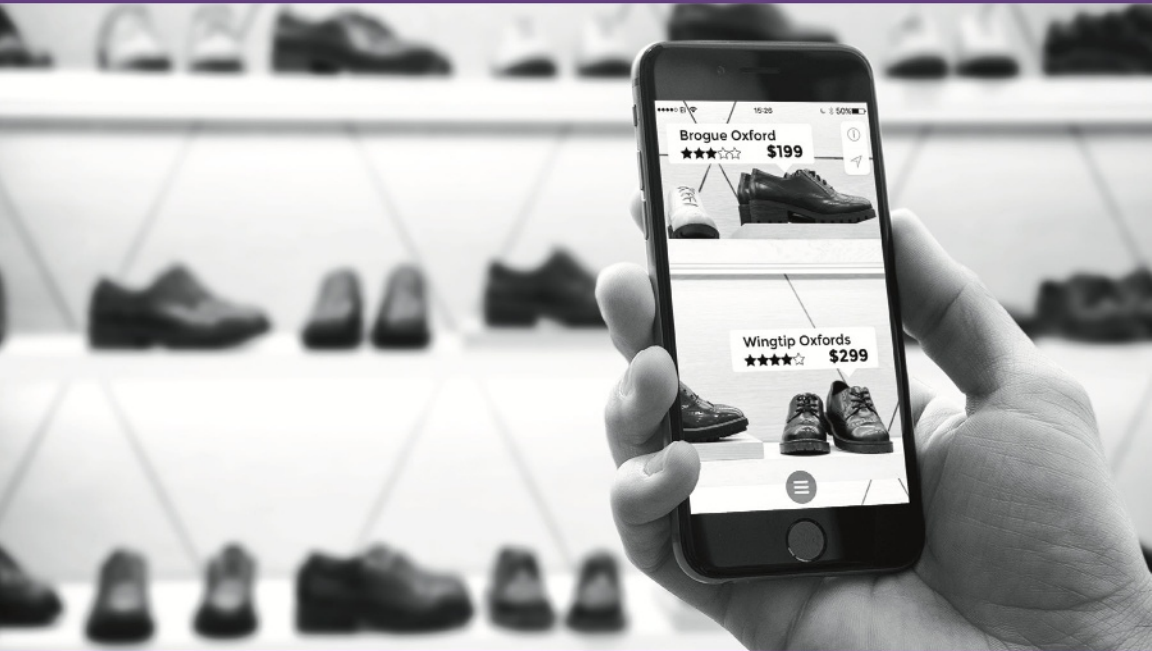
O'REILLY®

Compliments of commercetools
Next generation commerce

APIs for Modern Commerce

Enable Rich Customer Experiences Everywhere



Kelly Goetsch

commercetools
Next generation commerce

The commerce platform built for digital transformation

Quickly deliver new features to market

Rapidly iterate to improve customer experience

Improve developer engagement

commercetools offers the industry's first API and cloud-based solution for commerce which is built for use alongside microservices. Use all of our APIs or just the ones you need. Get the tools for next generation digital commerce with *commercetools* and microservices.

Learn more at: **commercetools.com**

APIs for Modern Commerce

Enable Rich Customer Experiences Everywhere

Kelly Goetsch

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

APIs for Modern Commerce
by Kelly Goetsch

Copyright © 2018 O'Reilly Media. All rights reserved.

Printed in the United States of America.

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

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

Editor: Brian Foster
Production Editor: Justin Billing
Copyeditor: Gillian McGarvey
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2017: First Edition

Revision History for the First Edition
2017-11-02: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *APIs for Modern Commerce*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-99523-5

[LSI]

Table of Contents

Foreword. . . . . **v**

Acknowledgments. . . . . **ix**

1. The API Economy. **1**
What Is an API? 2
Digitizing the World 4
APIs Are the Currency of Commerce 6
Final Thoughts 9

2. Modeling APIs. **11**
The Case for REST 13
Serialization Frameworks 14
API Modeling Best Practices 16
Final Thoughts 25

3. Implementing APIs. **27**
Identifying Needs of Clients 27
Applications Backing APIs 27
Handling Changes to APIs 29
Testing APIs 34
Securing APIs 38
Using an API Proxy 42
Exposing APIs Using GraphQL 44
Final Thoughts 47

4. **Consuming APIs** 49
Identify Clients 49
API Calling Best Practices 53
Final Thoughts 57

5. **Extending APIs** 59
Extending Traditional Enterprise Commerce Platforms 60
Approaches to Extending APIs 61
Final Thoughts 67

Foreword

We live in a connected world where virtually every aspect of our lives is choreographed by technology. Our mobile devices interface with the Internet of Things to monitor our health and provide real-time weather updates, which in turn connects us with friends to compete against during morning runs or share information with through crowdsourced weather networks. Businesses then access this information to deliver more relevant offers, such as the perfect pair of running shoes based on wear and weather conditions. Finally, companies may share this information across their ecosystems to better inform supply chain decisions.

These interactions are enabled by application programming interfaces (APIs), which have become the fabric of modern communication and a business currency so powerful they are reshaping the business world. Fundamentally, an API is an interface. In the same way that most applications have user interfaces to support human interactions, APIs are the interface that applications expose to facilitate interactions with other applications. APIs have existed since the first computer programs were developed, but the original APIs were rigid and required strict adherence to proprietary programming structures. In the early 2000s, web-connected APIs, including Amazon's Store API, eBay, and Salesforce, transformed the landscape and created a new network of open web APIs that anyone could consume. Since then, APIs have evolved from rigid interfaces to flexible and declarative platforms, establishing the modern bedrock of application development and integration.

At Adobe, we believe that the key to thriving in today's hyper-competitive landscape is being agile and differentiating through world-class experiences. As business velocity accelerates, the forces of creative destruction, disruptive innovation, and continuous change are reshaping every industry. APIs allow you to respond to these forces with agility, enabling brands to connect legacy systems and modern web applications with customers, partners, business ecosystems, and the internet in meaningful ways that unlock new value. They also accelerate innovation, making it easier to deliver new capabilities, amplify them by tapping into a network of complementary services, and expose those capabilities as omnichannel services.

As customers embrace an increasing number of omnichannel and mobile technologies, customer journeys are fractured into hundreds of real-time, intent-driven micro-moments. Each moment represents an opportunity to contextually engage customers and solve their problems as they move through their shopping journey. APIs provide the foundation for supporting these experiences, enabling customers to seamlessly access real-time information such as local store inventory, personalized offers, and updates on service requests. Customers are now able to enjoy frictionless experiences that connect with them in the moment, personally and contextually, rather than forcing them to interact based on how backend systems and processes are designed. The API-first approach provides the foundation for the experience-led business wave, with over 89% of companies expecting to compete on the basis of customer experience. This establishes deeper customer relationships, improves business performance, and establishes a more durable strategic advantage.

APIs for Modern Commerce provides the foundation you need to take action. The book will introduce you to the power of web APIs and will provide a framework for creating easily consumable APIs. Mr. Goetsch guides you through each phase of the process. Starting with modeling APIs, you will learn how to define and model stateless, easy-to-call APIs that are easy to integrate and extend. The building and deploying chapters cover best practices for how to build APIs and manage them through their life cycle. Finally, the APIs are consumed and extended to create an agile operating model.

This book will provide you with a clear understanding of what it takes to design, deploy, and extend your commerce environment with an API-first approach. Through the use of real-world examples

and proven best practices, you will learn both the technical and business principles necessary to embark upon your own API transformation.

— Errol Denger, Adobe,
Director of Commerce Strategy
October 8, 2017

Acknowledgments

Thank you to Tim Aiken, Leho Nigul, Drew Lau, Tony Moores, Eric Halvorsen, Rohit Mishra, and Matthias Köster for reviewing my manuscript and challenging me to write better. I sincerely appreciate the time, energy and professionalism they all brought to reviewing my work. Also, thank you to Dirk Hörig, Andreas Rudl, and the rest of the team at commercetools for giving me the encouragement and space to write.

CHAPTER 1
The API Economy

> The world can be seen as only connections, nothing else. We think of a dictionary as the repository of meaning, but it defines words only in terms of other words. I liked the idea that a piece of information is really defined only by what it's related to, and how it's related. There really is little else to meaning. The structure is everything. There are billions of neurons in our brains, but what are neurons? Just cells. The brain has no knowledge until connections are made between neurons. All that we know, all that we are, comes from the way our neurons are connected.
>
> —Tim Berners-Lee, *Weaving the Web (Harper Business)*

APIs facilitate today's digital revolution, similar to how steam-powered engines enabled the Industrial Revolution in the 1700s. When you withdraw money from an ATM, check the weather, or buy a new pair of shoes, you're using hundreds of APIs behind the scenes.

APIs are transformational because they allow for an organization's functionality and data to be exposed to third parties. When those third parties discover and consume that functionality and data, they often add more value than the sum of the APIs they consume. APIs democratize access to functionality and data, similar to and building on many principles of the World Wide Web.

> **NOTE**
>
> Metcalfe's law says that the value of a network is proportional to the square of the number of its users. The classic application of this law is to fax machines. A single fax machine in a network offers no value, but two fax machines facilitate communication between two individuals, three fax machines facilitate communication between nine individuals, and so on.
>
> Metcafe's law can also be applied to APIs over the internet. Individuals building world-changing applications can do so faster and with less cost than ever before because of the availability of these fundamental building blocks. When Facebook acquired WhatsApp for $18 billion, WhatsApp only had 55 employees. Instagram only had 13 employees when it was acquired for $1 billion.
>
> Modern businesses are able to build upon an enormous network of functionality and data that is often exposed over APIs. In the past, businesses of all sizes had to have enormous staffs of people to do what can now be done using an API for a few pennies per million calls. It's this compounding of innovations that adds exponential value to our economy.

APIs are especially important to commerce. Consumers of all types expect to shop where, when, and how they want, across any device. Every day, a new device that's capable of facilitating a commerce transaction enters the market, whether it's a dishwasher that's capable of ordering its own detergent or a wearable fitness tracker that reorders your favorite pair of running shoes when they're worn out. What underpins all these devices is their ability to consume commerce functionality and data over an API.

Before we get too far, let's look at what an API actually is.

What Is an API?

API is an acronym for *application programming interface*. An API is simply an interface on top of an application or library that allows callers to execute functionality (e.g., calculate sales tax, query inventory, and add to shopping cart) or create/read/update/delete data (products, orders, prices, etc.). Think of it as a contract between a provider and a consumer.

NOTE APIs have been with us since the beginning of software. This book on web APIs, where the caller of the API is decoupled from the producer and the transaction often occurs over a network boundary like the internet.

An API is conceptually similar to a legal contract, where two parties outline obligations to each other without going into too much detail about the implementation. For example, a contract between a steel manufacturer and a buyer calls for 100 tons of SAE grade 304L to be delivered in 60 days but doesn't specify which mine the iron must be extracted from or what temperature the furnace must be set to. So long as it's SAE grade 304L steel and it's delivered within 60 days, the obligations of the contract are met.

It's the exact same with APIs: specify the *what* but not the *how*. Providers of functionality or data document what they're offering in great detail but do not describe how the functionality or data they're exposing is produced. Additionally, the provider can offer promises pertaining to the uptime of the API, as well as response times, authentication and authorization schemes, pricing (often a charge per *x* number of API calls), and limits on how often the API can be called. If the consumer agrees to the terms offered by the producer and has the proper permissions from the provider, they're free to call the API.

Without an API, you'd be left to directly query databases and perform other tricks that expose the caller to too many of the implementation details of the application you're calling. An API abstracts away all of those details.

> **NOTE**
>
> Because there's often ambiguity, it's important to differentiate between APIs and microservices. Think of microservices as relating to *how* the application is built below the API. *Microservices* are individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines. Microservices always expose functionality and data through APIs. APIs are always required for microservices, but microservices are not required for APIs.
>
> APIs can be bolted on top of any application, whether on the microservice or monolithic end of the spectrum. For more about microservices, have a look at *Microservices for Modern Commerce*, by Kelly Goetsch (O'Reilly).
>
> We'll discuss this further in Chapter 3.

Now that we've defined APIs, let's talk about why they are so relevant to commerce today.

Digitizing the World

With software powering the digital revolution we're living through, APIs have emerged as the foundational currency. APIs are great for many different constituencies.

For end consumers, APIs allow for functionality and data to be consumed from any device, anywhere. Gone are the days when the only way to interact with an organization was through a browser-based website on a desktop. Consumers are now fully immersed in mobile devices, tablets, voice, wearables, and even "smart" devices like internet-connected refrigerators. They're accessing functionality and data through third parties like social media and messaging platforms, native apps accessed through proprietary app stores, and native clients like those found in cars and appliances.

For organizations building consumer-facing experiences, it's easier to consume functionality and data as a service over an API rather than downloading, installing, configuring, running, and maintaining large stacks of software and hardware. It makes a lot more sense to pay a vendor to run multitenant copies of the software for you at scale. You just get an API that you can code to, with the functionality delivered as a service. No need to install anything.

Software vendors like exposing their functionality and data as APIs because it allows outside developers to wire functionality and data into their applications in pieces. Rather than a large one-size-fits-all software package that must be entirely adopted and then customized, an API-based approach allows for developers to consume smaller, more granular pieces of functionality from specialized "best of breed" vendors. This strategy is how the public cloud vendors have changed the face of IT. As an example, have a look at the dozens of discrete services (all exposed as APIs) that you see when you log in to Amazon Web Services (Figure 1-1).

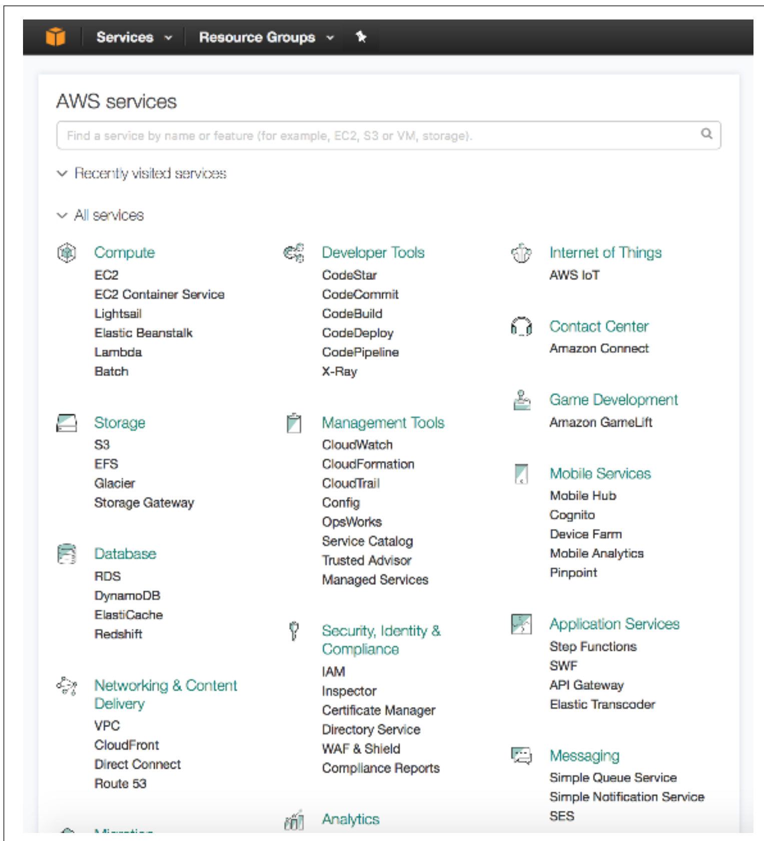


Figure 1-1. Amazon Web Services console

Some customers only use AWS for DNS. Some use it just for storage. It's entirely up to the customer. This is increasingly how organiza-

tions and individual developers want to buy software. And it makes so much sense.

Perhaps the most important advantage of APIs is time to market for all parties involved, which is arguably the most important competitive differentiator in today's digital revolution. Developers can simply consume little building blocks of functionality, similar to the Amazon Web Services model. Organizations don't have to download, install, configure, run, or maintain anything. Software vendors can release new functionality to APIs many times a day, provided each API is backed by a separate microservice with its own team, application, infrastructure, and release cycle. Unless the change was big enough that it necessitated a breaking change to the API, the consumer of the API can start using the new functionality immediately without having to do anything.

APIs Are the Currency of Commerce

The fundamentals of retail (whether B2C or B2B) remain unchanged: get the right product to the right person at the right time for the right price. How this occurs in today's technology-driven retail environment is completely different than ever before. For example, a 2016 study by Harvard Business School showed that 73% of consumers used multiple channels during their shopping journeys.

Technology and habits are quickly changing on the consumer side (the "C" in B2C), with the clear trend being toward mediation through consumer-focused platforms and electronics. This started with B2C commerce, but brands and B2B commerce are seeing these changes as well. Most consumers now engage with businesses through a handful of social media apps on consumer electronic devices, like smartphones. Even many of today's B2B transactions are now occurring over mobile devices. With all this change on the consumer side, the notion of what constitutes a channel has fundamentally changed as well.

Defining a Channel

A channel used to be a discrete physical or virtual interaction with a customer, such as physical stores, websites, mobile applications, and kiosks. It was clear when your customer engaged with your brand. If your customer opened up your app on their smartphone, they were

engaging with you directly through your mobile channel, with no other parties involved.

Things are different today. You now have to reach customers using dozens of smaller touchpoints. Social media (Instagram, Facebook, Pinterest, Twitter, Snapchat, YouTube, etc.), wearable devices, marketplaces (e.g., eBay, Amazon, and physical malls, which are beginning to have shoppable applications), smaller embedded "micro" applications on mobile devices (iMessage apps, etc.), and so on are now the primary means of interacting with retailers and brands. You have to be part of your customers' everyday lives in order to remain relevant. You have to give them contextual content in the location of their choosing. Very few brands have enough value that a consumer will consume an entire channel, like a mobile application.

To further complicate matters, there's a revolution in consumer electronics that's allowing customers to interface with devices in ways that were previously never possible. Voice, for example, is rising in prominence. Apple's Siri handles over two billion commands a week. An incredible 20% of Google searches on Android-powered handsets in the US are input by voice. Amazon Echo' was one of 2016's hottest holiday gifts. Commerce is now as simple as saying, "Alexa, buy me some AAA batteries."

Beginning in 2007, consumer engagement started to be mediated by new gatekeepers. Prior to the iPhone being introduced in 2007, consumers more or less directly interfaced with your website. Perhaps as important as the iPhone itself was the concept of the app store, where customers could download trusted apps from a walled garden. By setting standards for, reviewing, and approving apps, Apple became a mediator between you and your customer. Android adopted the same model. Social media's rise to prominence in 2010 brought even more mediation. In 2017, you have to reach customers where they are, on their own terms and through the platform of their choice. Experiences are now more and more mediated.

Mediated experiences require the use of APIs, which is why APIs are now the primary currency of commerce. Rather than build a single website or mobile app with its own stack, you should instead offer a suite of dozens, hundreds, or even thousands of discrete features as APIs. Some of those APIs can be built in-house, some can be outsourced to a partner, and some can be purchased from third parties. What matters is that the APIs are available and easily consumable

from anywhere. Frontends can then consume those APIs to build compelling experiences for customers, without any synchronization required. Each piece of functionality (add to cart, claim coupon, etc.) or piece of data (product, category, customer, etc.) is accessed only through one API, rather than having to access multiple backend systems (ERP, CRM, OMS, etc.), as is often the case for enterprises.

For example, Best Buy is openly courting developers to integrate its APIs (Figure 1-2).

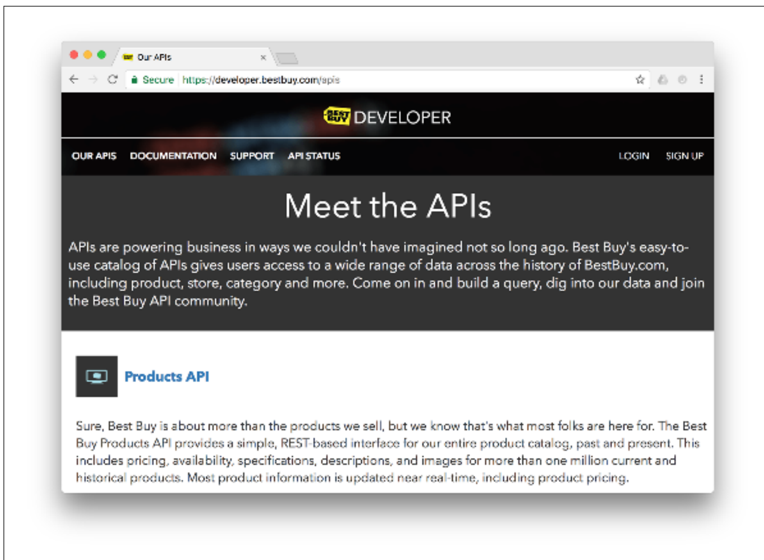


Figure 1-2. Best Buy's public API catalog

Additionally, Amazon.com, Walmart, Tesco, eBay, Target, and most of the world's other leading retailers are on board with offering functionality and data as APIs. Developers are then free to integrate the APIs wherever they see fit, often earning a commission on sales.

Jeff Bezos famously directed his staff at Amazon to adopt an API-first approach in a 2002 memo, which went something along these lines:

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.

3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—it doesn't matter.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design so that interfaces that can be exposed to developers in the outside world. No exceptions.

Today, Amazon famously has thousands of APIs that it uses to "inject" Amazon.com into thousands of different social media and consumer electronics clients.

Gartner, the research and advisory company, states the following in its Hype Cycle for Digital Commerce, 2017 report:

> API-based commerce will be critical for the future of *commerce that comes to you,* whereby commerce functions occur in the customer's context wherever and using whatever channels are most convenient to them. Commerce journeys will become more fragmented and an API-based approach is a fundamental enabler for cross-channel experiences.
>
> API-based commerce will need to be available as a set of discrete services that can be utilized independently (with an appropriate cost model), and these capabilities should no longer require a *whole platform* purchase or subscription.
>
> —Gartner, *Hype Cycle for Digital Commerce 2017, Mike Lowndes, July 2017*

Today's commerce platforms are expected to be API-first because APIs are the only way of injecting commerce into mediated experiences. APIs are faster to integrate, offer more flexibility, and—if implemented using a microservices approach—support all of your future commerce initiatives.

Final Thoughts

Now that we've discussed APIs and how transformational they are, let's look at how to model them.

CHAPTER 2
Modeling APIs

When building applications, start by first modeling your APIs. Then write the application(s) to back those APIs. Modeling APIs involves selecting the representation format (typically JSON or XML), defining the various resources (objects like `/Product`, `/StoreLocator`, etc.), modeling each resource's attributes (e.g., key/value pairs like `productId="12345"`), and finally modeling relationships to other resources (e.g., `<link rel = "customer" uri = "/Customer/c12345" />`) It's similar to defining an entity relationship diagram (ERD) for your database before writing a monolithic application.

When you model APIs first, you'll find that it's easier to write the application. Often, individual resources (e.g., `/Product`) map back neatly to individual microservices. If you start by writing your application and then retroactively expose functionality and data through APIs, you'll end up with APIs that mirror your application's idiosyncrasies rather than a well-thought-out API that is easy for developers to consume.

NOTE

How granular should your APIs be? Check out Eric Evans' iconic 2003 book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional). In it, he makes the case for a pattern called Bounded Contexts whereby APIs and the underlying applications should be modeled as closely as possible to mirror the data and behavior of your business domain. For example, Eric would call for separate product and pricing APIs, as the two are distinct business domains, even though the two have a direct relationship to each other and could be modeled as one API.

A perpetual issue in software development is parallelizing development. If you get all of the stakeholders in a room and have them centrally plan the APIs, you can then parcel out the development of the APIs to internal teams and systems integrators. A clearly documented API is easier for a team to implement when compared to an application whose API is unknown.

Your end goal should be to have an enterprise-wide catalog of APIs that anyone can consume, inside or outside your organization (Figure 2-1). A single API could be used by dozens, hundreds, or even thousands of different clients. Your API *is* your product.

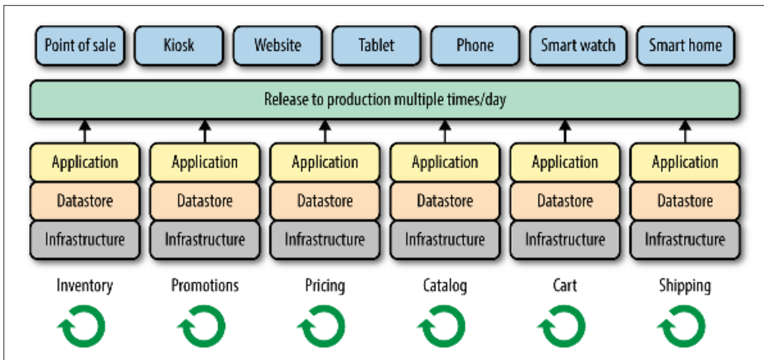


Figure 2-1. A vision of the future: a catalog of independently developed and consumed APIs

This catalog of APIs can then be handed to a systems integrator or creative agency to build a new experience for the latest consumer electronic device, for example.

Now that we've discussed why it's important to model your APIs first, let's step back a little and discuss REST.

The Case for REST

REST is assumed to be the default style because of its universality, flexibility, large supporting ecosystem of technology, and friendliness to both producers and consumers of APIs.

Why? Fundamentally, REST APIs are analogous to the web itself. REST was defined by Roy Fielding (one of the principal authors of the HTTP specification) in his 2000 PhD dissertation, "Architectural Styles and the Design of Network-Based Software Architectures," at UC Irvine. In it, he stated:

> Representational State Transfer (REST) is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.

What Roy describes could easily be equated to the World Wide Web Tim Berners-Lee conceived of in the early 1990s.

Let's clarify some terms before we go any further:

Resource
: Is an entity that can be interacted with using HTTP verbs, like `GET`, `POST`, `DELETE`, etc. It can be singular (`/Product/{`*`id`*`}`) or plural (`/Products`). Roy describes it in his dissertation as "the intended conceptual target of a hypertext reference."

Resource identifier
: A URL used to access the resource. *https://api.<yourcompany>.com/Product* or `/Product` would be the resource identifiers.

Representation
: The format of the data returned when an API is called. Often, it's XML or JSON.

Resource operation
: Maps back to HTTP verbs, like `GET`, `POST`, `DELETE`, etc.

Client
: Refers to whoever is calling the API, whether it's another enterprise application or a mobile application belonging to the end consumer.

The web has proven to be an extremely successful model for APIs to follow. It's only natural for APIs to adopt the principles from the web that work and to build on top of its infrastructure (HTTP, TCP, XML/JSON, etc.). The use of HTTP specifically is crucial because our digital world is built on top of it. Content delivery networks, web application firewalls, API gateways, and authorization and authentication frameworks all rely on the HTTP infrastructure. There's a prescribed set of verbs (`GET`, `DELETE`, etc.) for dealing with resources. As a principal author of the HTTP specification, REST was a natural next step for Roy.

> **NOTE**
>
> Which format is preferred—JSON or XML?
>
> JSON is more compact but also more difficult to read, especially when data is complex and hierarchical. XML is best for structured data, but it's more verbose than JSON. All modern tooling will work with JSON and XML. The XML ecosystem tends to be richer but is suffering from a lack of investment as JSON becomes dominant.
>
> Either will work just fine. Don't get pulled into endless debates. Use whichever you feel comfortable with and what works best for your organization. What matters is that you pick a format and use it consistently across your organization.

Serialization Frameworks

While REST is the default, its primary drawback is performance. The HTTP stack is well known for not being efficient. The documents must be parsed in their entirety and sometimes validated before data they contain can be accessed. Field names are stored in the documents. There are many "filler" characters, such as tabs, spaces, and special characters such as < and }. In short, REST is universally understood by humans and easy to work with, but the performance is sometimes lacking.

There are situations where high performance is an absolute requirement, including:

- Special use cases, like high-frequency trading
- Retrieval of a large amounts of data, like retrieving all orders placed in the past week
- Architecture that forces synchronous calls between microservices

It's important to note that you should rarely if ever make synchronous HTTP calls between microservices, especially if the client is waiting on the response. However, there are situations when synchronous calls are necessary. For example, your shopping cart may need to make a real-time call to validate inventory for a given product before checkout is initiated. In that case, your shopping cart now needs to call your inventory synchronously.

For the absolute best performance, you can use binary-level serialization frameworks. Examples include Apache Thrift, Apache Avro, and Google Protocol Buffers. These bypass the entire HTTP stack (including TCP), don't rely on text and therefore parsing, don't store field names, don't have extemporaneous characters, and don't need intermediary SDKs. These stacks are built for speed and they deliver on that promise. For example, Protocol Buffers from Google are 3 to 10 times smaller and 20 to 100 times faster than XML. Most of the time, the performance bottleneck is with the application you're calling as opposed to the overhead that REST introduces.

If you're a provider of APIs, you'll generally want to expose REST APIs publicly. These binary-level serialization frameworks offer very little support for security, readability, or a larger ecosystem, though these areas are improving over time. Binary is meant to be used when speed is your primary requirement.

> **NOTE**
> Google has estimated that 74% of all publicly accessible APIs are REST. Given its popularity, consider rest as the default when APIs are discussed henceforth.

API Modeling Best Practices

When modeling an API, look at who your consumers are. Are they internal or external? How technical are they? Who are your clients, and where are they making API calls from? These are all important considerations which we will cover next and in future chapters.

Here are some best practices that should be followed regardless of who your consumers and clients are.

Documented Using a Specification

All APIs require documentation. Documentation includes:

- Resources
- Attributes
- Relationships between resources
- Representation formats
- Supported verbs
- Error response codes

APIs also require an application that does what the API promises. They can optionally offer an SDK or some other form of client-side code to make it easier for clients to call the API. The problem for those building APIs is that the API documentation, server-side implementation and client-side SDKs must match at all times. If a "currency" attribute is added to your pricing resource, it must be available in the documentation, the server-side implementation, and any client-side SDKs.

There are a number of standards available that allow you to model your APIs using a high-level markup language like YAML. OpenAPI (formerly known as Swagger) and RAML have emerged as the two most commonly used standards for documenting REST APIs (Figure 2-2).

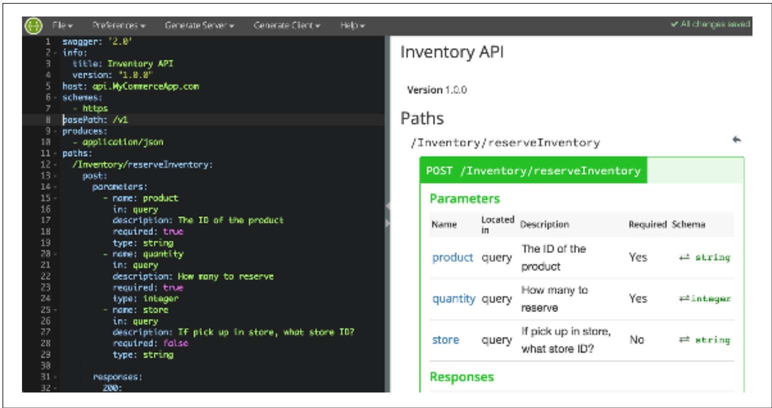


Figure 2-2. Inventory API modeled using OpenAPI

Once APIs are modeled using the specification, you can then automatically generate client and server stubs in the language of your choice (Figure 2-3).

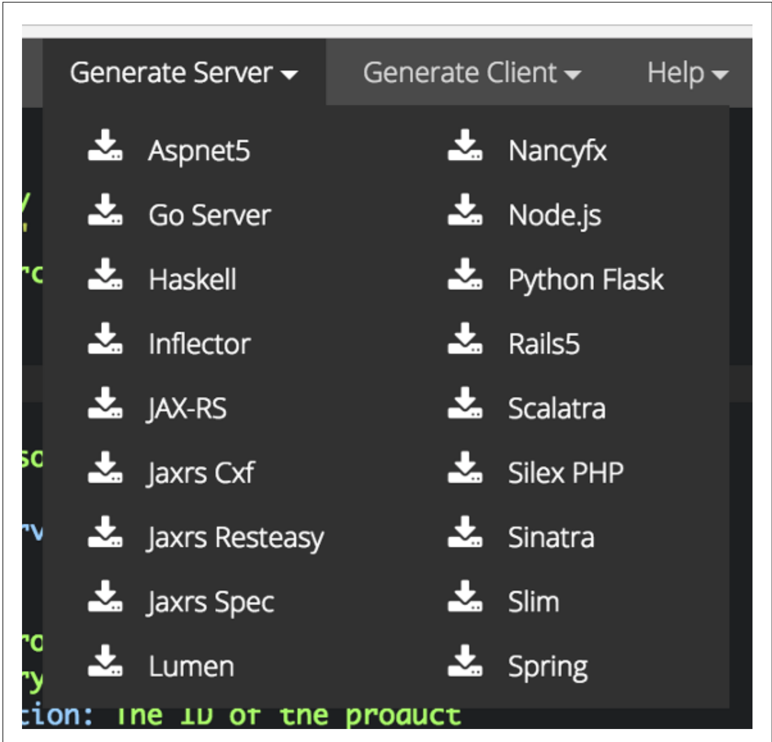


Figure 2-3. Client and server auto-generation

Being able to auto-generate documentation, server stubs, and client stubs dramatically simplifies the development of APIs because all three pieces are generated simultaneously from the same high-level markup language. These definitions should be checked in and treated as source code.

Though the technology and architecture backing each API can change, you should pick a specification and use it across your company. What matters most is that you use it consistently.

Independently Callable

APIs should be *independently callable*, meaning you shouldn't have to call one API before you call another. For example, many 2000s-era commerce APIs required that clients authenticate and authorize a customer (often as two separate calls) before an action was performed against that user's account, like adding an item to their shopping cart. While it does make sense that the client authenticate and authorize the user, requiring one or two API calls as a precondition doesn't make a lot of sense. It requires putting too much intelligence into the client. Imagine having to maintain that business logic across 10 different clients managed by 10 different development teams.

These hard dependencies between APIs are often introduced due to idiosyncrasies in the monolithic application because APIs were bolted on after the application was built. That's why it's best to start by modeling your APIs first, and then writing individual microservices to back each API.

Stateless

The applications backing each API should be *stateless*, meaning that no single client request is dependent on the server-side state of a previous request. This allows a call to the API to be served by any instance of the application. You could implement a round-robin load balancing strategy, for example.

As an example of why it's best to be stateless, consider that many top celebrities have more than 100 million followers on social media platforms. Individuals with large followings regularly pitch products to their followers (see Figure 2-4).

Figure 2-4. Celebrity-driven product endorsements

If someone with 100 million followers posts a link to your product detail page, your auto-scaling should kick in and create a few thousand instances of the applications that back your product, inventory, pricing, and other microservices used to render that page. If you have stateful sessions, you'll have to slowly wait for your application instances to be free of sessions, which could take many hours. If your instances are stateless, your auto-scaling mechanism can simply kill off the unnecessary instances following the rush.

Easy to Call

APIs should be written so that they are easily callable by any client. This primarily means modeling the APIs at the outset, offering SDKs in a variety of languages, and making the APIs as performant as possible. Let's look at all three.

First, as we've discussed, ensure that the APIs are modeled first, before any code is written. Then write your application to back the API you've modeled, preferably with a 1:1 relationship between the API and a backing application/microservice. This will help to ensure that your APIs are intuitively modeled and free of the idiosyncrasies of the implementation of your application. As we've also discussed, it's best to use a formal specification like OpenAPI or RAML

(Figure 2-5). This ensures that your clients can leverage the specification's large ecosystem of tooling.

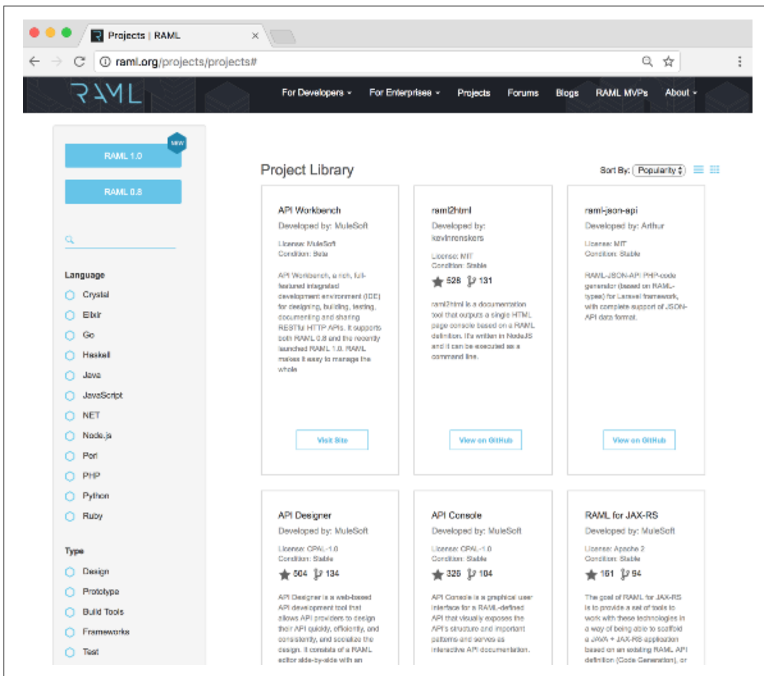


Figure 2-5. RAML ecosystem

But you can only leverage all that tooling if your API conforms to a specification.

Next, offer SDKs to make it easier for clients to access your APIs. While REST is a necessary common denominator, it's not very easy or performant to work with large JSON or XML documents.

For example, here's how you'd retrieve the products from the product service using raw JSON:

```
URL url = new URL("https://api.yourcompany.com/Product");
 try (InputStream is = url.openStream();
      JsonReader rdr = Json.createReader(is)) {

     JsonObject obj = rdr.readObject();
     JsonArray results = obj.getJsonArray("products");
     for (JsonObject result : results.getValuesAs(
       JsonObject.class)) {
         System.out.println("Product Name="
           + result.getJsonObject("product").getString("name"));
```

```
                ....
            }
        }
```

There are a few issues with this:

- There's no type safety. If your developer accidentally typed "prodducts" instead of "products," the compiler wouldn't flag the error.
- Developers aren't able to use auto-complete features in modern IDEs.
- You have to manually handle authentication and authorization if that's required. That code can get complicated.

Rather than interacting with XML or JSON directly, you can use the tooling of the specification you chose to generate a client. It takes two mouse clicks from the API modeling UI to generate a standalone library (*product-service-java-client-1.0.0.jar* in this example; see Figure 2-6).

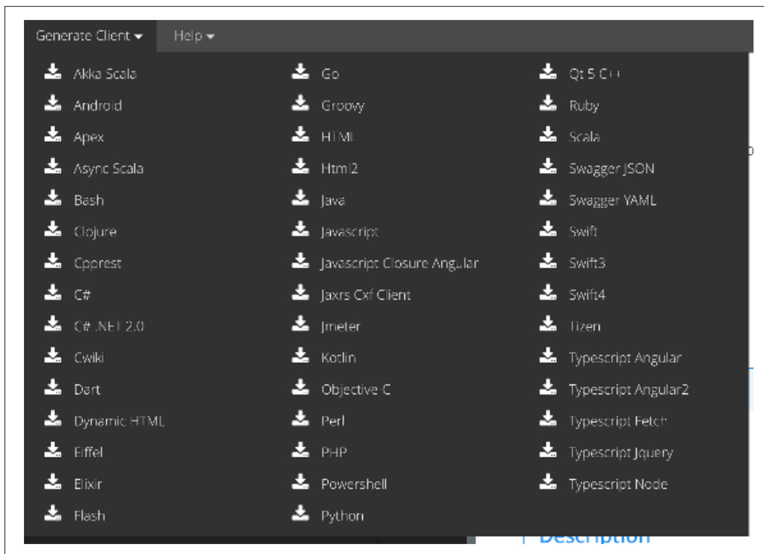


Figure 2-6. SDK generation using OpenAPI tooling

Simply import the library and interact with resources in your native language, rather than text-based documents:

```
for (Product product : new DefaultApi().productGetAll())
{
        System.out.println("Product Name=" + product.getName());
}
```

Developers love native libraries because they simplify development through the use of auto-complete in IDEs (see Figure 2-7).

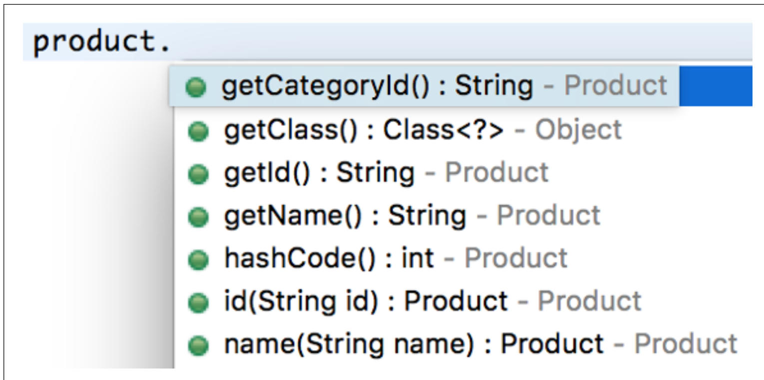


Figure 2-7. Auto-complete in IDE through the use of an SDK

Beyond auto-generating clients from a specification, an SDK can abstract away all the security for developers.

Consider offering client-side libraries for a variety of popular languages. As a provider, you should furnish all the client-side tooling that you can in order to entice developers to consume your APIs.

Finally, don't forget about performance. The worst thing you can do is provide an API with response times in the hundreds of milliseconds. It puts a big burden on client-side developers who then have to code around the poor performance.

Cacheable

To improve performance, it's important to aggressively cache API calls. The vast majority of calls from your frontend translate to HTTP GET requests, whether you use REST directly or indirectly through a native SDK. An HTTP GET call is easily cacheable by any intermediate layer between your frontend and your backend.

To cache properly, you'll need to represent resources by proper URIs. An HTTP request for */Cart/{id}* is very easily cacheable but an HTTP GET to */Cart/current* or some other ambiguous URI is not.

Similarly, it's also more difficult to cache when resources aren't defined independently. If you're HTTP `POST`ing to a */ProductCatalog* URI to retrieve products, you'll have a harder time than if you're referencing individual resources by URI, like */ProductCatalog/Product/12345*.

As individual resources, you have more control over caching policy. You could easily define a `"max-age=180"` HTTP request header for products and `"max-age=5"` for inventory, for example.

You'll also be able to cache more if you make full use of HTTP verbs, like `GET`, `POST`, `DELETE`, etc. Many developers take shortcuts and route all API calls over HTTP `POST`. When this is done, HTTP is used more as a tunneling mechanism, which reduces cacheability. It's far easier to cache `GET`s because you know they're read-only and therefore cacheable. A `POST` is hard to cache because you don't know if it's creating, reading, updating, or deleting data, or executing some functionality remotely.

Once you have a well-defined API, it's easy to cache through the use of standard HTTP-level caching techniques. Google and many others have great guides showing how to configure HTTP caching.

As for where to cache, you have a lot of freedom based on your environment. You could cache at the edge (often part of your content delivery network), an API load balancer and/or API gateway, a standalone caching proxy, a web server, a reverse proxy, or any one of the intermediaries between the client and the originating application backing the API. The leading content delivery networks even auto-cache by default, as seen in Figure 2-8, an example of Amazon Web Service's CloudFront.

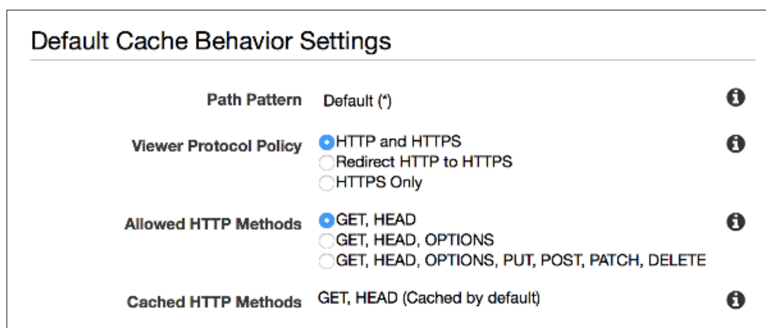


Figure 2-8. Auto-caching HTTP requests at the edge

The ability to have intelligent intermediary layers is one of the key advantages of using HTTP for your API calls rather than the binary-level protocols, which are almost entirely point to point.

Intuitive

The World Wide Web (linking through HTML) is the layer that made the internet a fixture in business and at home. That model succeeded because it made it easy for normal people to find information. A web page displaying a company's stock price also includes a link to view that company's balance sheet, for example. The value is in the connections.

Good API designers adopt many of the same principles as the web, often under the umbrella term *Hypermedia as the Engine of Application State* (HATEOAS). While arguably not the nicest sounding acronym, the principle is solid. HATEOAS means each response includes links to other related resources. Here's an example of an order object showing the caller how to delete the order:

```
<order id="{id}">
        ...
        <link rel = "delete" uri = "/Order/{id}/delete"/>
</order>
```

HATEOAS is like the web today with browsers, but applied to APIs instead. The value of this approach is as follows:

- It allows developers to much more easily consume functionality and data.
- Developers can code to the link named "delete," allowing the API paths to change.
- It eliminates hardcoded API paths and other business logic.
- It helps developers model APIs.

You can take HATEOAS further and also use it to control application state. For example, a cart could prevent the submission of an order by not having a `confirm` rel present when billing and payment details haven't been provided yet in the application state.

Try for HATEOAS as much as possible.

Idempotent

Idempotency means that an action (such as add to cart, increment inventory, or create product) can be performed multiple times without causing problems. For any number of reasons, an API may be invoked multiple times. Invoking an API multiple times should result in the same output every time.

For example, an API expecting the following input is not idempotent:

```
HTTP POST /Cart/{id}

{
        "add": {
                "skuId": "12345",
                "quantity": 1
        }
}
```

Every time this is invoked, one SKU will be added to the cart. If the API is accidentally executed 10 times, the customer will end up with 10 of SKU 12345 in their shopping cart.

The following input, on the other hand, would be idempotent:

```
{
        "add": {
                "skuId": "12345",
                "totalQuantity": "1"
        }
}
```

This could be executed 1,000 times and the result would still be the same: the customer would have only one SKU 12345 in their shopping cart.

It is best practice to assume that the plumbing between your client and your server is unreliable. APIs may be invoked multiple times when only one invocation was anticipated.

Final Thoughts

It should be clear why APIs should first be modeled before writing a single line of code. Don't get too dogmatic about which specification you use. What matters is that you use one and stick to it.

Next, let's explore the code-level implementation of the APIs.

CHAPTER 3
Implementing APIs

This chapter focuses on actually implementing what's behind your APIs. You've modeled your APIs, but now it's time to write the code. Here's how you get started.

Identifying Needs of Clients

Before you start writing code, remember who you're building APIs for and what they want. Are your APIs primarily for internal developers at your company? Or are they for external developers not employed by your company? Are your APIs for paying customers or for partners?

We'll discuss this more in Chapter 4, but for now, let's ask the simple question—who are your clients? Are they other microservices? Are they web frontends? Are they mobile frontends? Your client could be on the other side of the world or on the next rack over in the data center. These are all important considerations. Who's calling your APIs will have a direct impact on how you design, deploy, and manage them.

Keep these factors in mind as you look at building the code behind your APIs.

Applications Backing APIs

All large applications fall somewhere on a spectrum between monolithic and microservices. A traditional monolithic has multiple busi-

ness functions in the same codebase. Pricing, orders, inventory, and so on are all included in the same codebase and are deployed as a single large application requiring dozens or hundreds of people working in horizontal (frontend, backend, ops, etc.) teams. These monolithic applications often retroactively add APIs to access functionality contained within the monolith (see Figure 3-1).

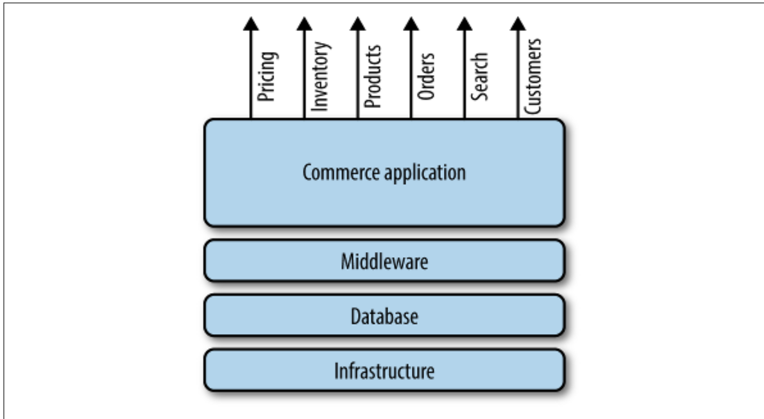


Figure 3-1. Traditional monolithic commerce application

Microservices are individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines. Characteristics of microservices include:

Single purpose
: Do one thing and do it well.

Encapsulation
: Each microservice owns its own data. Interaction with the world is through well-defined APIs (often, but not always, HTTP REST).

Ownership
: A single team of 2 to 15 (7, plus or minus 2, is the standard) people develop, deploy, and manage a single microservice through its life cycle.

Autonomy
: Each team is able to build and deploy its own microservice at any time for any reason, without having to coordinate with any-

one else. Each team also has a lot of freedom in making its own implementation decisions.

One of the key characteristics of microservices is encapsulation. Small, vertical microservice teams often start by modeling the API and then writing a microservice that implements it (see Figure 3-2).

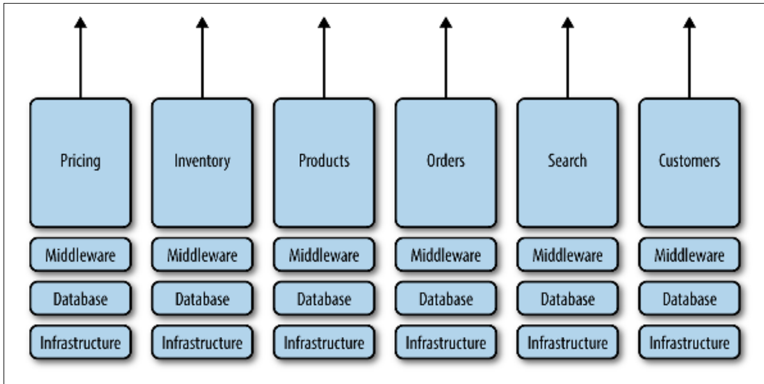


Figure 3-2. Microservices-based commerce application

The advantage of a microservices-based architecture in this context is that the APIs are more easily able to be consumed independently. The shopping cart microservice team, for example, needs to expose an API that can be called by anybody or anything, internally or externally. An API retroactively bolted on top of a monolithic application is inherently less callable because you're consuming a small piece of something much larger. There are always going to be dependencies. Going back to the shopping cart example, you may have to call inventory, pricing, and tax as you retrieve your shopping cart. But a shopping cart developed as a standalone microservice will already have those dependencies included. Microservices requires a substantially different approach to development that is ultimately very beneficial for APIs.

For more information about microservices, read *Microservices for Modern Commerce*, by Kelly Goetsch (O'Reilly).

Handling Changes to APIs

Traditionally, commerce applications have forced all clients to use the same API and implementation versions. In practice, this meant that the monthly or quarterly release to production would require

the clients to be updated at the same time, leading to a long weekend for the ops team. When the only client was a website, this was just fine. When it was mobile and web, it became more difficult because an update to the core platform meant you had to redeploy both mobile and web at the same time. But in today's omnichannel world, there could be dozens of clients, each with their own release cycles (see Figure 3-3). It is not possible to get dozens of clients to push new versions live at the same time. Each client must evolve independently, with its own release cycle.

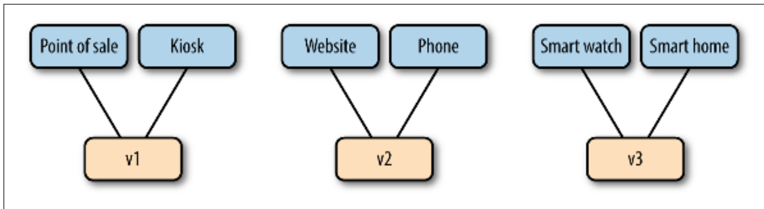


Figure 3-3. Multiple clients, each calling different versions

There are two basic approaches that producers of APIs can take: evolve or version. Let's start with evolving APIs.

Evolving APIs

Many APIs simply do not change that much, especially if they're designed by people who understand the domain extremely well. For example, most of the external tax calculators have static APIs. Have a quick look at Avalara's tax API as an example. The underlying tax rates and sometimes the formulas change, but the actual API you call is fairly static. The response you get back is also fairly static. The US could adopt a VAT-style tax system and the APIs still wouldn't change. Most APIs you interact with on a day-to-day basis are like this.

Inevitably, APIs need to evolve—but not necessarily change. Let's say you're building a customer profile API that allows simple create, read, update, and delete (CRUD) operations. The following code shows the object that the API would need for a new customer to be created.

```
{
  "id": "c12345",
  "firstName": "Kelly",
  "lastName": "Goetsch",
```

```
  "email": "kelly.goetsch@commercetools.com"
}
```

Now let's say that your business users want to capture your customers' shoe sizes so they can be targeted with better product offers. Your JSON object would now look like this:

```
{
  "id": "c12345",
  "firstName": "Kelly",
  "lastName": "Goetsch",
  "email": "kelly.goetsch@commercetools.com",
  "shoeSize": 12
}
```

This is an *evolution* of your API, which should be easily supported without versioning. If the client doesn't specify the `shoeSize` parameter, the application shouldn't break. This goes back to Postel's law, which states that you should be "liberal in what you accept and conservative in what you send." When applied to APIs, Postel's law essentially means you shouldn't do strict serializations/deserializations. Instead, your code should be tolerant of additional attributes. If `shoeSize` suddenly appears as an attribute, it shouldn't break your code. Your code should just ignore it. For example, the serializer we use allows for the following annotation:

```
// To ignore any unknown properties in JSON input wuthout exception:
@JsonIgnoreProperties(ignoreUnknown=true)
```

If you adopt a strict approach to serialization, any difference in the client and server is going to break the client:

```
Unhandled exception
org.springframework.oxm.jaxb.JaxbUnmarshallingFailureException:
  JAXB unmarshalling exception: unexpected element
  (uri:"http://yyy.org", local:"xxxResponse").
  Expected elements are <{}xxx>,<{}xxxResponse>;
```

The approach of having evolving APIs goes back to Bertrand Meyer's open/closed principle, which he documented in his 1988 book, *Object-Oriented Software Construction*. In it, he said that software entities (especially APIs) should be "open for extension but closed for modification." He went on to further say:

> A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

> A module will be said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

The majority of APIs you have will fall into this category. Simply add attributes where you can, and don't break existing functionality.

The major advantage of this approach is that the APIs remain fairly static, allowing clients to code to them more easily. It's one less dimension for developers to care about. The supplier of the API only has one version of the codebase to support in production at any given time, dramatically simplifying bug fixing, logging, monitoring, etc.

The disadvantage of this approach is that the APIs are fairly locked from the start. Vendors who solely adopt this approach lose the flexibility to radically change the APIs, which is perfectly acceptable in many cases.

For APIs that change more radically and where true A/B testing is necessary, versioning is the preferred approach.

Versioning APIs

With versioning, the provider of the APIs deploys more than one major version of an API to the same environment at the same time. For example, versions 1, 2, and 3 of the pricing API may be live in production all at the same time. All versions can serve traffic concurrently.

While there are many flavors of versioning, a common approach is to guarantee API compatibility at the major version level but continually push minor updates. For example, clients could code to version 1 of an API. The vendor responsible for the implementation of the API can then publish and deploy versions 1.1, 1.2, 1.3, and beyond over time to fix bugs and implement new features that don't break the published API. Later, that team can publish version 2, which breaks API compatibility with version 1.

Clients (e.g., point of sale, web, mobile, and kiosk) can request a specific major version of an API when making an HTTP request. By default, they should get the latest minor release of a major version. This is often done through a URL (e.g., */Inventory/v2/* or */Inventory?*

version=v2) or through HTTP request headers (`Accept: applica tion/vnd.example.api+json;version=2`).

This is great for vendors who are rapidly innovating. It allows them to release minimum viable products. When enough is learned, they can fork the codebase and then offer the old version 1 and have an entirely new breaking API as version 2. The vendor isn't "locked in" to a specific API, as is the issue with evolving APIs.

The major challenge you'll have with versioning is persistent data. Here, there are essentially two approaches: you can have one datastore per major API version (Figure 3-4), or you can have one datastore per environment (Figure 3-5).

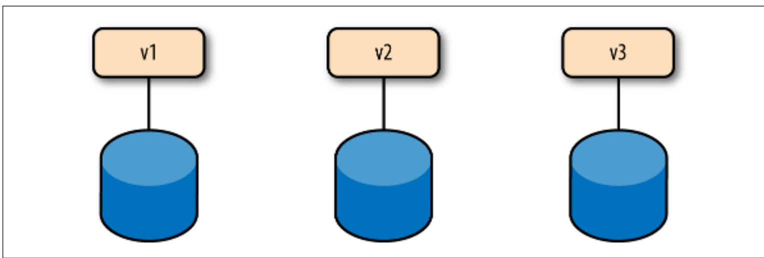


Figure 3-4. One data store per version

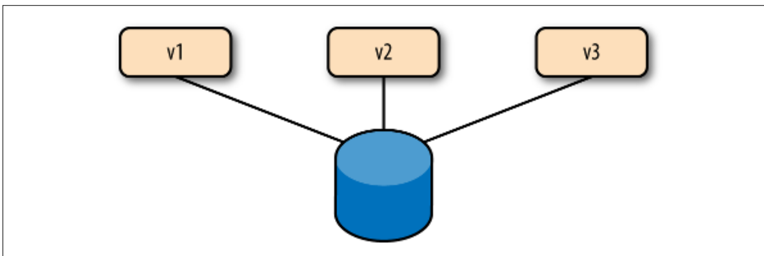


Figure 3-5. One data store per environment

If you have one datastore per major API version, then you need to migrate or continually synchronize the data between major versions. If your client was using version 1 of the order API, and then you start using version 2, you need to physically move or synchronize the data from version 1 to version 2. You can't just seamlessly switch over to version 2, for example. This is hard to do when you have multiple clients because it requires that you cut all your (potentially dozens of) clients over to the new version of the API at the same

time. Facebook has gone so far as to offer an upgrade utility to help developers transition from one version to another.

If you have one datastore per environment, with all API versions hitting the same datastore, you have the problem of "evolving" the objects. Your point-of-sale system could write an order object using version 1 of the API and, five seconds later, your iOS application could try to amend that order using version 2 of the same API. Any API version can write an object, and any API version can update that order at any time. This is by far the most common approach, but it's hard to do.

Versioning is only used because it offers more freedom to innovate, especially in a fast-changing environment. Evolving APIs are easier to support, but both the clients and the producer of the APIs tend to get locked in over time, slowing the pace of change.

Testing APIs

Testing is obviously important to all software development and must be taken seriously. Fortunately, APIs make testing easy. Before we go further, you must adopt a new mindset.

Traditional commerce applications were one large monolith, often deployed as a single multigigabyte EAR file or something similar. They had frontend and backend code, all contained in one application. The scope of testing was fairly well defined—whatever was in that single archive needed to be tested.

Commerce is now a collection of smaller APIs, often backed by separate microservice teams. You'll have a team that exposes an inventory API and another that exposes a pricing API. An enterprise could easily expose a catalog of a few hundred individual APIs for any client to consume. The providers of the APIs often have no idea how their APIs will be used by the dozens of clients out there. Think of APIs as LEGO blocks, available for use by anyone in any way.

With that in mind, let's look at the different methods of testing.

Local Testing

Testing APIs locally is pretty easy. Just download Postman, Advanced REST Client, Insomnia REST Client, or any of the myriad of GUI-based tools that allow you to execute HTTP requests against

a REST resource, and see the response (Figure 3-6). You can even use traditional cURL if you're inclined to use the command line. The purpose of this testing is to verify that uncommitted changes you've made locally don't break the API.

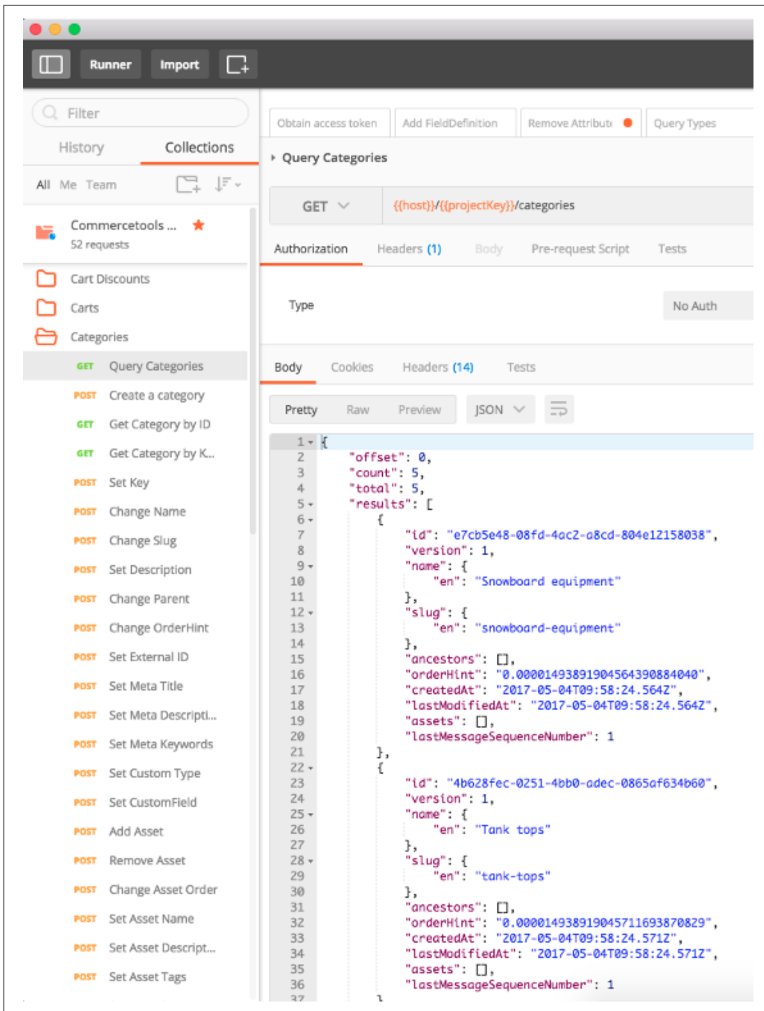


Figure 3-6. Postman HTTP client

Traditionally, local testing was hard because you'd have to run an entire multigigabyte application locally, including the UI, application server and database. There was no way to test out the backend functionality without exercising the frontend. But it's actually pretty

easy to test your APIs if you're just building APIs. The frontend developers can test their stack independently.

Unit Testing

While local testing is focused on individual developers testing uncommitted changes, unit testing is focused on ensuring that the API you're working on as a team (often a microservice team) works as expected. The API is the unit you're testing. An API is really just a contract, if you look at the big picture. When unit testing, you're verifying that the API is working as advertised.

Unit testing must be 100% automated and baked into your Continuous Integration/Continuous Delivery (CI/CD) pipeline. It should exercise all aspects of the API, including its functionality and especially the HTTP response codes it produces. If you version your APIs, your tests should cover supported versions of your APIs as well.

For example, let's pretend you have a `/Product/{id}` HTTP REST resource. When you call it with `/Product/12345`, you get back the following response:

```
{
   "product":
   {
      "id" : "12345",
      "name" : "Test product",
      "description" : "Long description..."
   }
}
```

To test this functionality, you can use any number of frameworks. Let's use a simple example using REST Assured. It would look something like this:

```
@Test public void
product_resource_returns_200_with_expected_product_name() {
    when().
            get("/Product/{id}", "12345").
    then().
            statusCode(200).
            body("product.name", equalTo("Test product"));
}
```

You can very easily hook this into any CI/CD pipeline so that each of your APIs is rigorously tested with each code commit.

Load Testing

In addition to testing the functionality of each API, you should also test its scalability limits. Again, you don't know who's consuming your API or what they're doing with it.

Any load testing framework out there can make an HTTP request to test an API. Common frameworks include ApacheBench, Gatling, and JMeter. If your API is backed by a microservice-style application and deployed to a public cloud with auto-scaling, you should have no problems scaling your APIs.

Integration Testing

Once you've verified that an individual API works and performs well under load, it then must be tested within the context of other APIs in the ecosystem.

Figure 3-7 shows a fairly common end-to-end flow you'd want to test.

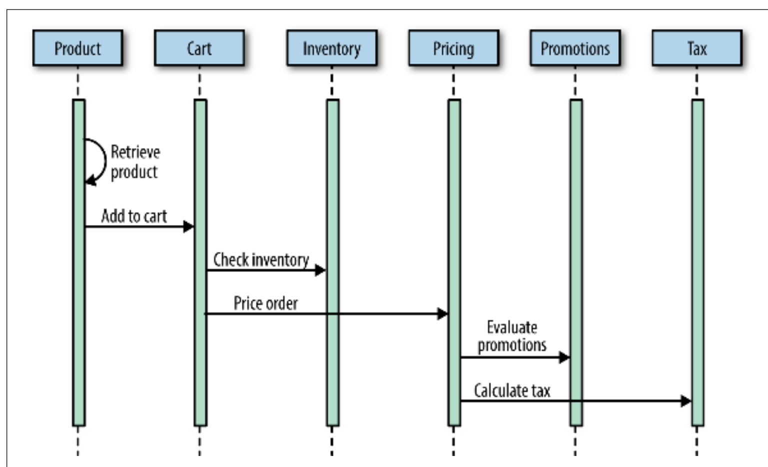


Figure 3-7. Synthetic integration testing

Repeat this for all the major flows for all the combinations of all the APIs you support. It might take a few minutes to execute the tests, but it'll be well worth the comfort in knowing that the APIs all work well together.

These integration tests should be executed every time code is checked in, as part of your CI/CD pipeline. If integration testing fails, stop everything and fix it.

Securing APIs

Before you can expose an API, you must secure it. Security starts with *authentication*: is this developer or application whom he/she/it purports to be? Next, you have to *authorize* the user: does he/she/it have access to this API? What type of behavior is permitted? Finally, you have to ensure that your users aren't abusing your APIs in some way. For example, you may want to cap the number of HTTP requests that can be made by any given client. A few thousand HTTP requests may be OK, but are 100 million HTTP requests per hour OK? Probably not.

What's great about using REST-based APIs is that the underlying stack (TCP, HTTP) is so widely used. There are well-established approaches to solving all of these security issues.

Authentication

Let's start with authentication. Authentication ensures that a user, whether a human or another system, is who he/she/it purports to be. It's like having your ID checked at the airport.

At a high level, your client needs to provide a "secret" of some sort (Figure 3-8). That secret can be a username/password or an API key of some sort, which is typically a long string encoded using base64. Keys are best because they're easier for developers to use, with many SDKs allowing you to supply the key via a configuration file.

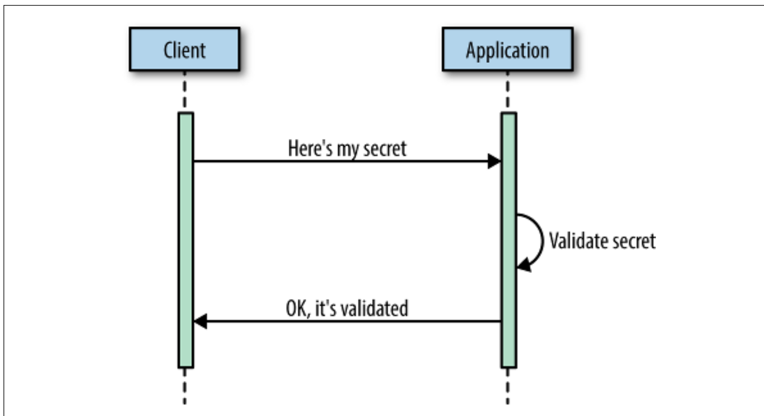


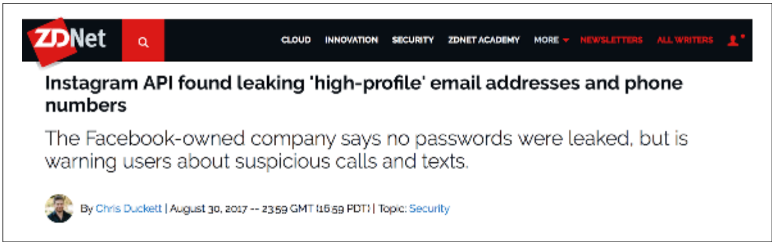
Figure 3-8. Simple authentication flow

Here's Tesco's developer documentation for how to pass in a key when making an HTTP GET request for products:

```
curl -v -X GET "https://dev.tescolabs.com/product/?gtin={string}&
  tpnb={string}&tpnc={string}&catid={string}"
-H "Ocp-Apim-Subscription-Key: {subscription key}"
```

> **NOTE**
>
> Never put your secret in a URL as an argument. URLs are public. Your secret should be private. Any number of intermediaries can sniff the URLs you're browsing, even if you use HTTPS. Your secret is safe if it's in the request header and you use HTTPS (HTTP + TLS or SSL).

The issue with simply supplying an API key or username/password is that the application now knows your secret. If you have one big monolithic application, it's fine, as it's less likely to leak out. But imagine if you have 100 APIs backed by 100 microservices/applications? That creates a whole new set of issues, which we'll discuss shortly (see Figure 3-9).



ZDNet CLOUD INNOVATION SECURITY ZDNET ACADEMY MORE NEWSLETTERS ALL WRITERS

Instagram API found leaking 'high-profile' email addresses and phone numbers

The Facebook-owned company says no passwords were leaked, but is warning users about suspicious calls and texts.

By Chris Duckett | August 30, 2017 -- 23:59 GMT (16:59 PDT) | Topic: Security

Figure 3-9. Don't let this happen to you!

Authorization

Once you've authenticated your client, you must now authorize that client to perform some action, like retrieve an order or query for inventory availability.

Authentication and authorization are often intermingled, but they're distinct. Going back to the airport analogy, authorization is scanning your boarding pass when boarding your flight. Your identity has already been validated, but now you need to be authorized to board a particular flight.

With one monolithic application backing your APIs, you can put an API gateway in front of your APIs. Begin by cataloging all your APIs and the HTTP verbs allowed by each (`POST`, `GET`, `PUT`, `PATCH`, and `DELETE`). Define which groups or individuals are allowed to access each API, and then within each API, which verb they're allowed to call. This is fairly simple (see Figure 3-10).

Proxy Endpoint	Method	Path	URL	Policies	Actions
default	GET	/*	.../retail/v2/products/*	1	Edit ✕ Delete
default	GET	/	.../retail/v2/products/	0	Edit ✕ Delete
default	POST	/*/reviews	.../retail/v2/products/*/reviews	17	Edit ✕ Delete
default	GET	/*/reviews	.../retail/v2/products/*/reviews	1	Edit ✕ Delete
default	GET	/*/buy	.../retail/v2/products/*/buy	0	Edit ✕ Delete

+ Resource

Figure 3-10. Securing your APIs through an API gateway

Now let's say you have 100 separate microservices, each backed by its own application and development team. A vulnerability in any one of those microservices will expose the secret to the public. The

secret will need to be regenerated and all clients using it will need to be updated.

OAuth 2 solves this problem by serving as a trusted intermediary between the client and the applications the client is trying to interact with. The client is able to authenticate once with an OAuth server. The server responds with a temporary access token, which the client sends as an HTTP request header with every request. The microservice/application receiving that temporary access token (such as a product or shopping cart microservice), then consults with the OAuth server, asking what rights the token has (see Figure 3-11).

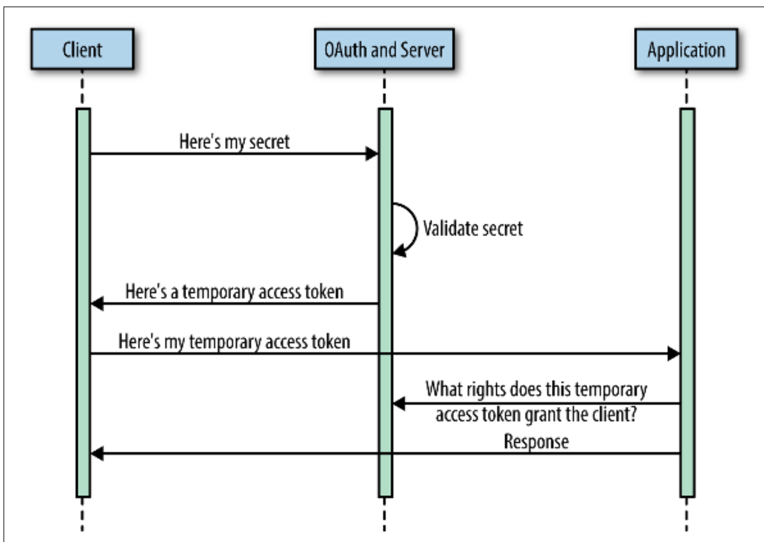


Figure 3-11. Advanced authentication flow

Access tokens are great because they:

- Are temporary
- Can be easily revoked
- Are granular, allowing for fine-grained access to resources
- Don't force each microservice/application to validate identity, as done by the OAuth server

Most API gateways allow you to define authorization policies, with the underlying implementation and enforcement being left to OAuth 2.

> **NOTE:** Both authentication and authorization are specialized domains that require experts. Hire an external consultant who specializes in this area. It's not worth doing yourself.

Request Rate Limiting

All your APIs should have throttling in place to ensure that they're not called too often by any given client, whether maliciously or not. Whether through error or poor architecture, a client can end up calling an API too many times. Establish limits for each type of client. Limit new developers to a thousand requests per hour, for example. But allow your web-based frontend to call your APIs without any limits.

Denial of Service attacks are rampant today. Use your content delivery network or alternate upstream systems to ensure that you're protected.

If the number of HTTP requests exceeds your policy, respond with an HTTP `429 Too Many Requests` response.

Data Validation

As with any application, all inputs must be validated. Since you're probably using REST APIs, you can use any HTTP-based web application firewall on the market. These firewalls are like traditional network firewalls except that they look more deeply at the HTTP traffic, applying rules that look for malicious behavior, such as cross-site scripting (injecting malicious code that the server then executes), SQL injection (getting the application to arbitrarily execute commands against the database), and the use of special characters or other behavior designed to cause application errors.

Using an API Proxy

APIs should always be protected behind an API proxy of some sort, whether it's an API load balancer or an API gateway. These proxies sit between your clients and backend, providing a number of valuable functions, including:

- Load balancing to individual instances of applications running your API

- Offering authentication and authorization
- Throttling abusive clients
- Conversion between representation formats, like XML → JSON
- Metering of API consumption
- Logging who's consuming your APIs and what they do with them

Where an API load balancer differs from an API gateway is *aggregation*, as seen in Figure 3-12. A web page or screen on a mobile device may require retrieving data from dozens of different APIs. Each of those clients will need data tailored to it. For example, a web page may display 20 of a product's attributes, but an Apple Watch may only display 1.

You could choose an API to serve as the intermediary.

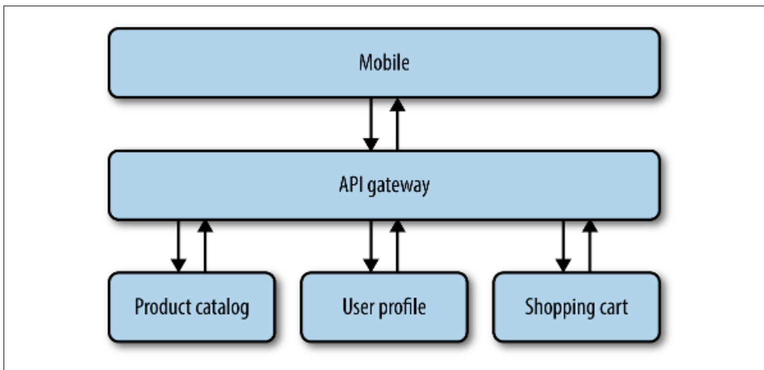


Figure 3-12. Aggregator pattern

The client makes the call to the API gateway, and the API gateway makes concurrent requests to each of the microservices required to build a single response. The client gets back one tailored representation of the data. API gateways are often called "backends for your frontend."

The issue with API gateways is that they become tightly coupled monoliths because they need to know how to interact with every client (dozens) and every microservice (dozens, hundreds, or even thousands). The very problem you sought to remedy with APIs and microservices may reappear in your pipes if you're not careful.

Whether you use an API load balancer or gateway, what matters is that you have one or more intermediaries between your clients and backend providing the functions outlined in this section.

Exposing APIs Using GraphQL

GraphQL is a query language specification for APIs that originated at Facebook in 2012, with the specification being open sourced in 2015. Facebook, Twitter, Yelp, GitHub, Intuit, Pinterest, and many others are now using it. GraphQL is analogous to what SQL queries brought to relational databases. Rather than querying the product and SKU tables independently, you can build a SQL query to retrieve data from both tables. GraphQL is the same but for APIs.

> **NOTE**
>
> GraphQL is a specification, not an implementation.

Let's say you wanted to render a page showing a given customer's last five orders, along with the products purchased in each order. Normally, you'd query the orders resource to find the orders belonging to the customer in question. Then you'd query the customer resource to retrieve the customer's name and other details. Then you'd query the product resource to retrieve the name of the products contained in the orders. To render just one page, you'd hit at least three different resources, retrieving kilobytes or even megabytes of data that is unnecessary to rendering the page (Figure 3-13).

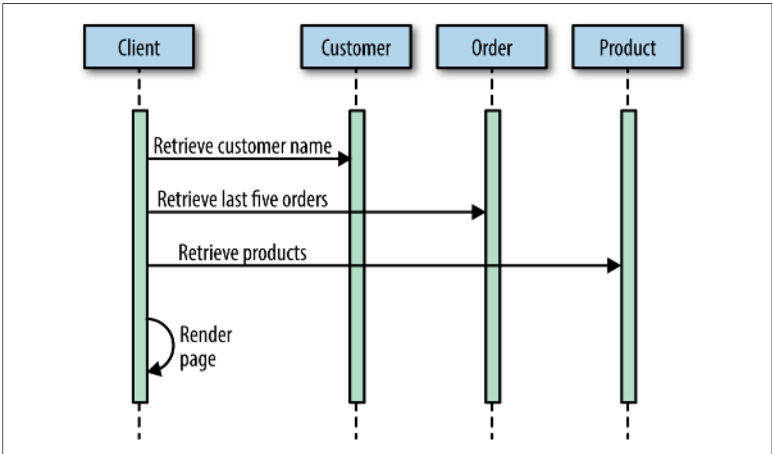


Figure 3-13. Retrieving customer, order, and product information using separate APIs

There are many solutions to this problem that involve inserting some form of an aggregation layer between your client and the different APIs you need to render (Figure 3-14). While that certainly works and is more elegant than hitting individual APIs, it forces the pages/screens to be rendered to match what data views are available. Now there's coupling between the different layers.

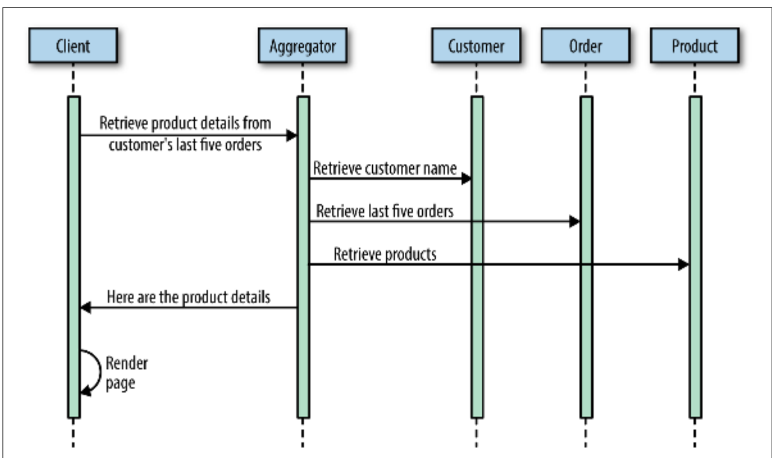


Figure 3-14. Retrieving customer, order, and product information using an aggregation layer

This is where GraphQL comes in. You can build a simple query that retrieves customer, orders, and products in one single request:

```
query {
  customer(id: "25484d8d45") {
    id
    firstName
    lastName
    orders (last: 5) {
      id
      datePlaced
      products: {
        displayName
      }
    }
  }
}
```

An intermediary layer then queries the individual APIs (*/Customer*, */Order*, */Product*) and exposes the data according to the GraphQL specification:

```
{
  "data": {
    "customer": {
      "id": "25484d8d45",
      "firstName": "Kelly",
      "lastName": "Goetsch",
      "orders": [
        {
          "datePlaced": "2017-06-10T21:33:15.233Z"
          "products": [
              "displayName": "Magformers Construction Set",
              "displayName": "Puzzle Doubles Find It!",
              "displayName": "LEGO Marvel Super Heroes 2"
          ]
        },
        {
          "datePlaced": "2017-08-19T08:07:55.007Z"
          "products": [
              "displayName": "Fast Lane Live Streaming Drone"
          ]
        },
        ....
```

The advantages of GraphQL include the following:

- Each client retrieves exactly the data it needs. This can be especially beneficial in low-bandwidth environments.

- JSON objects can be retrieved and used as is, without logic on the client side. Everything the client asks for is right there in the response.
- Your clients can remain independent from how the APIs are defined. There's no need to build these static intermediary layers.
- Pairs perfectly with React. GraphQL and React were co-developed and are extensively used together.

APIs are still necessary. But GraphQL is a perfect complement to them.

Final Thoughts

Now that we've discussed the mechanics of building an API, let's explore clients and how they consume APIs.

CHAPTER 4

Consuming APIs

Up to this point, we've discussed why APIs are important, how to model them, and how to implement the code behind them. In this chapter and the next, we'll discuss how to best consume APIs.

Identify Clients

Broadly, your clients can fall into one of the following three categories:

Internal applications
: Internal microservices that rely on functionality exposed by your APIs

Digital Experience Platforms
: Platforms that allow non-technical and semi-technical business users to build experiences for different customers. Grew out of web content management space

Custom frontends
: Custom web UIs, native mobile, embedded devices, social media, etc.

These three categories of clients have different requirements that will change how you model, build, and consume APIs.

Start by building an inventory of all known clients. If you have an existing API that you're replacing, trace the source of all the inbound API calls. Your existing API may have clients you never knew about.

Then, imagine what kinds of clients you'll see more of. If you're selling consumer packaged goods, many of your future clients will probably be Internet of Things (IoT) devices, like smart refrigerators. If you're selling apparel, many of your future clients will probably be magic mirrors and similar devices.

Finally, look at the characteristics and capabilities of each type of client:

Language
: What type of language can be used to consume the APIs? Does it have to be plain REST, or can you use SDKs based on Java, .NET, or other languages?

Latency
: Where physically is the client relative to the origin of your API? Are you running a custom web UI out of the same physical data center where you have a millisecond of latency, or are you consuming the API from a mobile phone that's 200 milliseconds from the origin?

Bandwidth
: Similar to latency, what are bandwidth constraints? Your end-consumers will get upset if their smart watch consumes hundreds of megabytes every time your app is opened.

Processing power
: Some clients may have very limited processing power, which impacts how you call the API. An internet-connected coffee pot that reorders coffee pods will be able to consume APIs very differently than a microservice running on a new data center's server.

Security
: Does your client support the libraries required for common authentication and authorization libraries? Native clients, especially, may severely restrict your ability to include third-party libraries.

Ability to cache
: Is the client capable of any client-side caching? To what degree? An internet-connected coffee pot might not be able to cache anything, whereas a full data center server can probably cache hundreds of millions of responses.

The capabilities of each client may force the producer of your API, whether internal or external, to cater to the lowest common denominator.

Let's look at the three classes of clients and explore what specifically sets them apart and what you should keep in mind.

Internal Applications

By far, the top client for your APIs will be other internal applications. These applications can be legacy back-office applications (OMS, ERP, CRM, etc.), newer cloud-native microservices, or anything in between.

Legacy applications can be excessively chatty with your APIs because they weren't built to support the number of integrations that are common within enterprises today. The big back-office applications that serve as the backbone of enterprises (OMS, ERP, CRM, etc) once largely operated within their own silos but today are connected by hundreds of touchpoints. To make matters worse, these applications don't tolerate latency. The world they were built for was deployment to on-premise data centers, with all applications physically deployed next to each other with no latency. Because responses were expected to be instantaneous, there often isn't any support for asynchronous programming of any sort.

Newer-style cloud-native applications, often backed by microservices, are built for the heavily interconnected world that is common within enterprises today. They are more intelligent about how they call other APIs, often relying on batching or calls that retrieve more data than is immediately required. They are better able to cache data. They support latency, often through advanced asynchronous programming models.

As we discussed in Chapter 2, serialization frameworks like Apache Thrift, Apache Avro, and Google's protocol buffers sometimes must be used. Clients like internal applications are best able to leverage these frameworks because you can often include the required libraries and have more control over how the APIs are called.

Digital Experience Platforms

Digital Experience Platforms (DXP) allow non- or semi-technical business users to define experiences for customers on different devi-

ces. An experience is an interaction with an end consumer, whether it's a traditional web page view, an alert on a smart watch, or a moment in front of a magic mirror (Figure 4-1).

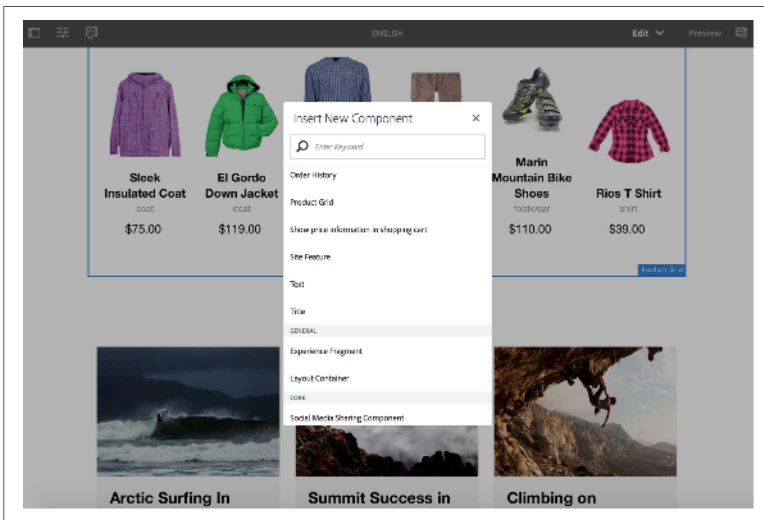


Figure 4-1. Example of a Digital Experience Platform (Adobe Experience Manager)

Marketing should own these touchpoints with customers because they're closest to customers. Yet in most organizations, IT owns all digital interactions with customers. DXPs give control of experiences to marketers and other business users. Here's how Gartner defines these platforms:

> A digital experience platform is a rationalized, integrated set of technologies used to provide and improve a wide array of digital experiences to a wide array of audiences. This includes web, mobile and emerging IoT experiences. DXP frameworks evolved from portals and web content management (WCM), yet differ from them with a broad collection of supporting services. Examples include app/API framework, search, analytics, collaboration, social, mobile and UX framework, and may include features like digital commerce or digital marketing.
>
> —Gartner, *Hype Cycle for Digital Commerce 2017, Gene Phifer, July 2017*

Commerce APIs are then injected into the various experiences. In this model, business owns the experiences, and IT owns the APIs that are consumed as part of those experiences.

These platforms are usually deployed to data centers that are near or well connected to where your APIs are served from. You often have your choice of programming language and are able to use third-party libraries, allowing you to use SDKs, serialization frameworks, and security libraries.

> **NOTE**
> Unless you have compelling reasons to build custom UIs, you should use a DXP.

Custom UIs

Custom UIs can be anything. They could be web, mobile, embedded devices, social media, etc. It's hard to find a device that isn't capable of facilitating a commerce transaction in some way.

These clients have wide ranges of capabilities, but many times you're left working directly with REST APIs. Remember to design and deploy your APIs in a way that allows the lowest common denominator of a client.

GraphQL can be of particular value to native clients, where latency and bandwidth are serious constraints. Facebook developed and then later open-sourced GraphQL for precisely this reason.

API Calling Best Practices

As a consumer of an API, what's behind the API shouldn't be of concern to you. As a consumer, you're given a URL, some documentation, and an SLA. The provider of the API should make sure that the API is functional and available. That being said, there are some things you can do to protect yourself from issues on the client side.

Only Request What You Need

It's easy to over-request data. A simple request to */Product/12345* might result in a response that's hundreds of kilobytes. A single product may have hundreds of attributes. Unless you're a merchandiser who's editing that information in a business administration UI, it is unnecessary for clients to be requesting that much. Displaying a product on a smart watch may only require retrieving three or four properties.

Your SDK may allow you to specify which attributes you're requesting. Or if you're making an HTTP request directly to a REST resource, you could simply specify the properties you want as part of the HTTP request:

```
GET /Product/12345?attributes=id,displayName,imageURL
```

If you're using GraphQL, it's even easier:

```
query {
  product(id: "12345") {
    id
    displayName
    imageURL
}
```

However you approach this problem, ensure that every single client is requesting only what is truly necessary.

Don't Make Too Many Calls

For performance reasons, it's important to not make too many calls to your APIs. Your clients may be accessing APIs over a mobile internet connection. Like traditional web browsers, your client may have limits on the number of parallel HTTP requests or even open sockets. The farther away your clients are from your APIs, the more you have to worry about this problem.

> **NOTE**
>
> HTTP/1 only supports one outstanding HTTP request per TCP connection. With browsers supporting only a handful of HTTP requests per distinct origin URL, HTTP/1 suffers from a head-of-line blocking problem whereby one slow response can block the rendering of an entire page.
>
> HTTP/2 is fully multiplexed, meaning it allows multiple HTTP requests in parallel over a single TCP connection. A client can send multiple requests over one connection, and the server can respond as each response becomes available. This makes it dramatically faster and easier to hit dozens of APIs in parallel to render a single page.

You can minimize the number of calls from the client by using some form of an aggregation layer. The layer may be an API gateway of some sort. It may be another application or microservice. It may be

GraphQL. What matters is that your remote client isn't making dozens or even hundreds of API calls to render pages or screens.

Use a Circuit Breaker

Calls from your client to your API should ideally be routed through a circuit breaker. Circuit breakers are a form of bulkheading in that they isolate failures.

If your client calls an API without going through a circuit breaker, and that API is fails, the client is likely to fail as well. Failure is likely because your client's request-handling threads end up getting stuck waiting on a response from your API. This is easily solved through the use of a circuit breaker (see Figure 4-2).

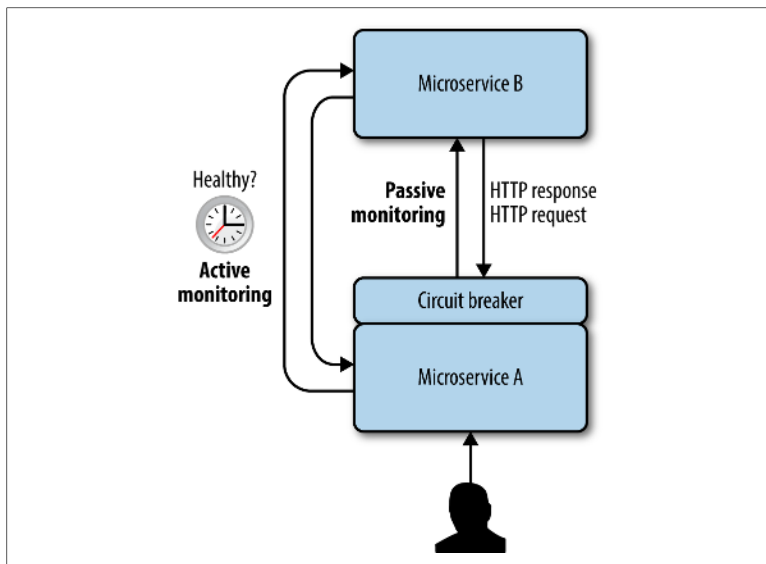


Figure 4-2. Example of the circuit breaker pattern

A circuit breaker uses active, passive, or active-plus-passive monitoring to keep tabs on the health of the microservice you're calling. Active monitoring can probe the health of a remote microservice on a scheduled basis, whereas passive monitoring can monitor how requests to a remote microservice are performing. If a microservice you're calling is having trouble, the circuit breaker will stop making calls to it, as seen in Figure 4-3. Calling a resource that is having trouble only exacerbates its problems and ties up valuable request-handling threads.

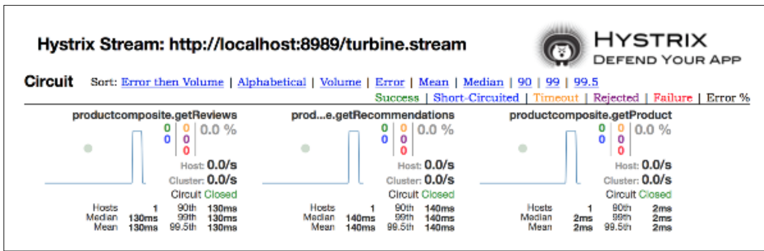


Figure 4-3. Example of Hystrix, the popular circuit breaker from Netflix

To further protect callers from downstream issues, circuit breakers often have their own threadpool. The request-handling thread makes a request to connect to the remote microservice. Upon approval, the circuit breaker itself, using its own thread from its own pool, makes the call. If the call is unsuccessful, the circuit breaker thread ends up being blocked, and the request-handling thread is able to gracefully fail.

The exact technology you use doesn't matter so much as the fact that you're using something.

Cache on the Client Side

A client can be anything—a mobile device, a web browser, another application or microservice, an API gateway, etc. For performance reasons, it's best to cache as aggressively as each client allows, as close to the client as possible.

What's good about REST APIs, is that they leverage the HTTP stack, which is extremely cacheable. The vast majority of HTTP requests use the GET verb, which is almost entirely cacheable. `GET /Product/12345?attributes=id,displayName,imageURL` will always return the same JSON or XML document. Through existing ETags and Cache-Control HTTP headers you can finely control how your clients cache responds. Google has a great guide on this topic.

Some clients are essentially transparent passthroughs that make caching transparent. For example, many clients access APIs through a Content Delivery Network. Many APIs are exposed through API gateways. With clients like this, you set up caching and forget about the internals of how it's handled.

If your client is another application or a custom frontend, you'll have to deal with caching on your own. In this case, consider using Redis, Memcached, or some other object store to cache entire resources or collections of resources.

Final Thoughts

Now that we've discussed consuming APIs, let's turn our attention to how to extend/customize APIs to suit your specific business needs.

CHAPTER 5

Extending APIs

Your business is unique, and few APIs will offer the exact functionality required by every client. Whether provided by a third-party software vendor, a systems integrator, or an in-house team, you'll often have to extend the APIs you consume. Common extensions in the commerce space include:

- Sending notifications when an event has occurred, like sending an email when an order has been shipped
- Capturing additional properties on resources, like capturing a customer's shoe size at registration
- Validating data, like checking user-submitted data for SQL injection attacks
- Performing real-time data checks, like making sure inventory is available during checkout
- Adjusting the behavior of the API, like changing how prices are calculated

In this chapter, I'll explain the three different approaches to extending APIs, highlighting which approach is best for which type of extensions.

Extending Traditional Enterprise Commerce Platforms

If you were consuming a legacy commerce platform, you'd essentially be getting two things:

- A framework
- A bunch of libraries

The framework, platform, or whatever you want to call it often includes some type of extensibility mechanism, allowing you to plug your custom code inside the framework. This is often implemented with IoC.

Libraries are immutable, precompiled pieces of functionality, like JAR files and NPM packages. Libraries are similar to APIs, with the only difference being how the functionality is consumed. With a library, you're embedding the vendor's code in your application. With an API, you're still embedding the vendor's code in your application but rather than executing locally, it's executing somewhere else.

Martin Fowler draws the distinction between frameworks and libraries as follows:

> Inversion of Control is a key part of what makes a framework different to a library. A library is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.
>
> A framework embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

Commerce platforms are no longer just something you deploy off to the side of your business. Commerce *is* your business. Small, vertical teams are building and exposing granular pieces of functionality to the rest of your business, often as microservices.

In this model, there is no longer a single packaged commerce solution providing both the framework and the libraries with a vendor telling you how to extend out-of-the-box functionality.

Approaches to Extending APIs

Let's explore four different approaches to extending API-based commerce platforms.

Extending the Default Object Model

Many customizations are simply a matter of collecting additional attributes or defining custom objects. If you sell shoes, you'll want to capture the shoe size of your customer. If you sell auto parts, you'll want to capture the make/model/year of the customer's car. These are all fairly standard requirements that any API-based commerce platform should be able to easily support.

Event-Based

Many extension use cases can be solved through the use of events. An *event* is essentially a message with a payload—often a JSON or XML-based representation of an object—like an order or a customer profile. What differentiates an event from a message is volume. Traditionally, messaging was limited to passing important bits of data (orders, customer profiles, etc.) between applications. Messaging often used heavyweight protocols like JMS and relied on expensive commercial products.

Eventing is a central characteristic of modern software development, especially microservice-based development. Everything is represented as an event. Individual lines in log files; small changes to orders, customer profiles, and products; container instantiations, API calls, and so on—all are represented as unique events. It's not uncommon to have millions or tens of millions of events per second in a microservice-based ecosystem.

Once an event is emitted, it must be consumed and passed to custom code that can process it. Here, you have two options.

The first option is to write a small application. Using a small framework like Spring Boot, Play, or Node.js, you can quickly write a small application whose sole responsibility is to pull events and do something. The "something" may be connecting to your backend CRM system and updating a customer record. It may be sending an email to a customer. It may be charging a credit card. While these applications can be easily built, they must be maintained over years or even decades. The development framework you use will need to

be upgraded. You'll inevitibly have to upgrade your continuous delivery pipeline. Docker will continue to change as it matures. Maintaining applications is difficult.

The second option is to use a function-as-a-service/serverless framework. AWS Lambda, Google Cloud Functions, Azure Functions, and others allow you to essentially route specific types of events back to arbitrary functions/methods.

Let's take an example. A fairly routine requirement is to send an email to the customer when the order has been successfully sent to the OMS. You'd start by defining your AWS Lambda function, as shown in Figure 5-1.

Figure 5-1. Configuring your Lambda function

Then you'd write your code, as shown in Figure 5-2.

Figure 5-2. Lambda function code

Then, you'd bind your function to your event (Figure 5-3) so that your code is executed whenever a message is published.

Figure 5-3. Binding a Lambda function to an event

You'd follow a similar approach with the other cloud vendors and their function-as-a-service offerings.

Function-as-a-service works great because it's simple and it allows different pieces of the application to be changed and deployed at different times. You can change your order confirmation email template without redeploying your entire monolithic application, which is revolutionary for enterprise-level commerce.

To summarize, it's best to use event-based extensions for:

- Asynchronously synchronizing data between systems, often from your commerce platform to legacy backend systems
- Sending notification emails
- Logging important data for audit purposes
- Computationally heavy activities, like generating product recommendations

Posting Data Using Webhooks

Rather than the provider of the API publishing events, they may additionally or instead offer webhooks. *Webhooks* are URLs that the vendor of the API posts data to.

For example, you can often register webhooks for when an order is placed, a product is added, or a customer record is updated. If you register a webhook when an order is placed, the vendor of the API will post the entire order and maybe the customer's profile to the URL that you define. It's essentially the same model as events, with the following exceptions:

- HTTP requests are often made synchronously, whereas events are often posted asynchronously.
- Rather than pulling messages, you have to provide your vendor with a URL to which they can post data.
- An event-based model allows any number of consumers to consume an event; webhooks often allow just one URL.
- The vendor has to explicitly provide hook points. This often results in fewer hooks being defined. Events are just emitted from the application.

One advantage of webhooks is that it is theoretically possible to post data to backend systems without having to write an intermediary application or serverless function. The problem is that these backend systems have different authentication and authorization schemes. Sometimes they require VPNs. It's hard to do a direct post from a third-party vendor's API to an application that's not meant to be public.

> **NOTE**
>
> Webhooks are difficult for API vendors to properly implement. What happens if the callback fails? How often should retries be made? Are calls idempotent? How can these callbacks be monitored by third parties? It's hard. Vendors are increasingly exposing events, which do not suffer from many of the same problems as webhooks.

Going back to the previous example of using events, you can also define URLs that trigger serverless functions. Figure 5-4 shows an example of how you'd do it with AWS Lambda and AWS API Gateway:

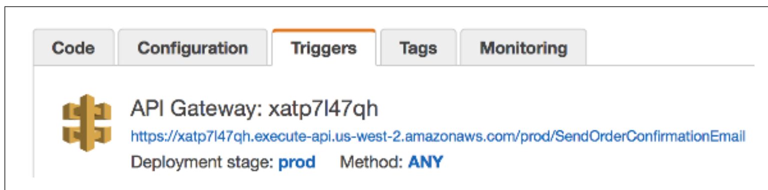


Figure 5-4. Binding Lambda function to a URL

In this example, you'd register *https://xatp7l47qh.execute-api.us-west-2.amazonaws.com/prod/SendOrderConfirmationEmail* as the

webhook URL for the "placeOrderWebhookURL" property or whatever your vendor defines.

An issue to be aware of is that there is little in terms of standards. If you switch providers, you'll have to rewrite your functions.

Wrapping API Calls

While extending the default object model, events and webhooks solve a wide range of use cases, there is a class of extensions that are inherently harder to implement. Examples include:

- Validating data, like checking user-submitted data for SQL injection attacks
- Performing real-time data checks, like making sure inventory is available during checkout
- Adjusting the behavior of the API, like changing how prices are calculated

All these examples require synchronously executing custom code before or after an API call is made. The process for this flavor of extensions is exactly the same as you're used to with old third party commerce platforms.

Let's take inventory as an example. Let's say you want to perform a real-time inventory check when inventory dips below 100 units.

With a traditional commerce platform, you'd use their IoC framework to extend the out-of-the-box inventory code. Out of the box, a client calling the inventory resource through an SDK would return an instance of `com.bigcorp.inventory.Inventory` or whatever the out-of-the-box class from your vendor is. You'd then create `com.yourcorp.inventory.CustomInventory` with a method as follows:

```
public class CustomInventory extends Inventory
{
        public int queryInventory(String productId,
      String skuId)
        {
                int inventory = super.queryInventory(productId,
          skuId);
                if (inventory < 100)
                {
                        inventory =
```

```
                    SAPConnection.realTimeInventoryLookup(productId,
                      skuId);
                      }
                      return inventory;
            }
    }
```

Now, the `Inventory` resource resolves back to instantiations of `com.yourcorp.inventory.CustomInventory`. Simple.

The difference is that API-based commerce platforms don't have an IoC framework. You're consuming APIs from your commerce platform vendor, from third-party vendors (payment, tax, product recommendations, etc.), and from custom applications/microservices that you build in house. There is no longer a single platform—it's just a bunch of APIs from different sources.

To perform this simple extension, you'd start by using your API vendor's SDK. Then, pick a small framework like Spring Boot, Play, or Node.js and build a standalone application that queries the API using your vendor's SDK:

```
public class InventoryService
{
    @RequestMapping(value = "/Inventory/{productId}/{skuId}",
      method = RequestMethod.GET)
    public int queryInventory(
      @PathVariable("productId") String productId,
      @PathVariable("skuId") String skuId)
    {
        int inventory =
        MyVendorSDK.getInventoryService().queryInventory(
          productId, skuId);
           if (inventory < 100)
                  {
                        inventory = SAPConnection.
                realTimeInventoryLookup(productId, skuId);
        }
           return inventory;
    }
}
```

This example is based on Spring Boot.

Once you've defined your application, deploy it behind your API gateway. Clients would then query inventory by accessing *https://api.yourcompany.com/Inventory* and passing `productId` and `skuId` as HTTP `GET` arguments.

Final Thoughts

Using these three approaches, you can implement just about any requirement you can think of. And if you can't, just build a brand-new API backed by a new microservice to accomplish your requirement.

Stepping back, I hope this introduction to APIs gives you enough guidance to get started on your journey. There are a lot of things to think about; but with proper planning, I have no doubt you'll be able to quickly get started and realize some quick wins.

About the Author

Kelly Goetsch is Chief Product Officer at commercetools, where he oversees product management, development, and ops. He came to commercetools from Oracle, where he led product management for its microservices initiatives. Kelly previously held senior-level product development and go-to-market responsibilities for key Oracle cloud products representing billions of dollars of revenue for Oracle. Prior to Oracle, he was a senior architect at ATG (acquired by Oracle), where he was instrumental to 31 large-scale ATG implementations. In his last years at ATG, he oversaw all of Walmart's implementations of ATG around the world.

Kelly has expertise in commerce, microservices, and distributed computing, having spoken and published extensively on these topics. He is the author of two books—*Microservices for Modern Commerce: Dramatically Increase Development Velocity by Applying Microservices to Commerce* (O'Reilly) and *E-Commerce in the Cloud: Bringing Elasticity to E-Commerce* (O'Reilly).

He holds a bachelor's degree in entrepreneurship and a master's degree in management information systems, both from the University of Illinois at Chicago. He holds three patents, including one key to distributed computing.