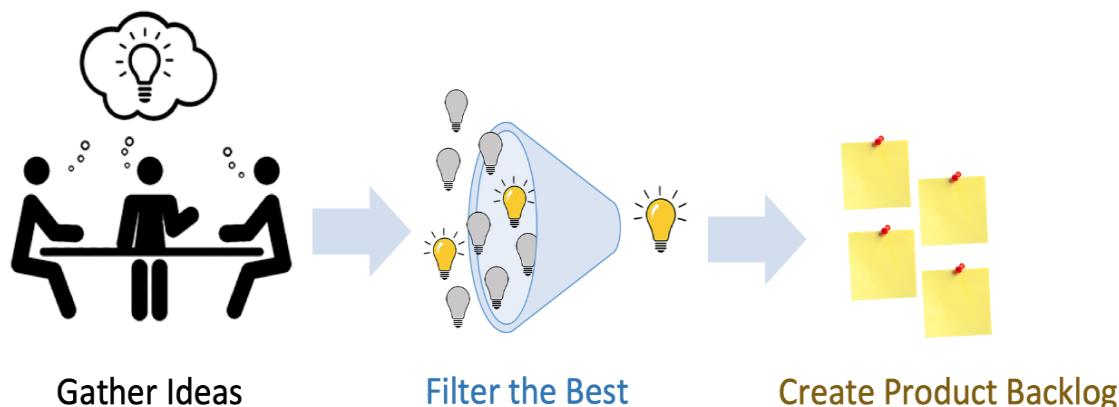


Figure 7.3: The ideation process in action

*Ideation is a fundamental phase of **human-centered design** and **design thinking**. Refer to <http://www.designkit.org> and <https://www.interaction-design.org/literature/topics/design-thinking> for more information.*

In the context of APIs, a series of ideation workshops that bring together business and IT stakeholders (and also, when applicable, end users) can be planned and executed with the objective to collectively identify new APIs that have good potential to deliver customer and business value, and thus can be packaged, marketized, and sold as products.

Such workshops require careful preparation, especially when it comes to identifying the right stakeholders and preparing adequate content. It might be desirable to organize separate workshops for each functional area, so participants of each session all talk a similar language and share a common understanding of the business domain being discussed.

Part of the planning process should also involve creating introductory content that participants can easily understand and relate to; for example, describing APIs that may already exist in the functional domain and, in simple terms, explaining how they help the business and add value to its consumers.

The actual sponsor for the API ideation workshops could be any executive within the organization with a desire (or an interest) to generate value for the business and its customers. However, it should be someone with a sufficient mandate to get the right people together and the budget to cover the costs of the workshop (which, at this stage, typically just means covering people's time and, on occasion, bringing in third-party organizations to assist in facilitating the workshop).

To this end, the API value chain and use cases described in [Chapter 1](#), The Business Value of APIs, and [Chapter 3](#), Business-Led API Strategy, respectively, may serve as inspiration for creating such sample content.

Note that because the term "API" may be unfamiliar to many people from a non-technical background, it is imperative to explain in a common language (preferably at the start of the session) what APIs are and how they can bring value to the business. Failing to do so may result in counterproductive outcomes or lead to only part of the audience contributing to the workshop.



Figure 7.4: API ideation workshop outcomes

As illustrated, a successful workshop should result in a series of ideas being captured, whereby each idea answers at least one of the following questions:

- **What exactly is the API for and what would it do?** For example, a telecoms company with the ability

to obtain satellite imagery may want to create an API that allows access to satellite images on demand.

- **How would the business benefit from it?** Answers could be a new revenue stream whereby access to images is charged for, the enablement of new channels (for example, a mobile app to sell existing products), and access to new markets, to name just a few.
- **Who would be the potential customers?** This could be any organization that could make better decisions based on a better understanding of its land. An obvious example is agriculture organizations that could use such imagery to gain better insights into their crops.
- **Why would they buy and/or use the product?** Because the satellite images would always be up to date (for example, updated every day), as opposed to content that is free online but is unreliable and, in the majority of cases, dated. Furthermore, the images could increase sales and the **return on investment (ROI)**.
- **What could be the business model? How would this be monetized?** Images could simply be charged for based on kilobytes downloaded (for example, the larger the area covered and the better the quality of the image, the larger the image size would be). It could also be charged on demand (per call) or based on a monthly fee (with a predefined limit of kilobytes that could be downloaded).

- **Who is the competition?** A product that is not found on Google these days is probably not a good product. Thus, an initial analysis could simply be conducted by performing a series of Google searches based on different keywords. Searches could also be conducted in public API marketplaces and directories, such as the ones described in [Chapter 4, API-Led Architectures](#).
- **What business capabilities does it require?** In the example, the telecoms organization should already have the required business capabilities to take the images; however, it may lack the capabilities to render such a number of images on demand. Thus, it may require some sort of digital asset management and image streaming technical capability.
- **What are the constraints and regulations?** For example, certain countries may not allow satellite images to be taken of their territory.
- **Any obvious showstopper(s)?** For example, the organization may not be in a position to make any sort of investment, even though there could be a return on it.

Note that for internal APIs, similar factors equally apply, with the difference that buyers would be internal customers (for example, different business units) and the competition would be different teams/departments working on similar capabilities. This could be a game-changing approach toward how the central IT team offers services and capabilities to the rest of the organization.

Lastly, and as illustrated in the preceding diagram, the best API ideas could be determined based on their implementation feasibility (availability of business and technical capabilities to deliver the API), their uniqueness (no similar API can be easily found based on an initial search), and the potential business and customer benefits.

API ideas deemed to be the best should subsequently be broken down into smaller requirements and added to a product backlog for their delivery.

A product backlog is a prioritized list of all items known to be required to deliver a product. In agile methodologies, this backlog is the single source of requirements for a product.

The backlog should ideally prioritize the items so initial iterations focus on speed to market by delivering a **minimum viable product (MVP)** as opposed to a final product with all the bells and whistles. Once the MVP is delivered, subsequent iterations can focus on other differentiating features. The reason for this approach is the fact that if an idea is deemed to be good, this doesn't necessarily mean it will be a success. For this reason, limiting the scope of a first delivery will not just lower the implementation costs, but it will also help in delivering a working product faster.

For more information on MVPs, refer to https://en.wikipedia.org/wiki/Minimum_viable_product.

Note that the party responsible for defining and a product backlog should be a designated **API product owner**, who

would also be responsible for the product's subsequent delivery and the continuous improvement of it post go-live. Refer to [Chapter 8, API Products' Target Operating Model](#), for more information on this role.

Design

This is the stage of the life cycle where requirements are translated into something tangible that can be built and delivered. It requires an understanding of all functional and non-functional requirements. A domain model and a conceptual design are produced based on a series of well-thought-out design decisions.

The concept design should, among other things, answer questions: what is the business domain of an API and its bounded context? What business capability does the API offer? What API architectural style is to be adopted (for example, GraphQL for public interface, or gRPC for inter-service communication)? What does the end-to-end solution look like, including the patterns (for example, API aggregator and CQRS) and technical capabilities required, naming conventions, and documentation?

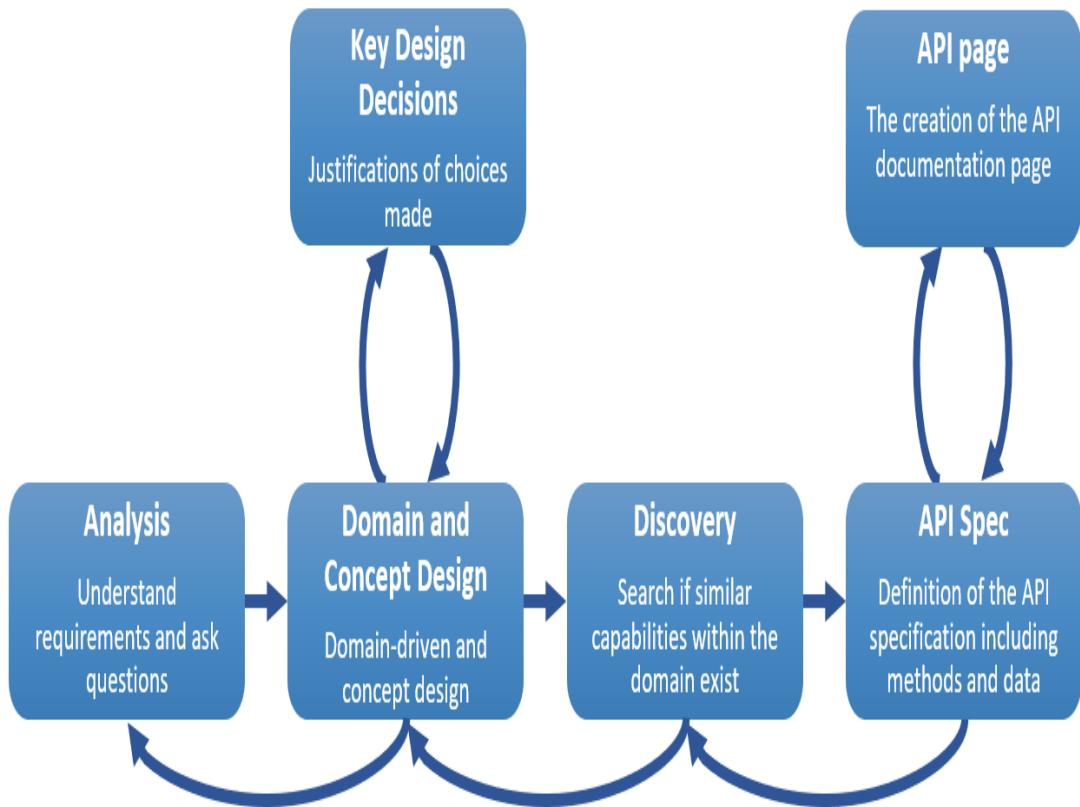


Figure 7.5: The API design process

The design phase should ideally consist of the following activities:

Analysis

This is the process of understanding all the business requirements in the backlog and, if necessary, organizing **question and answer (Q&A)** sessions to clarify doubts and ensure that all needs are well understood and that nothing is left out. If needed, the backlog items can be refined further.

During this activity, it is important to properly understand functional as well as non-functional requirements. Key for the

latter are things such as expected throughput, geographical regions for users and source data, and security requirements. These may be key factors influencing what design decisions to make.

The analysis activity is typically carried out by **API architects**, but also typically involves input from the product owner and business analysts who contributed to the creation of the backlog. Refer to [Chapter 8, API Products' Target Operating Model](#), for more information on these roles.

Domain and concept design

A domain design is a **domain model** (for example, an entity relationship diagram in **Unified Modeling Language (UML)**) describing the business domain and problem being solved in notation that both business and IT teams can relate to. The model should therefore act as a **ubiquitous language**, as it reflects a shared understanding of the domain.

Domain-driven design (DDD) is a well-known and, in fact, recommended approach to such domain models.

The book, Domain-Driven Design, by Eric Evans describes DDD in full:

<https://www.youtube.com/watch?v=7MaYeudL9yo>

The model should, at a minimum, describe three things:

- The business domain (for example, sales, finance, or human resources) and actors.

- The different **bounded contexts**.
- The **business entities** within each bounded context and their relationships (both within and outside their context).

The following article by Martin Fowler provides a great explanation of the bounded context pattern:

<https://martinfowler.com/bliki/BoundedContext.html>

On many occasions, a domain model may already be available, in which case this step should be about identifying the domain, the bounded contexts, and the entities that are relevant for the design.

Once a domain design is available, a concept or conceptual design can be produced that describes the following:

- The different technical components that build up the solution and how they relate to each other.
- What technical capabilities (from the reference architecture) are required on each component.
- Which capabilities are to be introduced (as they don't exist) and which are to be reused.
- Key design decisions made.
- What API policies are applicable to secure the service endpoints.
- Assumptions made.

- Known gaps in the solution.
- Information views in the form of entity relationship diagrams.
- If required, sequence/interactive views describing how the different components interact.

*A conceptual design should not be a long and extensive document, as few will, in practice, read it. Instead, it should be a living document (for example, a markdown page in a **Git** repository or a dynamic document produced in wikis and/or content management systems) that only includes the information relevant to the design; nothing more and nothing less.*

The conceptual design activity is typically also carried out by API architects.

Key design decisions (KDDs)

KDDs are design choices that have a notable impact on the way a product is realized or on the final product itself. They are decisions because multiple viable options exist and, therefore, a choice has to be made (if only one option exists, then it will not be a design decision).

Although this is not a mandatory step in the design process, it is a highly recommended one as it ensures not only that all design choices are properly justified, but also that decisions are recorded on a log for anyone in the future to understand why certain design choices were made. The log, just like the conceptual design, could simply be a markdown page in a **Git** repository or a dynamic page in a wiki or equivalent content

system. It should ideally consist of a high-level summary of all decisions (each with a unique ID) and a more detailed page for each decision including:

- The creator of the decision and the date.
- The person who approves or rejects the decision. This can vary from organization to organization. In some cases, decisions are approved or rejected in a consensus-based approach in some form of design authority or design decision forum whereby relevant stakeholders (typically those who, in some shape or form, have an interest in the outcome of the decision) take part to review and vote on the final recommendation, or it could just be the senior-most architect within the team that has the authority to decide.
- By when a decision is expected and the impact if a decision is not made in time.
- A description of the problem statement and its context; for example, what API architectural style to adopt.
- The main options (ideally no more than three) to solve the problem and why (including any reason why some options may have been left out). For example, options could be GraphQL, REST, and gRPC, leaving out options such as SOAP/WSDL given their lack of industry support in terms of technology and skill levels. A good description of each option should be included.

- The pros and cons for each option, as a minimum from an architectural, delivery, operations, and **total cost of ownership (TCO)** point of view.
- A final recommendation with a proper justification.
- The status of the decision (for example, pending, approved, or rejected) and the date.

Discovery

During this activity, existing business capabilities that offer similar functionality to the items in the backlog are searched for. Ultimately, the objective is to avoid reinventing wheels. If an API(s) already exists that offers functionality that addresses the requirements of the backlog, instead of duplicating functionality, reuse should be considered.

In an ideal world, this step should be as simple as just conducting a search in a common enterprise API catalogue and seeing whether any APIs match the search criteria. However, in practice, few organizations have a single catalogue as a source of truth. Therefore, this activity may also require having to search multiple catalogues or having to contact different stakeholders who may be able to offer additional information.

This task should also include searching public API catalogues such as programmableweb.com and rapidAPI.com, as, on many occasions, it might be more cost-effective and quicker to just make use of

a public API as opposed to building one. The discovery activity is also typically carried out by API architects.

Versioning approach

As discussed in [Chapter 6](#), *Modern API Architectural Styles*, depending on the API architectural style adopted, there are different opinions on how API versioning can be handled. Therefore, the versioning strategy to be adopted is also an important design consideration that requires careful consideration. For example, in the case of REST, versions are commonly defined within the **Uniform Resource Identifier (URI)** (for example, `/v1/orders`) or the HTTP header (for example, `version: 1`). In this approach, new versions (for example, `/v2/orders` or `version: 2`) are created when there isn't **backward compatibility**.

Backward compatibility refers to the ability of an iterated version of an interface to be compatible with its older versions, meaning that for consumers binding to the older version of the interface, it should continue to work once the new version is released.

When a deployment artifact is backward compatible, it can be deployed to a running environment superseding its older version. API consumers should not be impacted from it and the release should be smooth and straightforward. This type of change is typically referred to as minor versions.

However, if the release isn't backward compatible for any reason, then superseding an existing API and/or service with it may result in breaking the API consumer's code, which is far from ideal. At this point, being able to run parallel versions of the same API/service may be unavoidable.

In the case of GraphQL, however, multiple versions of a service are discouraged and, by definition, all changes should be backward compatible. To this end, tools and approaches are available to deprecate types such as objects and operations. In gRPC, backward compatibility could just be based on versioning the proto file and respective servers/clients derived from it.

Regardless of what architectural style is chosen, the fact remains that a versioning strategy of some sort will be needed, especially to handle scenarios in which, for whatever reason, an API and/or service is not backward compatible. Furthermore, this should be documented as well, since consumers of the API will also be interested in the approach taken.

API specification

This is a document that describes the **technical contract** of an interface, such as:

- All methods/operations supported by the interface
- Data definitions for all inputs and outputs on each method/operation

- Any HTTP headers and metadata used
- Technical constraints
- A technical description of each method/operation, ideally with examples
- A technical description of all data entities and what each field means
- Sample requests and responses (bearing in mind that these can be used later on to test the interface)

An API specification is defined in accordance with the **interface description language (IDL)** for the API architectural style chosen and, depending on this, different tools and techniques can be adopted, some more dynamic than others.

Refer to [Chapter 6, Modern API Architectural Styles](#), for more details on IDLs for REST, GraphQL, and gRPC.

An API specification can be created by API architects and/or **senior developers**. Regardless of the role, however, the person that works on the creation of API specifications is typically also referred to as an **API designer**. Refer to [Chapter 8, API Products' Target Operating Model](#), for more information on this.

Note that modern API management tools typically come with rich capabilities to create and edit API specifications as part of the API life cycle. Refer to [Chapter 4, API-Led Architectures](#), for more details on capabilities.

API page

This is a web page that, in addition to incorporating the specification of an API, also includes a human-readable (meaning less technical) description of the API, including what it offers (in terms of functionality) and plenty of examples on how to use it. This document should be customer-oriented, meaning the objective is to make it as simple and quick as possible for potential API consumers to understand and start using the API.

Note that modern API management tools typically come with rich capabilities to create and edit the API page as part of the API life cycle. Refer to [Chapter 4, API-Led Architectures](#), for more details on capabilities.

Because of the target audience and nature of content required for the API page, it is recommended that a **techno-functional writer** (or someone with previous experience in creating customer-facing technical content) is responsible for producing it. Refer to [Chapter 8, API Products' Target Operating Model](#), for more information on this role.

Mock and try

API mocking is a technique by which the methods/operations specified within an IDL are simulated by means of a mocking server. The idea behind this approach is to enable API consumers and developers to try the API through its mock before the actual implementation takes place.

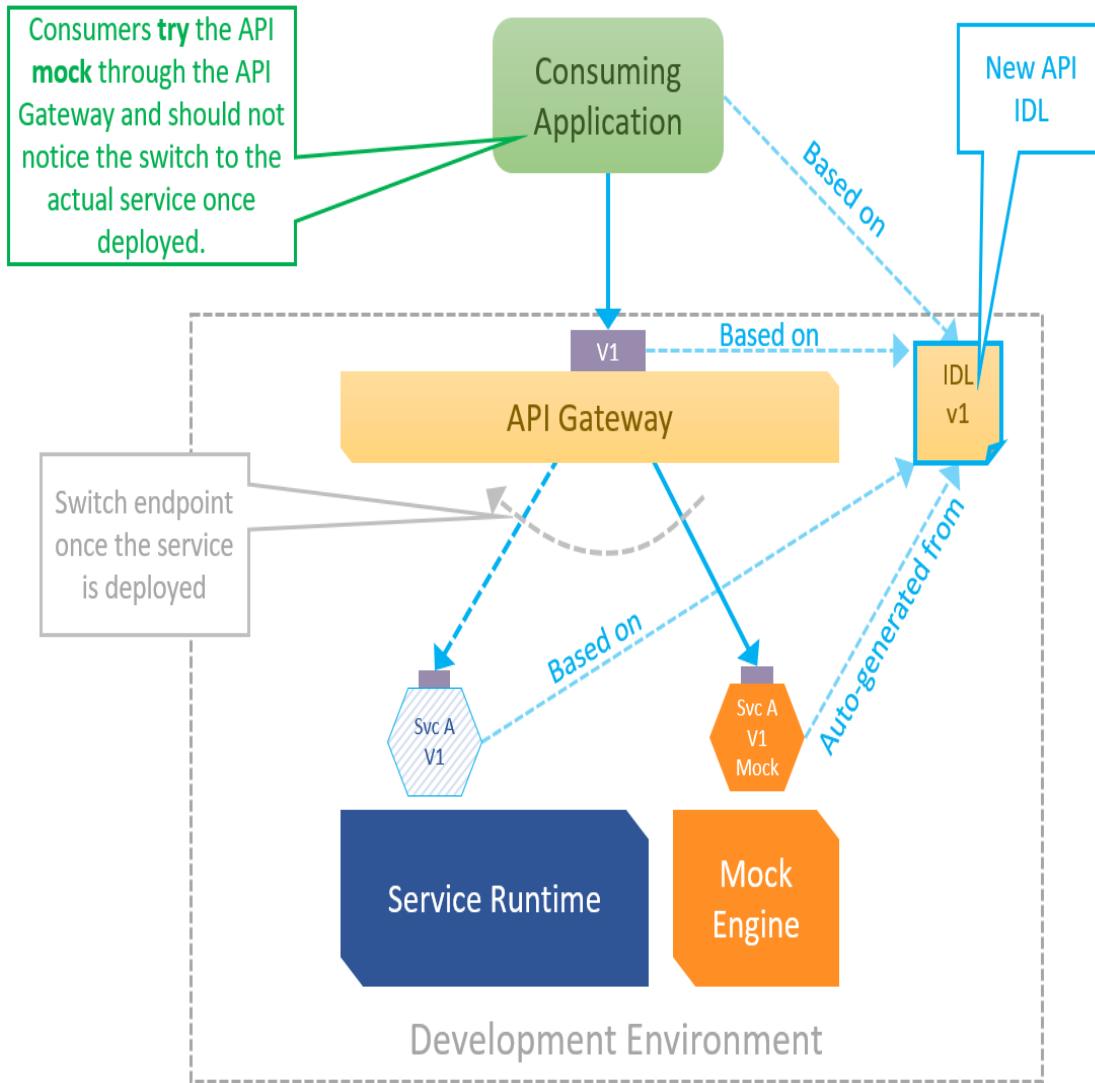


Figure 7.6: API mocking

As the preceding diagram illustrates, an API mock should behave in accordance with the API specification itself (the IDL). As long as the mock makes use of representative (sample) data, API consumers can use it to try out the API early in the life cycle.

This is useful for many reasons, but especially because it can speed up the development process, and also the quality of the API, by collecting feedback early in the life cycle. Doing so

means that in theory, the API should not undergo many changes later in the life cycle as a consequence of mismatched requirements.

Furthermore, assuming that the mock is exposed through an API gateway (as illustrated), once the actual API is built, then the gateway can switch to it without impacting consumers of the mock. So, from an API consumer standpoint, this is seamless.

Note that modern API management tools typically come with rich capabilities to create and expose API mocks interactively during the API design process. Refer to [Chapter 4](#), API-Led Architectures, for more details on capabilities.

When mocking APIs, the following activities typically take place:

- The mock server is generated based on the API specification. In the case of REST APIs, there are several tools that can do this automatically and on the fly as the API specification is created. For REST, plentiful tools are available on the market for API mocking. Good examples are **Apiary** (apiary.io), **SwaggerHub** (swagger.io), **Mocky**, (mocky.io), and **WireMock** (wiremock.org).

*In the case of GraphQL and gRPC, not so many commercial tools are available (at the time of writing); however, open libraries such as **Graphql-Faker** (github.com/APIs-guru/graphql-faker) for the former, and **GripMock** (github.com/jekiaapp/gripmock) for the latter, are rapidly emerging.*

- Sample request and response data is created for all methods described in the API specification.

Note that more advanced mocking engines also allow for the API to behave differently based on different request payloads. This is important as it enables consumers to also try different request scenarios, such as failures, different filters/parameters, and payloads.

- Calls made to the mock server are monitored, as this too can help with understanding how the API mock is being used.
- Test clients are created that can be used later on to test the actual implementation of the API. Likewise, for test clients, there are plenty of examples for REST APIs, such as **API Fortress** (apifortress.com), **Dredd** (dredd.org), and **PostMan** (getpostman.com).

*In the case of GraphQL and gRPC, not so many commercial tools are available (at the time of writing); however, open libraries and tools, such as **EasyGraphQL** (github.com/EasyGraphQL/easygraphql-tester) and **Altair** (altair.sirmuel.design) for GraphQL, are rapidly emerging. For gRPC, several open libraries for testing and other purposes can be found at github.com/grpc-ecosystem/awesome-grpc.*

Create/configure

Once an API mock is available or a service has been implemented and/or iterated (for example, a new version or enhancement), an API can be created using API management capabilities (as described in [Chapter 4, API-Led Architectures](#)), so different policies, such as authorization, rate limiting, monetization plans, and mediation, can be applied to it.

This step is typically carried out by developers and/or hands-on API architects. It involves using a policy editor (typically, a centralized web application in newer tools) in order to:

- Create and edit an API and its metadata (for example, its description, and stage in the life cycle).
- Define the version of the API.
- Attach any related API specification and/or documentation.
- Define the service endpoints. In some cases, this could be just the API mock if the service is under development.
- Apply, edit, or remove API policies, such as OAuth 2.0 authorization, API-key validation, throttling, and rate limiting.

- Configure environment-specific properties as required.

Note that modern API management tools typically come with rich API policy editors. Refer to [Chapter 4](#), API-Led Architectures, for more details on capabilities.

Depending on the number of API policies to be applied, this process can be very quick or time-consuming (especially if custom policies are deemed to be required).

Deploy

In simple terms, deployment is a process by which a code artifact (typically referred to as a deployment unit), such as an API and/or service, is prepared and moved into a runtime environment for its execution and use. However, in practice, deployment is carried out within a **continuous integration and continuous deployment (CICD)** pipeline that takes care of the packaging, verification, regression testing, and deployment of the code.

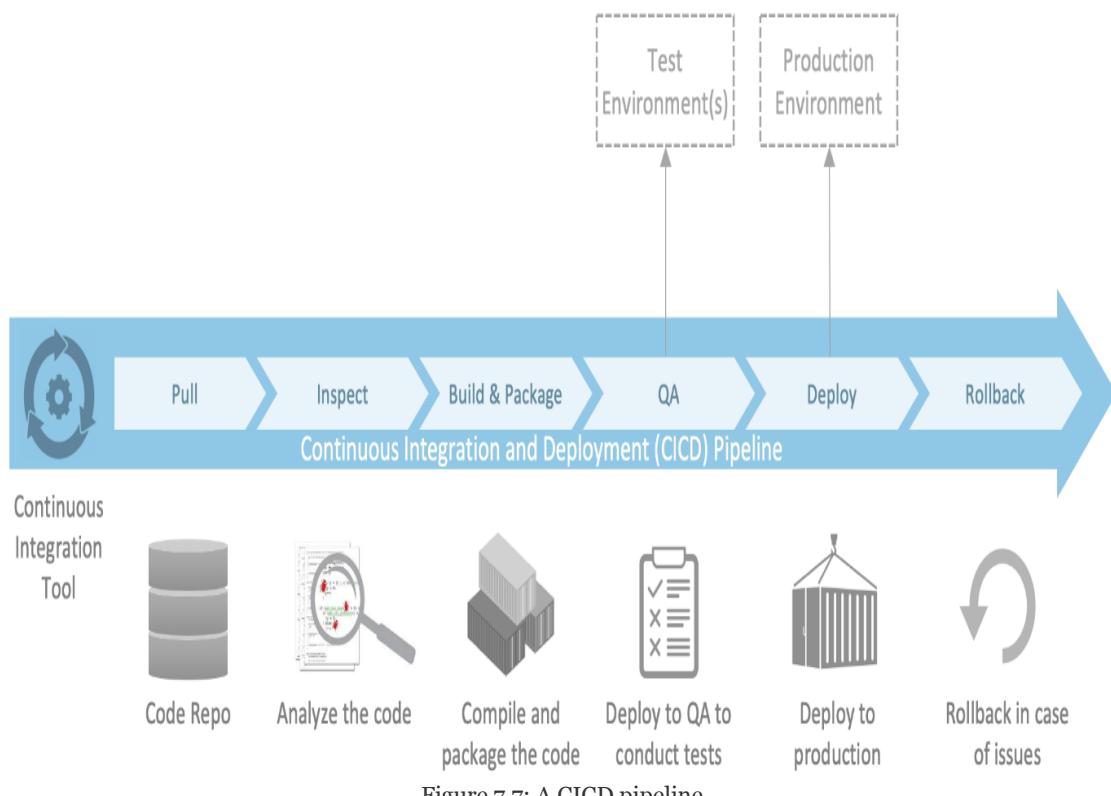


Figure 7.7: A CICD pipeline

Although the topic of defining and creating CICD pipelines won't be covered in detail in this book, as illustrated in the preceding diagram, the process typically involves adopting a continuous integration tool (for example, Jenkins, TeamCity, or CircleCI), which, based on preconfigured conditions, such as a merge into a main development branch, a scheduled task, or even a manual action, executes a series of tasks. The tasks are as follows:

Pull

This means pulling the code from a specific branch in a code repository, which could be a source control system, such as **Git** (for example, GitHub, Gitlab, or Bitbucket), or even older repositories such as **Subversion (SVN)** and **Concurrent Versions System (CVS)**.

Inspect

This means inspecting the code and its dependencies in search of potential errors and/or vulnerabilities. This can be done with tools such as Sonarqube (sonarqube.org), Coverity (scan.coverity.com), or Fortify (microfocus.com).

Build and package

This involves compiling and packaging the code, along with its dependencies, into a release package, tagged with its respective version. For example, in the case of Java, this typically involves

using tools such as Maven (maven.apache.org) or Gradle (gradle.org) to generate .jar files.

This task also involves the publishing of a generated released artifact into a repository manager such as Nexus (sonatype.com/nexus-repository-oss) and/or Artifactory (jfrog.com/artifactory), along with generating and tagging Docker containers and pushing them into a container registry (for example, hub.docker.com or internal ones).

Quality assurance (QA)

This consists of deploying the released artifacts into a QA environment with the objective of conducting a series of tests, such as:

- **Interface testing:** This is verifying that an interface matches its interface definition. For example, in the case of REST APIs, this can be done with tools such as Dredd (dredd.org) and Swagger Inspector (inspector.swagger.io).

Note that in the case of GraphQL and gRPC, this step is not typically required as their IDL is closely coupled with the service implementation. However, in the future, as API-design tools emerge for these architectural styles, new tools may arise that conduct such tasks.

- **Functional testing:** This is completed by making a series of pre-defined API calls to verify that the API behaves as expected. This can be done using tools such

as Postman (getpostman.com), API Fortress (apifortress.com), or ReadyAPI (smartbear.com/product/ready-api/overview).

Note that although the above tools are better suited for REST, they could potentially also be used for GraphQL by testing different HTTP POST calls with the previously generated payloads.

Alternatively, tools such as the `apollo-server-testing` package could be used (apollographql.com/docs/apollo-server/features/testing). For gRPC, refer to the following URL for tools: <https://github.com/grpc-ecosystem/awesome-grpc>.

- **Performance testing:** This is carried out by simulating a large number of concurrent calls to ensure that the API can handle the expected throughput. In the case of REST and GraphQL, tools such as Apache benchmark (httpd.apache.org/docs/2.4/programs/ab.html), Fortio (github.com/fortio/fortio), and JMeter (jmeter.apache.org) can be used to simulate a large number of HTTP 1.1 calls. For gRPC, as long as the tools support HTTP/2 then, in theory, they too could be used (for example, Fortio supports HTTP/2 and gRPC). However, for more tools, refer to <https://github.com/grpc-ecosystem/awesome-grpc>.
- **Security testing:** This is typically a complex task as it involves testing an interface against common threads, such as the ones defined in the OWASP Top 10 project (www.owasp.org/index.php/Category:OWASP_Top_Ten_Project). The tools to be used for this type of testing heavily depend on the API architectural style and what type of vulnerability tests are executed. In many cases, it requires custom scripts and the use of frameworks such as Metasploit (metasploit.com).

Deploy

Should no issues be encountered during the tests, the solution is deployed into the production environment.

Rollback

Should any issues be encountered post-deployment, then there should be the ability to roll back to the previous working version. The creation of the deployment pipeline is typically carried out by **platform engineers** with support from developers as required. Note that depending on the API management and service life cycle capabilities available, this process can be fully automated, semi-automated (requiring human intervention for certain steps), or even fully manual, although the latter should be avoided.

Promote, deprecate, and retire

As APIs and services evolve through multiple design and development iterations, naturally, new versions are created to reflect the fact that changes have taken place, such as new features being added, improvements to existing features, or even just bug fixes. Regardless of the type of change, the fact remains that handling versions is a critical aspect of any software development life cycle.

However, having a strategy in place to handle API versioning isn't enough. If the process of rolling out changes isn't carefully thought through, such as how to deal with non-backward-compatible versions, there can be negative repercussions, such as breaking the API consumer's code or even ending up with too many versions of the same API in production, thereby adding additional complexity and costs.

Such a fate can be avoided by having the ability to promote, deprecate, and retire different versions of the same API and its corresponding service:

- **Promotion:** This refers to the ability to make a new version of an API (and related documentation) available

for general use. When an API is promoted, it becomes the default version for use, meaning that all API catalogues and developer portals should display information related to the promoted API and not its predecessor.

- **Deprecation:** An API becomes deprecated when it is superseded by a new version. When an API is deprecated, its endpoint is still fully functional and its documentation is available, but only for a period of time. Because of this, consumers of the deprecated API should be notified of the fact that the API will only be available for a given timeframe.

How long exactly depends on factors such as how many API consumers there are, and how critical the features added to the new version are.

- **Retirement:** An API is retired when its endpoint is no longer functional and calling it will result in an error.

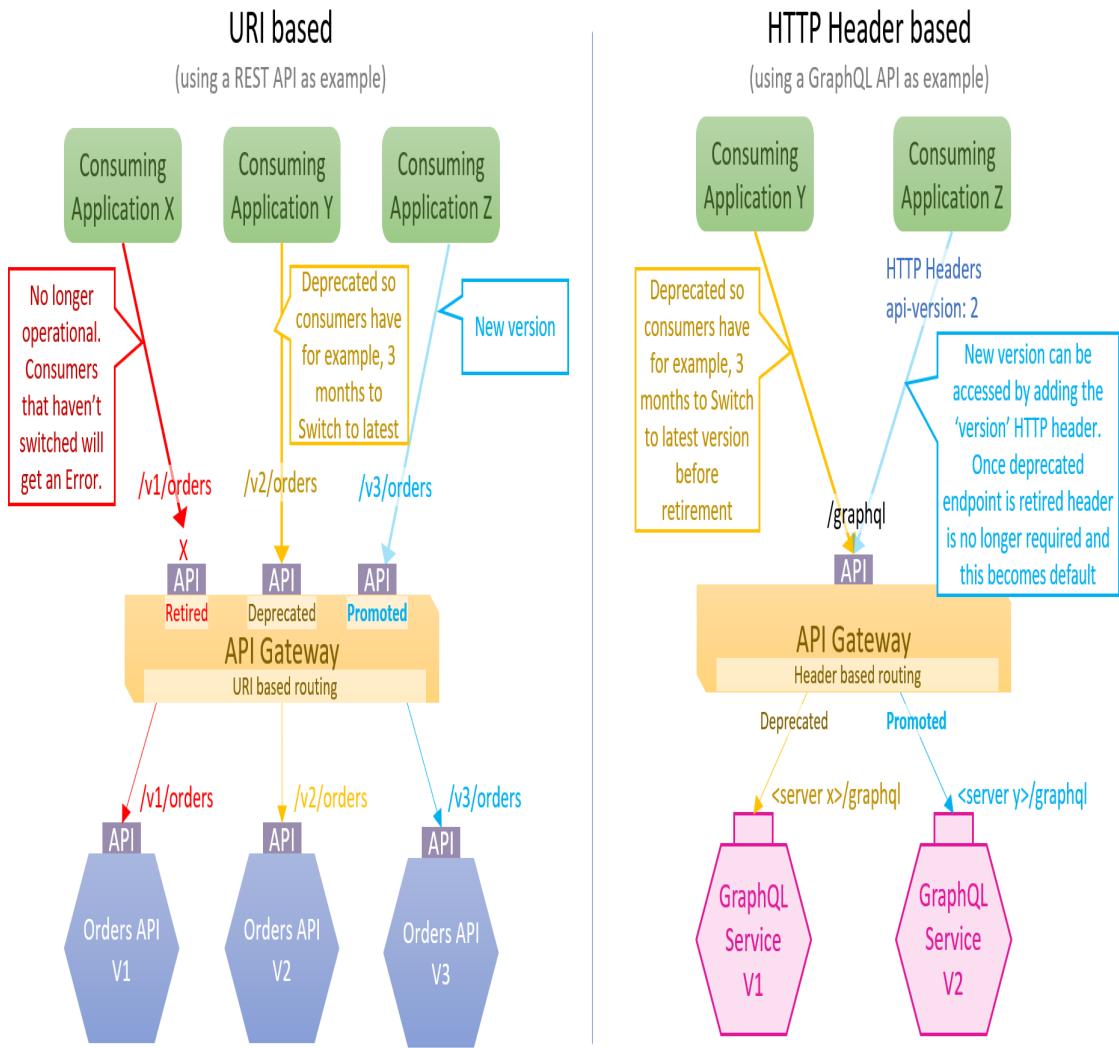


Figure 7.8: Promote, deprecate, and retire examples

For example, as illustrated in the preceding diagram, a REST API that adopts a URI versioning approach could have concurrent versions of its services running in production, though a rule of thumb is to have no more than two: a deprecated version and a new version. API consumers binding to the deprecated version are given a notice period (for example, three months) to switch to the new API before the current one is retired and they get an error.

A header-based approach could also be adopted, even in the case of version-less APIs (for example, GraphQL), especially in scenarios whereby backward compatibility simply isn't possible. In this case, the API gateway takes care of the routing to the right service version based on a version HTTP header. In this example, consumers could be notified that the current API (accessed without any version header) is deprecated and therefore they have a period of time to switch to the new one by adding a version HTTP header.

Once the deprecated API is retired, the default API (meaning accessed without the version header) becomes the newer one, at which point consumers that didn't switch may face issues when calling it. Needless to say, this approach is more intrusive and requires careful consideration.

In terms of tools, modern API management products should be capable of supporting the above mentioned activities at the API exposure level. Modern container runtimes, such as **Kubernetes** (kubernetes.io), provide robust support for dealing with this, especially if additional capabilities are introduced into the runtime, such as the **Helm** (helm.sh) package manager, which can be used to heavily simplify the process of releasing inter-related containers and defining environment-specific metadata, and **Istio** (istio.io), a service mesh that, among other things, introduces support for **canary releases**, which, in turn, enable techniques such as **A/B testing**.

Refer to the following link for more information on canary releases:

<https://martinfowler.com/bliki/CanaryRelease.html>

For more details on A/B testing, refer to the following link:

https://en.wikipedia.org/wiki/A/B_testing

Lastly, this task is typically carried out by platform engineers with input from API architects and developers as required.

Observe

In control theory, **observability** is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. In other words, it is the ability of a system to externalize internal state data, namely **logs** (verbose and text-based representations of system events), **traces** (data representing a specific event that occurred within the application), and **metrics** (a numeric representation of point-in-time data, such as counters and gauges; for example, CPU, RAM, and disk usage). These are referred to as the three pillars of observability, which are needed for monitoring and analyzing the whole system.

Refer to the following link for more information on the three pillars of observability: https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/c_h04.html

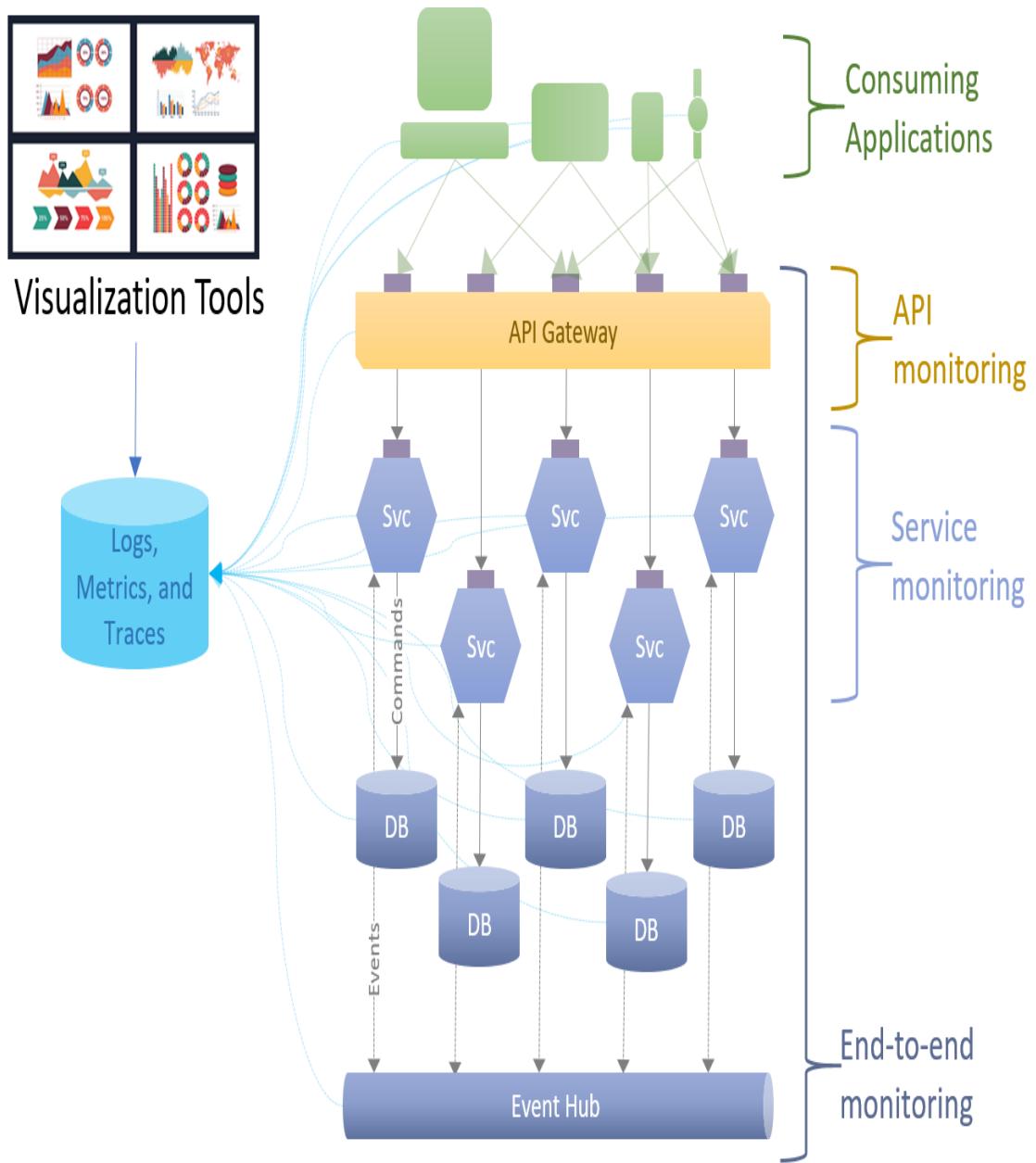


Figure 7.9: End-to-end observability

The preceding diagram illustrates multiple layers of an application stack that have been instrumented in order to send traces, metrics, and logs to centralized storage for analysis and visualization.

Although observability isn't new (for example, logging has been around since the beginning of programming), in distributed systems, such as microservice architectures, it has become paramount. Without properly instrumenting all components of the stack (as illustrated), performing tasks such as understanding the overall health of the system, debugging/troubleshooting issues, identifying potential performance bottlenecks, monitoring compliance against important **service-level agreements (SLAs)** (for example, availability and throughput), and even discovering potential security breaches can become a very difficult task, if not impossible.

Therefore, assuming that adequate capabilities have been put in place to instrument, collect, and visualize logs, metrics, and trace data, this step of the life cycle typically involves the following tasks:

API monitoring

The aim of this type of monitoring is to get real-time and historical insights on individual API metrics, such as the current health of an API, the total number of calls received, success/error call ratios, average response times, and even geo-related data (for example, where calls come from). This data should ideally be used to create a proper and holistic understanding of how APIs are being used, by whom, when, and from where. Such insights will help to drive important business and architectural decisions, such as which APIs

deserve further investment and enhancing as they're being heavily utilized, or which APIs should perhaps be retired as they're underutilized and their running costs are difficult to justify.

Another important aim of API monitoring is the definition of alerts and notifications to ensure that SLAs, such as availability, throughput, and response times, are not being breached.

In terms of tools, modern API management tools typically provide the means to conduct this type of monitoring. If not, there is always the possibility to instrument the API gateways so data can be collected in order to use specialized monitoring tools (see end-to-end monitoring for more details).

This type of monitoring is typically carried out by platform engineers and developers during day-to-day operation activities and/or when troubleshooting (though this type of monitoring isn't necessarily the most insightful for this type of activity). It can also be carried out by API product owners and API architects wishing to gain more information about an API's usage.

Service monitoring

The purpose of this type of monitoring is to gain real-time and historical-based insights about individual services, their interactions with other services, and their overall performance.

In runtimes such as Kubernetes, there are several capabilities available in order to conduct this type of monitoring; for example, the **Istio** service mesh can be configured to instrument log and trace data, which can later be visualized and analyzed with tools such as **Prometheus** (prometheus.io) and **Grafana** (grafana.com). Other tools, such as Kiali (kiali.io), can be configured as well to get proper visibility about the service mesh.

This type of monitoring is typically carried out by platform engineers and developers during day-to-day operations and when debugging and troubleshooting services.

End-to-end monitoring

As its name suggests, this type of monitoring aims to gain holistic and complete understanding of the entire system, as opposed to just its individual pieces. This type of monitoring is more complex, as it requires instrumentation across relevant components of the distributed system in order to collect meaningful log, metrics, and trace data. Assuming this has been done, end-to-end monitoring then can be classified into the following categories.

Log analytics

This type of monitoring involves collecting, aggregating, indexing, and analyzing log data produced by applications. The aim is to use this data in aid of activities such as debugging,

troubleshooting, root cause analysis, SIEM (described subsequently), or just day-to-day operations.

There are several log analytics tools on the market; however, **Splunk** (splunk.com) and the **ELK stack** (elastic.co/elk-stack) are very popular choices. This type of monitoring is typically also carried out by platform engineers and developers during day-to-day operations, and when debugging and troubleshooting APIs and their services.

Security information and event management (SIEM)

This is a sub-discipline of log analytics. The aim of this type of monitoring is to continuously analyze log and trace data in search of potential security alerts that may compromise the system. This task involves defining correlation rules and patterns to analyze structured and unstructured data, with the aim of identifying potential security vulnerabilities.

In terms of tools, there are several options on the market for this; from log analytics such as **Splunk** that also support SIEM (though as an additional capability), to specialized vendors such as **LogRhythm** (logrhythm.com) and even traditional security vendors such as **McAfee**, which also plays in this space. This type of monitoring is typically carried out by a security team, which will normally explain what type of data is expected to be collected.

Application performance monitoring (APM)

The aim of this type of monitoring is to continuously observe the performance and availability of applications. This type of monitoring is extremely useful for detecting and diagnosing complex application performance problems in distributed systems.

In terms of tools, vendors such as **AppDynamics** (appdynamics.com) and **New Relic** (newrelic.com) are popular choices for specialized APM software.

However, large log analytics vendors, such as **Splunk**, also offer APM add-ons.

This type of monitoring is typically carried out by platform engineers and **performance engineers** who are conducting day-to-day operation activities, but also when they are troubleshooting performance issues and/or when performance tuning the system.

Distributed tracing

Given that, in distributed systems, a single business transaction may span across several architectural components (for example, API gateways, load balancers, services, databases, or event hubs), understanding and analyzing transaction flows can be extremely complicated. Therefore, in order to reconstruct the series of events that build up the flow, each component of the distributed system must generate traces.

Such traces must therefore share some form of **correlation ID**, so the end-to-end flow can later be reconstructed.

The correlation ID is typically generated by the first component that handles a request (for example, an API gateway or sometimes even the client application itself) and is then propagated to all subsequent components that take part in the flow.

Once distributed traces are available for different transaction flows, this monitoring capability becomes extremely powerful for pinpointing which component in a transaction flow is responsible for potential issues (for example, performance bottlenecks or errors).

Zipkin (zipkin.io) and **Jaeger** (jaegertracing.io) are perhaps the most popular open source tools for distributed tracing; however, tools such as **Splunk** and **ELK** could also be used for this purpose (but will most likely require additional add-ons).

This type of monitoring is typically carried out by platform engineers during day-to-day operation activities and also by developers when debugging and troubleshooting APIs and services.

Note that some large cloud vendors, such as Microsoft, Oracle, and IBM, do offer packaged tools that support all of the aforementioned types of monitoring. However, whether to use them or not really depends on the context of implementation.

The API design-first life cycle

As briefly described in [chapter 4, *API-Led Architectures*](#), the purpose of this cycle is to establish an iterative and interactive process by which API consumers get early visibility of, and access to, an API specification and its corresponding mock while the API is still being designed. By receiving feedback as early as possible in the life cycle, reworking can be avoided.

Although the majority of steps in the cycle were described earlier in this chapter, the following diagram completes the concept by illustrating the fact that feedback should be collected in order to start the cycle all over again until the right design (one that satisfies the needs of its consumers) is accomplished.

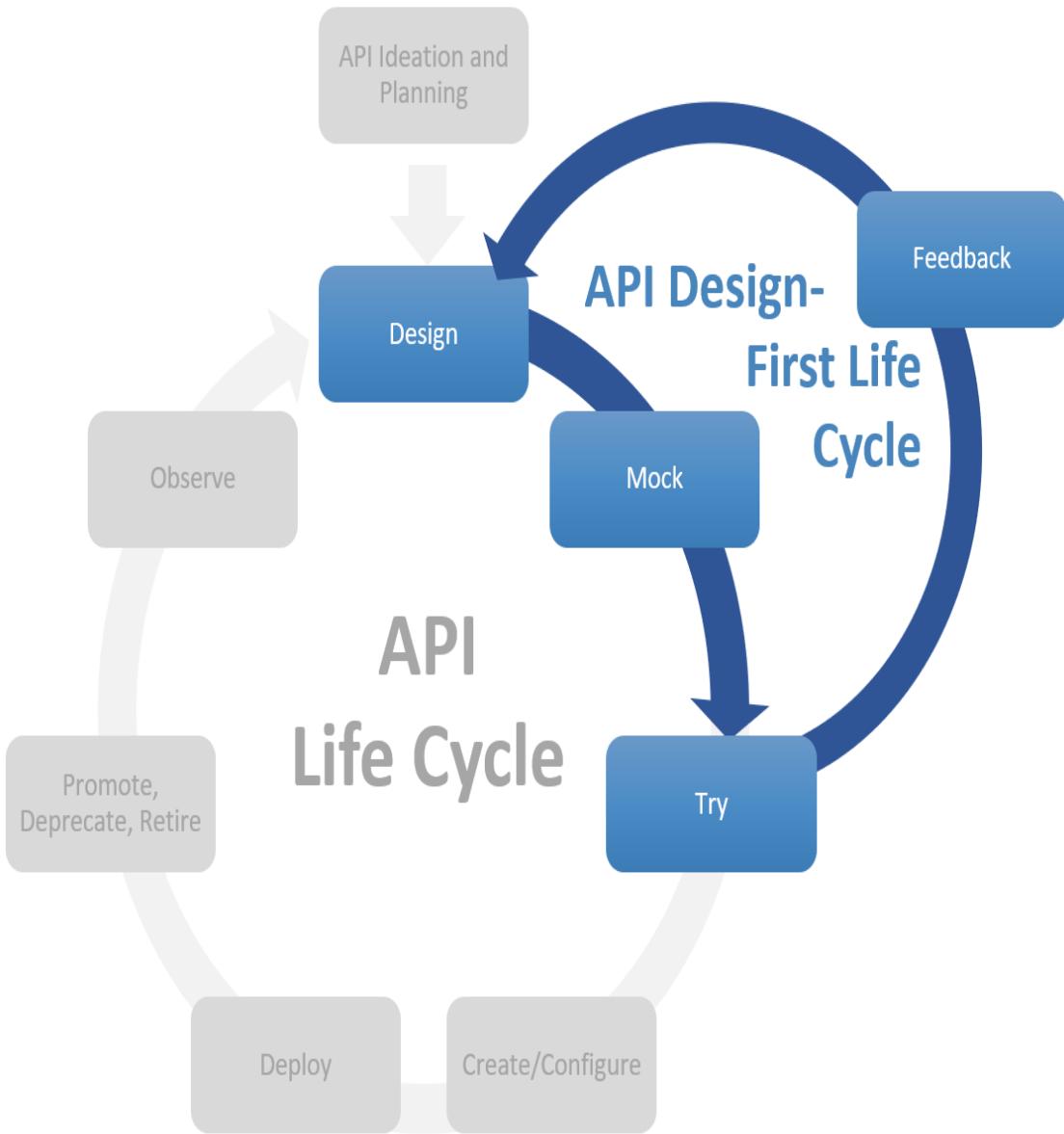


Figure 7.10: The API design-first life cycle

Don't forget that the process is meant to be iterative and interactive. Therefore, establishing efficient and fluent communication channels between designers and consumers of APIs becomes key, as otherwise, any feedback collected might be limited or vague, thus defeating the purpose of the process itself.

Adopting team collaboration tools such as **Slack** (slack.com) or **HipChat** (hipchat.com) can really help in establishing such channels. Furthermore, API product owners should have an active role in establishing such communication channels and in ensuring that feedback is being constantly collected.

For detailed information on how to implement an API design process using Apiary, please refer to the following book:

<https://www.packtpub.com/virtualization-and-cloud/implementing-oracle-api-platform-cloud-service>

Service life cycle

This crucial cycle defines a process by which services can be implemented in accordance with API specifications and designs. The process starts once an API design is deemed complete (refer to the previous section), meaning that, in addition to the conceptual design, an IDL and an API mock are both available.

As the topic of implementing services is extensive and lots of public bibliography is already available online, the aim of this section is not to describe each step in detail, but rather important considerations to bear in mind when implementing services in support of API-led architectures.

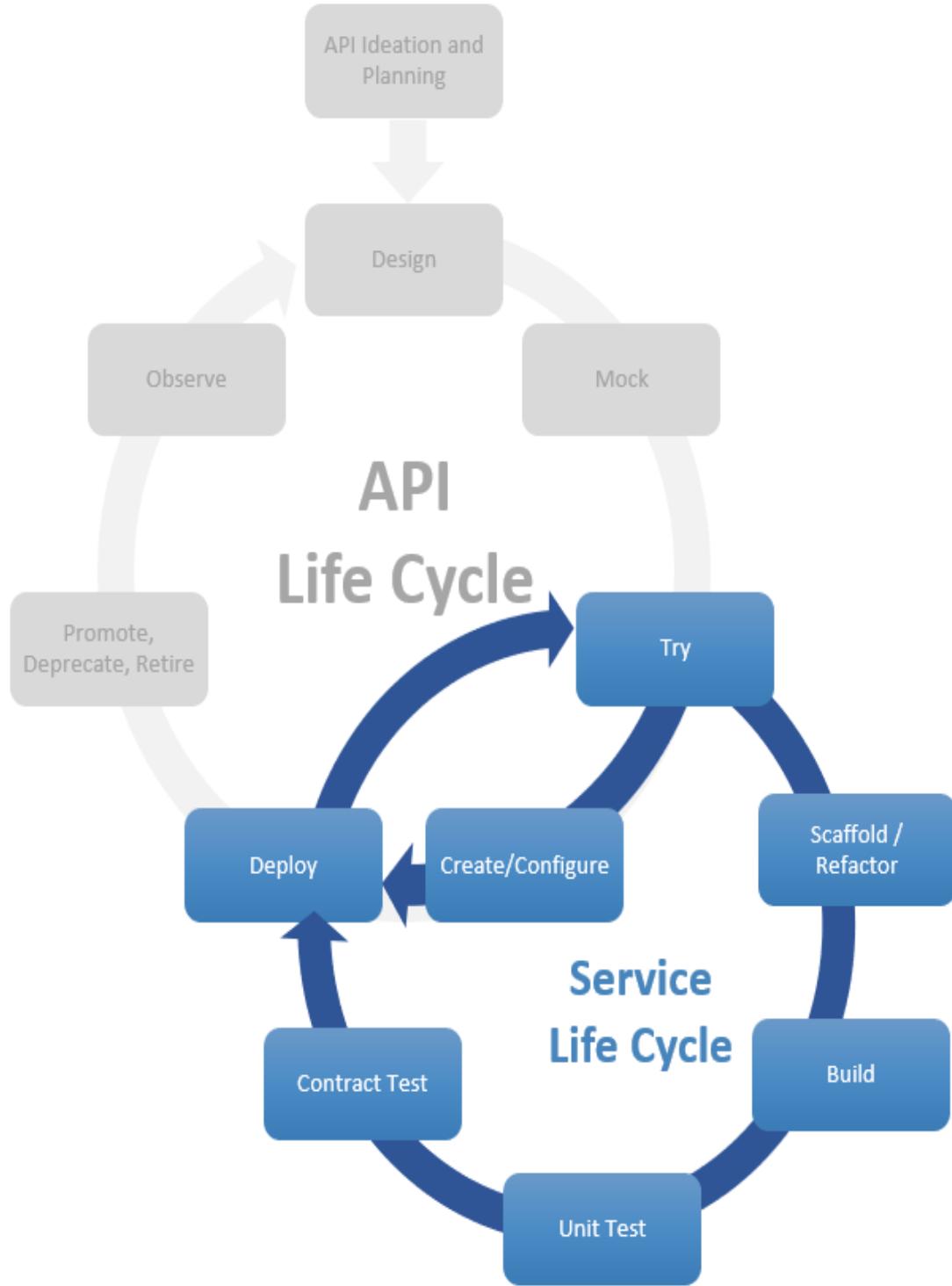


Figure 7.11: The service life cycle

Scaffold/refactor

The first step of the process is to generate a skeleton of the code required to implement the interface, which is a process sometimes referred to as "scaffolding an API server." In the case of gRPC, for example, this is done by simply making use of the `protoc` command line to generate both client stubs and servers in several languages.

Refer to the following link for tutorials on how to do this in multiple languages:

<https://grpc.io/docs/tutorials/>

In the case of REST, perhaps the most popular tool out there that is used to scaffold servers is the **OpenAPI generator** (openapi-generator.tech), which can be used to scaffold REST API servers from **OpenAPI Specification (OAS)** in about 40 different programming languages and technologies.

For GraphQL, tools are rapidly emerging to provide scaffolding capabilities directly from GraphQL schemas. A good example is the **graphql-cli** project (github.com/graphql-cli/graphql-cli), which can be used to rapidly scaffold a GraphQL server from a GraphQL schema file.

Lastly, it is important to also appreciate that although scaffolding a server for the first time can be a straightforward

task, as the design is iterated, the task can become more complicated as the scaffolded server would have already been extended with its implementation logic. This can be avoided by, for example, not just simply overriding a previously generated server code, but rather using tools that can help to differentiate and highlight what has changed in newly generated servers.

Build and unit test

This is the step where developers get to write the code that implements the business logic expected of a service. Although the implementation approach and best practices vary depending on the programming language and/or framework used, developing code can be a labor-intensive task and its successful execution heavily relies on the skills of the developers performing the job.

Some common good practices when implementing code are:

- Adopting a robust and modern **source code control system (SCCS)**, preferably based on **Git** (git-scm.com), not just because of its vast popularity, but also because of the fact that it is feature-rich and mature, there is tons of public information online, and the majority of open-source projects use it.
- Related to the previous point, adopting an adequate branching strategy also becomes extremely important, especially in large development teams where parallel development is a common practice. Not doing so means that adding new features to services and handling multiple releases can be messy and error-prone.

- **GitFlow** (nvie.com/posts/a-successful-git-branching-model) is perhaps the most widely adopted **Git** workflow in large projects. However, other branching strategies exist, such as Feature Branches (atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) and Forking Flow (atlassian.com/git/tutorials/comparing-workflows/forking-workflow).
- Complete and robust **unit testing** as part of the service codebase. This type of testing is all about ensuring that all of the different units that build up a service (for example, classes and methods, properties, and modules) behave as designed. Mature languages and development frameworks, such as Java or Spring Boot (a Java framework), have several tools for unit testing (for example, Junit or Mockito). Regardless of the technology choices made, unit testing is always possible, even if it means writing and executing custom classes or functions for such a purpose.
- Choosing an adequate tool to write/edit/debug/test the code for the technology choices made. Although it might sound obvious, it's trickier than it sounds. For example, should you adopt a fully featured **interface development environment (IDE)** such as Eclipse (eclipse.org), NetBeans (netbeans.org), or IntelliJ (jetbrains.com/idea), or a **source code editor** such as VSCode (code.visualstudio.com), Atom (atom.io), or Sublime (sublimetext.com)? Making such choices is becoming increasingly difficult, especially as code editors such as

VSCode are not just very lightweight, but come with hundreds of add-ons for several languages and frameworks (Java and Spring Boot included). However, using fully fleshed IDEs such as IntelliJ also has its advantages in the context of Java, at least given its maturity and vast number of features. In summary, selecting the right tool or tools for editing code is not a trivial task and deserves the right level of attention.

Contract test

Also referred to as an interface test, this type of test focuses exclusively on verifying that a service public interface matches its specification. In the case of GraphQL and gRPC, this test is less relevant as their IDLs are tightly coupled to their service implementations. But for REST, where multiple IDL options exist (for example, OAS, API Blueprint, or **RESTful API Modeling Language (RAML)**), the implementation of the API (its service) is normally decoupled from its IDL (assuming an API design-first approach was followed) and, therefore, executing this test makes a lot of sense to prevent services from being released that don't match its specification.

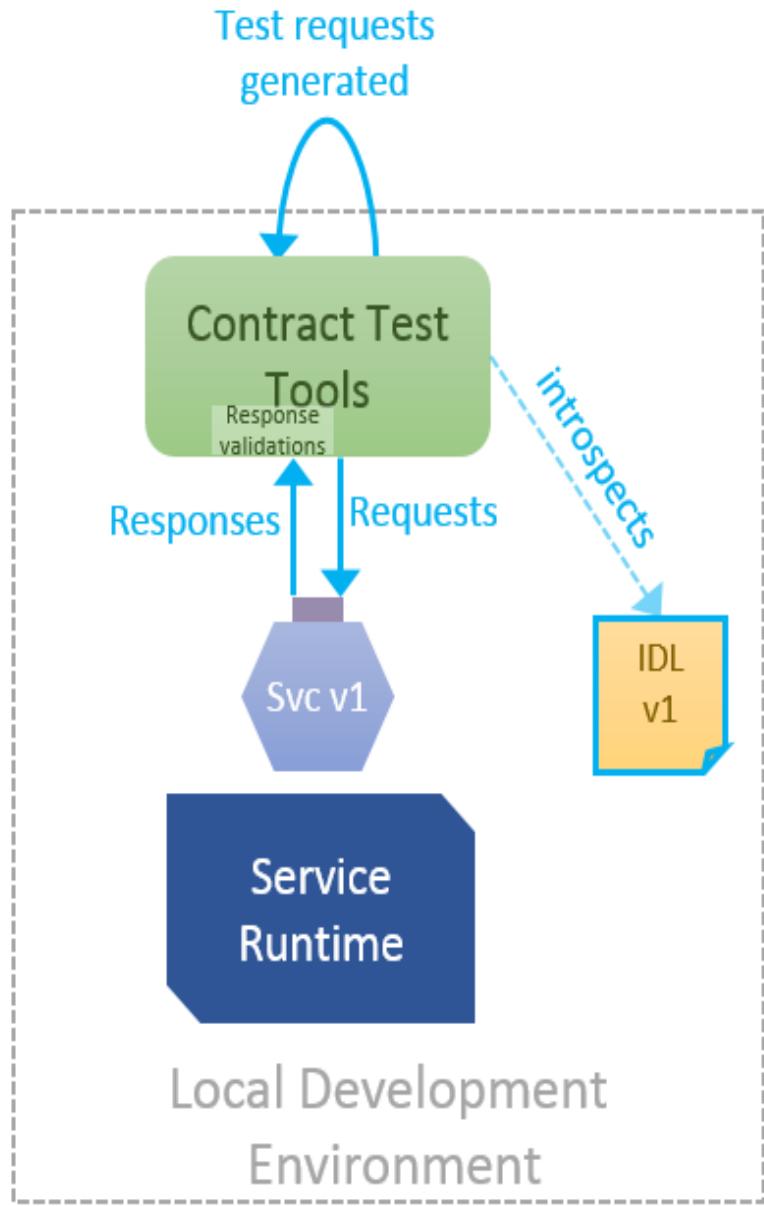


Figure 7.12: Verifying the service implementation

The process is simple. As illustrated in the diagram, a contract test tool feeds from an IDL in order to verify that the service implementation corresponds to its interface definition.

*Tools such as **Dredd** (dredd.org), **ReadyAPI** (smartbear.com/product/ready-api), and **PostMan** (getpostman.com) can be used for this purpose. However, when selecting a tool, bear in mind that some actually require tests to be manually defined. Other tools, such as **Dredd**, automatically generate tests by introspecting an IDL.*

Lastly, this activity should be carried out by service developers. As a good practice, code should not be considered completed until all contract tests are passed, as otherwise, a service might be released that is either not fully compliant or lacks functionality and features.

Customer life cycle

This life cycle defines, at a high level, key activities that are carried out by application developers when implementing functionality within their applications that require calling APIs. This flow is important as it steps away from the details of designing and implementing an API; rather, it looks at the implementation and usability aspects of an API from the point of view of its consumers.

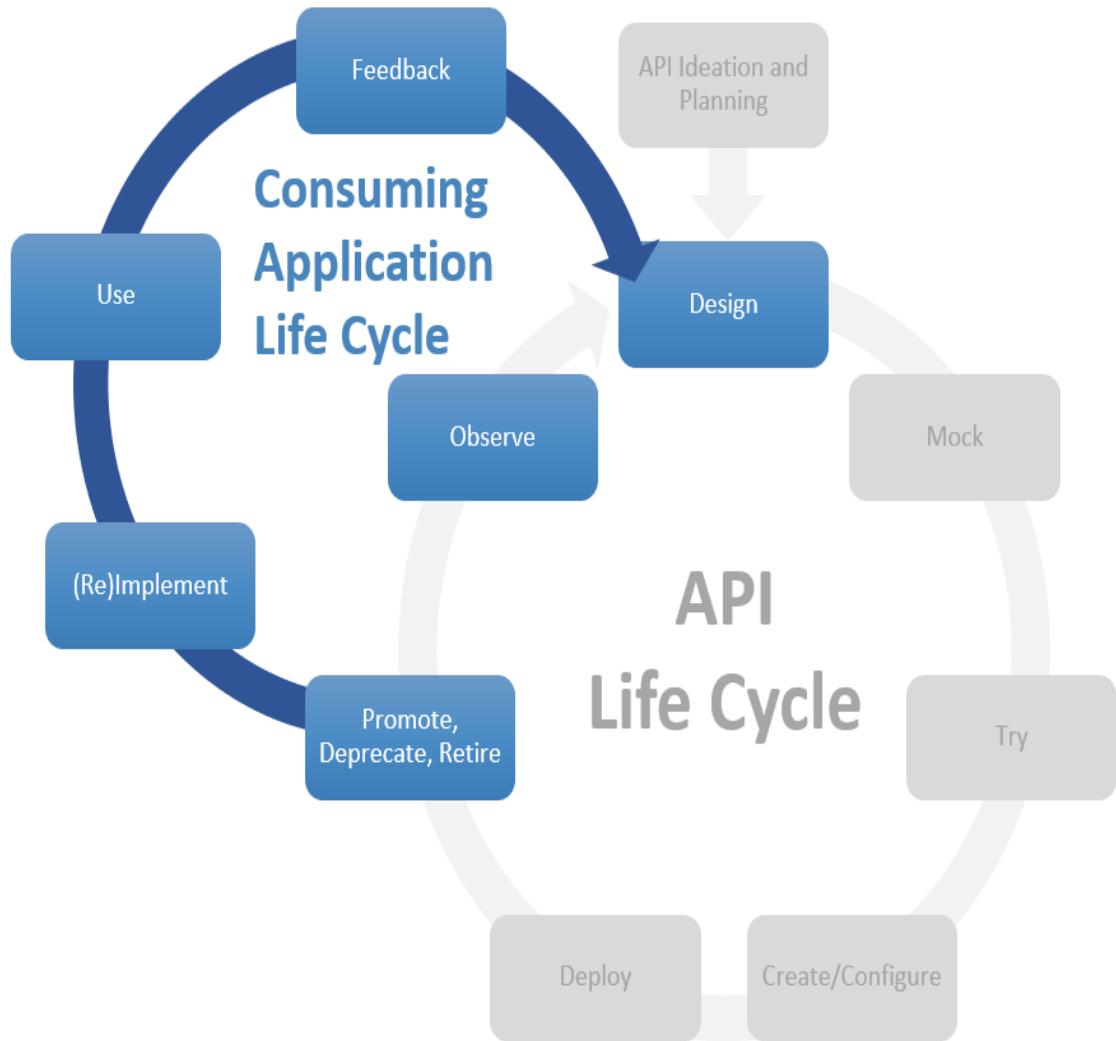


Figure 7.13: The customer life cycle

Note that although the flow indicates that the cycle starts once an API has been promoted, in practice, the promoted API does not necessarily have to be an actual service but can be a service mock, as described in the API design-first steps covered earlier in the chapter.

Implementation and use

Implementation is the process by which an application developer creates the code artifacts required to call a given API (typically as an application model). Once the model is created, then it can be used by the viewmodels (in model-view-viewmodel architectures) or controllers (in model-view-controller architectures) as required.

When using REST, API design tools such as **Apiary** and **SwaggerHub** also provide capabilities to scaffold client application code in several languages.

In gRPC, the process of generating gRPC stubs is similar to that of generating servers that was described earlier in the chapter (basically just using the `protoc` compiler).

For GraphQL, there are several client libraries available, each providing different features, such as response-caching and call handlers.

The following articles provide a good overview of GraphQL client options:

<https://medium.com/open-graphql/exploring-different-graphql-clients-d1bc69de305f>

Furthermore, there are a few considerations to bear in mind when implementing clients. Client caching is not a given. In the case of REST APIs, this is much easier, as not only is it one of

the fundamental constraints of REST (refer to [Chapter 6, Modern API Architectural Styles](#), for details), but it is closely aligned to HTTP. Headers, such as Cache-Control and Expires, can be used for this purpose. In the case of GraphQL, it is more difficult because each call can be different. Therefore, GraphQL clients can't rely on HTTP caching. Instead, it has to be part of the client application code, though most clients do provide rich support for this (for example, the Apollo GraphQL client provides rich client caching support). In the case of gRPC, libraries are emerging to accomplish this task (for example, <https://godoc.org/k8s.io/client-go/tools/cache>).

Adopting patterns such as the **Tolerant Reader** (find more information at <https://martinfowler.com/bliki/TolerantReader.html>) really helps in making the consumer code less exposed to server-side changes. Implementing a tester client for external APIs and running tests regularly within the application CICD process (or some form of scheduler) can be very useful for quickly detecting whether, by any chance, a non-backward compatible change has been introduced, or to simply check the health and status of APIs that the application depends on.

Feedback

Lastly, just like in the case of the API design-first life cycle, frequently collecting feedback from API consumers, not just on their experience of adopting the API, but also on the API performance once the API is used by an application, is extremely valuable in order to continually improve the API throughout the API life cycle. Therefore, in order to make the process of collecting feedback easy, in addition to the communication channels described earlier (for example, Slack), additional capabilities could be added in the developer portal so that consuming application developers can directly rate and comment on individual APIs.

Summary

This chapter delivered a comprehensive overview of not just the entire API life cycle, but also related cycles that are derived from it.

The chapter started by explaining how good APIs could be identified by running a series of well-structured and interactive ideation workshops. Next, the chapter described the different activities required when designing an API and its associated services. The subsequent sections then focused on elaborating on each step with the use of examples.

The next chapter will look at the organizational aspects of APIs by describing what it means to treat APIs as products and a **target operating model (TOM)** suitable for this purpose.

API Products' Target Operating Model

The previous chapter delivered a comprehensive walk through of the end-to-end API life cycle. This chapter elaborates on the non-technical aspects of implementing APIs by focusing on the organizational implications of handling APIs as business products. To this end, the chapter describes what it means to think of APIs as products, as opposed to IT assets. The chapter explores the implications of doing so and the impact this has on the organization, teams, roles, responsibilities, and even communication structures.

The chapter outlines a **target operating model (TOM)** that is suitable for handling APIs and related teams as **profit generating organizations**, as opposed to just IT cost centers.

Products in the real world

Before talking about APIs as products, it is important to first understand what products actually are in the real world, how they are classified, and how they are perceived by buyers.

According to many definitions, a product is any good and/or service that satisfies a need (a lack of a basic requirement) or a want (a specific requirement for products or services to match a need) and thus can be offered to a market. From a customer standpoint (someone who acquires the good or service: the buyer), a product is something that delivers a benefit, as otherwise there would be no point in acquiring it.

Products can be **tangible** or **intangible**. Tangible products are physical goods (objects), such as phones, cars, drones, or anything that can be seen and touched. Such products are easily identifiable and don't require further explanation.

Intangible products, on the other hand, are trickier to understand, as they can't be seen and/or touched and are not physical objects. They can be virtual goods, such as mobile apps purchased through an app store, **Software as a Service (SaaS)** purchased from a cloud vendor, or services offered by an organization and/or individual, such as catering services, hospitality services, or even software development services.

Note that the word "intangible" in the preceding definition is used just to define the type of product and not its potential benefits, which could well be tangible in the literal sense. Refer to the following link for a more detailed explanation:

https://en.wikipedia.org/wiki/Intangible_good



Figure 8.1: Examples of tangible versus intangible products

In some cases, a product may combine tangible and intangible goods. For example, the vast majority of modern cars today are smart in the sense that they have onboard computers, rich displays, and lots of proximity sensors. In addition, they have apps such as a navigation system, real-time traffic information, and auto-pilot. However, without Internet access, many of these features are of limited use and, in some cases, they simply wouldn't work. Therefore, when such cars are sold, they are typically bundled with data connectivity as an add-on to the main purchase.

Regardless of their type, however, products must always deliver customer benefits. In fact, according to marketing experts, most notably **Philip Kotler**, a prominent figure in the marketing world and an author of over 60 books on the topic, "*products are the means to an end wherein the end is the satisfaction derived from addressing a need or want.*"

According to Kotler, customers choose a product based on its perceived value, and thus are only satisfied if the actual value of the product equals or exceeds the original perceived value. To this end, Kotler defined the **five levels** of a product, commonly referred to as the **customer value hierarchy**, because the higher a product reaches on the circle, its higher the perceived value.

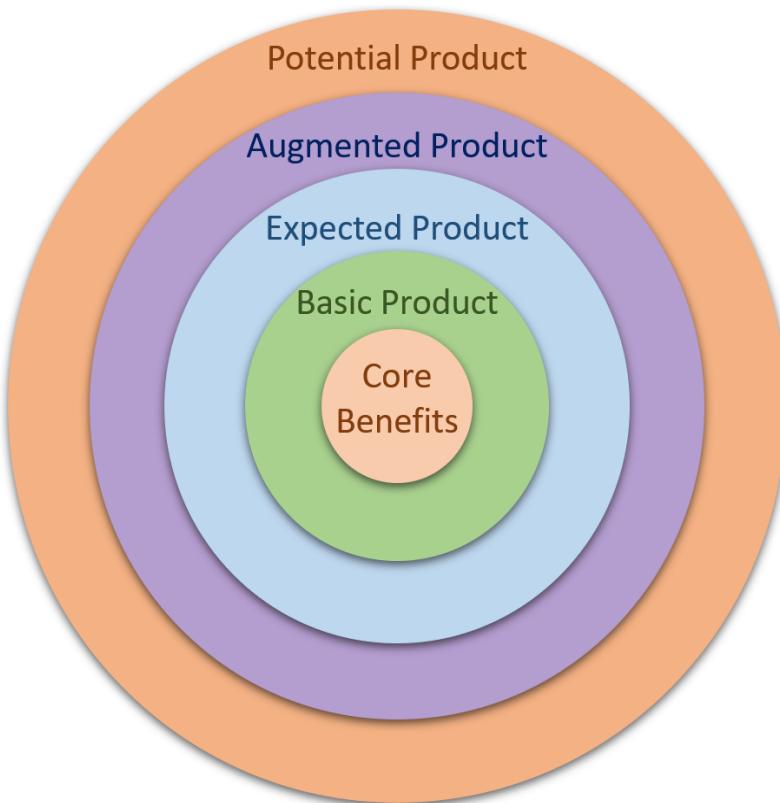


Figure 8.2: The customer value hierarchy

The levels are as follows:

1. **Core benefit:** This is the most fundamental level as it refers to the actual service or benefit that a customer is buying. For example, what a hotel guest really buys is rest and sleep.

2. **Generic product:** This turns the core benefits into a tangible or intangible product. Using the previous example, this means offering the hotel guest, at the very least, a room with a bed and a bathroom.
3. **Expected product:** This refers to the typical benefits a customer expects when acquiring the product. For example, the average hotel guest expects a clean room, fresh sheets, towels, and tranquility.
4. **Augmented product:** This refers to additional factors that set the product aside from its competition. This could be the differentiators that exceed customer expectations or features that the customer simply did not expect, but that add further value. For example, for a hotel, this could be a prime hotel room view (for example, of the sea), the latest smart TV with free Internet access and apps such as Netflix, Amazon Prime, and other streaming services, or a virtual concierge.

Note that at this level, a product typically faces severe competition. Furthermore, each augmentation adds to the cost of the product and also increases customer expectations, meaning that the augmented benefits soon become expected product benefits.

5. **Potential product:** This is augmentations and transformations that the product could undergo in the future, typically referred to as the **product roadmap**.

More information on Kotler's five product levels can be found at the following URL:

<https://www.provenmodels.com/16/five-product-levels/philip-kotler>

APIs as products

Now that context has been provided about what products are in the real world, it is easier to explain and appreciate what it means to treat APIs as products. **API products** are intangible goods (because APIs are digital goods, not physical objects) that satisfy the needs of application developers and/or owners who seek quicker access to information, functionality, and/or innovation that is deemed necessary in order to deliver a basic, expected, or augmented product.

For example, API products could meet the following requirements:

- Quick access to innovation that would otherwise be nearly impossible and/or very expensive to build from scratch. A good example is embedding a maps capability within an application. Instead of building a new maps API, it would be cheaper and quicker to just make use of the Google Maps API.
- Obtaining high-quality and up-to-date data deemed crucial for certain business transactions whereby data that's already a few seconds old could have a considerable business impact. For example, an application that performs currency conversions (for

example, an expense claim application) will need access to up-to-date forex exchange rates information. Instead of having to extract this data from multiple public sources and in real time (which would be time-consuming and expensive), adopting one of the many forex exchange APIs will save time and money.

Typically, the more up to date the data offered is, the costlier the API will be.

- Executing complex calculations and/or analysis that would otherwise require a huge amount of compute power and many days of programming. For example, crop yield predictions based on historical weather and geo-special data, and using complex machine learning algorithms, could be made by leveraging machine learning and prediction APIs.

Note that [Chapter 1](#), The Business Value of APIs, describes in detail the value that APIs can offer based on different use cases.

Although these are just a few examples, they do have one thing in common: API products always deliver a clearly understood customer benefit, meaning that they satisfy a need or a want, and thus are worth someone acquiring, either internally within the same organization (**internal API products**), or externally by partners or third parties (**external API products**).

Treating APIs as products, however, doesn't just happen by magic.

The implications of treating APIs as products

Considering APIs as business products has fundamental implications for the organization, and especially for how software is delivered. This is because, as I've emphasized throughout the chapters of this book, APIs and the teams that deliver them can no longer be considered as IT cost centers; rather, they must be seen as revenue-and-profit-generating organizations that make use of APIs. This has deep consequences for how such teams are organized, measured, and managed.

From a development life cycle standpoint, this means:

- Moving away from a traditional delivery approach, whereby requirements are gathered at the start and it isn't until late in the life cycle that real feedback is collected from the end users (API consumers), to adopting agile approaches that encourage the collection of API consumers' feedback throughout the entire design phase. This means that when a design is deemed complete, it is because it truly satisfies the needs of customers.

- Ensuring that analytical and operational data is properly collected and presented in a way that makes it possible to get insights that help to continuously improve the product but also ensure that legally binding **service level agreements (SLAs)** are followed.
- Streamlining the software release process, so that the time it takes to deploy a working product is dramatically reduced, but without compromising on quality. Ideally, it should be possible to make multiple releases a day into production.

Chapter 7, API Life Cycle, described an API life cycle suitable for APIs as products.

From an organizational standpoint, this means:

- Moving away from a traditional project delivery approach whereby development teams, having completed an application delivery (for example, an API), hand over the running software to a support team for on-going support and maintenance. The focus of the support team is mainly on keeping the lights on and fixing defects as they appear, so new features are rarely introduced.
- Given the lack of innovation post go-live, in time, the running API moves away from being augmented (with differentiation) to being expected, and, soon after, it

moves to being basic, at which point the API can be considered legacy.

- Because both development and support teams are typically organized around technologies (for example, an architecture team for the software architecture and API design, a development team for implementing the API, and a database team for database design), the final software product is directly exposed to the observations made by **Melvin Conway** in his thesis, *How Do Committees Invent?*, which was later popularized as simply **Conway's law**.

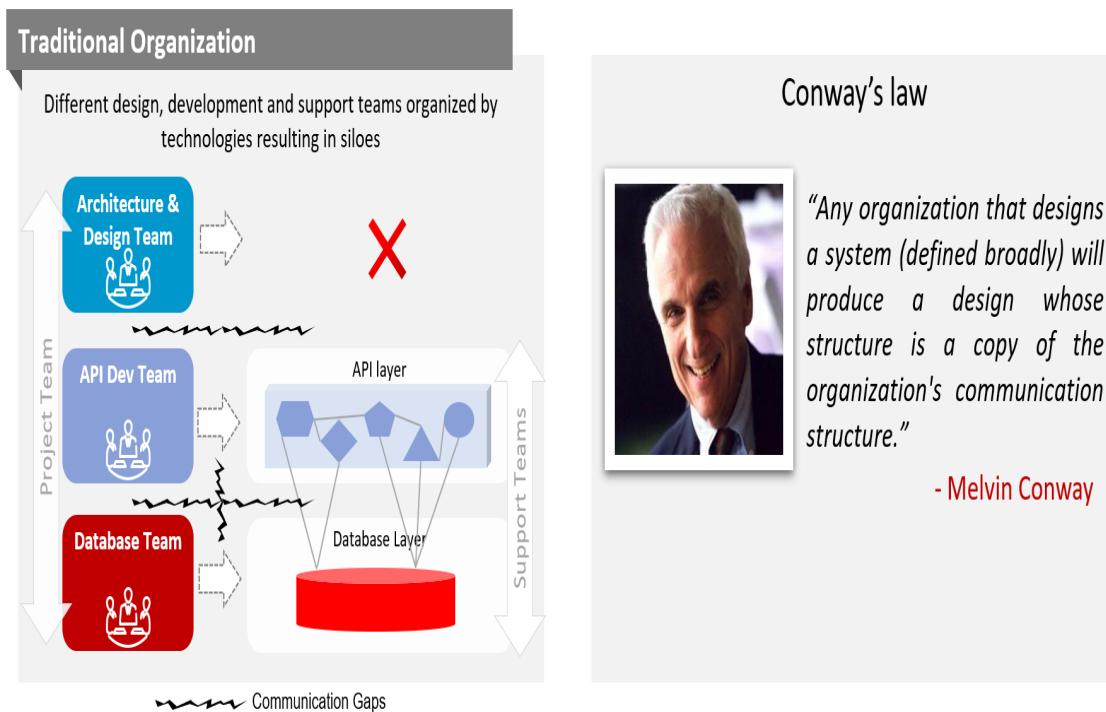


Figure 8.3: Conway's law

More information on Conway's law can be found on this website:

http://www.melconway.com/Home/Conways_Law.html

Although, in practice, Conway's law can have multiple interpretations, an undisputed one is that if the different teams responsible for designing, implementing, and supporting a system don't communicate effectively among one another, the communication gaps will eventually be reflected in the software they produce. **Fred Brooks**, another prominent computer scientist best known for his work on developing IBM's System/360 family of computers, discussed this in his paper entitled *The Mythical Man-Month*:

"Conway's law was not intended as a joke, a Zen Koan, but as a valid sociological observation. The law is a consequence of the fact that two software modules, A and B, cannot interface correctly with each other unless the designer and implementer of A communicates with the designer and implementer of B. Thus, the interface structure of a software system necessarily will show a congruence with the social structure of the organization that produced it."

We can therefore conclude that trying to deliver API products from teams organized around application tiers will likely result in APIs that look more like three-tier monolithic applications, with the APIs sharing a lot of the codebase, a runtime, and a database. In time, such APIs will be difficult to change and scale (as a change in a single API may impact others, and scaling a single API means that the entire system has to be scaled), and will rarely evolve.

Many attempts at demystifying and even proving Conway's law have been published and are publicly available. Therefore, this chapter will not elaborate further on the topic.

In order to avoid the negative effect of Conway's law, a better and more effective approach is to first define the business products from which APIs will be derived (refer to the *API*

ideation section of [chapter 7, API Life Cycle](#)), and then assemble multi-disciplinary teams around each of these products for their subsequent delivery.

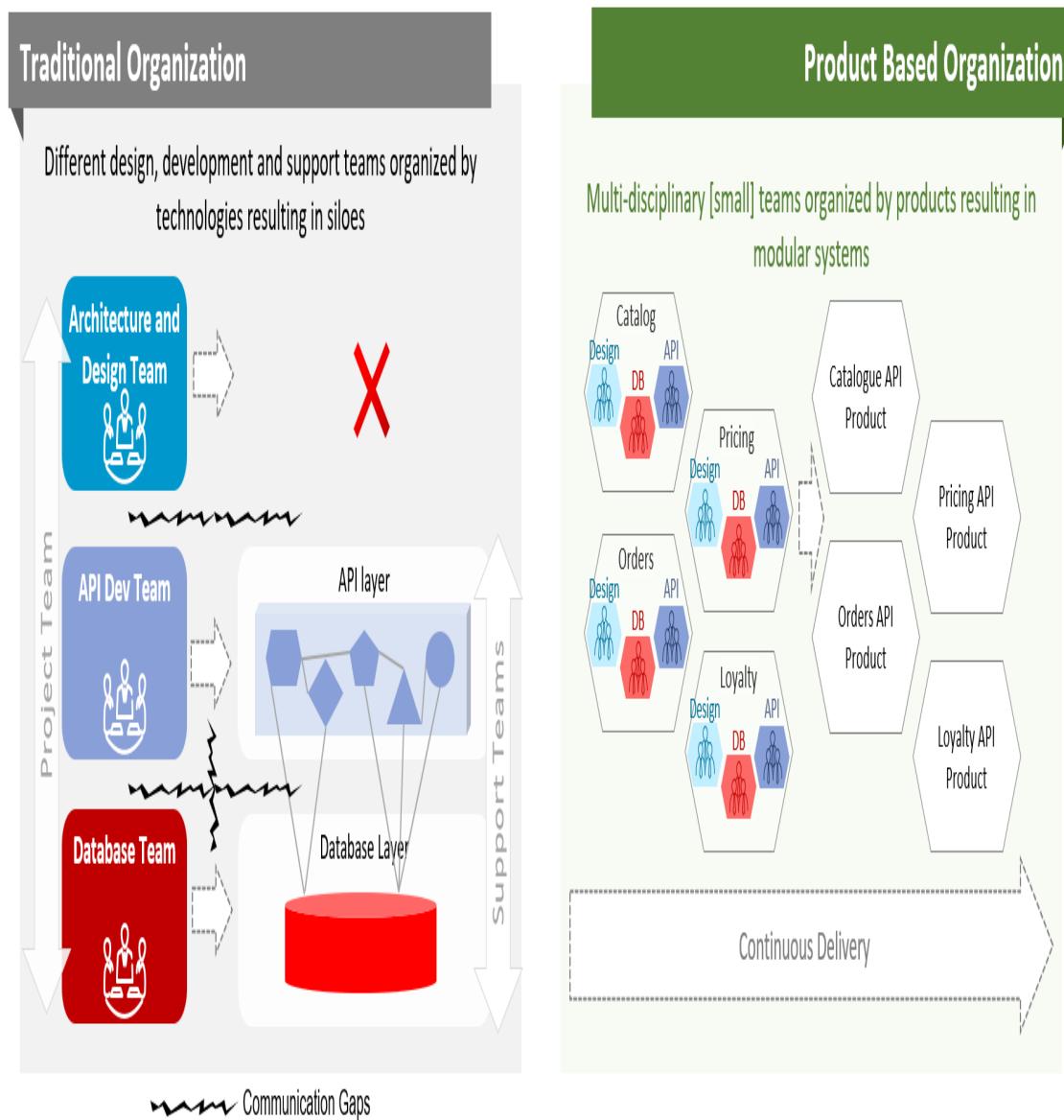


Figure 8.4: Traditional versus product-based operating models

This approach is sometimes referred to as **Conway's law inside out** because teams and communication structures are

created in support of how the resulting system is expected to look.

This approach is referred to by the ThoughtWorks Technology Radar as the Inverse Conway Maneuver.

<https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver>

Furthermore, product-based teams typically have the following characteristics:

- They are organized around the entire product life cycle and not project phases, meaning that teams remain consistent over time and there isn't such a thing as a project end.

*Note that a product is typically aligned to an existing **business capability**. For example, most organizations will have a business capability around pricing, meaning that a pricing API is productizing that pricing business capability. In some cases, though, new business capabilities may have to be created in support of a new product that can't be mapped to an existing one.*

- Each team has full accountability for the success and/or failure of their product.
- Each team has a product owner who is responsible, among other things, for setting the product vision, defining the requirements (for example, agile epics and user stories), and executing the vision.
- The team puts customers central and thus defines customer value hierarchies to ensure that the product is continuously successful and not just at one point in time.

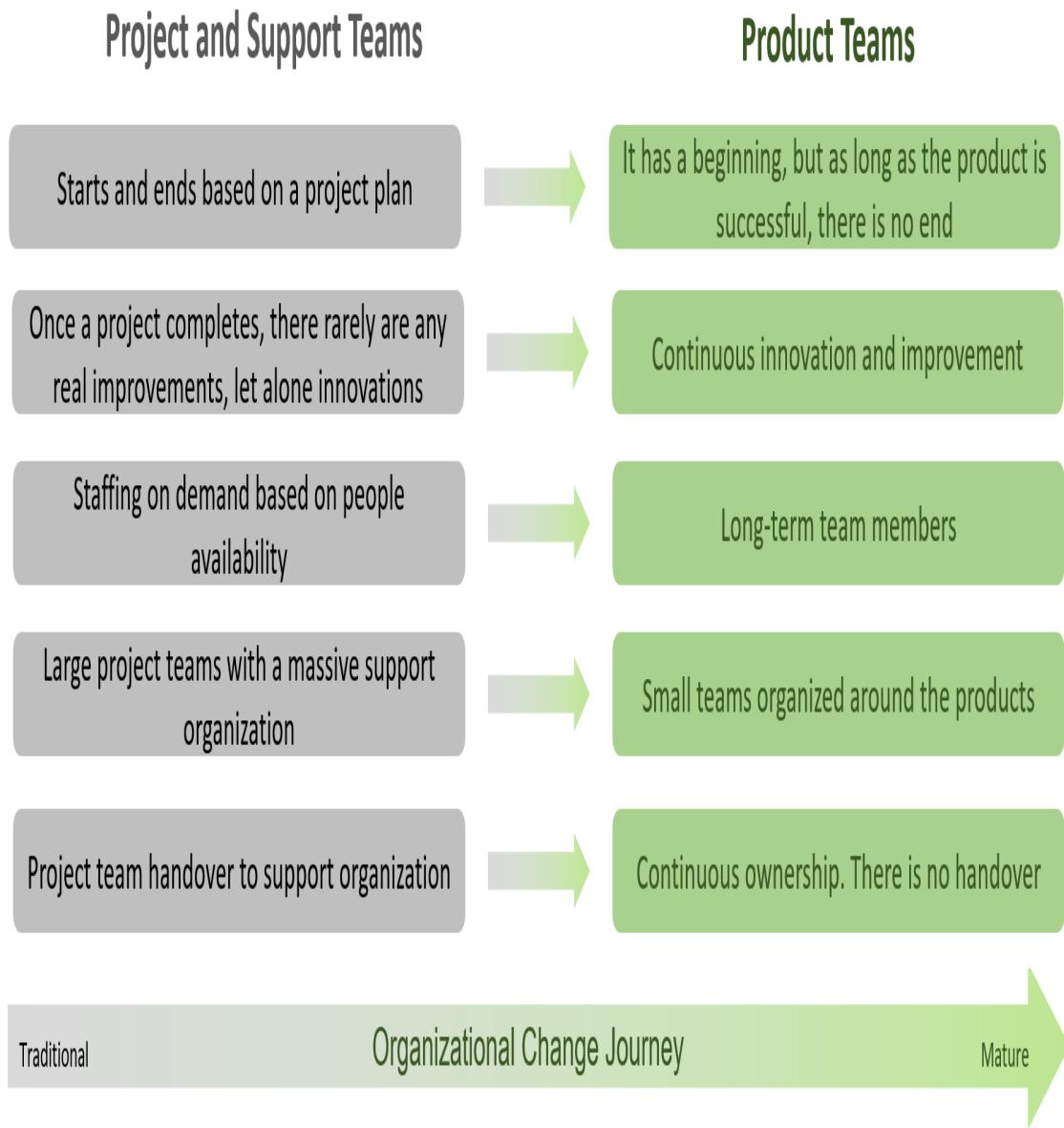


Figure 8.5: Project versus products

Lastly, from a business standpoint, it is important to appreciate that API products:

- Are business assets that can generate revenue and thus deserve the same attention (investment, backing, and support) as other products.

*Refer to [Chapter 4](#), API-Led Architectures, and the section entitled API monetization and billing, for more information on different **API monetization models**.*

- Will too require involvement from other parts of the organization, such as sales, marketing, finance, and even customer service departments. This support is as fundamental for APIs as it is for any other products offered by an organization.
- Will be bound to **key performance indicators (KPIs)** and SLAs. Therefore, having an operating model suited for APIs as products becomes crucial.
- Will require executive sponsorship in order to drive the organizational change that is required to put in place a suitable operating model for API products.

The subsequent sections of the chapter explore a TOM that is suitable for API products.

What is a TOM?

An **operating model** is a visual representation of how an organization, or parts of the organization, operates in order to produce products that deliver customer value and profit. Put simply, it describes the present (current) state of how an organization (or part of it) runs.

A TOM, however, does not focus on the present, but rather on the future. The purpose of a TOM is to define how the organization will run in order to execute a strategy, such as a new or revised API strategy that now considers APIs as business products and thus takes into account the numerous implications, as described previously.

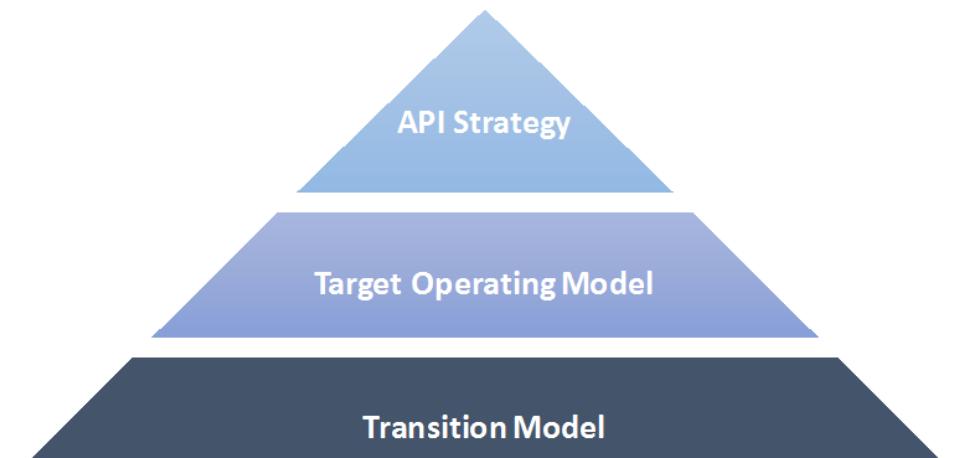


Figure 8.6: TOM in context

As a TOM naturally implies change, not just in the way of doing things, but also in how teams are organized, and given roles

and responsibilities, a TOM should always be accompanied by a **transition model** describing how the organization can move from its current state to the target state. This is bearing in mind sociocultural implications, such as the aforementioned Conway's law, and the fact that most humans don't like change.

As Chapter 3, Business-Led API Strategy, already described in detail an approach for creating API strategies aimed at delivering business and customer value, the focus of this chapter is to elaborate on the TOM and transition model.

Defining the model

There are multiple ways to define a TOM and, in fact, an extensive bibliography exists on it. For example, enterprise architecture frameworks, such as **The Open Group Architecture Framework (TOGAF)**, **Zachman**, or the **Aris House**, incorporate within their approach the means to model a business in a deliverable typically referred to as the **business architecture**.

Other frameworks, such as **POLISM**, or the **Operating Model Canvas**, are less broad and are designed specifically for creating TOMs. There are also industry-specific frameworks, such as **eTOM** in telecommunications, and **IAA** in insurance.

Although all these frameworks can serve as a solid foundation for businesses to rely on when defining their TOMs, when it comes to subject-matter areas such as API products and their management, a pragmatic approach, more focused on addressing key concerns, rather than delivering standard documentation, will arguably be easier and quicker to produce.

The concerns that should be addressed are:

- 1. Organization:** Will there be a single centralized and all-encompassing team offering all API-related capabilities? Or will each business unit and/or department have its own capability? What about a mix of the two?
- 2. Roles and responsibilities:** What are the roles and responsibilities required in the new organization?
- 3. Collaborations model:** How will the different teams interact and collaborate? What will the handshakes be among different teams?
- 4. Transition approach** (aka the transition model):
How can you move from the current organization to the target one as smoothly as possible and with the least amount of disruption?

Once all these concerns are addressed in a way that technical and non-technical people can understand, the outcomes produced can then be incorporated into a broader business architecture that may already exist within the organization. The subsequent sections of the chapter address each of the preceding points in more detail.

Organization

When it comes to defining where enterprise-wide capabilities, such as API product design, implementation, and support, actually reside, organizations have traditionally favored two main approaches, as follows.

Central organization

As its name suggests, this approach consists of grouping all of the expertise and tools required to deliver a given capability into a single team. This type of organization is also referred to as a **Center of Excellence (CoE)**, although, in practice, this approach is rarely used due to practical reasons, such as finding the right expertise and/or costs.

It is also not uncommon to see large enterprises outsource entire CoEs to offshore and/or near-shore delivery centers. Although the motivations for this may vary, one of the common reasons is to save on costs.

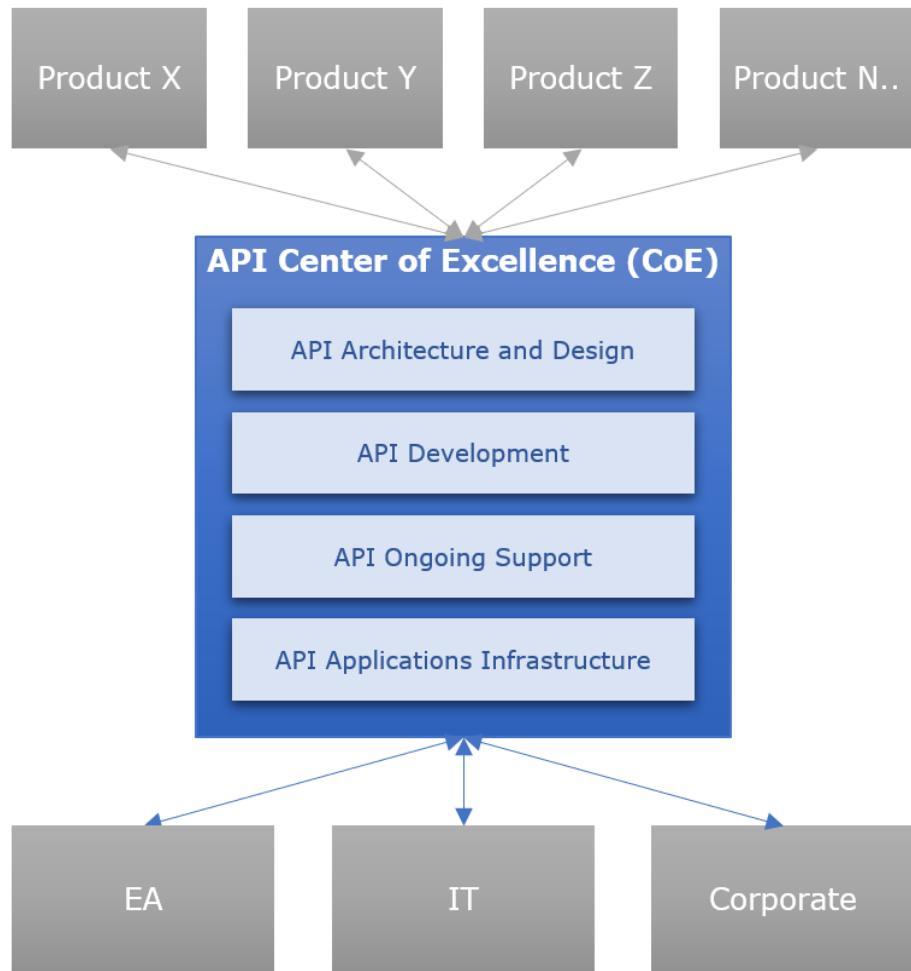


Figure 8.7: Centralized API organization

Under this model, any product team in need of API-related capabilities has to engage with the CoE in order to obtain the required level of support.

Note that in the model, a product can be an API product and/or any other business product that requires APIs. The same applies to the subsequent diagrams shown.

From a management standpoint, having all API-related capabilities under one roof brings some notable advantages. For example, knowledge exchange should be easier, which, in turn, should help with onboarding new personnel and delivering a common and industrialized delivery methodology.

This should result in getting higher-quality results quicker. Having common KPIs to track performance over the entire CoE should, in theory, also be easier.

In practice, though, this approach has proven to be inefficient, not just because it is bound to the downsides of Conway's law (and this alone should be a reason to seriously question this model), but also because, as requirements start to emerge from all over the enterprise (often in parallel), the central organization will struggle to scale and deliver on time, meaning it can't meet the demand. Therefore, the chances of the central organization becoming a bottleneck are high.

Added to this is the fact that different teams have different deadlines to meet, so handling priorities will also become extremely challenging. This often results in conflicts and escalations, as the CoE has no choice but to prioritize certain engagements over others.

Federated organization

Instead of having a single (and typically large) central organization, this approach favors a federated approach, wherein different product teams have the freedom and flexibility to build up their own expertise and use the tools that they deem necessary.

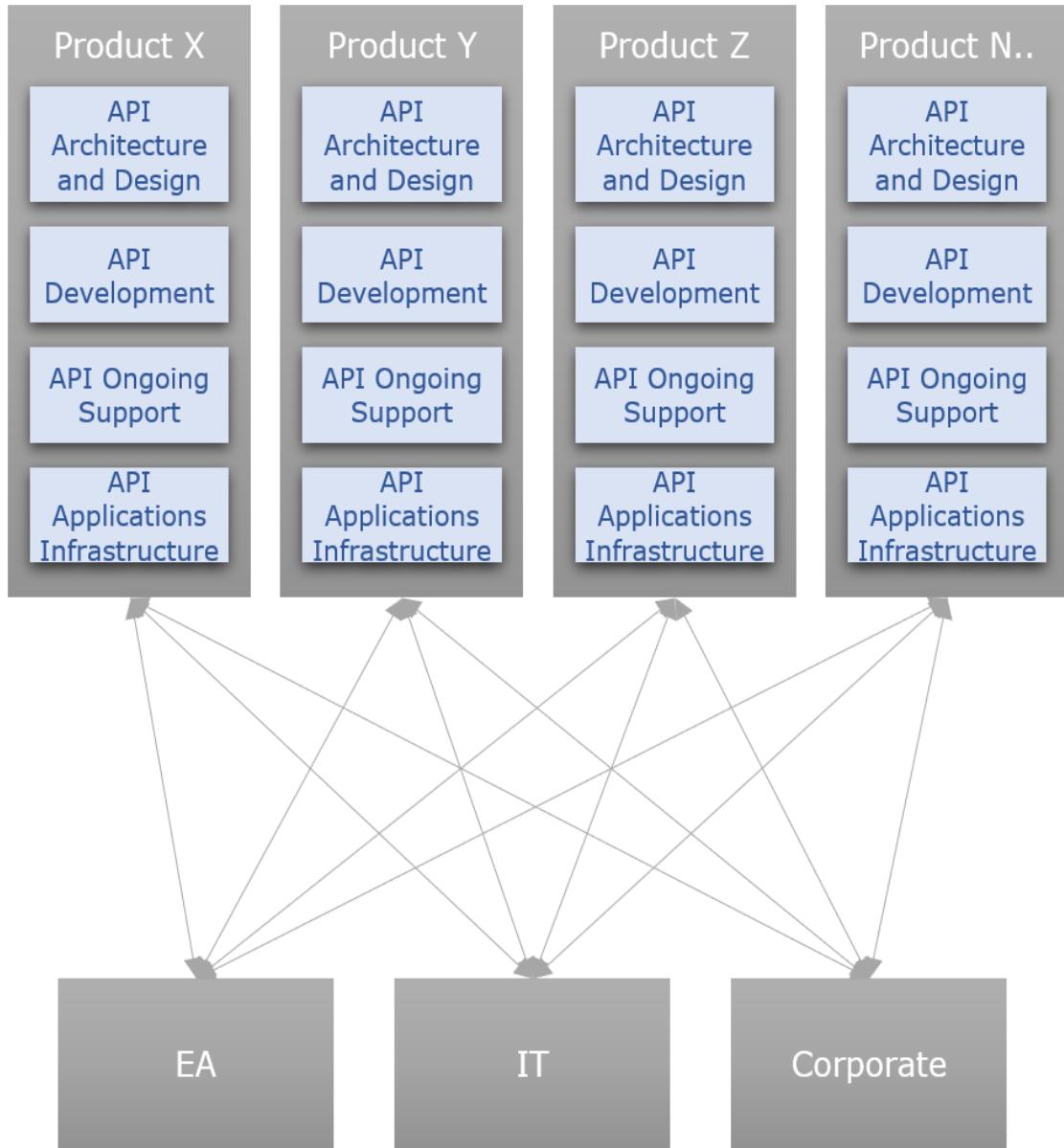


Figure 8.8: Federated API organization

Under this model, an **enterprise architecture (EA)** function tends to act as a coordinator and governance function, so a level of knowledge/experience exchange and standardization does takes place. For example, the EA function may offer a catalog of tools and capabilities to choose from, which can prevent a situation whereby hundreds of different technologies remain.

When done right, such a flexible model offers significant advantages; for example, a higher degree of innovation and speed. Also, because teams work under much fewer constraints, increased employee satisfaction can be experienced.

There are, however, some notable disadvantages and risks associated with this model, especially when applied to large enterprises. First of all, it is highly unlikely that the size of the EA function (or any other function acting as a mediator/coordinator body) will increase in proportion to the number of product teams, thus causing a notable degradation in the level of knowledge and experience exchange. Unless teams talk to each other directly in an efficient and collaborative manner (which, in large enterprises, can be a challenge due to factors such as deadlines, team locations, and even politics), the risk of teams becoming siloed is very high, and so is the risk of technologies proliferating to the extent that no one, other than the product teams themselves, really understands what each team is doing.

Talking specifically about APIs, this is really bad news, as one of the main advantages associated with API-led architectures is being able to discover and reuse APIs. This will be highly unlikely if each team does its own thing and uses its own tools.

To summarize, whereas this model can be effective for organizations of a small and medium size, implementing it in organizations of a large size is risky.

A platform-based approach

Before describing what this approach is and what it entails, it's important to first understand what a platform is. We all use platforms in our day-to-day lives.



Figure 8.9: Platform samples

As illustrated in the preceding diagram, apps we commonly use, such as Facebook, Netflix, Uber, YouTube, Spotify, Amazon, and eBay, are, in fact, platforms. They all share a set of common characteristics:

- **Platforms are self-service:** You don't need to go to a store or ring a call center in order to create a Facebook or Amazon account, or upload pictures or products. It's all done online through self-service. Even if you need to learn how to do something, there will be online tutorials and interactive guides.

- **Platforms enable us to connect and interact:**
Users of the platform can connect with each other and interact almost instantaneously and without the need for third parties.
- **Platforms create value:** Throughout all the interactions that occur in the platform, value is created. For example, the Amazon platform allows its users to publish their own products so others can find and buy them. This creates value for the users that publish products, for the buyers who ideally find a great deal, and for Amazon, as the platform owner, in addition to a fee that is typically charged for the transaction and service.

Extrapolating these concepts into the world of APIs looks like this:

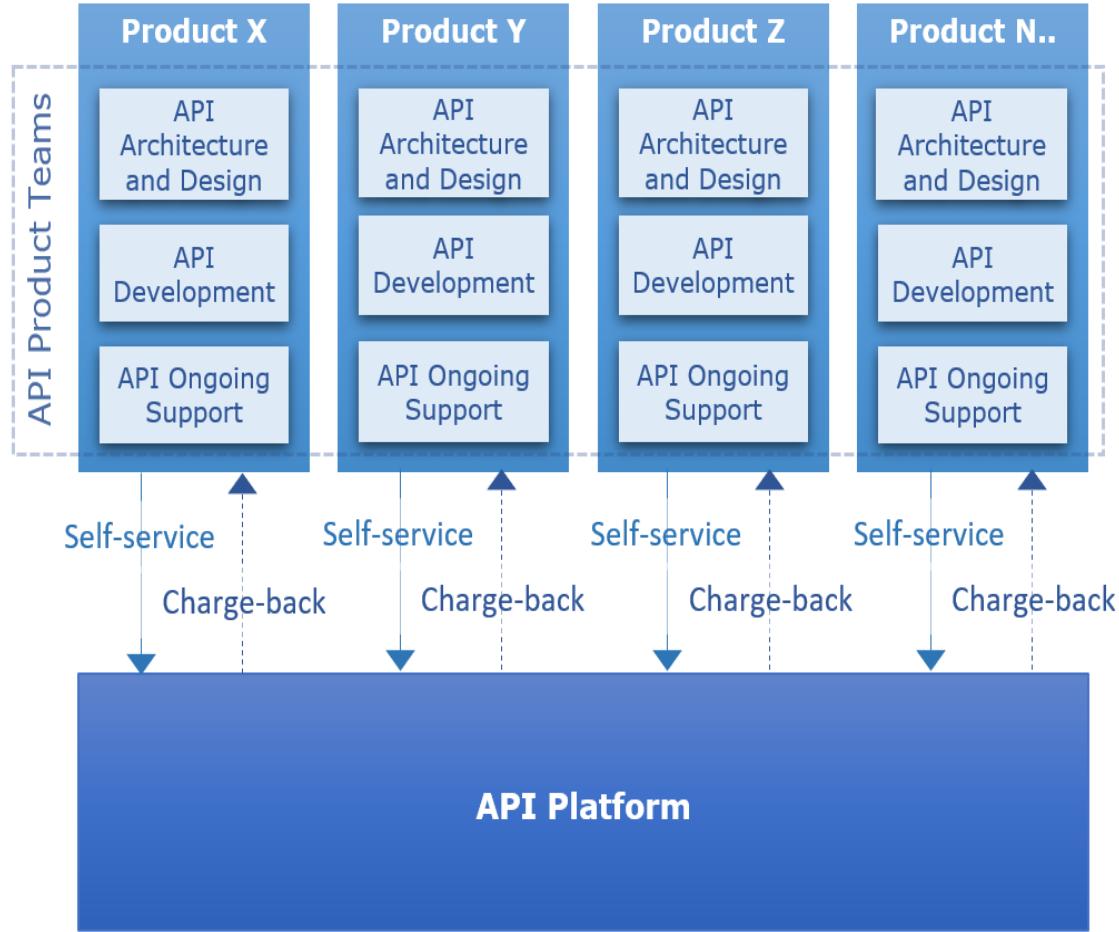


Figure 8.10: The platform model

The core idea is that by offering an API platform as a set of self-service capabilities (for example, those described in [Chapter 4, *API-Led Architectures*](#)), product teams don't have to undergo heavy processes and/or have to spend lots of time liaising with a central organization in order to get access to the tools and capabilities required to deliver API products.

This basically means that while API product teams will retain their independence, because a common set of capabilities will be used, there will intrinsically be commonality in what

technologies different teams use and how they use them. This, in turn, should also translate into lower operations costs.

Furthermore, as the platform itself should be non-monolithic, multi-cloud, microservices-oriented, and basically third generation (refer to [Chapter 2](#), *The Evolution of API Platforms*, for more details), product teams should also have a good degree of flexibility regarding their API architectures and deployment models, which is important in order to ensure that creativity and innovation isn't affected by forcing all users of the platform down a single path.

It is crucial to understand, though, that in order for this model to be successful, the platform must truly deliver on expectations. Failing to do so will most likely result in product teams being frustrated and being creative in finding ways to avoid utilizing the platform.

The subsequent sections continue to elaborate on a TOM that puts the platform concept at the epicenter.

*The book *Platform Revolution* is highly recommended for learning more about platform-based models and the impact they've had on digital revolutionaries such as Uber, Airbnb, Amazon, Apple, and PayPal.*

Roles and responsibilities

Under a platform-based TOM, there are two main teams for which clear roles and responsibilities are to be defined.

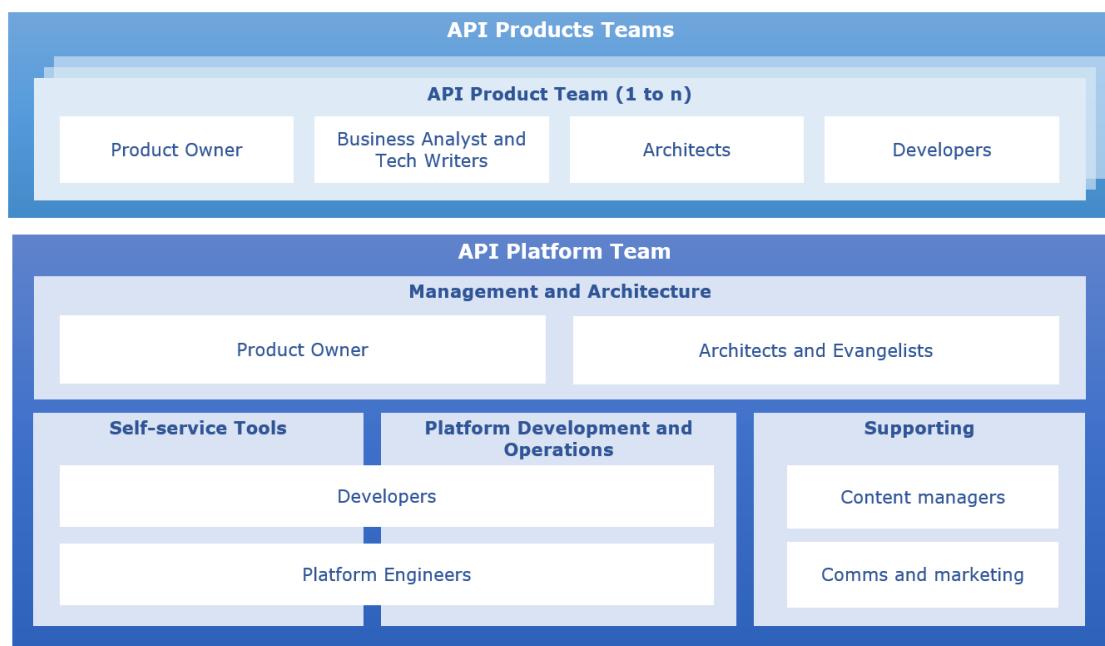


Figure 8.11: TOM roles and responsibilities for API products

API product teams

The **API product teams** are accountable and responsible for individual API products, and are therefore also responsible for executing all of the activities of the API life cycle (refer to [Chapter 7, API Life Cycle](#), for more details on the API life cycle).

There can be as many API product teams as required, as long as there is a clear understanding of which team owns which API. Ideally, there should be one-to-one mapping (one product to one team); however, on occasion, it might make sense to give a team more than one API. This should be done with caution in order to avoid the team becoming too stretched to the point that features are no longer delivered frequently.

Furthermore, it is recommended that API product teams should be relatively small. As a rule of thumb, the size of an API product team should follow Jeff Bezos' **two pizza rule**: "*If a team couldn't be fed with two pizzas, it was too big.*"

The following article explains the science behind Bezos' rule:

<https://buffer.com/resources/small-teams-why-startups-often-win-against-google-and-facebook-the-science-behind-why-smaller-teams-get-more-done>

In terms of how to measure success, different performance metrics could be measured against APIs produced by each API product team:

- **Time to market:** How long it takes from the moment a new requirement is accepted into the API product backlog to the moment it is released into production.
- **Meantime time to repair (MTTR):** This represents the average time it takes to fix a failed component. This metric is useful as it indicates how quickly the solution can recover from issues across the stack. A high MTTR means that it takes a long time to recover from issues. Therefore, in order to reduce this, investment in automation and resilient infrastructure should be undertaken.

More information on MTTR can be found at the following link:

https://en.wikipedia.org/wiki/Mean_time_to_repair

- **Mean time to deploy (MTTD):** This measures the average time it takes to successfully execute a production deployment. A low MTTD indicates that deployment to production can be done quickly and consistently, which, in turn, shows maturity in the deployment pipelines.
- **Successful API calls:** The average number of successful API calls processed per period of time.
- **Average response times:** Each API deployed in the platform should have an expected response time. For example, a given API may have an expected response time of 200 milliseconds. This KPI should therefore measure whether the overall response time for this API

over a period of time has been above or below the 200 millisecond threshold.

- **Availability:** This measures the overall availability of an API over a period of time. This metric can't be higher than the availability of the underlaying platform; it can either match it or be lower.
- **Customer success ratings:** Users of the API should be asked to indicate how satisfied they are with the functionality offered and their experience in using it. For example, if customers are asked to score an API from one to five, with one being the lowest score, then a good API should consistently score above four.

Whereas many other metrics can be defined, the preceding metrics typically are a good starting point. For more detailed information on success metrics in agile organizations, the following book is highly recommended:

<https://itrevolution.com/book/accelerate/>

The performance metrics listed (along with others) should be used as a basis for defining KPIs and SLAs. This is another important characteristic of an API product. For example, consumers of an API will normally have expectations about how an API performs, so defining KPIs and SLAs around successful API calls, average response times, and availability makes complete sense.

The difference between KPIs and SLAs is that the former focuses on past performance (for example, how an API

performed against individual SLAs and/or other metrics), whereas the latter defines future commitments, or, in other words, sets the expectations of API consumers about the performance of an API product (for example, expecting sub-second response times from API calls).

However, not all KPIs have to be SLAs. For example, customer success rating makes sense as a KPI, but not as an SLA. Therefore, understanding the differences and similarities between KPIs and SLAs is also an important factor in the success of an API product.

The following article provides more information on KPIs and SLAs:

<https://tinyurl.com/slavskpi>

Roles and responsibilities

The roles and responsibilities within an API product team are as follows.

API product owner

This person is responsible for defining the API product vision and the API customer value hierarchy. To this end, the product owner owns the definition of the product backlog and its priorities (for example, which features take precedence over others).

In order to be successful, the product owner must understand market trends that apply to the product and must come up with

innovative features to augment the product. In addition, the product owner should maintain as much interaction as possible with the users of the API, as their feedback can also help with defining new product features that add customer value.

This role also entails having interactions with other parts of the organization, such as the sponsoring business unit to report on progress, the sales team, the marketing team to promote the API product, or even the **API platform team** if additional support is required beyond that available with self-service tools.

Business analyst

A business analyst is a functional expert in the business domain that an API product relates to. For example, an API that offers eCommerce functionality will require someone that fully understands the eCommerce domain. Although, arguably, the product owner should be the domain expert, in practice, they may require additional support in order to properly define all of the epics and stories for an API. Architects, developers, and technical writers as well may need additional functional support, and this role can therefore offload some of these activities from the product owner so that they can focus on other value-adding activities.

This role may be a part-time one or only required for a certain amount of time. Especially in the initial stages and testing

stages of an API product, having someone with these skills can prove to be of great value.

Architects

They are responsible for converting all business and non-functional requirements, coming in the form of epics, stories, or features, into a solution that is fit for purpose for the API product in question. Architects are, therefore, responsible for analyzing requirements, producing an end-to-end conceptual design, making key design decisions, and working with developers to produce adequate API specifications and with technical writers to ensure that high-end documentation is produced for the API.

During the initial delivery, the architect is also responsible for mentoring and providing technical support to the rest of the team.

Once an API goes live, the architect should make use of observability tools offered by the platform (for example, how/when APIs are used, from where, common sources of errors, throughput) in order to identify areas of improvement or new features that could be added to augment the product.

Whether many architects are needed truly depends on the size and scope of the API. Typically, a single individual with the right skills is enough.

Developers

Developers are responsible for the actual coding and unit testing of services in accordance with the API specification and solution design. During testing phases, the developers should also provide ongoing support (for example, fixing bugs identified during the testing phase and request deployments for an API) as deemed required. Post go-live, developers should continue to deliver backlog features and fix defects as prioritized by the product owner.

Technical writers

Technical writers are techno-functional individuals who are well versed in the art of communications and language. They can translate documents of a technical nature (for example, an API specification or an architecture design) into more human-readable content fit for more general audiences. The idea is that the content produced should be easy to read and follow, as only then will consumers of the API appreciate the capabilities offered.

On some occasions, a business analyst may also be capable of undertaking a technical writer role and vice versa.

API platform team

The API platform team, on the contrary, is not responsible for individual APIs, but rather for the underlying platform on top of which APIs run. This team is therefore responsible for providing all of the self-service tools and capabilities required by API product teams to smoothly deliver APIs throughout their entire life cycle.

This team is also responsible for promoting the capabilities offered by the platform, with the objective to ensure that the platform gets used, so it becomes self-funding, assuming a fee is charged for the use of the platform. To this end, the API platform itself should be considered a product as well, along with also being subject to the customer value hierarchy.

In terms of how to measure success, many of the same performance metrics described for the API product team can also be applied to the platform itself. However, there are platform-centric metrics that won't apply to individual APIs:

- **Platform availability:** This measures the overall availability of the platform over a period of time, such as in a given year. For example, a platform that should be available 99.98% of the time should not experience more than 1h 45m of downtime over an entire year.

The following site is a great tool for doing this sort of calculation:

<https://uptime.is>

- **Total number of APIs:** The overall number of APIs deployed to production in a given period of time is a good indicator of how much the platform is being used. Ideally, the number of APIs should consistently grow over time.
- **Total number of API calls:** Related to the previous one, this KPI measures whether the total number of API calls is within the expected range in order for the platform to deliver value. The number of calls should increase over time, as this is also a good indicator that platform usage is also increasing.

The roles and responsibilities within this team are as follows.

API platform owner

This role is, in essence, similar to the product owner role, with the difference being that the product in question is the platform itself as opposed to individual API products.

The API platform owner's main goal is to ensure that the platform gets used and that value is derived from it, while, at the same time, ensuring that the platform meets all SLAs and KPIs. To this end, the platform owner should have a deep understanding of popular capabilities, those that aren't as popular (or not working well) and therefore require

improvement or retiring, and new features required to augment the product.

In addition, the product owner should understand different stakeholders within the organization in order to promote the platform with support from the platform architects, evangelists, and communication and marketing specialists.

Platform architects and evangelists

Platform architects are subject matter experts in the different technologies that underpin the API platform as a whole.

Ideally, platform architects should also act as evangelists, meaning that they should be capable of enthusiastically showcasing and demoing the platform as a whole and its key features.

Platform architects are also responsible for producing detailed technology designs and later on following up with the engineers and developers to make sure that the platform is delivered.

Architects should work closely with the product owner in identifying technology features that should be added to the backlog, such as required product patches and upgrades, additional compute capacity to comply with non-functional requirements, changes required to comply with regulation, bug fixes, and enhancements based on feedback.

As the platform may comprise different technologies and areas of expertise, typically, there is more than one architect to cover

the entire stack.

Platform engineers

They are responsible for provisioning, configuring, monitoring, and operating all runtime components of the platform.

Platform engineers must also be well versed on how the platform delivers against performance metrics. This involves API gateways and container runtimes, such as Kubernetes, along with **continuous integration and continuous delivery (CICD)** pipelines with tools such as Jenkins, Sonar, and Nexus.

If the platform runs entirely on cloud compute, then engineers should ideally be well versed in also configuring things such as virtual network configuration, load balancers, and edge services, such as traffic managers, web application firewalls, and DNS services.

Note that in some cases, however, these responsibilities fall under a different team (sometimes referred to as a cloud infrastructure team), in which case the platform architect should work together with the platform engineers in ensuring that all dependencies are delivered as expected.

Platform developers

Platform developers are responsible for the actual development and support of the self-service tools used by all platform users.

This is a critical role as its focus is to deliver on one of the fundamental pillars of a platform: the self-service capabilities.

Developers will typically have to work closely with the engineers to ensure that self-service capabilities are adequately delivered end to end throughout the stack.

Content manager

The main responsibility of the content manager is to create rich content, such as video tutorials, interactive how-to guides, and well-documented wiki pages, in order to assist users of the platform at different stages of engagement. This includes onboarding in the early stages of using the platform, using more advanced features, and knowing what to do and how to make contact in the case of issues.

The content manager should work closely with the platform owner and the self-service tool developers in order to ensure that the right content is created in the right place, aiding with using the self-service capabilities and also using the platform as a whole.

This role is extremely important since, even if a platform is robust, scalable, and modern, without proper guidance and tutorials with lots of examples, it will be difficult for new users to start using the platform, and existing users will struggle to fully leverage all features available, and won't know what to do

in certain scenarios, such as when facing common errors and workarounds.

The content created should also include information about the platform team, who the main points of contact are, and a summary of the vision of the platform and its roadmap. That way, everyone has clear visibility about the future of the platform, which features will be added, and when.

Communication and marketing specialist

The main purpose of this role is to promote the platform product across the organization. This, in turn, means advertising, across multiple internal channels, the platform capabilities, the team, and all success stories. This person is also responsible for organizing in-person and/or virtual educative sessions, such as podcasts, webcasts, workshops, or even hackathons, that could help to raise awareness about what the platform has to offer.

As this role shares some similarities with the content manager role, it's not unusual to have a single individual do both, especially during the early stages of the platform implementation, when demand is low. As demand increases, then the roles can again be split into two.

Communication and collaboration model

The success of a platform-based model isn't just determined by having good self-service capabilities. Having a fluent and dynamic communications structure that ensures the platform team communicates and collaborates effectively, not only among themselves but also with other parts of the organization, is crucial to avoid silos and also prevent the negative effects of Conway's law.

Therefore, a comprehensive view, illustrating the sorts of communication channels and interactions that would be expected across the team, can be extremely useful in achieving an organization that is not just effective in terms of providing tools, but that can also communicate and collaborate effectively.

The following is an example of how such a view could look:

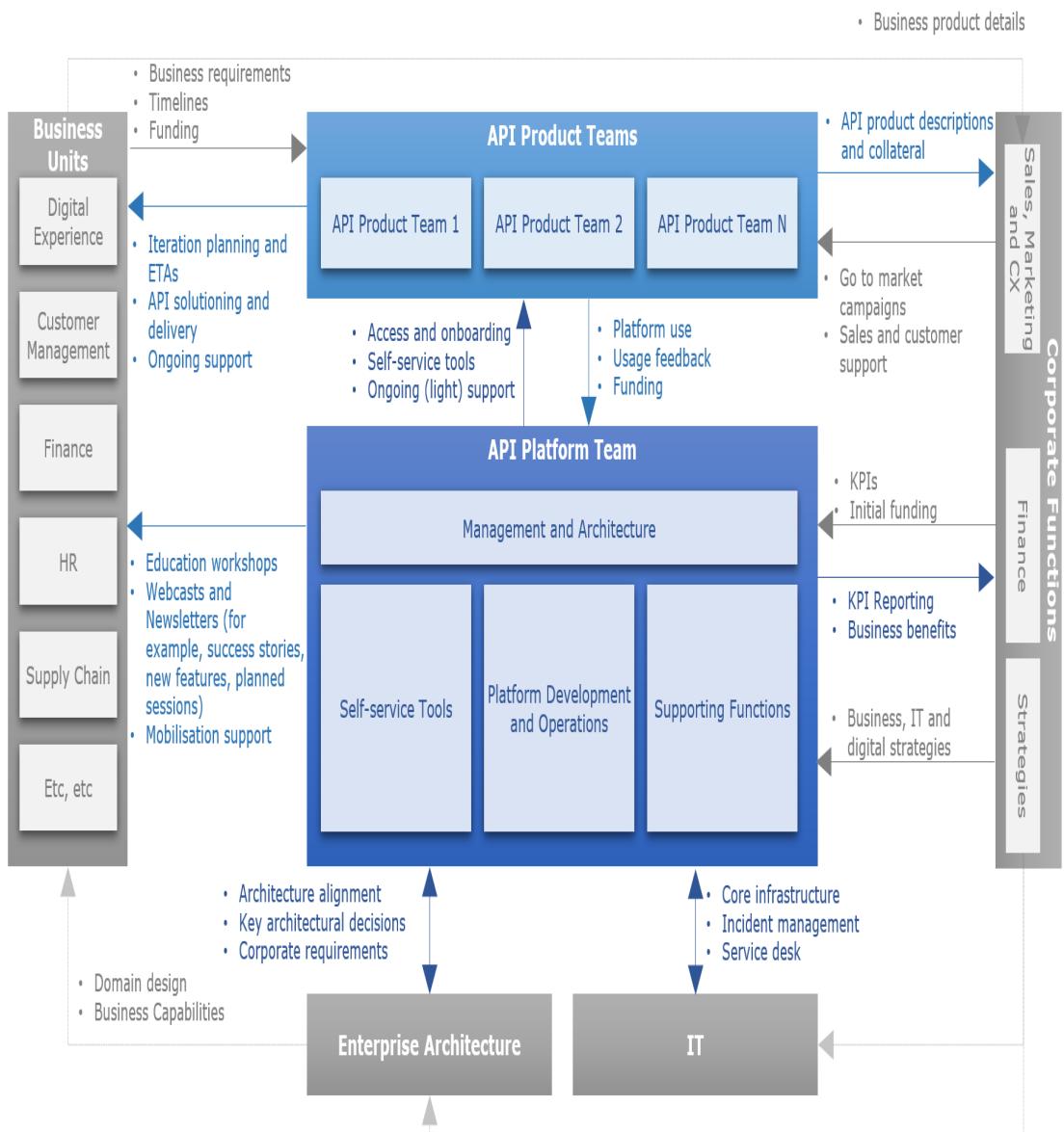


Figure 8.12: The API platform communication and collaboration model

As can be seen, the model incorporates which parts of the organization both the API platform and product teams are expected to interact with. Determining such blocks requires a good understanding of the business and how it is structured, and this will drive the interactions that are required.

The first and most obvious set of interactions is between the API platform and the API product teams. Although the intent of the platform model is that most interactions should be self-service, there will still be a need to provide access, onboarding, and a level of ongoing support to the teams using the platform (API product teams), especially at the beginning as the self-service tools mature. During this period, the platform team should go out there and promote its capabilities to the rest of the organization (other business units) to ensure that the platform gets used. This, too, should be reflected in the model.

Expanding on this further, the majority of large organizations have enterprise architecture and IT functions. Whereas the former is typically focused on producing and overseeing enterprise reference architectures, principles, and standards, the latter is mainly focused on technology tools and running business-critical systems.

The enterprise architecture function will normally expect alignment with, and a degree of influence over, the API platform team in regards to the architecture of the platform and other designs products. Therefore, such interactions should be modeled. Failing to reflect (or acknowledge) this may result in unnecessary escalations and/or frictions, all of which will have a negative impact on the overall delivery.

Similarly, in the case of the IT function, in order to address important dependencies, such as access to corporate tools, connectivity, critical systems, and provision cloud

infrastructure, engaging with IT is inevitable, and not doing so may result in roadblocks. These interactions should therefore be captured too.

Other important API platform team interactions are with strategy teams regarding direction and alignment, and finance for initial funding and performance metrics reporting.

For API product teams, obvious interactions are with the business units (or initiatives within the units) that produce the business requirements from which API products are derived and that also provide funding. These units/initiatives will likely expect ongoing progress reports and estimated times of arrival (especially as the API product being delivered may be part of a broader initiative).

Other important interactions for the API product teams are with corporate functions such as sales, marketing, and customer experience (sometimes referred to as customer management or CRM) departments. Whereas sales and marketing can help in identifying the right customer segment and running marketing campaigns, the customer experience team can help not just in making sure that the product delivers a rich user experience, but also that it is customer-centric.

Transition approach

Changing an organization and its communication structures is no easy task. In fact, it is quite the opposite. History tells us that initiatives that fail to recognize (or at least fail to recognize early) how difficult it can be to drive organizational change will seriously struggle to succeed. This is either because there isn't enough buy-in from key stakeholders, or simply because whichever capabilities the initiative delivers fail to gain the expected adoption.

Therefore, coming up with a transitional and incremental approach toward driving any sort of organizational change becomes a necessity. A transitional approach doesn't have to be complicated and could consist of only a few steps, each indicating what sort of change is required, and what benefits should be expected for each step accomplished. For example, as the organization (or part of the organization) transitions from being project-centric to being product-centric, ideally the customer value delivered should increase, while the time to market should decrease, both delivering better products faster.

The following model presents a simple transition approach that can be used as inspiration when defining a broader or more specific organizational transition strategy:

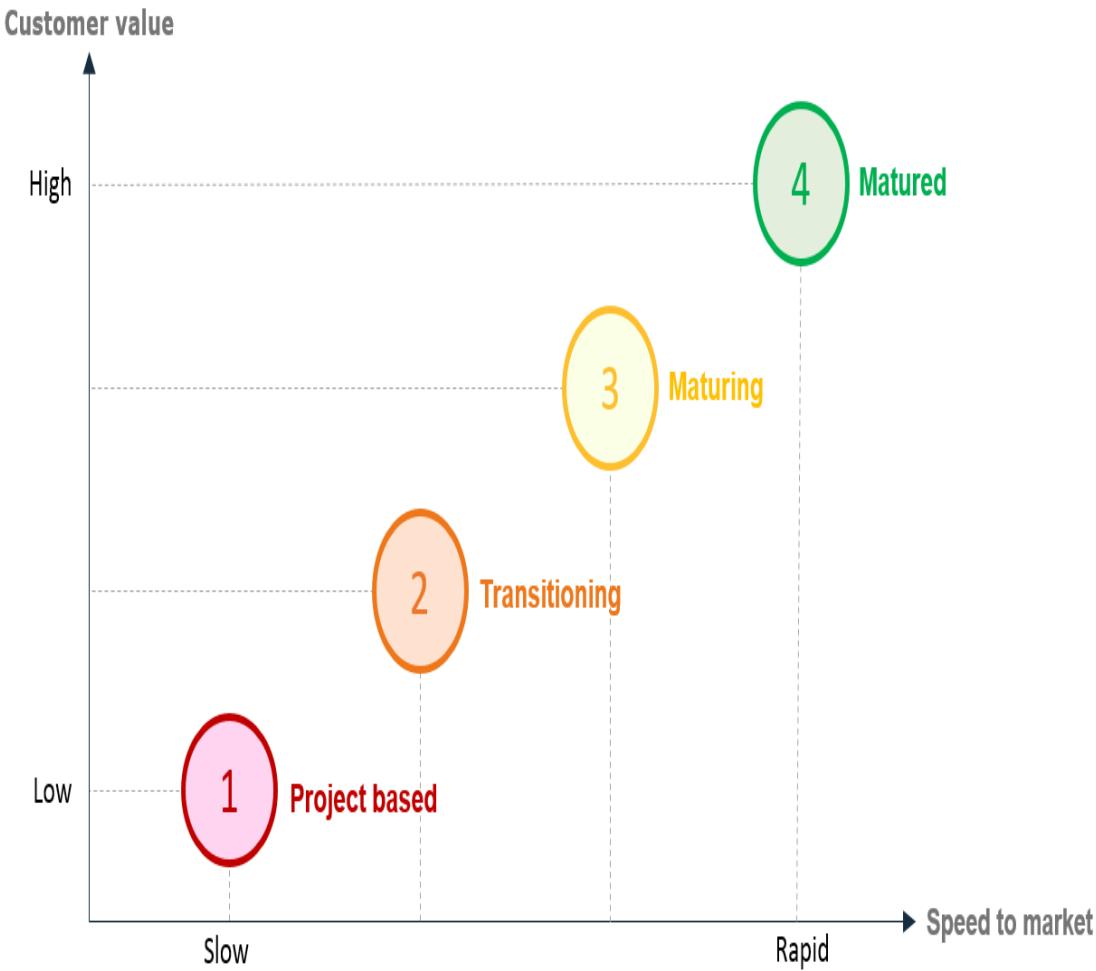


Figure 8.13: The transition model

The model consists of four levels:

- 1. Project based:** This is the starting point and represents an organization that is fully project-centric, meaning development and operation teams are not just different but part of completely different organizations. At this level, teams are organized based on technologies as opposed to the business products that the implemented technologies actually enable. Communication silos are commonly in place.

Deployments to production are far from frequent and require a lot of planning and coordination.

2. **Transitioning:** Having recognized that organizational change is required in order to drive better products and faster, the journey begins by slowly reorganizing teams around the business products they deliver, as opposed to just based on technologies. At this level, both development and support functions become the responsibility of the product team and, therefore, the role of the product owner is created as the accountable party for the success of the product. At this level, investments are made in automation and self-service capabilities so teams can independently and programmatically drive the life cycle of their products. At this stage, deployments to production should start to occur more frequently.
3. **Maturing:** As the organizational transition continues, more and more teams are organized around products and, thus, are solely responsible for the full cycle of their individual products. A new culture also starts to emerge whereby communication and interaction across teams is more agile and objective. Less time is spent on bureaucracy and more on addressing needs to drive value. At this point, deployments should be more frequent and driven almost entirely through CICD pipelines that, with little human interaction, take care of things such as regression, quality testing, packaging, and deployment.

4. Matured: At this level, all teams are product-centric, meaning that development and operation teams operate as one. As the entire product life cycle is automated through CICD pipelines, introducing new features is quicker and easier, thus enabling the product to truly differentiate itself from its competition. Accountability is also clear, along with the communication structures in teams.

Summary

Throughout the sections of this chapter, it was explained what it really means to treat APIs as business products. From a simple definition of what products in the real world actually are, to concepts such as the customer value hierarchy and Conway's law, the chapter described in practical terms the actual organizational implications of treating APIs as true business products.

The chapter, however, wasn't just about concepts: it walked through an illustrative and representative approach to delivering a TOM that is suitable for large enterprises wishing to build API products. To this end, the chapter exemplified different organizational models, each with their pros and cons, but ultimately focusing on a platform-based model, favored because of its potential to drive the most customer and business value, while being agile and scalable.

We then looked at the different elements expected of a TOM fitted to a platform model, such as roles, responsibilities, communication, and collaboration structures across different teams.

The chapter concluded by describing a sample transition model suited for driving organizational change through a series of

steps, each aimed at getting the organization a level closer to the set target.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Implementing Oracle Integration Cloud Service

Phil Wilkins, Robert van Molken

ISBN: 978-1-78646-072-1

- Use ICS to integrate different systems together without needing to be a developer
- Gain understanding of what a number of technologies and standards provide – without needing to understand the fine details of those standards and technologies
- Understand the use of connectors that Oracle provide from technology based connections such as file and database connections to SaaS solutions ranging from Salesforce to Twitter

- Enrich data and extend SaaS integration to route to different instances
- Utilize a number of tools to help develop and check that your integrations work before connecting to live systems
- Introduce and explain integration concepts so that the integrations created are maintainable and sustainable for the longer term
- Provide details on how to keep up to date with the features that Oracle and partners provide in the future
- Get special connections developed to work with ICS



Implementing Oracle API Platform Cloud Service

Phil Wilkins, Andrew Bell, Sander Rensen, Luis Weir

ISBN: 978-1-78847-865-6

- Get an overview of the Oracle API Cloud Service Platform
- See typical use cases of the Oracle API Cloud Service Platform
- Design your own APIs using Apiary
- Build and run microservices
- Set up API gateways with the new API platform from Oracle
- Customize developer portals
- Configuration management
- Implement Oauth 2.0 policies
- Implement custom policies
- Get a policy SDK overview
- Transition from Oracle API Management 12c to the new Oracle API platform

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!