O'REILLY®

Mastering API Architecture

Defining, Connecting and Securing Distributed Systems and Microservices

Early Release

RAW & UNEDITED

James Gough,
Daniel Bryant &
Matthew Auburn

Mastering API Architecture

Defining, Connecting, and Securing Distributed Systems and Microservices

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

James Gough, Daniel Bryant, and Matthew Auburn

Mastering API Architecture

by James Gough, Daniel Bryant, and Matthew Auburn

Printed in the United States of America.

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

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

Acquisitions Editor: Melissa Duffield

Development Editor: Nicole Tache

Production Editor: Deborah Baker

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

January 2022: First Edition

Revision History for the Early Release

- 2021-03-18: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492090632 for release details.

978-1-492-09056-4

[LSI]

Chapter 1. API Architecture Primer

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be Chapter 1 of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at feedback@mastering-api.com.

Anyone who has taken a long journey will no doubt have encountered the question (and possibly persistently) *"are we there yet?"*. For the first few inquiries we look at the GPS or a route planner and provide an estimate - hoping that we don't encounter any delays along the way. The journey for an API Architect is usually complex and often coupled with changing infrastructure, cloud migration and possibly organizational change. Even in Greenfield projects, Type 1 Decisions [1] can have drastic consequences, despite seeming straightforward at the time. The ability for an API Architect to recognize and focus on key decisions is an important skill. An API Architect's challenge does not stop here, in order to get internet scale and meet performance demands considering a microservices based architecture and horizontal scale will be necessary.

One of the challenges with architecture is in reality we are never *there*,[2] or never quite where we planned to be. Trade offs are made in order to unlock value at different stages of the journey. For example, Service Mesh is a

fantastic technology and is an enabler of a microservices based architecture and APIs. However, if your architecture began before Service Mesh was possible, what approach do you take? Do you stop and rebuild? The culminating effect of delivering incremental value combined with new emerging technologies leads to the concept of evolutionary architecture. Evolutionary Architectures are a critical idea we will embrace and discuss in the chapters to follow.

If we don't have the API equivalent of a GPS system and expect our architecture to evolve, how do we best arrive at a system that is in the best possible state? In this book we will cover a variety of references, books and blogs for you to explore, whilst bringing together the context of why these topics are important to an API Architect. There are many media formats of information and everyone learns in a different way, finding the right approach for you personally will be an important balance to strike. As authors we came up with the idea of writing a book on API Architecture at the O'Reilly Software Architecture Conference in New York. In talking together and with other delegates we felt many key concepts of API architecture are covered extremely well independently, but often do not provide the big picture perspective of building out an API architecture. Although architects in different organizations are all travelling on different journeys, asking others for directions and challenging the planned route helps set us up for success.

In this chapter we will set the groundwork that will set us heading in the right direction for the rest of the book. We will explore key concepts and definitions and discuss how they fit within the context of an API architecture. We will introduce some tools and ideas for sharing architecture and decisions with other developers and architects. The chapter will close with the proposed architecture for our conference system, that we will discuss how to design and build throughout the book.

A Brief Introduction to APIs and Microservices Based Architecture

In the field of software architecture there are a handful of terms that are

incredibly difficult to define. The term API (Application Programming Interface) falls into this categorization, as the concept first surfaced as many as 80 years ago. Broadly speaking APIs can be broken into two categories depending on whether the API invocation is *in process* or *out of process*. For example, Java's Date Time Library `java.util.time` is a library provided by the core platform. The invocation mechanism and data exchange is well defined, it is within the language constraints. The invocation is compile time safe, so it is understood at the time of writing code the conditions under which the exchange mechanism will be successful. An *out of process* exchange is significantly more complicated, the external process can be written in any language and potentially located across the world. There needs to be an agreement on the protocol, expected behavior and compatibility. This focuses on *out of process* exchange, in Chapter 2 we will explore many of the options available to us.

In the field of software architecture there are a handful of terms that are terribly named. Microservices is a term that is almost universally agreed as one of the most misleading names in our industry, the determination of size rarely comes into it. In the book we prefer the term Microservices Based Architecture, which involves the API Architect taking a pragmatic balance between Service Oriented Architecture and a Microservices Based Architecture. One thing we can agree on when it comes to Microservices is that there will be an increased number of services, looser coupling and a requirement to define interfaces and behavior between services.

The combination of Microservices Based Architecture, Public Cloud and APIs has lead to an explosion in demands for considering API Architectures. Designing for both elastic compute and expected failure has become more common with The Twelve-Factor App methodology. Although *out of process* exchange is not new, the paradigm shift over the recent years makes decoupled process interaction more important than before.

WHAT IS WRONG WITH A MONOLITHIC APPROACH?

It is possible to deliver an API Architecture using infrastructure routing and a monolithic service, and for smaller organizations this may be the best way to start. As the number of APIs offered and

number of consumers increase, rapid bottlenecks in the process may start to form both in terms request latency and deploying new services. Chapter 6 will provide insights into the process of evolving existing services and will be worth considering for existing monolithic deployments.

Before we dive too far into what is the right type of deployment for our problem or drawing up an architecture proposal, we need to consider an important concept for API Architectures, traffic patterns.

Traffic Patterns: API Consumption From All Cardinal Points

Imagine walking into a coffee shop 5am on Broadway New York, mildly jet lagged and ordering a coffee. The barista takes your order and you pay, meanwhile a label is attached to the cup with your request and passed along to be fulfilled. Another barista picks up the cup and completes the request, handing it to you with warm or cold caffeinated goodness. You sit back and watch the world go by, and instead of the usual concurrency example with coffee shops you start to appreciate traffic patterns (or maybe that's just us!).

Two more customers walks in (or ingresses) the coffee shop through the door with a request. The barista stops the first and checks what they want, the new customer is looking for waffles - which is not available. The barista turns them away [3], the next customer has a valid order. The order is passed along to be fulfilled by the hot beverage barista. Figure 1-1 displays the different interactions, lets formally define the flow of traffic through our system.

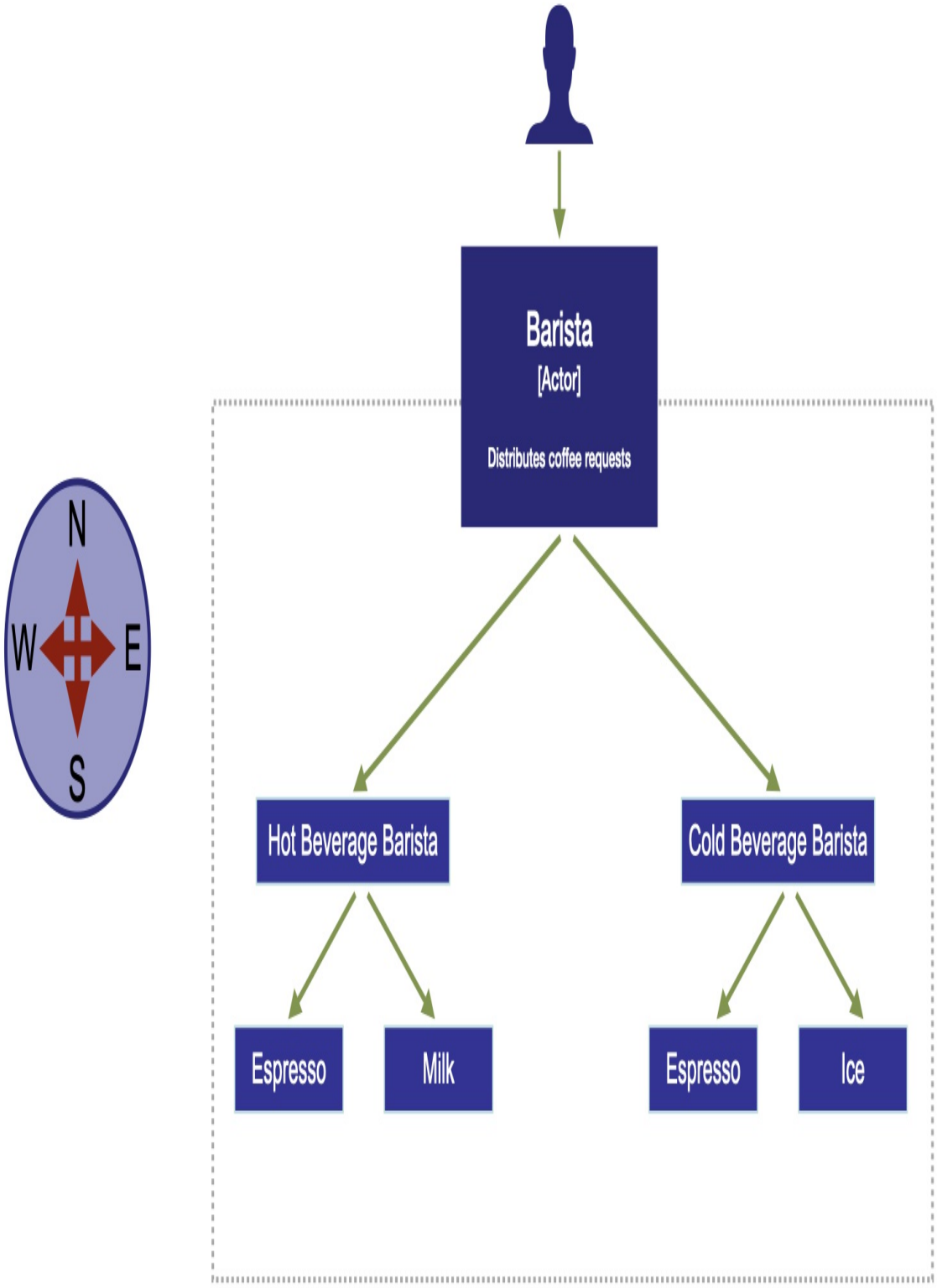

Barista
[Actor]
Distributes coffee requests
N
W
E
S
Hot Beverage Barista
Cold Beverage Barista
Espresso
Milk
Espresso
Ice

Figure 1-1. Traffic Patterns in a Coffee Shop

North-South *Ingress* Traffic

North-south, or ingress, traffic is network traffic entering (or "ingressing") into a data center, network, or a Platform-as-a-Service (PaaS) or container orchestration cluster (e.g. Kubernetes). Traditional network diagrams were drawn with the ingress traffic entering the data center at the top of the page and flowing down (north to south) into the network, and the name has stuck. In our example this is the customer, entering through the door to make the request to the barista.

NORTH-SOUTH KEY POINT: YOU DON'T KNOW THE CLIENT

Any location of your network that is exposed publicly will be accessed by unknown clients, often by design, and more often than not by clients you hadn't intended. This means that any component handling north-south traffic must make valid assumptions about client identity and also include appropriate defenses. An API gateway that handles north-south traffic can often be the first line of defense (in depth) for an application and organization, and so security should be top priority: authentication, authorization, transport security, rate limiting etc.

East-West *Service-to-Service* Traffic

East-west, or service-to-service, traffic is network traffic flowing within a data center, network, PaaS, or cluster. Traditional network diagrams were drawn with the service-to-service (inter-data center) traffic flowing from left to right (east to west) in the diagrams. In our analogy this was the barista at the checkout handing the cup over to the cold beverage/hot beverage barista in order to be fulfilled. In turn the barista needs to request contents from other systems to fulfill the request.

EAST-WEST KEY POINT: TRUST, BUT VERIFY

Most east-west traffic, particularly if the origin is within your IT estate, can be trusted to some degree. For example, it is often easier to implement mutual TLS (mTLS) within an organization's network, and identity access management (IAM) and network access control lists (NACLs) have a

smaller scope in comparison with dealing with the global (internet) context. However, this isn't to say you can be lax with service-to-service security, as you still want to design applications and network architecture to make it hard for accidental issues (such as an internal denial of service by a broken service) or bad actors to cascade and move through your systems.

Cross-datacenter traffic is by no means a new concept — both on-premise/co-located infrastructure and cloud platforms have supported this since disaster recovery/business continuity (DR/BC) became a priority — but it has seen increased resurgence over the past several year with many of the service mesh technologies surfacing this as a first-class concept. So, you are probably asking yourself: is traffic flowing from one data center to another, or from one cluster to another, north-south traffic or east-west? You've asked a good question, and like all interesting software development questions that answer is "it depends"!

Generally speaking, if the traffic's origin is trusted and the traffic is flowing over an internal network or over secured network (IPSec, VPNs, mTLS etc) it is often treated as east-west traffic. However, modern service mesh multi-cluster paradigms, such as service mirroring utilise an ingress controller (or API gateway) to route traffic that is egressing from one cluster to a service in another. So, here the traffic could be looked at as north-south. The best answer most likely points to the intent of the communication i.e. is the origin traffic from a trusted source and within some definition of "your" system.

Traffic Patterns and Traffic Management

Once we have a handle on our traffic patterns, the next challenge is how we manage traffic. Consider when it is 8am and the coffee shop is busy as people head to work, we'd want to be able to scale East → West services to handle the increase in traffic. Perhaps we want to deploy a new coffee machine that improves the speed coffee is dispensed, how do we ensure this machine isn't overloaded whilst it is tested? This is where traffic management becomes and important concept in continuous delivery and deployments patterns in an API Architecture.

Different Types of API

API Architecture Taxonomy

Core Concepts

The Open Systems Interconnection (OSI) Networking Model

Control Planes and Data Planes

Sidecar Pattern

Evolving Architecture

API Architecture Toolbox

There are a lot of in-depth conversations about what makes a great architect and anti patterns in the role of an architect. The reality is that the size and maturity of the organization will determine what works best for individuals in this position. There are however several focus areas that the authors would recommend architects prioritize when evolving their API Architecture.

- The ability to clearly communicate the architecture to stakeholders and the importance of components
- When making decisions clearly record the decision, including the context and background for making the decision
- Be amenable to an evolutionary architecture, work in the open and review the platform whilst continuing to deliver value

- Aim to be aware of upcoming patterns and technologies from industry and have an unbiased forum to review and present ideas

When rolling out an API Architecture there will be stakeholders across infrastructure, security, devops, application development teams and of of course the API consumers too. This could be a few people in a smaller organization or thousands of people, depending on the organization and the consumers of the API. Regardless of the size of organization the authors believe there are several key tools to have in the architects toolbox.

Architecture Decision Records (ADR)

As developers, architects and indeed humans we have all been in the position where we ask the question *"what were they thinking??"*. If you have ever driven on the M62 between Leeds and Manchester in the United Kingdom you may have been baffled by the construction of the motorway. As you climb the hill on the 3 lane highway it starts to deviate away from the traffic contraflow, until eventually Scott Hall Farm emerges surrounded by around 15 acres of farming land nestled between the carriages. Local legend of what happened described the owner of the land as stubborn and refused to move or handover his land, so the engineers simply built around him [4] 50 years later a documentary surfaced revealing that the real reason for this was a geological fault beneath the land, meaning the motorway had to be built that way. When people guess why something was done in particular way expect rumour, humour and possibly criticism to emerge.

In software architecture there will be many constraints that we have to build around, however we want to ensure the decision is recorded and transparent. Architecture Decision Records (ADRs) help make decisions clear in software architecture.

> An Architecture Decision Record (ADR) is a document that captures a decision, including the context of how the decision was made and the consequences of adopting the decision.
>
> —Engineering at Spotify Blog

There are several templates available that help to capture ADRs consistently

and group ADRs together. Some teams use a git repository and pull request new ADRs in, asking for feedback on the PR. Some companies keep ADRs centrally (or pull them from project repositories to a central location). The important thing to bear in mind is that a rejected ADR is probably more valuable than an accepted one, so ensuring they are available for comment and input is important.

C4 Model

The Unified Modeling Language (UML) provides an extensive dialect for communicating software architectures. A major challenge is that the vastness of what UML provides is not committed to memory of architects and developers, people quickly revert to boxes/circles/diamonds. It becomes a real challenge to understand the structure of diagrams before getting onto the technical content of the discussion. Many diagrams are only committed to a project history if someone accidentally uses a permanent marker instead of dry wipe marker by mistake.

The C4 Model provides a simplified set of diagrams that act as a guide to your project architecture at various different levels of detail. Context is an important part of reviewing a software architecture, and progressing through the series of diagrams is a progressive disclosure for the reader or during presentations.

Table 1-1. C4 Model Primary Diagrams

Level 1: System Context	The System Context diagram is the big picture top level view of the software system. The technical details are not the focus here, but instead the aims of the software system and how it interacts with users and other dependencies of the system. This diagram should be understandable by both technical and non-technical stakeholders, it's the diagram you would get out when speaking to any new team.
Level 2: Container Diagram	Once the context and overall big picture is established, the container diagram helps to look at the detailed breakout of the major participants in the architecture. A container is defined as *"something that needs to be running in order for the overall system to work"* for example a database or mobile application.

	The focus on these diagrams is technical in nature and builds on the context to discuss the technical implications. It is likely this diagram will be used in architecture reviews for feedback and discussion.
Level 3: Component Diagram	For each container, the component diagram discusses what makes up the containers functionality. A component is really tricky to define, and it really does depend on what language/stack you are building upon. If you are using Java this can be thought of a well defined interface that is used across the application - for example `com.masteringapi.routing` might supply the information for routing requests. The component diagram helps to define what the roles and responsibilities are, along with the interactions expected within the system. This diagram might be reviewed if the detail of a container is queried in discussion, it is also a very useful map to the codebase. Think about the first time starting work on a new project, you can browse the *self documenting* codebase - but it can be difficult to piece everything together. This diagram is really a map for developers and architects to discuss the purpose and evolution of the container.
Level 4: Code Diagram	The code diagram is where UML lives within the C4 model and entity relationship (ER) diagrams. Ideally this is an automatically generated diagram that gives the architect/developer full detail about the classes, interfaces in the container.

The astute reader may have noticed that the diagrams do not talk about infrastructure/deployment details and only talk about a single system. The C4 Model has a series of supplementary diagrams that will be extremely useful to provide some of the missing details that you may need to capture. For example, the System Landscape Diagram provides the context of the given system within other systems in the plant.

LINES AND ARROW HEADS

One of the most confusing parts of UML is the degree of complexity in the arrows and lines used within the diagram. In C4 an arrow represents a unidirectional relationship, which has a label that conveys the intent of the relationship between the entities. The goal should be that the reader can see there is a relationship and understand what it is for and why it is there at each level.

Lets practically explore the use of the C4 model by exploring the conference

system example we will use throughout the book.

Conference System Example

> **COMING SOON**
>
> As we continue to build later chapters of the book the example system is in flux and evolving to provide the best possible learning experience. In later early access releases we will add some of the diagrams here to introduce and explore the overall solution.

Structurizr Tool

Summary

---

1 Irreversible decisions that require careful planning and consideration.

2 Spoiler alert.

3 404 waffle not found is the worst type of exception.

4 Local stubborn traits fueled this likely explanation.

Chapter 2. Defining Rest APIs and Alternative API Approaches

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be Chapter 2 of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at feedback@mastering-api.com.

Microservices based architectures and Service Oriented Architectures promote an increased number of independent services. Often services are running multi-process, on multiple machines in data centers around the globe. This has led to an explosion in mechanisms to communicate between processes and deal with the challenges of distributed communication between services. The software development community has responded by building a variety of creative API protocols and exchanges including REST, gRPC and GraphQL learning from vintage protocols. With a range of options available an API architect needs an overall understanding of the exchange styles available and how to pick the right technologies for their domain.

In this chapter we will explore how REST addresses the challenges of communicating across services, techniques for specifying REST APIs using OpenAPI and the practical applications of OpenAPI specifications. With modern tooling building REST APIs is easy, however building a REST API that is intuitive and follows recommended practices requires careful design

and consideration. In order to be successful with microservices, a focus on continuous integration, deployment speed and safety is essential. REST APIs need to complement DevOps and rapid deployments rather than become a bottleneck or deployment constraint. In order to keep our system free to evolve and not disrupt consumers, it is also necessary to consider versioning in the release process of APIs.

Due to the simplicity of REST and wide support it is usually one of the best approaches to create an API. However there will be situations and constraints where we must consider alternative approaches for our API implementation. This may come down to performance or the requirement to create a query layer across multiple APIs.

The chapter concludes by looking at whether we can combine multiple API formats in a single service. We will also discuss whether the complexity of multi specifications takes us further away from the fundamentals of REST and other API approaches.

Introduction to REST

Roy Fielding's dissertation Architectural Styles and the Design of Network-based Software Architectures provides an excellent definition of the architectural applications of REST. REpresentational State Transfer (REST) is one of the primary mechanisms for exchanging data between client and server and is extremely popular within the API ecosystem and beyond. If we adjust *"distributed hypermedia system"* in Roy Fielding's definition below to *"an API"* it describes the benefit that we get from using REST APIs.

> The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based

application.

—Roy Thomas Fielding

To explore REST in more practical terms Figure 2-1 describes a typical REST exchange and the key components involved in the exchange over HTTP. In this example the consumer requests information about the *attendees resource*, which is known by the *resource identifier* *http://mastering-api.com/attendees*. Modelling an exchange in REST involves a request method (also known as the request verb), in this case a `GET` to retrieve data, and the resource identifer to describe the target of our operation. REST defines the notion of a *representation* in the body and allows for *representation metadata* to be defined in the headers. In this example we are informing the server we are expecting *application/json* by stating this in the `Accept` header.

The response includes the status code and message from the server, which enables the consumer to interrogate the result of the operation on the server-side resource. In the response body a JSON representation containing the conference attendees is returned. In "REST API Standards and Structure" we will explore approaches to modelling the JSON body for API compatibility.

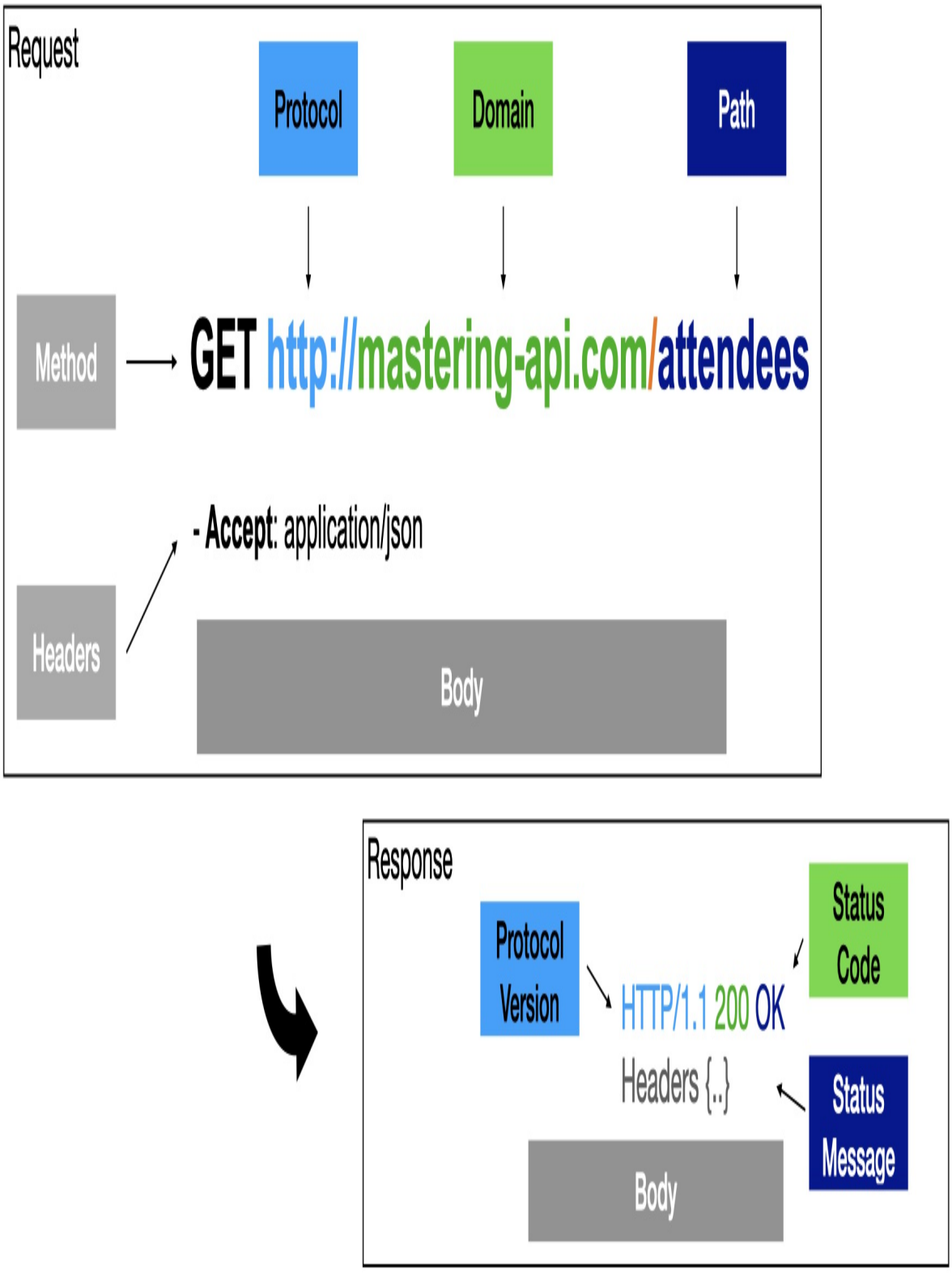


Figure 2-1. Anatomy of a RESTful Request and Response over HTTP

NOTE

Although there is nothing in the REST specification that states HTTP is required, the data elements and their usage were designed with HTTP in mind.

The Richardson Maturity Model

The REST specification does not enforce a high degree of restriction in terms of how developers and architects build and design REST APIs. Speaking at QCon in 2008 Leonard Richardson presented his experiences of reviewing many REST APIs. Martin Fowler also covered Leonard Richardson's maturity model on his blog. The model presents levels of adoption that teams apply to building APIs.

Table 2-1. Richardson Maturity Model Levels

Level 0 - HTTP/RPC	Establishes that the API is built using HTTP and has the notion of one URI and one HTTP method. Taking our example above of `GET /attendees`, if we stopped there the example would match Level 0. Essentially this represents an RPC (Remote Procedure Call) implementation, a consideration we will compare and contrast throughout this chapter.
Level 1 - Resources	Establishes the use of resources, and starts to bring in the idea of modelling resources in the context of the URI. In our example if we added `GET /attendees/1` returning a specific attendee it would start look like a level 1 API. Martin Fowler draws an analogy to the classic object oriented world of introducing identity.
Level 2 - Verbs (Methods)	Starts to introduce the correct modelling of multiple resources URIs accessed by different request methods (also know as HTTP Verbs) based on the effect of the resources on the server. An API at level 2 can make guarantees around `GET` methods not impacting server state and presenting multiple operations on the same resource URI. In our example adding `DELETE /attendees/1`, `PUT /attendees/1` would start to add the notion of a level 2 compliant API.
Level 3 - Hypermedia Controls	This is the epitome of REST design and involves navigable APIs by the use of HATEOAS (Hypertext As The Engine Of Application State). In our example when we call `GET /attendees/1` the response would contain the

	actions that are possible on the object returned from the server. This would include the option to be able to update the attendee or delete the attendee and what the client is required to invoke in order to do so.

Most APIs reach API maturity level 2, but with this wide range of options there is a lot for the developer to consider. The practical applications of HATEOAS are limited in modern RESTful APIs and this is where the theoretical ideal REST concept meets the battle testing of production code. In order to be consistent we need to establish baseline expectations of the APIs implemented using REST and provide that consistency across all the APIs offered.

REST API Standards and Structure

REST has some very basic rules, but for the most part the implementation and design of an API is left as an exercise for the developer. It is useful to have a more practical definition around the APIs to provide a uniformity and expectations across different APIs. This is where REST API Standards or Guidelines can help, however there are a variety of sources to choose from. For the purposes of discussing REST API design we will use the Microsoft REST API Guidelines, which represent a series of internal guidelines that have been OpenSourced to create a dialog in the community. The guidelines use RFC-2119 which defines terminology for standards such as MUST, SHOULD, SHOULD NOT, MUST NOT etc allowing the developer to determine whether requirements are optional or mandatory.

TIP

As REST API Standards are evolving, an open list of API Standards are available on the book's Github. Please contribute via Pull Request any open standards you think would be useful for other readers to consider.

Lets evolve our *attendees* API using the Microsoft REST API Guidelines and introduce an endpoint to create a new `attendee`. If you are familiar with

REST the thought will immediately be to use `POST`, but the recommended response header might not be so obvious.

```
POST http://mastering-api.com/attendees
{
    "displayName": "Jim",
    "givenName": "James",
    "surname": "Gough",
    "email": "jim@mastering-api.com"
}
---
201 CREATED
Location: http://mastering-api.com/attendees/1
```

The *Location* header reveals the location of the new resource created on the server, and in this API we are modelling a unique ID for the user. It is possible to use the *email* field as a unique ID, however the Microsoft REST API guidelines recommend in section 7.9 that PII should **not** be part of the URL.

> **WARNING**
>
> The reason for removing sensitive data from the URL is paths or query parameters might be inadvertently cached in the network, for example in server logs or elsewhere.

Another aspect of APIs that can be difficult is naming, as we will discuss in "API Versioning" something as simple as changing a name can break compatibility. There is a short list of standard names that should be used in the Microsoft Rest API Guidelines, however teams should expand this to have a common domain data dictionary to supplement the standards. Lets now take a look at patterns for retrieving data.

Collections and Pagination

It seems reasonable to model the `GET` */attendees* request as a response containing a raw array. The source snippet below shows an example of what that might look like as a response body.

```
GET http://mastering-api.com/attendees
---
200 OK
[
    {
        "displayName": "Jim",
        "givenName": "James",
        "surname": "Gough",
        "email": "jim@mastering-api.com",
        "id": 1,
    },
    {
        "displayName": "Matt",
        "givenName": "Matthew",
        "surname": "Auburn",
        "email": "matt@mastering-api.com",
        "id": 2,
    }
]
```

Let's consider at an alternative model to the `GET` */attendees* that nests the array of attendees inside an object. It may seem strange that an array response is returned in an object, however the reason for this is that allows for us to model bigger collections and pagination. This is reaping the benefits of hindsight, adding pagination later and converting from an array to a object in order to add a `@nextLink` (as recommended by the standards) would break compatibility.

```
GET http://mastering-api.com/attendees
---
200 OK
{
    "value": [
        {
            "displayName": "Jim",
            "givenName": "James",
            "surname": "Gough",
            "email": "jim@mastering-api.com",
            "id": 1,
        }
    ],
    "@nextLink": "{opaqueUrl}"
}
```

Filtering Collections

Our conference is looking a little lonely with only two attendees, however when collections grow in size we may need to add filtering in addition to pagination. The filtering standard provides an expression language within REST to standardize how filter queries should behave, based upon the OData Standard. For example we could find all attendees with the `displayName` Jim using

```
GET http://mastering-api.com/attendees?$filter=displayName eq 'Jim'
```

It is not necessary to complete all filtering and searching features from the start. Designing an API in line with the standards will allow the developer to support an evolving API architecture without breaking compatibility for consumers.

Updating Data

When designing an API the developer would need to make an active decision on whether to use `PUT` or `PATCH` to update an attendees details. A `PUT` is used to replace the resource entirely with the content of the request, where as a `PATCH` would only update the attributes specified by the request. Depending on the number of fields and expected parallel updates the API should be designed accordingly. For example two `PUT` operations would likely involve a lost update whereas two `PATCH` requests may be successful independently.

Error Handling

An important consideration when extending APIs to consumers is defining what should happen in various error scenarios. Error standards are useful to define upfront and share with API producers to provide consistency. It is important Errors describe to the API Consumer exactly what has gone wrong with the request, this will avoid increased support of the API.

WARNING

Ensure that the error messages sent back to the consumer does not contain stack traces and other sensitive information. This information can help an hacker aiming to compromise the system.

We have just scratched the surface on building REST APIs, but clearly there are important decisions to be made at the beginning of the project to build an API. If we combine the desire to present intuitive APIs that are consistent and allow for an evolving compatible API, it is worth adopting an API Standard early.

Checklist: Choosing an API Standard

Table 2-2. API Standards Checklist

Decision	Which API standard should we adopt?
Discussion Points	Does the organization already have other standards within the company? Can we extend those standards to external consumers? Are we using any third party APIs that we will need to expose to a consumer (e.g. Identity Services) that already have a standard? What does the impact of not having a standard look like for our consumers?
Recommendations	Pick an API standard that best matches the culture of the Organization and formats of APIs you may already have in the inventory. Be prepared to evolve and add to a standard any domain/industry specific amendments. Start with something early to avoid having to break compatibility later for consistency. Be critical of existing APIs, are they in a format that consumers would understand or is more effort required to offer the content?

Specifying REST APIs

As we have seen the design of an API is fundamental to the success of an API platform. The next challenge to overcome is sharing the API with developers consuming our APIs.

API marketplaces provide a public or private listing of APIs available to a consumer. The developer can browse documentation and quickly trial an API in the browser to explore the API behavior and functionality. Public and private API marketplaces have placed REST APIs prominently into the consumer space. Architecturally REST APIs are increasingly important in support of both microservices based architectures and Service Oriented Architectures. The success of REST APIs has been driven by both the technical landscape and the low barrier to entry for both the client and server.

Prior to 2015 there was no standard for specifying REST APIs, which may be quite surprising given that API specifications are not a new concept. XML had XML Schema Definitions (or XSD), which were a core mechanism in ensuring compatibility of services. However, it is important to remember that REST was designed for the web, rather than specifically for APIs. As the number of APIs grew it quickly became necessary to have a mechanism to share the *shape* and structure of APIs with consumers. This is why the OpenAPI Initiative was formed by API industry leaders to construct the OpenAPI Specification (OAS). The OAS was formerly known as Swagger and documentation and implementation use OAS and Swagger interchangeably.

OpenAPI Specification Structure

Let’s explore an example OpenAPI Specification for the attendees API.

```
"openapi": "3.0.3",
"info": {
    "title": "Attendees Mastering API",
    "description": "Example accompanying the Mastering API Book",
    "version": "1.0"
},
"servers": [
    {
        "url": "http://mastering-api.com",
        "description": "Demo Server"
    }
],
```

The specification begins by defining the OAS version, information about the API and the servers the API is hosted on. The `info` attribute is often used for top level documentation and for specifying the version of the API. The version of the API is an important attribute, which we will discuss in more detail in “API Versioning”. The `servers` array is one of the new additions in OpenAPI Specification 3, prior to this only a single host could be represented. The `openapi` object key is named `swagger` in older versions of the specification.

> **NOTE**
>
> As well as defining the shape of an API the OpenAPI Specification often conveys full documentation about the API.

```
    "paths": {
        "/attendees": {
            "get": {
                "tags": [
                    "attendees-controller"
                ],
                "summary": "Retrieve a list of all attendees stored within the
system",
                "operationId": "getAttendees",
                "responses": {
                    "200": {
                        "description": "OK",
                        "content": {
                            "*/*": {
                                "schema": {
                                    "$ref":
"#/components/schemas/AttendeeResponse"
                                }
                            }
                        }
                    }
                }
            }
        }
    },
```

The `paths` tag conveys the possible operations on the RESTful API and the

expected request and response object formats. In this example on a 200 status response the consumer can expect to receive an object `AttendeeResponse`. The `components` object will describe the response has a key `value` containing the `Attendee` array. The `$ref` tag indicates that this will be represented elsewhere in the specification.

```
"components": {
    "schemas": {
        "Attendee": {
            "title": "Attendee",
            "required": [
                "email",
                "givenName",
                "surname",
                "displayName"
            ],
            "type": "object",
            "properties": {
                "displayName": {
                    "type": "string"
                },
                "email": {
                    "maxLength": 254,
                    "minLength": 0,
                    "type": "string"
                },
                "givenName": {
                    "maxLength": 35,
                    "minLength": 0,
                    "type": "string"
                },
                "id": {
                    "type": "integer",
                    "format": "int32"
                },
                "surname": {
                    "maxLength": 35,
                    "minLength": 0,
                    "type": "string"
                }
            }
        },
        "AttendeeResponse": {
            "title": "AttendeeResponse",
            "type": "object",
```

```
                "properties": {
                    "value": {
                        "type": "array",
                        "items": {
                            "$ref": "#/components/schemas/Attendee"
                        }
                    }
                }
            }
        }
    }
}
```

> **NOTE**
>
> In addition to documentation the author can also supply example responses to demonstrate how the API should be used.

`Components` holds the various object schemas [1] for the specification and in this case defines our `Attendee` and `AttendeeResponse` object.

OpenAPI specifications can also include a wide range of additional metadata and useful features for developers and API consumers. In the Attendees example the required fields of an Attendee are all fields except for `id`, at the time of generating an `Attendee` the consumer does not know the id. The specification also sets out maximum lengths for some of the strings, it also possible to set a regular expression to pattern match `email`. Full details of the OpenAPI Specification are hosted on the book's GitHub.

Example requests and responses, like the one we've shown here, demonstrate a typical data exchange supported by the API. The OAS also documents the OAuth2 flows that are supported by an API, which we will explore further in Chapter 9. Over the course of the chapter it should become clear how important the OpenAPI Specification is to offering any type of REST API platform.

Visualizing OpenAPI Specifications

It’s quite difficult to read a specification in JSON or in YAML (which is also supported by OAS), especially as APIs grow beyond a handful of operations. The example specification above includes no user documentation. When documentation is added specifications can rapidly become thousands of lines long, which makes the specification difficult to read without a tool. One of the big success stories of OpenAPI Specifications has been the number of tools available in many different languages. There are tools that enable the developer to generate OpenAPI Specifications directly from their code or use the Swagger Editor.

Practical Application of OpenAPI Specifications

Once an OpenAPI Specification is shared the power of the specification starts to become apparent. OpenAPI.Tools documents a full range of open and closed source tools available. In this section we will explore some of the practical applications of tools based on their interaction with the OpenAPI Specification.

Code Generation

Perhaps one of the most useful features of an OpenAPI specification is allowing the generation of client side code to consume the API. As discussed in “Specifying REST APIs” we can include the full details of the server, security and of course the API structure itself. With all this information we can generate a series of model and service objects that represent and invoke the API. The OpenAPI Generator project supports a wide range of languages and tool chains. For example, in Java you can choose to use Spring or JAX-RS and in Typescript you can choose a combination of Typescript with your favorite framework. It is also possible to generate the API implementation stubs from the OpenAPI Specification.

This raises an important question about what should come first the specification or the server side code? In the next chapter we are going to

discuss “API Contracts” which presents a behavior driven approach to testing and building APIs. The challenge with OpenAPI Specifications is that alone they only convey the shape of the API. OpenAPI specifications do not fully model the semantics (or expected behavior of the API) under different conditions. If you are going to present an API to external users it is important that the range of behaviors is modelled and tested to help avoid having to drastically change the API later.

A common challenge with API modelling, as discussed in “The Richardson Maturity Model”, is determining whether you need a RESTful API or whether you need RPC. We will explore this idea further in “Alternative API Formats”. It is important that this is an active decision, as delivering RPC over REST can result in a modelling mistake and a challenge for consumers. APIs should be designed from the perspective of the consumer and abstract away from the underlying representation behind the scenes. It is important to be able to freely refactor components behind the scenes without breaking API compatibility, otherwise the API abstraction loses value.

OpenAPI Validation

OpenAPI Specifications are useful for validating the content of an exchange to ensure the request and response match the expectations of the specification. At first it might not seem apparent where this would be useful, if code is generated surely the exchange will always be right? One practical application of OpenAPI validation is in securing APIs and API infrastructure. In many organizations a zonal architecture is common, with a notion of a DMZ (Demilitarized Zone) used to shield a network from inbound traffic. A useful feature is to interrogate messages in the DMZ and terminate the traffic if the specification does not match.

Atlassian, for example, Open Sourced a tool called the swagger-request-validator, which is capable of validating JSON REST content. The project also has adapters that integrate with various mocking and testing frameworks to help ensure that API Specifications are conformed to as part of testing. The tool has an `OpenApiInteractionValidator` which is used to create a `ValidationReport` on an exchange.

```
//Using the location of the specification create an interaction validator
//The base path override is useful if the validator will be used behind a
gateway/proxy
final OpenApiInteractionValidator validator = OpenApiInteractionValidator
        .createForSpecificationUrl(specUrl)
        .withBasePathOverride(basePathOverride)
        .build;

//Requests and Response objects can be converted or created using a builder
final ValidationReport report = validator.validate(request, response);

if (report.hasErrors()) {
    // Capture or process error information
}
```

Examples and Mocking

The OpenAPI Specification can provide example responses for the paths in the specification. Examples, as we've discussed, are useful for documentation to help developers understand the API usage. Some products have started to use examples to allow the user to query the API and return example responses from a mock service. This can be really useful in features such as a Developer Portal, which allows developers to explore documentation and invoke APIs.

Examples can potentially introduce an interesting problem, which is that this part of the specification is essentially a string (in order to model XML/JSON etc). openapi-examples-validator [2] validates that an example matches the OpenAPI Specification for the corresponding request/response `component` of the API.

Detecting Changes

OpenAPI Specifications can also be useful in detecting changes in an API. This can be incredibly useful as part of a DevOps pipeline. Detecting changes for backwards compatibility is incredibly important, but first it is useful to understand versioning of APIs in more detail.

API Versioning

We have explored the advantages of sharing an OpenAPI specification with a consumer, including the speed of integration. Consider the case where multiple consumers start to operate against the API. What happens when there is a change to the API or one of the consumers requests the addition of new features to the API?

Let's take a step back and think about if this was a code library built into our application at compile time. Any changes to the library would be packaged as a new version and until the code is recompiled and tested against the new version, there would be no impact to production applications. As APIs are running services, we have a couple of upgrade options that are immediately available to us when changes are requested:

- **Release a new version and deploy in a new location.** Older applications continue to operate against the older version of the APIs. This is fine from a consumer perspective, as the consumer only upgrades to the new location and API if they need the new features. However, the owner of the API needs to maintain and manage multiple versions of the API, including any patching and bug fixing that might be necessary.
- **Release a new version of the API that is backwards compatible with the previous version of the API.** This allows additive changes without impacting existing users of the API. There are no changes required by the consumer, but we may need to consider downtime or availability of both old and new versions during the upgrade. If there is a small bug fix that changes something as small as an incorrect fieldname, this would break compatibility.
- **Break compatibility with the previous API and all consumers must upgrade code to use the API.** This seems like an awful idea at first, as that would result in things breaking unexpectedly in production.[3] However a situation may present itself where we cannot avoid breaking compatibility with older versions. One example

where APIs have had to break compatibility for a legal reasons was the introduction of GDPR (General Data Protection Regulation) in Europe.

The challenge is that each of these different upgrade options offer advantages, but also drawbacks either to the consumer or the API owner. The reality is that we want to be able to support a combination of all three options. In order to do this we need to introduce rules around versioning and how versions are exposed to the consumer.

Semantic Versioning

Semantic Versioning offers an approach that we can apply to REST APIs to give us a combination of the above. Semantic versioning defines a numerical representation attributed to an API release. That number is based on the change in behavior in comparison to the previous version, using the following rules.

- A **Major** version introduces non-compatible changes with previous versions of the API. In an API platform upgrading to a new major version is an active decision by the consumer. There is likely going to be a migration guide and tracking as consumers upgrade to the new API.
- A **Minor** version introduces a backwards compatible change with the previous version of the API. In an API service platform it is acceptable for consumer to receive minor versions without making an active change on the client side.
- A **Patch** version does not change or introduce new functionality, but is used for bug fixes on an existing Major.Minor version of functionality.

Formatting for semantic versioning can be represented as `Major.Minor.Patch`. For example 1.5.1 would represent major version 1, minor version 5 with patch upgrade of 1. Whilst reading the above the reader may have noticed that with APIs running as services there is another

important aspect to the story. Versioning alone is not enough, an element of deployment and what is exposed to the consumer at what time is part of the challenge. This is where the API Lifecycle is important to consider, n terms of versioning.

API Lifecycle

The API space is moving quickly, but one of the clearest representations of version lifecycle comes from the now archived PayPal API Standards. The lifecycle is defined as follows:

Table 2-3. API Lifecycle (adapted from PayPal API Standards)

Planned	Exposing an API from a technology perspective is quite straightforward, however once it is exposed and consumed we have multiple parties that need to be managed. The planning stage is about advertising that you are building an API to the rest of the API program. This allows a discussion to be had about the API and the scope of what it should cover.
Beta	Involves releasing a version of our API for users to start to integrate with, however this is generally for the purpose of feedback and improving the API. At this stage the producer reserves the right to break compatibility, it is not a versioned API. This helps to get rapid feedback from consumers about the design of the API before settling on a structure. A round of feedback and changes enables the producer to avoid having many major versions at the start of the APIs lifetime.
Live	The API is now versioned and live in production. Any changes from this point onward would be versioned changes. There should only ever be one live API, which marks the most recent major/minor version combination. Whenever a new version is released the current live API moves to deprecated.
Deprecated	When an API is deprecated it is still available for use, but significant new development should not be carried out against it. When a minor version of a new API is released an API will only be deprecated for a short time, until validation of the new API in production is complete. After the new version is successfully validated a minor version moves to retired, as the new version is backwards compatible and can handle the same features as

	the previous API. When a major version of the API is released the older version becomes deprecated. It is likely that will be for weeks or months, as an opportunity must be given to consumers to migrate to the new version. There is likely going to be communication with the consumers, a migration guide and tracking of metrics and usage of the deprecated API.
Retired	The API is retired from production and is no longer accessible.

The lifecycle helps the consumer fully understand what to expect from the API. The main question is what does the consumer see with respect to the versioning and lifecycle? With Semantic Versioning combined with the API Lifecycle the consumer only needs to be aware of the major version of the API. Minor and patch versions will be received without updates required on the consumers side and won't break compatibility.

One often controversial question is how should the major version be exposed to the user. One way is to expose the major version in the URL i.e. *http://mastering-api.com/v1/attendees*. From a purely RESTful perspective however the version is not part of the resource. Having the major version as part of the URL makes it clear to the consumer what they are consuming. A more RESTful way is to have the major version as part of the header, e.g. VERSION: 1. Having the version in a header may be slightly hidden from the consumer. A decision would need to be made to be consistent across APIs.

You may be wondering how APIs with multiple versions can be presented side-by-side during deployments and route to specific API services. We will explore this further in Chapter 4 and in Chapter 5.

OpenAPI Specification and Versioning

Now that we have explored versioning we can look at examples of breaking changes and non breaking changes using the attendees API specification. There are several tools to choose from to compare specifications, in this example we will use openapi-diff from OpenAPITools.

We will start with a breaking change. We will change `givenName` to be a

field called `firstName`. We can run the diff tool from a docker container using the following command:

```
$docker run --rm -t \
   -v $(pwd):/specs:ro \
   openapitools/openapi-diff:latest /specs/original.json /specs/first-name.json
================================================================================
==                            API CHANGE LOG                                  ==
================================================================================
                          Attendees Mastering API
--------------------------------------------------------------------------------
                             What's Changed                                  --
--------------------------------------------------------------------------------
- GET    /attendees
  Return Type:
    - Changed 200 OK
      Media types:
        - Changed */*
          Schema: Broken compatibility
          Missing property: [n].givenName (string)
--------------------------------------------------------------------------------
                                 Result                                     --
--------------------------------------------------------------------------------
                API changes broke backward compatibility
```

TIP

The `-v $(pwd):/specs:ro` adds the present working directory to the container under the /specs mount as read only.

We can try to add a new attribute to the `/attendees` return type to add an additional field `age`. Adding new fields does not break existing behavior and therefore does not break compatibility.

```
$ docker run --rm -t \
 -v $(pwd):/specs:ro \
openapitools/openapi-diff:latest --info /specs/original.json /specs/age.json
================================================================================
==                            API CHANGE LOG                                  ==
================================================================================
                          Attendees Mastering API
--------------------------------------------------------------------------------
```

```
                          What's Changed                                --
-------------------------------------------------------------------------
- GET    /attendees
  Return Type:
    - Changed 200 OK
      Media types:
        - Changed */*
          Schema: Backward compatible
-------------------------------------------------------------------------
                                Result                                  --
-------------------------------------------------------------------------
                  API changes are backward compatible
```

It is worth trying this out to see what would be compatible changes and what would not. Introducing this type of tooling as part of the API pipeline is going to help avoid unexpected non compatible changes for consumers. OpenAPI specifications are an important part of an API program, and when combined with tooling, versioning and lifecycle they are invaluable.

Alternative API Formats

REST APIs work incredibly well for extending services to external consumers. From the consumer perspective the API is clearly defined, won't break unexpectedly and all major languages are supported. But, is using a REST API for every exchange in a microservices based architecture the right approach? For the remainder of the chapter we will discuss the various API formats available to us and factors that will help determine the best solution to our problem.

Remote Procedure Call (RPC)

Remote Procedure Calls (RPC) are definitely not a new concept. RPC involves executing code or a function of another process. It is an API, but unlike a REST API it generally exposes the underlying system or function internals. With RPC the model tends to convey the exact functionality at a method level that is required from a secondary service.

RPC is different from REST as REST focuses on building a model of the

domain and extending an abstraction to the consumer. REST hides the system details from the user, RPC exposes it. RPC involves exposing a method from one process and allows it to be called directly from another.

gRPC is a modern open source high performance Remote Procedure Call (RPC). gRPC is under stewardship of the Linux Foundation and is the defacto standard for RPC across most platforms. Figure 2-2 describes an RPC call in gRPC, which involves the Schedule Service invoking the remote method on the Attendees Service. THe gRPC Attendees Service creates a server, allowing methods to be invoked remotely. On the client side, the Schedule Service, a stub is used to abstract the complexity of making the remote call into the library.

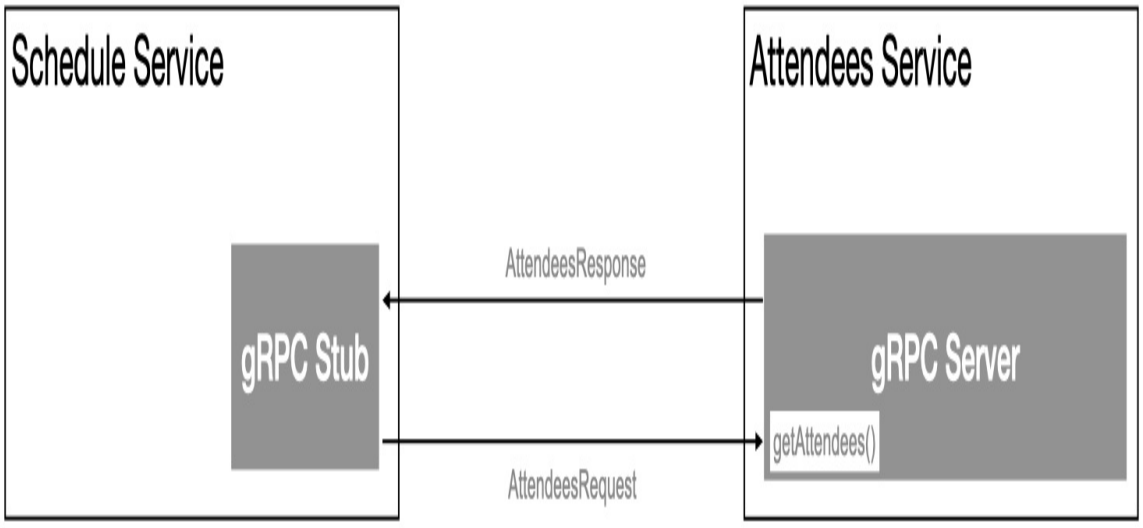


Figure 2-2. Example Attendees with RPC using gRPC

Another key difference between REST and RPC is state, REST is by definition stateless - with RPC state depends on the implementation. The authors have seen huge systems built around SOAP, which was the XML-RPC successor of the noughties! Many systems have been built around content based routing of messages to specific services with the right cached state to handle the request. In state based RPC systems the developer must have a detailed understanding of each exchange and expected behavior. In order to scale, systems start to leak business logic into routing, which if not

carefully managed can lead to increased complexity.

Implementing RPC with gRPC

The Attendees service could model either a North→South or East→West API. In addition to modelling a REST API we are going to evolve the Attendees service to support gRPC.

> **TIP**
>
> East→West such as Attendees tend to be higher traffic, and can be implemented as microservices used across the architecture. gRPC may be a more suitable tool than REST for East→West services, owing to the smaller data transmission and speed within the ecosystem. Any performance decisions should always be measured in order to be informed.

Let's explore using a Spring Boot Starter to rapidly create a gRPC server. The Java code below demonstrates a simple structure for implementing the behavior on the generated gRPC server classes.

```
@GrpcService
public class AttendeesServiceImpl extends
AttendeesServiceGrpc.AttendeesServiceImplBase {

    @Override
    public void getAttendees(AttendeesRequest request,
        StreamObserver<AttendeeResponse> responseObserver) {
          AttendeeResponse.Builder responseBuilder =
AttendeeResponse.newBuilder();

          //populate response
          responseObserver.onNext(responseBuilder.build());
          responseObserver.onCompleted();
    }
}
```

The following `.proto` file defines an empty request and returns a repeated `Attendee` response. In protocols used for binary representations it is important to note that position and order of fields is important, as they govern the layout of the message. Adding a new service or new method is backward

compatible as is adding a field to a message, but care is required. Removing a field or renaming a field will break compatibility, as will changing the datatype of a field. Changing the field number is also an issue as field numbers are used to identify fields on the wire. The restrictions of encoding with gRPC mean the definition must be very specific. REST and OpenAPI are quite forgiving as the specification is only a guide [4]. Extra fields and ordering do not matter in OpenAPI, versioning and compatibility is therefore even more important when it comes to gRPC.

The following `.proto` file models the same `attendee` object that we explored in our OpenAPI Specification example.

```
syntax = "proto3";
option java_multiple_files = true;
package com.masteringapi.attendees.grpc.server;

message AttendeesRequest {
}

message Attendee {
  int32 id = 1;
  string givenName = 2;
  string surname = 3;
  string email = 4;

}

message AttendeeResponse {
  repeated Attendee attendees = 1;
}

service AttendeesService {
  rpc getAttendees(AttendeesRequest) returns (AttendeeResponse);
}
```

The Java service modelling this example can be found on the Book GitHub page. gRPC cannot be queried directly from a browser without additional libraries, however you can install gRPC UI to use the browser for testing. `grpcurl` also provides a command line tool:

```
$ grpcurl -plaintext localhost:9090 \
```

```
    com.masteringapi.attendees.grpc.server.AttendeesService/getAttendees
{
  "attendees": [
    {
      "id": 1,
      "givenName": "Jim",
      "surname": "Gough",
      "email": "gough@mail.com"
    }
  ]
}
```

gRPC gives us another option for querying our service and defines a specification for the consumer to generate code. gRPC has a more strict specification than OpenAPI and requires methods/internals to be understood by the consumer.

GraphQL

RPC offers access to a series of individual functions provided by a producer, but does not usually extend a model or abstraction to the consumer. REST extends a resource model for a single API provided by the producer. It is possible to offer multiple APIs on the same base URL by combining REST APIs together using API Gateways. We will explore this notion further in Chapter 4. If we offer multiple APIs in this way the consumer will need to query the APIs sequentially to build up state on the client side. This approach is also wasteful if the client is only interested in a subset of fields on the API response. Consider a user interface that models a dashboard of data on our conference system using visual tiles. Each individual tile would need to invoke each API to populate the UI to display the tile content the user is interested in. Figure 2-3 shows the number of invocations required.

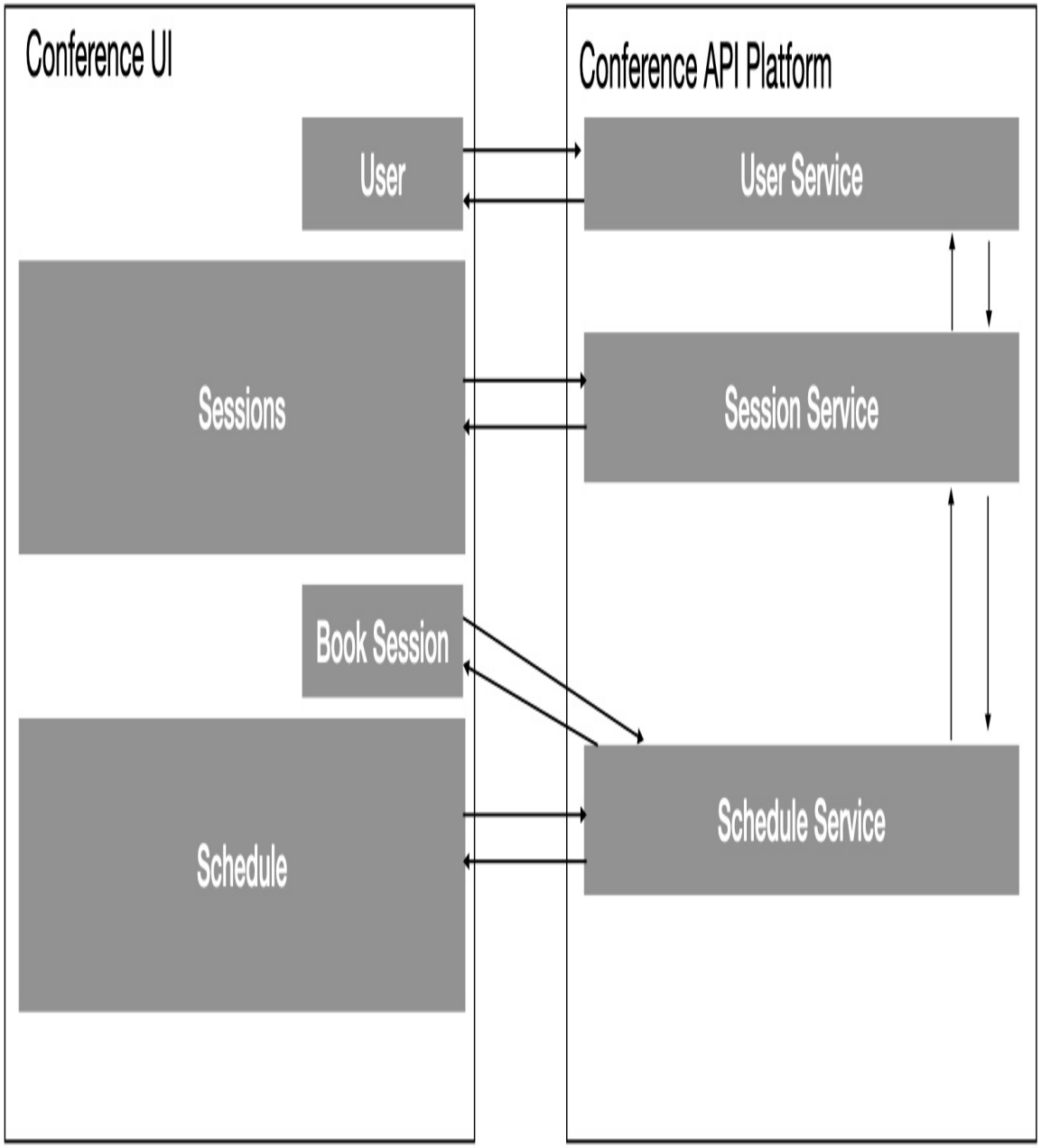


Figure 2-3. Example User Interface model

GraphQL introduces a technology layer over existing services, data stores and APIs that provides a query language to query across multiple sources. The query language allows the client to ask for exactly the fields required, including fields that span across multiple APIs.

GraphQL introduces the GraphQL schema language, which is used to specify

the types in individual APIs and how APIs combine. One major advantage of introducing a GraphQL schema is the ability to provide a single version across all APIs, removing the need for potentially complex version management on the consumer side.

After defining the schema the next consideration is the implementation of behavior and defining how data is retrieved and if necessary converted to match the schema. A `Resolver` is a function that a GraphQL server implementor creates to source the data for data elements in the GraphQL schema. Mark Stuart has written an excellent blog on GraphQL resolver best practices for PayPal Engineering.

> **TIP**
>
> One mistake that API developers and architects often make is assuming that GraphQL is only a technology used with User Interfaces. In systems where vast amounts of data is stored across different subsystems GraphQL can provide an ideal solution to abstracting away internal system complexity.

Lets implement a very trivial GraphQL schema for the `/attendees` call to look at what GraphQL looks like to the consumer.

```
var schema = buildSchema(`
  type Attendee {
    givenName: String
    surname: String
    displayName: String
    email: String
    id: Int
  }

  type Query {
    attendees: [Attendee]
  }
`);

// Logic to resolve and fetch content

var app = express();
app.use('/graphql', graphqlHTTP({
```

```
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);
```

GraphQL has a single `POST /graphql` endpoint (not too dissimilar from RPC), however unlike RPC it is not single a method with set behavior that is invoked but a declarative request for specific elements of the schema.

```
curl -X POST -H "Content-Type: application/json" \
    -d '{"query": "{ attendees { email } }"}' http://localhost:4000/graphql

{"data":{"attendees":[{"email":"jpgough@gmail.com"}]}}
```

Figure 2-4 shows how GraphiQL (a UI tool for GraphQL) provides the consumer with a graphical mechanism for building up queries, along with the ability to explore the schema data types.

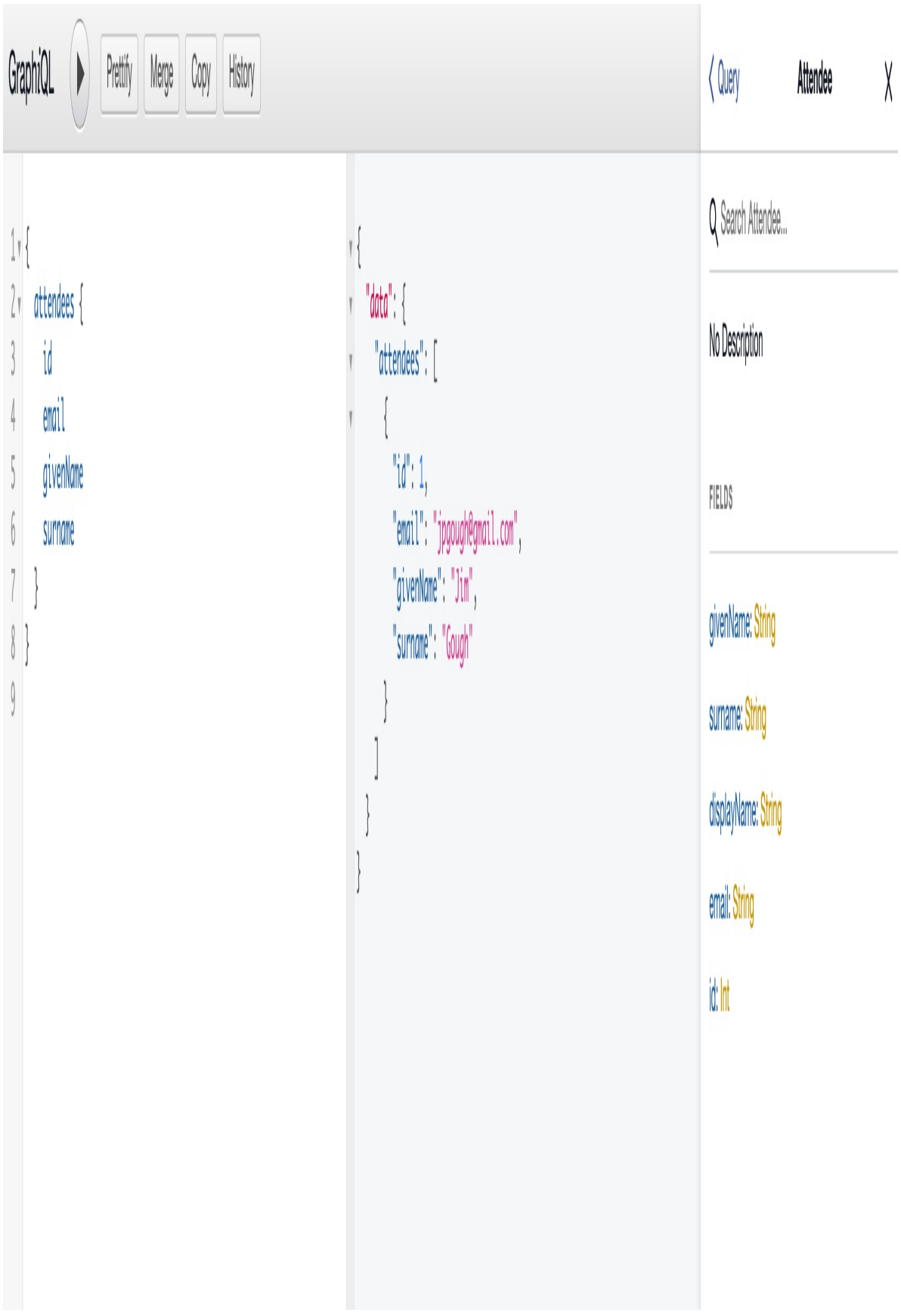

GraphiQL
Prettify
Merge
Copy
History
Query
Attendee
{
attendees {
id
email
givenName
surname
}
}
{
"data": {
"attendees": [
{
"id": 1,
"email": "jpgough@gmail.com",
"givenName": "Jim",
"surname": "Gough"
}
]
}
}
Search Attendee...
No Description
FIELDS
givenName: String
surname: String
displayName: String
email: String
id: Int

Figure 2-4. Executing a Query from the GraphiQL UI

GraphQL is a fascinating technology and offers a complement to REST and RPC and in some cases will be a better choice. Learning GraphQL by Eve Porcello and Alex Banks offers an in-depth exploration for the curious architect. GraphQL works very well when the data and services that a team or company present are from a specific business domain. In the case where disparate APIs are presented externally GraphQL could introduce a complicated overhead to maintain if it tried to do too much. Whilst you can use GraphQL to normalize access to a domain, maintenance may be reduced if the services behind the scenes have already been normalized.

Exchanges and Choosing an API Format

Earlier, we discussed the concept of traffic patterns, and the difference between requests originating from outside the ecosystem and requests within the ecosystem. Traffic patterns are an important factor in determining the appropriate format of API for the problem at hand. When we have full control over the services and exchanges within our microservices based architecture, we can start to make compromises that we would not be able to make with external consumers.

It is important to recognize that the performance characteristics of an East→West service are likely to be more applicable than a North→South service. In a North→South exchange traffic originating from outside the API producer's environment will generally involve the exchange using the internet. The internet introduces a high degree of latency, and an API architect should always consider the compounding effects of each service. In a microservices based architecture it is likely that one North→South request will involve multiple East→West exchanges. High traffic East→West exchange need to be efficient to avoid cascading slow-downs propagating back to the consumer.

High Traffic Services

In our example Attendees is a central service. In a microservices based architecture components will keep track of an `attendeeId`. APIs offered to consumers will potentially retrieve data stored in the Attendees service, and at scale it will be a high traffic component. If the exchange frequency is high between services, the cost of network transfer due to payload size and limitations of one protocol vs another will be more profound as usage increases. The cost can present itself in either monetary costs of each transfer or the total time taken for the message to reach the destination.

Large Exchange Payloads

Large payload sizes may also become a challenge in API exchanges and are susceptible to increasing transfer performance across the wire. JSON over REST is human readable, and will often be more verbose than a fixed or binary representation.

> **TIP**
>
> A common misconception is that “human readability” is quoted as a primary reason to use JSON in data transfers. The number of times a developer will need to read a message vs the performance consideration is not a strong case with modern tracing tools. It is also rare that large JSON files will be read from beginning to end. Better logging and error handling can mitigate the human readable argument.

Another factor in large payload exchanges is the time taken by components to parse the message content into language level domain objects. Performance time of parsing data formats varies vastly depending on the language a service is implemented in. Many traditional server side languages can struggle with JSON compared to a binary representation for example. It is worth exploring the impact of parsing and include that consideration when choosing an exchange format.

HTTP/2 Performance Benefits

Using HTTP/2 based services can help to improve performance of exchanges

by supporting binary compression and framing. The binary framing layer is transparent to the developer, but behind the scenes will split and compress the message into smaller chunks. The advantage of binary framing is it allows for a full request and response multiplexing over a single connection. Consider processing a list in another service and the requirement is to retrieve 20 different attendees, if we retrieved these as individual HTTP/1 requests it would require the overhead of creating 20 new TCP connections. Multiplexing allows us to perform 20 individual requests over a single HTTP/2 connection.

gRPC uses HTTP/2 by default and reduces the size of exchange by using a binary protocol. If bandwidth is a concern or cost gRPC will provide an advantage, in particular as content payloads increase significantly in size. gRPC may be beneficial compared to REST if payload bandwidth is a cumulative concern or the service exchanges large volumes of data. If large volumes of data exchanges are frequent it is also worth considering some of the asynchronous capabilities of gRPC, which we will cover in Chapter 10.

Vintage Formats

Not all services in an architecture will be based on a modern design. In Chapter 6 we will look at how to isolate and evolve vintage components, however as part of an evolving architecture older components will be an active consideration. Many older services will use formats such as SOAP/XML over HTTP/TCP. It is important that an API architect understands the overall performance impact of introducing vintage components.

Performance Testing Exchanges

Recognizing the performance characteristics of exchanges is a useful skill for an API Architect to develop. Often it is not the network alone that needs to be considered - the test should include the network, parsing, responding to the query and returning a response. Smaller benchmarks do not capture the full picture, so it is important to look at the performance in the context of your

system. Let’s explore the approach of a simple end-to-end performance test in the gRPC ecosystem.

Performance is at the heart of every build of the gRPC libraries, and a Performance Dashboard monitors each build and the impact of changes on performance. Buying into the gRPC ecosystem will provide a full stack of complementing libraries that work together in the target language for building services.

If performance is a primary concern for a service it is important to build a benchmark that can be used to test changes to code and libraries over time. We can use a gRPC benchmarking tool `ghz` to get an idea of the performance of the attendees service.

```
brew install ghz
ghz --insecure --format=html --total=10000 \
    --proto ./attendees.proto \
    --call com.masteringapi.attendees.grpc.server.AttendeesService.getAttendees
\
    -d {} localhost:9090 > results.html
```

Figure 2-5 shows a graphical representation of the the performance of 10,000 requests. The average response time was 2.49 milliseconds, the slowest response time was 14.22 milliseconds and the fastest was 0.16ms.

Summary • Histogram • Latency Distribution • Status Distribution • Data

Summary

Count	10000
Total	519.74 ms
Slowest	14.22 ms
Fastest	0.16 ms
Average	2.49 ms
Requests / sec	19240.53

Options


```
{
  "host": "localhost:9090",
  "proto": "./attendees.proto",
  "import-paths": [
    "."
  ],
  "call": "com.masteringapi.attendees.grpc.server.AttendeesService.getAttendees",
  "insecure": true,
  "total": 10000,
  "concurrency": 50,
  "connections": 1,
  "timeout": 20000000000,
  "dial-timeout": 10000000000,
  "data": {},
  "binary": false,
  "CPUs": 8
}
```


Histogram

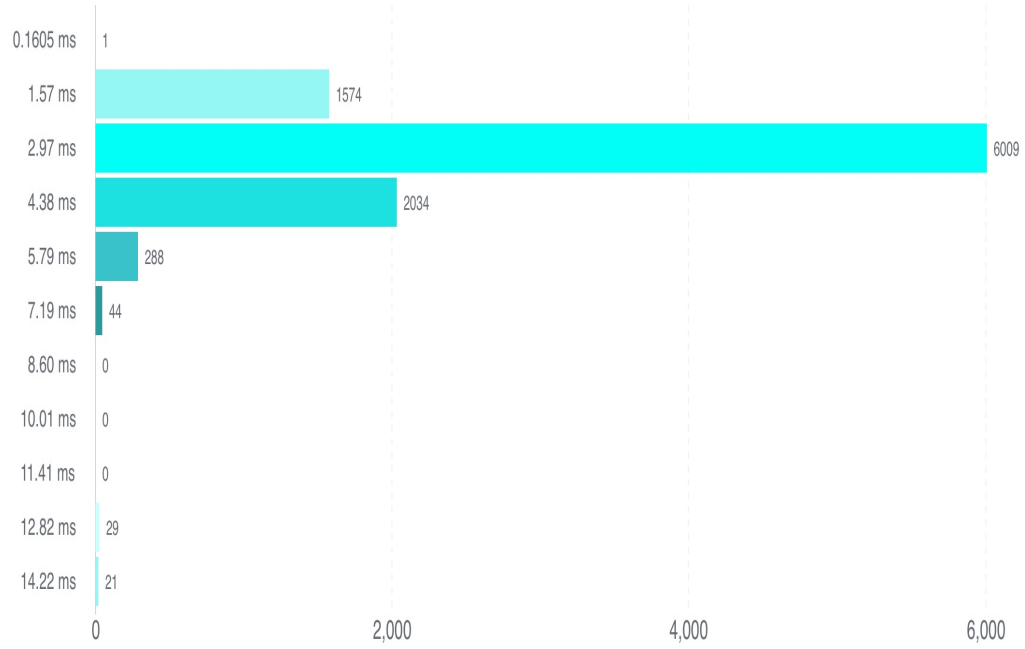


Figure 2-5. GHZ gRPC Benchmark Tool - https://github.com/bojand/ghz

> **TIP**
>
> We want to avoid premature optimization and benchmarks without analysis can lead to a confirmation bias. gRPC will provide performance benefits, but it is important to consider the consumer and their expectations of the API.

gRPC also supports asynchronous and streaming APIs, we will spend Chapter 10 discussing asynchronous approaches to APIs. If services are constantly exchanging information an open asynchronous pipe would offer advantages over an individual request/response model. In Chapter 5 we will explore alternative approaches to testing and monitoring the behavior of applications and exchanges in production.

Checklist: Modelling Exchanges

Table 2-4. Modelling Exchanges Checklist

Decision	What format should we use to model the API for our service?
Discussion Points	Is the exchange a North→South or East→West exchange? Are we in control of the consumer code? Is there a strong business domain across multiple services or do we want to allow consumers to construct their own queries? What versioning considerations do we need to have? What is the deployment/change frequency of the underlying data model. Is this a high traffic service where bandwidth or performance concerns have been raised?
Recommendations	If the API is consumed by external users REST is the lowest barrier to entry and provides a strong domain model. If the APIs offered connect well together and users are likely to use the API to query across APIs frequently consider using GraphQL. If the API is interacting between two services under close control of the producer or the service is proven to be high traffic consider gRPC.

Multiple Specifications

In this chapter we have explored a variety of API formats for an API Architect to consider and perhaps the final question is *"Can we provide all formats?"*. The answer is yes we can support an API that has a RESTful presentation, a gRPC service and connections into a GraphQL schema. However, it is not going to be easy and may not be the right thing to do. In this final section we will explore some of the options available for a multi-format API and the challenges it can present.

The Golden Specification

The `.proto` file for attendees and the OpenAPI specification do not look too dissimilar, they contain the same fields and both have data types. Is it possible to generate a `.proto` file from an OpenAPI specification using the openapi2proto tool? Running `openapi2proto --spec spec-v2.json` will output the `.proto` file with fields ordered alphabetically by default. This is fine until we add a new field to the OpenAPI specification that is backwards compatible and suddenly the ID of all fields changes, breaking backwards compatibility.

The sample `.proto` file below shoes that adding `a_new_filed` would be alphabetically added to the beginning, changing the binary format and breaking existing services.

```
message Attendee {
    string a_new_field = 1;
    string email = 2;
    string givenName = 3;
    int32 id = 4;
    string surname = 5;
}
```

OpenAPI specifications support the idea of extensions, and by using the openapi2proto specific OpenAPI extensions it is possible to generate the compatibility between the two mechanisms of specification.

> **NOTE**
>
> There are other tools available to solve the specification conversion problem, however it is worth noting that some tools only support OpenAPI Specification version 2. The time taken to move between version 2 and 3 in some of the tools built around OpenAPI has led to many products needing to support both versions of the OAS.

An alternative option is grpc-gateway, which generates a reverse-proxy providing a REST facade in front of the gRPC service. The reverse proxy is generated at build time against the `.proto` file and will produce a best effort mapping to REST, similar to `openapi2proto`. You can also supply extensions within the `.proto` file to map the RPC methods to a nice representation in the OpenAPI specification.

```
import "google/api/annotations.proto";
//...
service AttendeesService {
  rpc getAttendees(AttendeesRequest) returns (AttendeeResponse) {
        option(google.api.http) = {
                get: "/attendees"
        };
}
```

Using grpc-gateway gives us another option for presenting both a REST and gRPC service. However, grpc-gateway involves several commands and setup that would only be familiar to developers who work with the go language or build environment.

Challenges of Combined Specifications

It’s important to take a step back here and consider what we are trying to do. When converting from OpenAPI we are effectively trying to convert our RESTful representation into a gRPC series of calls. We are tying to covert an extended hypermedia domain model into a lower level function to function call. This is a potential conflation of the difference between RPC and APIs and is likely going to result in wrestling with compatibility.

With converting gRPC to OpenAPI we have a similar issue, the objective is trying to take gRPC and make it look like a REST API. This is likely going to create a difficult series of issues when evolving the service.

Once specifications are combined or generated from one another, versioning becomes a challenge. It is important to be mindful of how both the gRPC and OpenAPI specifications maintain their individual compatibility requirements. An active decision should be made as to whether coupling the REST domain to an RPC domain makes sense and adds overall value. Rather than generate RPC for East→West from North→South, what makes more sense is to carefully design the microservices based architecture (RPC) communication independently from the REST representation, allowing both APIs to evolve freely.

GraphQL offers a mechanism that is version-less from the consumers perspective, they interact with only the data that they wish to retrieve. This is at the cost to the producer in maintaining a GraphQL Schema and logic that is used to fetch and resolve data from the underlying services. It is possible to offer REST APIs to external users and then use the separate GraphQL server to aggregate together APIs that have combined domains. It is also possible to use GraphQL to present RPC based services in a normalized schema to clients.

Summary

In this chapter we have scratched the surface of a variety of topics that an API Architect, API developer or individuals involved in an API program perspective must appreciate.

- The barrier to building a REST API is really low [5] in most technologies.
- REST is a fairly loose standard and for building APIs, conforming to an agreed API Standards ensures our APIs are consistent and have the expected behavior for our consumers.

- OpenAPI specifications are a useful way of sharing API structure and automating many coding related activities. Teams should actively select OpenAPI features within their platform and choose what tooling or generation features will be applied to projects.
- Versioning is an important topic that adds complexity for the API producer but is necessary to ease API usage for the API consumer. Not planning for versioning in APIs exposed to consumers is dangerous. Versioning should be an active decision in the product feature set and a mechanism to convey versioning to consumers should be part of the discussion. Versioning alone is usually not enough and ensuring we have an API Lifecycle to help govern versioning will lead to a successful API offering.
- REST is great, but is it not always the best option especially when traffic patterns and performance concerns are factored in. It is important to consider how we approach and model exchanges of information in our microservices based architecture. gRPC and GraphQL provide options that need to be considered when we design our exchange modelling.
- Modelling multiple specifications starts to become quite tricky, especially when generating from one type of specification to another. Versioning complicates matters further but is an important factor to avoid breaking changes. Teams should think carefully before combining RPC representations with RESTful API representations, as there are fundamental differences in terms of usage and control over the consumer code.

The challenge for an API architect is to drive the requirements from a consumer business perspective, create a great developer experience around APIs, and avoid unexpected compatibility issues.

---

1 The schema object is an extended subset of the JSON Schema Specification Wright Draft 00.

2 *https://github.com/codekie/openapi-examples-validator*

3 The authors have been in this situation many times, usually first thing on a Monday!

4 Validation of OpenAPI specifications at runtime helps enforce a greater strictness of OpenAPI Specifications.

5 Anecdotally, whilst researching this chapter some developers claim it is 10 minutes.

Chapter 3. Testing APIs

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 3 of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at feedback@mastering-api.com.

In Chapter 2 we explained APIs of different flavors and the value that they bring to your architecture. Now that we know about APIs and want to incorporate them into our ecosystem it is crucial to understand how to build a quality product. When creating any sort of product, such as a mouthguard [1] or a building, the only way to verify that it works correctly is to test it. These products are stretched, hit, pushed and pulled in all different manners, simulations will be made to see how the product works under different scenarios. This is testing, and testing gives us confidence in the product that is being produced.

The testing of APIs in software development is no different.

Creating a well crafted and tested product gives us confidence to ship a API and know unequivocally that it will respond correctly for our consumer. As a consumer there is nothing worse then using an API that does not work as specified in the documentation. As discussed in "Specifying REST APIs", an API should not return anything unexpected from its documented results. It is also infuriating when an API introduces breaking changes or causes timeouts due to the large duration of time to retrieve a result. These types of issues

drive customers away and are all entirely preventable by creating quality tests around the API service. Any API built should be able to handle a variety of scenarios, including sending useful feedback to users who provide a bad input, being secure and returning results within a specified SLA (Service Level Agreement).

This chapter will introduce the different types of testing that should be applied to any API. We will highlight the positives and the negatives of each type of testing to understand where the most time should be invested. Throughout the chapter there we will reference some recommended resources for those readers seeking to gain a significantly more in depth and specialist knowledge about a subject area.

Scenario for this chapter

For this chapter let's imagine a new greenfield project has been created. This is the Schedule service as part of our Conference API System. In each section the Schedule service will be referenced to demonstrate the testing practices that were applied during its development. While we are designing a microservice in this project, the information being presented should not be thought of just applying to a microservice design. It applies to all developments of a module—either a service like our schedule service or a module of code as part of a monolithic application.

Figure 3-1 shows an image of the Schedule service and you can seen it connects to the Sessions service and the Attendee Service.

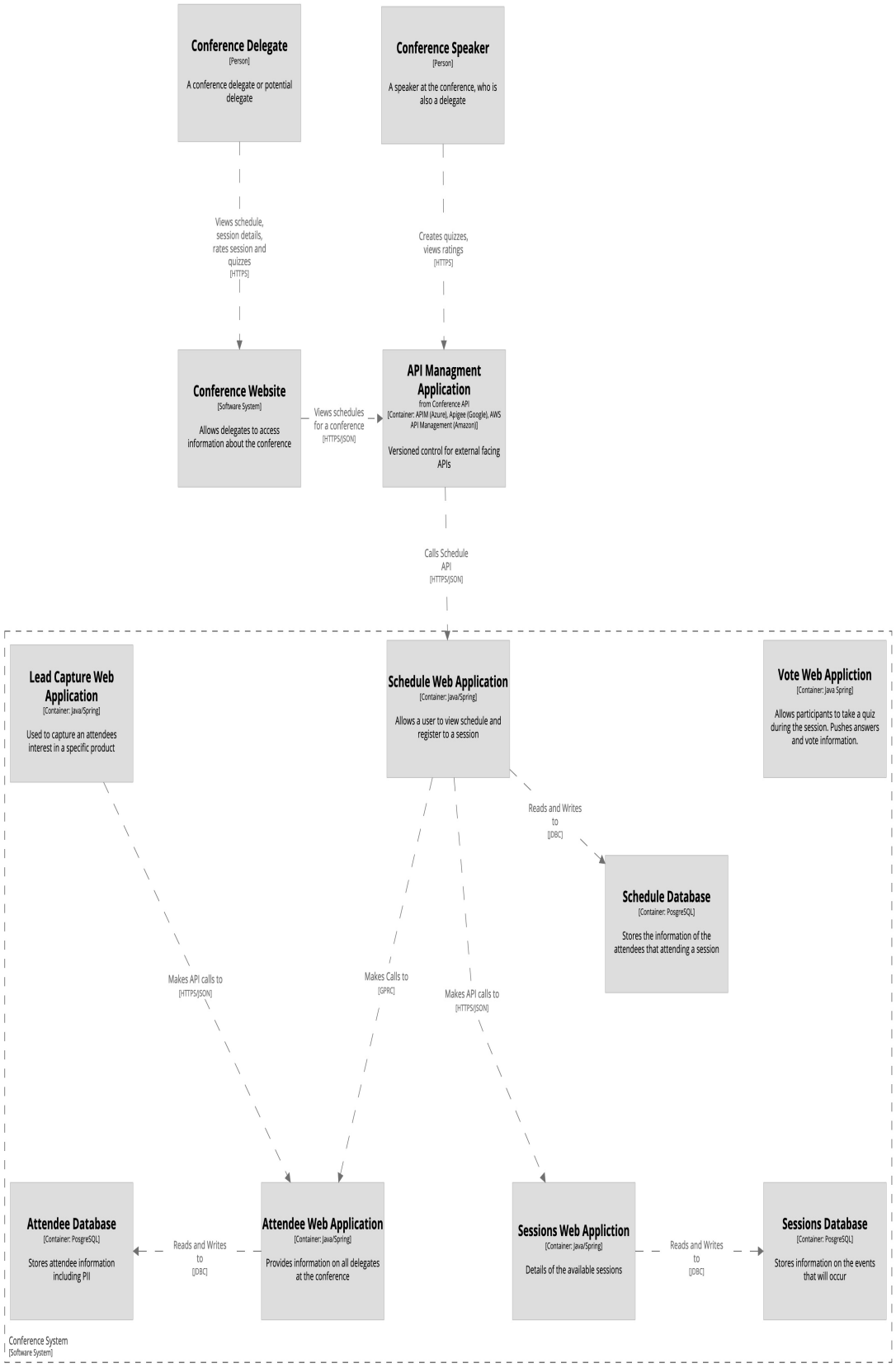

Conference Delegate
[Person]
A conference delegate or potential delegate
Conference Speaker
[Person]
A speaker at the conference, who is also a delegate
Views schedule, session details, rates session and quizzes
[HTTPS]
Creates quizzes, views ratings
[HTTPS]
Conference Website
[Software System]
Allows delegates to access information about the conference
Views schedules for a conference
[HTTPS/JSON]
API Managment Application
from Conference API
[Container: APIM (Azure), Apigee (Google), AWS API Management (Amazon)]
Versioned control for external facing APIs
Calls Schedule API
[HTTPS/JSON]
Lead Capture Web Application
[Container: Java/Spring]
Used to capture an attendees interest in a specific product
Schedule Web Application
[Container: Java/Spring]
Allows a user to view schedule and register to a session
Vote Web Appliction
[Container: Java Spring]
Allows participants to take a quiz during the session. Pushes answers and vote information.
Reads and Writes to
[JDBC]
Schedule Database
[Container: PosgreSQL]
Stores the information of the attendees that attending a session
Makes API calls to
[HTTPS/JSON]
Makes Calls to
[GPRC]
Makes API calls to
[HTTPS/JSON]
Attendee Database
[Container: PosgreSQL]
Stores attendee information including PII
Reads and Writes to
[JDBC]
Attendee Web Application
[Container: Java/Spring]
Provides information on all delegates at the conference
Sessions Web Appliction
[Container: Java/Spring]
Details of the available sessions
Reads and Writes to
[JDBC]
Sessions Database
[Container: PosgreSQL]
Stores information on the events that will occur
Conference System
[Software System]

Figure 3-1. Conference system

The Sessions application is being migrated from SOAP to a RESTful JSON service. This needs to be factored in as part of build for the Schedule service as the two are integrated. The Schedule service is going to be called externally, meaning that it will be internet facing. The Schedules service is called by the API Management Application which could be initiated by the Conference website or third parties.

The Testing Quadrant

Before implementing some tests and researching what testing tools and frameworks are available, it is important to understand the strategies that can be used when testing a product. The testing quadrant was first introduced by Brian Marick in his blog series on agile testing. This became popularized in the book *Agile Testing* by Lisa Crispin and Janet Gregory (Addison-Wesley). The testing quadrant is bringing together Technology and the Business for testing the service that is being built. Technology cares that the service has been built correctly, the pieces respond as expected, are fault tolerant and continue to behave under abnormal circumstances. The Business cares that the right service is being developed i.e. in the case of the Conference System does the service show all the conference talks available, will the conference system send a push notification before the next talk.

There is a distinction between the priorities from each perspective and the Testing Quadrant brings these together to create a nice collaborative approach to develop testing. The popular image of the test quadrant is:

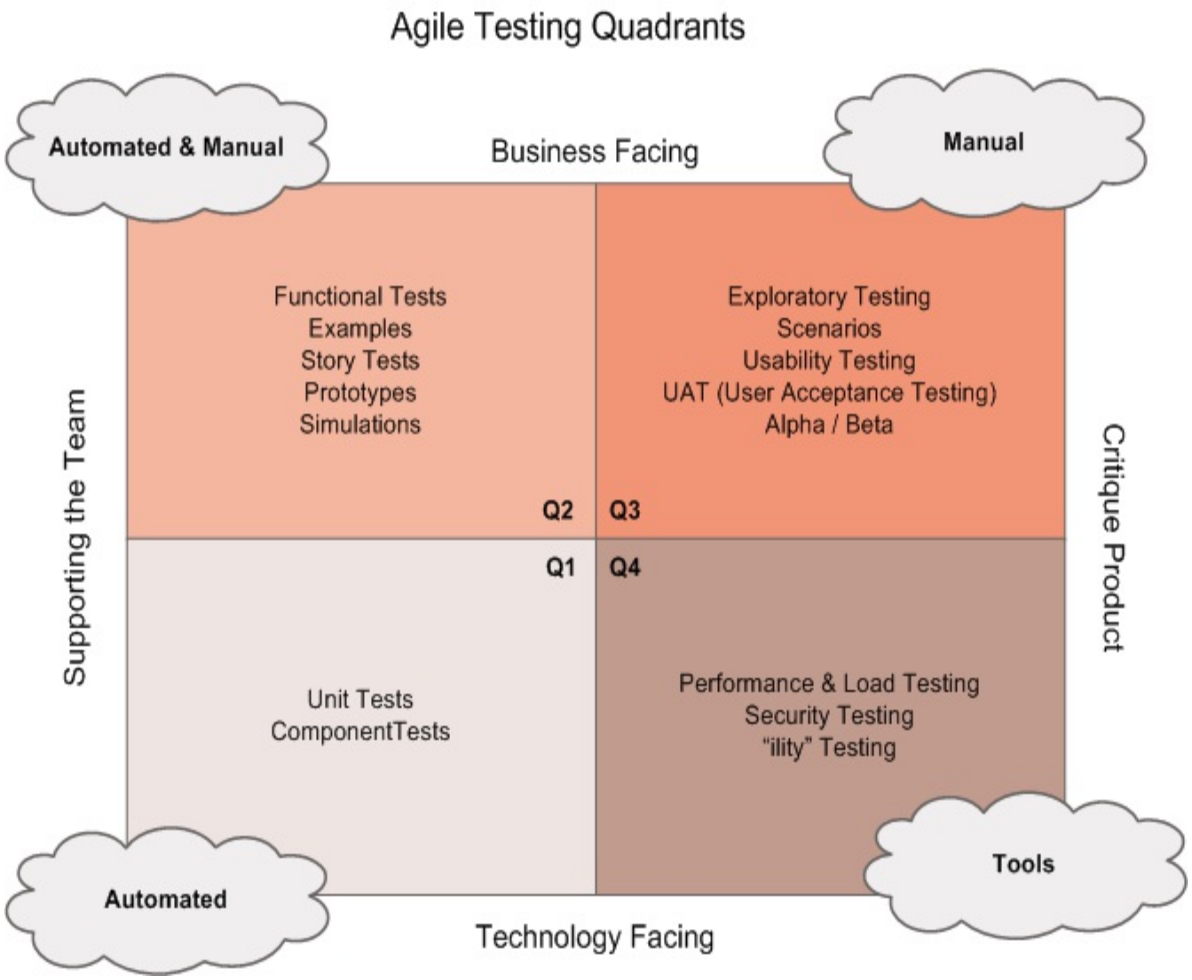


Figure 3-2. Agile Testing Quadrants from Agile Testing (Addison-Wesley) by Lisa Crispin and Janet Gregory

The testing Quadrant does **not** depict any order, this is a common source of confusion that Lisa describes in one of her blog posts. The four quadrants can be generally described as follows:

- Q1 Unit and Component tests for technology, what has been created works as it should, automated testing.
- Q2 Tests with the business to ensure what is being built is serving its purpose, combination of manual and automated.
- Q3 Testing for the business, what should this system be and what should it do, exploratory and expectations being fulfilled, manual testing.

- Q4 The system will work as expected from a technical standpoint, including aspects such as, security, SLA integrity, handle spike loads.

Again, the quadrants do not need to be fulfilled in any order. If our conference system was looking at selling tickets, which is a system that must handle large traffic spikes, it may be best to start at Q4.

The testing quadrant is a fantastic area to delve into and it is recommended that the reader look into this area more. When building APIs you will look at each of the quadrants, though in this chapter the primary focus will be on automating the testing.

The Testing Pyramid

The Testing pyramid was first introduced in the book Succeeding with Agile by Mike Cohn. This pyramid the notion of how much time should be spent on a testing area, the difficulty and the returned value that it brings. If you search online for images of a testing pyramid thousands will appear, they all consist of different blocks, colours, arrows drawn on them and some even have clouds at the top. However, the Testing Pyramid at its core has remained unchanged. The Testing Pyramid has Unit Tests as its foundation, Service Tests as the middle block and UI tests at the peak of the pyramid.

Figure 3-3 shows a Testing pyramid which comes from Martin Fowlers online post *https://martinfowler.com/bliki/TestPyramid.html*

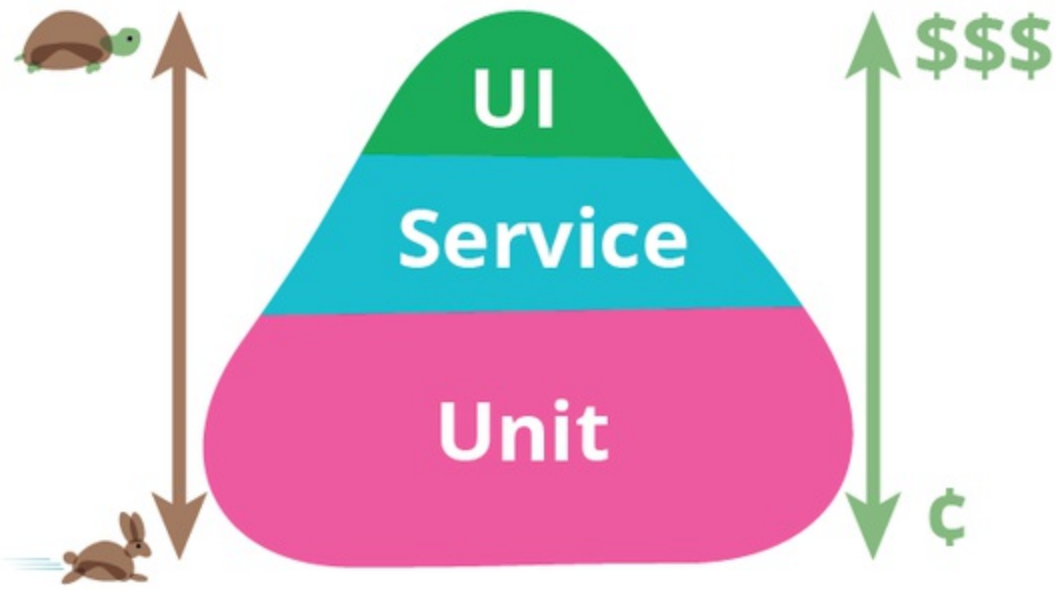


Figure 3-3. Martin Fowler Testing Pyramid

Firstly lets examine the test pyramid and then we can start to delve into what exactly is a unit, service and ui test.

When we look at this image we see intuitive icons that allow us to understand where we should dedicate our time and energy in creating tests for a product, in our case for the Schedule service.

Unit tests are at the bottom of the test pyramid and most importantly they form the foundation of this pyramid. The diagram highlights that they have the lowest cost in terms of development time to create and maintain and also they should run quickly. Unit tests are testing small, isolated units of your code to ensure that your defined unit is running as expected.

Service tests are next in the pyramid with a higher development and maintenance cost than unit tests and a slower run time than unit tests. The reason for the increased maintenance cost in a service test is that they are more complex as multiple units are tested together to verify they integrate correctly. As they are more complex this will mean that they will run more slowly than unit tests. This cost explains why there are less service tests than that of unit tests and the increased costs and decreased speed mean that the returned value decreases.

Finally we have UI tests which are are the peak of the pyramid. These are the

most complex tests so they will have the most cost in terms of creating and maintaining and will also run the most slowly. The UI tests will test that an entire module is working together with all its integrations. This high cost and low speed demonstrates why UI tests are the peak of the pyramid as the cost to benefit diminishes.

This does not mean that one type of test has more value than another. When building a service it is important to have all of these types of test. We do not want a service that only has unit tests or just UI tests.

The testing pyramid is a guide to the proportions of each type of testing that should be done.

The reason that we need a testing pyramid is that from a high level as an Architect or a project owner standpoint, UI tests are the most tangible. Having something tangible makes these tests feel the safest and by giving someone a list of step by step tests to follow this will catch all the bugs. This gives the false sense of security these higher level tests provide better testing than the results of unit tests which are not in an architects control. This false fallacy gives rise to the ice cream cone which is the opposite of a testing pyramid. For a robust argument on this please read Steve Smiths blog post End-To-End Testing considered harmful

Here we have an image of the anti-pattern testing ice cream cone *https://alisterbscott.com/kb/testing-pyramids/*.

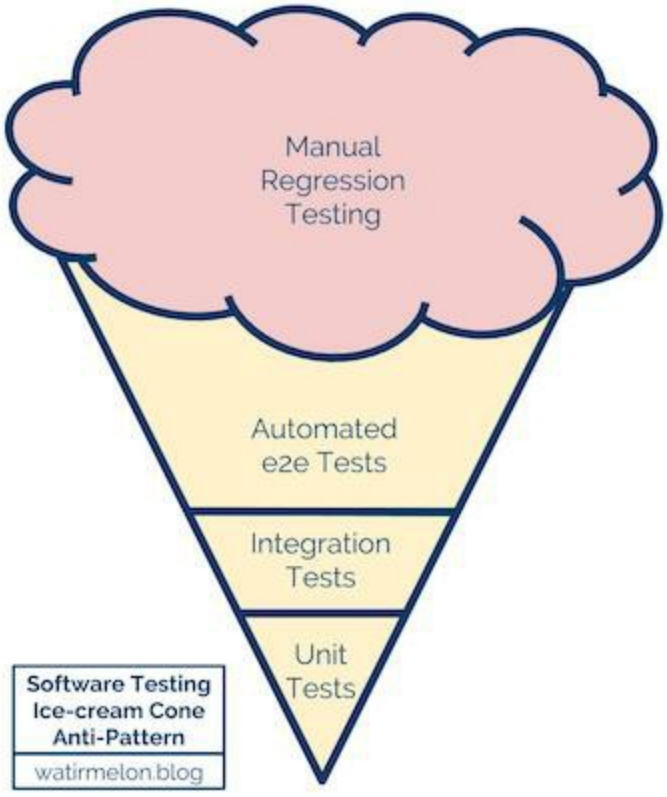


Figure 3-4. Software Testing Ice-Cream Cone by Alister B Scott

As can be seen the emphasis here is on the high level manual and end to end testing over the lower level and more isolated unit tests. This is exactly what you do not want to do, however, when looking at your own services.

Unit Testing

The foundation of the testing pyramid is unit testing and we have outlined some characteristics that unit tests should have, they should be fast numerous and cheap to maintain. The first question that might be raised by an architect is why are we spending time looking at unit tests in this discussion on APIs? Unit tests are for testing logic at small isolated parts. This gives a developer who is building the unit the knowledge that this piece has been built "correctly" and doing what it is supposed to do. When building an API it must be solid for the customer and it is essential that all these foundational units are working as expected.

At this point unit tests have been talked about,but only in the context of their purpose.

A Unit test should firstly be isolated and it should test a single unit. However, what is the unit?

The typical example of a Unit in Object Oriented (OO) languages is a Class though some make unit tests per public method of a Class. For a declarative language a unit may be defined as a Function or some Logic.

A Unit need not be defined exactly as one of these, however, you will need to define a unit for your case. To give a concrete example of this blurred line. Here we see Kevin Wittek defining a unit test to be anything that is interprocess.

Ken Fogel @omniprof · Jul 31

I have a Java public method that can find a record in a DB. To verify its operation am I writing a unit test or an integration test? I've been told that running a test that depends on a framework, i.e. JDBC, is an integration test while tests that do not are unit tests. Correct?

18 4 31

Mihai 🌵 @amihaiemil · Jul 31

If you ask me, a unit test should cover solely your class, with all dependencies (i.e. constructor arguments) mocked.

Anything else is an "integration test" because it integrates more classes/objects together.

1 2

Kevin Wittek
@Kiview

Replying to @amihaiemil and @omniprof

Here I have a different opinion. If two objects of classes that are part of my codebase interact with each other in-process, I still call this an Unit Test.

8:04 AM · Aug 1, 2020 · Twitter for Android

2 Likes

Figure 3-5. Twitter discussion on Unit Tests

Let us look at the Schedule service that we are building. Here, since we are using Java, we shall define a unit as a Class. The code is functional in style and will be commented to explain what it does if you are not familiar with the language.

Before we write the implementation let us look at what we want this unit to do.

Test Driven Development

Here, we'll test the class. We want to decorate the Session that we get from the Session Service with additional information on how it fits in the Schedule.

```
public class SessionDecorator {

}
```

The criteria that we wish to fulfill (i.e., what we'll be testing) are the following:

- Given a `ConferenceSession` is to be decorated, When a request to the database is performed and a corresponding schedule exists, Then a `ConferenceSchedule` will be returned without errors or warnings
- Given a `ConferenceSession` is to be decorated, When a request to the database is performed and no corresponding schedule exists, Then a NoSuchElementException will be returned.
- Given a `ConferenceSession` with no id value, Then an IllegalArgumentException will be returned

We can write the tests for this simple class before we write the implementation. This is known as TDD (Test Driven Development). The TDD process is defined as:

- Write Test - Write the test that you wish your unit to fulfill

- Run all the Tests - Your code should compile and run, however, your tests should not pass.
- Write the code to make test pass
- Run all the Tests - Validates that your code is working and nothing is broken
- Refactor code - Tidy up the code
- Repeat

The process is known as Red, Green, Refactor

This simple iterative process was discovered/re-discovered by Kent Beck and to learn more about this area refer to his works Test-Driven Development: By Example.

In our example we are using JUnit which is an xUnit framework for the Java language. Again to learn more about xUnit frameworks refer to the works of Kent Beck's *Test Driven Development: By Example*(Addison-Wesley).

Here we see a sample test for a positive outcome. This example uses stubs for the test. A stub is an object used for a test that can return hard coded responses.

```
  @Test
  void decorate_a_conference_session() {
//    Given a ConferenceSession is to be decorated
    final ConferenceSession conferenceSession = ConferenceSession.builder()
        .id("session-123456")
        .name("Continuous Delivery in Java")
        .description("Continuous delivery adds enormous value to the "
            + "business and the entire software delivery lifecycle, "
            + "but adopting this practice means mastering new skills "
            + "typically outside of a developer's comfort zone")
        .speakers(Collections.singletonList("dan-12345"))
        .build();

//    When a request to the database is performed and a corresponding schedule
exists
    final DatabaseSchedule foundSchedule = DatabaseSchedule.builder()
        .id("schedule-98765")
```

```
            .sessionId("session-123456")
            .startDateTime(LocalDateTime.now())
            .endDateTime(LocalDateTime.now().plusHours(4))
            .build();
        final StubScheduleRepository stubScheduleRepository =
            new StubScheduleRepository(foundSchedule);


//    Then a ConferenceSchedule will be returned without errors or warnings
        final SessionDecorator sessionDecorator =
            new SessionDecorator(stubScheduleRepository);
        final ConferenceSchedule returnedConferenceSchedule =
            sessionDecorator.decorateSession(conferenceSession);
        final ConferenceSchedule expectedResult =
            ConferenceSchedule.builder()
                .id(foundSchedule.getId())
                .name(conferenceSession.getName())
                .description(conferenceSession.getDescription())
                .startDateTime(foundSchedule.getStartDateTime())
                .endDateTime(foundSchedule.getEndDateTime())
                .build();
        Assertions.assertEquals(expectedResult, returnedConferenceSchedule);
    }
```

We next present a test that uses mocks. Mocks are pre-programmed objects that work on behavior verification.

```
    @Test
    void
return_NotFoundException_when_a_linked_schedule_is_not_in_the_repository() {
//    Given a ConferenceSession is to be decorated
        final ConferenceSession conferenceSession = ConferenceSession.builder()
            .id("session-123456")
            .name("Continuous Delivery in Java")
            .description("Continuous delivery adds enormous value to the "
                + "business and the entire software delivery lifecycle, "
                + "but adopting this practice means mastering new skills "
                + "typically outside of a developer's comfort zone")
            .speakers(Collections.singletonList("dan-12345"))
            .build();

//    When a request to the database is performed and no corresponding schedule
exists
        when(mockScheduleRepository.findAllBySessionId(anyString()))
            .thenReturn(Collections.emptyList());
```

```
//    Then a NotFoundException will be returned
    final SessionDecorator sessionDecorator =
        new SessionDecorator(mockScheduleRepository);

    Assertions.assertThrows(NoSuchElementException.class,
        () -> sessionDecorator.decorateSession(conferenceSession)
    );
  }
```

Please refer to the book's Github repository for further tests.

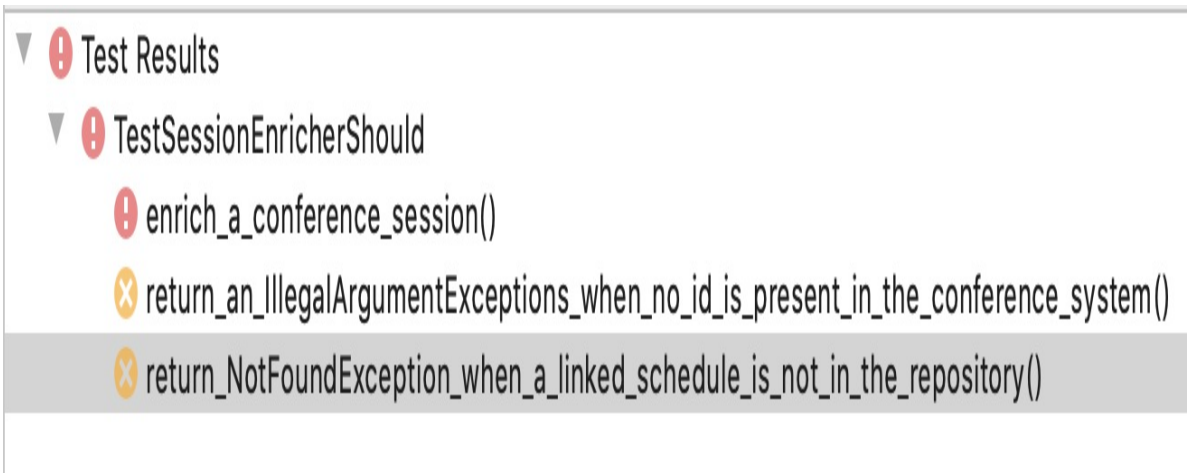
Classicist and Mockist

In these tests we show the use of using mocks, stubs and dummy's. This is highly exaggerated case to show some of the test objects available. The tests shown have both a Classicist and Mockist approach. [2] The Classicist approach uses real objects, like the stub example while the Mockist will use mock objects instead of real objects. Is there a purest way to go? No there is not a single approach to take and both have there pros and cons. To learn a lot more on this, please read this excellent article by Martin Fowler titled "Mocks Aren't Stubs".

Now lets create the method though not write any implementation and run the test suite again.

```
public class SessionDecorator {
  public ConferenceSchedule decorateSession(ConferenceSession session){
    throw new RuntimeException("Not Implemented");
  }
}
```

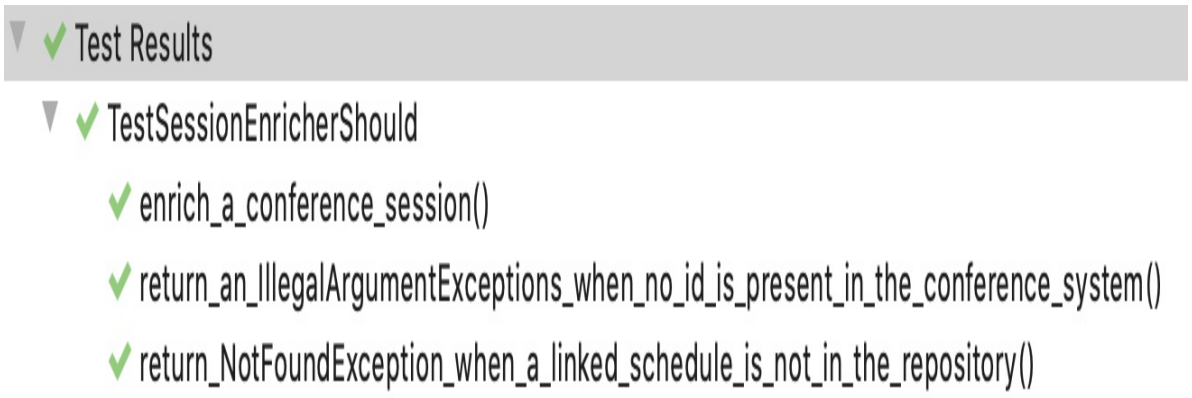
From this we will see that no tests pass.



Let us create the implementation to make our tests pass.

```
public ConferenceSchedule decorateSession(
    ConferenceSession session) {
  if (Objects.isNull(session.getId())) {
    throw new IllegalArgumentException();
  }
  final List<DatabaseSchedule> allBySessionId =
      scheduleRepository.findAllBySessionId(session.getId());
  if (allBySessionId.isEmpty()) {
    throw new NoSuchElementException();
  }
  else {
    final DatabaseSchedule databaseSchedule = allBySessionId.get(0);
    final ConferenceSchedule conferenceSchedule = ConferenceSchedule.builder()
        .id(databaseSchedule.getId())
        .name(session.getName())
        .description(session.getDescription())
        .startDateTime(databaseSchedule.getStartDateTime())
        .endDateTime(databaseSchedule.getEndDateTime())
        .build();
    return conferenceSchedule;
  }
}
```

Excellent the tests now pass.



The code has got a bit messy so let's refactor it:

```
public ConferenceSchedule decorateSession(
    ConferenceSession session) {
  Preconditions.checkArgument(Objects.nonNull(session.getId()));
  final List<DatabaseSchedule> allBySessionId =
      scheduleRepository.findAllBySessionId(session.getId());
  if (allBySessionId.isEmpty()) {
    throw new NoSuchElementException();
  }
  final DatabaseSchedule databaseSchedule = allBySessionId.get(0);
  return ConferenceSchedule.builder()
      .id(databaseSchedule.getId())
      .name(session.getName())
      .description(session.getDescription())
      .startDateTime(databaseSchedule.getStartDateTime())
      .endDateTime(databaseSchedule.getEndDateTime())
      .build();
}
```

Finally the tests should be run again to ensure that nothing has broken by the refactor.

We are now confident in our implementation.

NOTE

When first performing the refactor the Author made a mistake which broke one of the tests. Even with such a small and simple piece of code mistakes can happen. Tests make refactoring safe and we have immediately gained value from having them.

This example should give you a flavour of the Test Driven Development process. When building applications the value of TDD has historically spoken for itself. Studies have shown that Developers who user TDD spend less time debugging and, on greenfield projects, build something more solid.

For more information on Test Driven Development in Industry and the value it gives, we recomend these two papers:

- An Initial Investigation of Test Driven Development in Industry
- Realizing quality improvement through test driven development

Unit testing is a valuable way to build a system and using TDD gives you a firm foundation for your testing pyramid. These unit tests give you isolated testing of your defined unit and coverage of your application. Being fast to run and cheap to maintain they give you a fast development cycle and reassurance your units are doing what you expect.

> **TIP**
>
> As an Architect it is important that foundations of the modules being built use these fundamental testing techniques. If TDD is not being applied then it is worth allowing developers to invest time learning to apply this process to develop better modules and in the long term allow teams to iterate faster with confidence.

Service Testing

Service tests are more wide ranging in what they consist of. In this section, we will be exploring what is a Service test, the value that they bring and pros and cons. Service tests consist of integration tests and testing your application across different boundaries.

There are two boundaries that exist. The first is are between your module (service if building a microservice) and an external component. The other is between multiple units together internal to your module. It is important to keep these distinctions and have these boundaries separated for your testing.

This is why we will have two definitions for our service tests.

- Component test are testing our module and do not include external services (if they are needed then they are mocked or stubbed).
- Integration tests are tests that verify egress communication across a module boundary (e.g. DB Service, External service, other Module).

Component tests

Component testing to see how our module works with multiple units is really valuable. It allows validation that the modules work as expected and our service can return the correct response from incoming consumer requests.

As component tests do not include external dependencies, any that are required will be mocked or stubbed. This is useful if you want to force a module to fail and trigger issues such as client errors and internal service errors which can help verify that responses are generated correctly.

Introducing REST-Assured

REST-Assured is a tool that will will be useful for our component tests when sending incoming requests to our Schedule service. This is a DSL that uses the Given When Then semantics and makes clear readable tests.

REST-Assured allows us to construct requests and to closely control the payloads that are sent to the controller. It is possible to pass specific headers and query params to the URL. REST-Assured then allows us to inspect the result and interrogate exactly what the response is. This includes the HTTP Status code, headers and response body.

REST-Assured has a variety of language implementations so should not just be seen as a Java tool. There are other implementations that employ a similar style for testing.

Security component test example

In our example, the components that are being tested are the incoming requests to our Schedules service. When a client makes a request to the

endpoint “/schedules” it is important that it is routed to the correct Controller that will perform the business logic upon this request. There are further criteria that should be validated when requests are made such as to ensure that when making calls to our services that no information is leaked if a consumer probes our service. Validation that our service is not leaking any of this information is really important and where component testing is useful. Another key area to validate is security. In our Schedule service there is a security requirement that all authorized clients can make a request to see the schedules, however, only specific authorized end users are able to create a new schedule. Again a selection of units are grouped together to ensure that when a request is made to our service that it will allow an authorized user access to see the schedules.

For our Schedule service a controller has been created that will take requests from an authenticated user

```
@GetMapping
public Mono<ValueEntity<ConferenceSession>> getSchedules(
    @AuthenticationPrincipal JwtAuthenticationToken principal) {
  if (!READ_SCHEDULE_CLAIM
      .equals(principal.getToken().getClaimAsString("scope"))) {
    return Mono.error(
          new AccessDeniedException("Not required Authorization")
      );
  }
  return sessionService.getAllSessions()
    .flatMap(sessionDecorator::decorateSession)
    .collectList()
    .map(conferenceSchedules -> new ValueEntity(conferenceSchedules));
}
```

NOTE

This security check uses Spring Security. The Spring Security library has the ability to demonstrate this in a neater fashion, however, this more verbose method has been chosen for the purpose for brevity.

This controller was written using TDD, however, the Authenticated Principal

passed in was created in the unit test. This is why it is important to verify the security of this controller and the Spring Security framework is working with our component. Component tests will verify the controller fulfilled the following criteria:

- Requests will not reach the controller if no Authenticated principal is provided
- Requests that have an incorrect scope will return a status of 403
- Requests that are successful have response of 200

Here we show the second test case in the list above. Please refer to the github for the other test cases

```
@Test
void return_a_forbidden_response_with_missing_read_scope_supplied() {
  given()
    .webTestClient(
      webTestClient.mutateWith(
        mockJwt().jwt(jwt -> jwt.claim("scope", "schedules:WRITE"))
      )
    )
    .when()
      .get("/schedules")
    .then()
      .status(HttpStatus.FORBIDDEN);
}
```

What can be seen here is that the tests assert the expected behavior that was laid out. This essential testing gives us confidence that the endpoint is secure and will not leak data to unauthorized parties.

Component testing shows us the value of the units working together and can confirm whether the units work together as expected.

Integration tests

Integration tests in our definition are testing across boundaries between the module being developed and any external modules.

The schedule service talks to three other external services which are the PostgreSQL Database a RESTful Sessions Service and a gRPC Attendee service. All of the code that is used to communicate with these boundaries needs to be tested. For the Sessions Service the `SessionsService` Class is used to communicate between our service and RESTful service. For the Attendee Service the `AttendeeService` Class is used to communicate between our service and the gRPC service For communicating with the PostgreSQL database this is the `ScheduleRepository` Class, which uses Spring Data to create our connection between our application and our Database.

The external boundaries have been identified and the next step is to establish across each one what should be tested and verified. Each boundary will have its own set of tests and each one should be tested in its own environment. It is not advisable to have one test file that is used for all boundaries. This section of the book will give in depth coverage of testing the Sessions and Attendees services as well as mention the connection to the Database connection used for the Schedule Service.

External Service integration

A common requirement for a project is to integrate with another service or module.

The validation that should be performed is to confirm that when communicating across the boundary that the interaction is correct with this external service. For this external service integration the case that will be examined is the Schedule service communicating with the Sessions service.

How do we know that an external service integration is validated

- Ensuring that an interaction is being made correctly e.g. for a RESTful service this may be specifying the correct URL or the payload body is correct
- Is the data being sent in the correct format?
- Can the unit that is interacting with external service handle the

responses that will be returned?

Ideally, we would have a list of interactions with an external service and be able to validate these without having to speak to the actual service itself. If the Schedule service speaks to the actual Sessions service then this would be part of End to End testing which is later on in "UI Testing". So, how can we validate that the two services can talk to each other?

Options for testing with an external service

The first and most obvious solution is to use what was learned in "Introduction to REST" and to look at an API specification that a Sessions service produces. By looking at the API specification it is possible to see all the endpoints and the requests and responses that are available to us. This also means that if an API first approach is taken to a system ("Specifying REST APIs") an OpenAPI spec will be available for use.

Introduction to stubbing with Wiremock

Another option is to create a stub server of the Sessions service API that can be run locally as part of a test. It is possible to use a tool such as Wiremock to facilitate this stubbing.

By using WireMock it is possible to construct a stub server directly in code in tests or to create mappings files which are JSON files that describe an API interaction. WireMock can be ran as a standalone process so this does not limit you to a programming language or enforce only running it for testing. It is possible to run a stub server and use it in an environment as a replacement for a real service. An example case and mappings file is shown towards the end of "Why use API Contracts".

The process of using WireMock is to standup the stub server for your tests to represent an external service. The WireMock server is then fed interactions that it should respond to. So when representing the Schedules service a mapping will be fed to it that effectively states, when a request is made to this endpoint (e.g. "/schedules"), then respond with this body. If no matches are made an error is returned.

However, there is an issue here that immediately causes concern. The concern is actually writing the stub code. As the developers wish to have a stubed external service they need to create it by hand with their understanding of how the Schedule service should work. What if interaction is incorrectly defined, such as the expected response is written incorrectly?

There is a solution to this if the API service is already available. WireMock again can be used to capture real requests and responses by proxying the requests through it. More information on how to do this is available in the WireMock documentation These captured requests and responses are saved as mapping files and then can be used to use WireMock for testing or as a stub service for numerous cases such as a dev test environment when actual requests to an API may cost actual money.

By capturing the API requests with another service the concerns of writing mappings by hand are averted as they are instead being generated. This does not solve the issue when a new API interaction is being defined or the API interaction is extended and again the solution is to write the new API interaction or extend the stub server.

Stubbing is a common practice and something that the Authors have used in the past as it works and provides value. As long as the team is aware of the limitations these are fair techniques to use and provides invaluable testing across the external boundary.

There is another technique to use for testing with external services, though first let’s walk through a typical scenario for a new project that many readers will be familiar with or looking to undertake: A new project is created and two teams want or need to integrate their products together. They decide on the core interactions that they need and write a document with a list of endpoints for the new RESTful service. Payloads are defined of what is sent and what will be received. After time passes the two teams deploy the first iteration and they find that some or all of the interactions are not working.

Classic examples of why interactions are not working include:

- The URLs do not match the document as someone has decided on

new and “better” convention

- The client of the service has mistyped a property name
- The response is expected as an object but gets returned as an array of objects as this will make it “future proof”

An API first approach has been tried in this example, all parties try to do the right thing and make the best product. However, there is nothing to hold the services to account on the agreement made in the document. The producer of the API needs to uphold what they will respond with and the consumer needs to validate it is calling the API correctly.

For a real life situation what is being described for this document is at best handshake agreement of what each party will do. What is desired is something like a that looks more like a contract!

There is good news that API Contracts do exist.

API Contracts

AAs a consumer of an API it has always been a good idea to test with a Test Stub service (like the WireMock services just described and as a producer of an API there is huge value in being able to know how your service is going to be called.

API Contracts are a written definition of an interaction between two entities a Consumer and a Producer. A Producer responds to the API requests, it is producing data. e.g. a RESTful Web Service A Consumer requests data from an API. e.g. web client, terminal shell

Producers are also known as Providers. In this book the word Producer will be used, though the reader should be aware that they are interchangeable and provider is the term of choice in some contract frameworks.

When looking at the C4 model of the the interactions of our Schedule service it is valuable to see the UI is calling the service through the API Management solution and that the service is requesting data from both the Attendee and Sessions service.

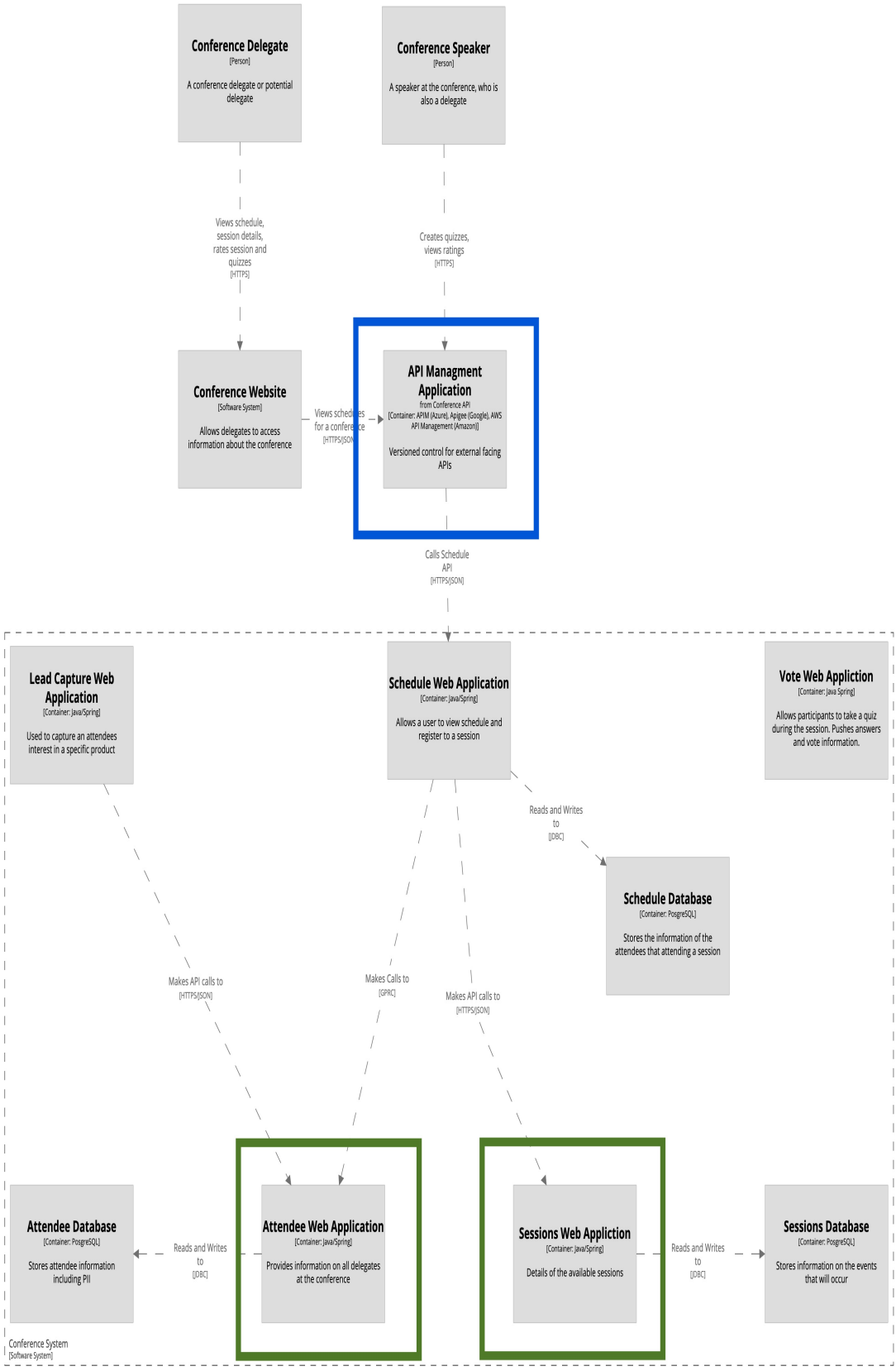

Conference Delegate
[Person]
A conference delegate or potential delegate
Conference Speaker
[Person]
A speaker at the conference, who is also a delegate
Views schedule, session details, rates session and quizzes
[HTTPS]
Creates quizzes, views ratings
[HTTPS]
Conference Website
[Software System]
Allows delegates to access information about the conference
Views schedules for a conference
[HTTPS/JSON]
API Managment Application
from Conference API
[Container: APIM (Azure), Apigee (Google), AWS API Management (Amazon)]
Versioned control for external facing APIs
Calls Schedule API
[HTTPS/JSON]
Lead Capture Web Application
[Container: Java/Spring]
Used to capture an attendees interest in a specific product
Schedule Web Application
[Container: Java/Spring]
Allows a user to view schedule and register to a session
Vote Web Appliction
[Container: Java Spring]
Allows participants to take a quiz during the session. Pushes answers and vote information.
Reads and Writes to
[JDBC]
Schedule Database
[Container: PosgreSQL]
Stores the information of the attendees that attending a session
Makes API calls to
[HTTPS/JSON]
Makes Calls to
[GPRC]
Makes API calls to
[HTTPS/JSON]
Attendee Database
[Container: PosgreSQL]
Stores attendee information including PII
Reads and Writes to
[JDBC]
Attendee Web Application
[Container: Java/Spring]
Provides information on all delegates at the conference
Sessions Web Appliction
[Container: Java/Spring]
Details of the available sessions
Reads and Writes to
[JDBC]
Sessions Database
[Container: PosgreSQL]
Stores information on the events that will occur
Conference System
[Software System]

Figure 3-6. Producer and Consumer elements of schedule service highlighted

Therefore we can say that the Schedule service is a Producer for the API Management Application and it is a Consumer of the Sessions service and Attendee service.

Why use API Contracts

The beauty of defining an interaction with a contract is that it is possible to generate tests and stubs from this. Automated tests and stub services are what give us the ability to perform local testing without reaching out to the actual service, though still allow us to verify an interaction.

To understand of how tests can be auto generated and stub servers can be created it is important to see an API contract. For our example as the Schedule service is being written in Java the Contract framework that will be used is Spring Cloud Contracts. However, the section "API Contract Frameworks" discusses other API Contract frameworks.

The Schedule service is communicating with the Sessions Service and it was decided as part of the build that the Sessions service would use API Contracts. We can see from the image the three contracts. These contracts are the Get sessions as a GET, a Create Session as a POST and a Remove Session as a DELETE.

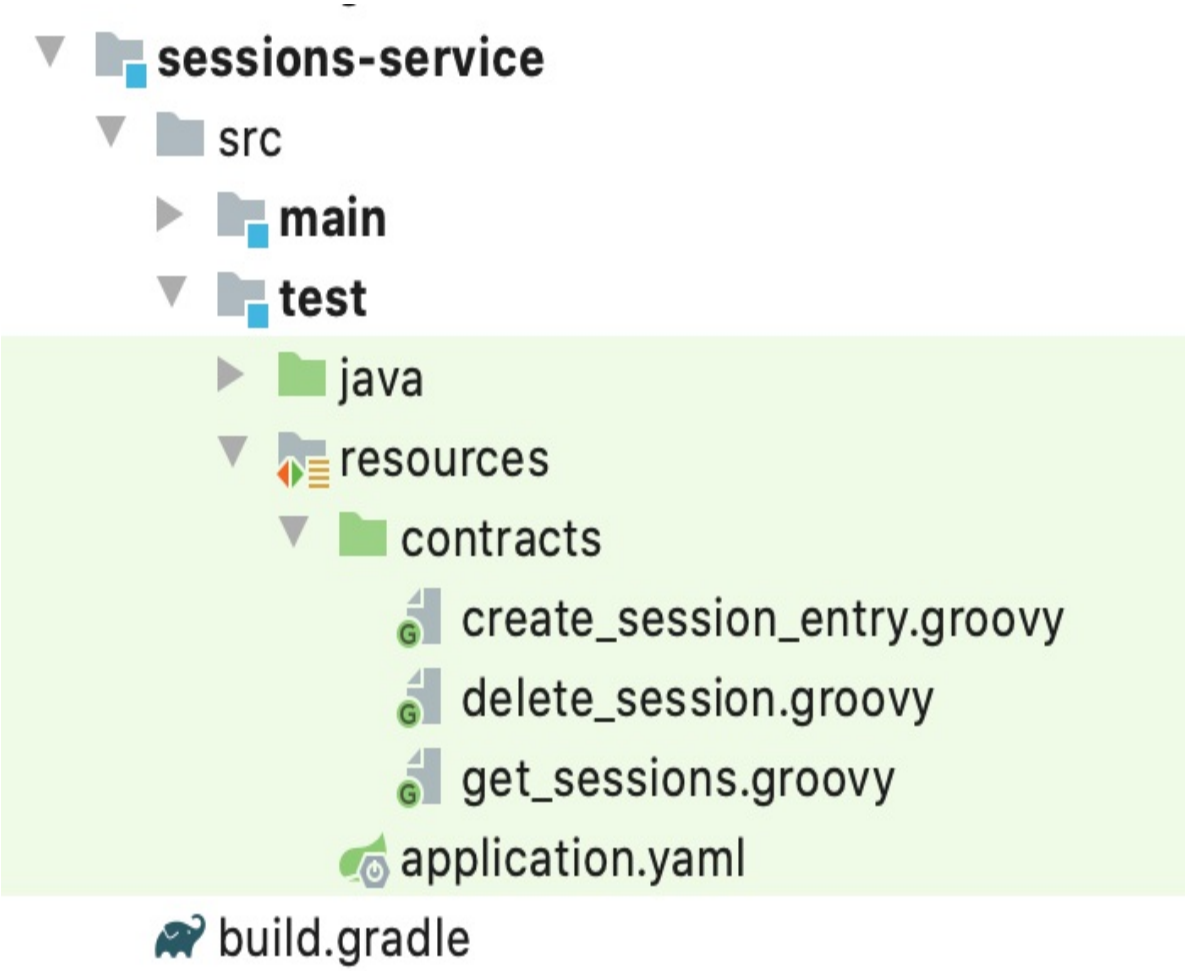


Figure 3-7. Contracts of sessions service

The piece of functionality that the sessions service is lacking is the ability to modify a session. This is a candidate for submitting a contract for this new interaction with the Sessions service.

Here is the contract for the new interaction. It shows how the Producer shall respond and how the Consumer should interact with the new PUT endpoint.

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
  // (1) Request entity
  request {
    // (2) Description of what the contract does
    description("""Modify a Schedule Resource""")
    // (3) The HTTP Method that the consumer will call
    method PUT()
```

```
    // (4) The URL endpoint that will be hit
    urlPath '/sessions/session-id-1'
    // (5) The Headers that should be passed by the callee
    headers {
      // (6) Header of ContentType with value 'application/json'
      contentType('application/json')
    }
    // (7) The Body of the request
    body(
        // (8) A name property
        name: "An Introduction to Contracts",
        // (9) A description property
        description: "A beginners guide to Contracts",
        // (10) A speaker id array property
        speakerIds: [
          // (11) An array of speaker ids
          "bryant-1",  "auburn-1"
        ]
    )
  }
  // (12) Response entity
  response {
    // (13) The status from producer should be NO CONTENT, 204
    status NO_CONTENT()
  }
}
```

This should hopefully look familiar as it has the look and feel of a generic Request and Response that you see in HTTP interactions. This is a real benefit as it makes writing and reading contracts simple for people not familiar with Groovy.

Let us go through this contract step by step and explain what is happening.

The first part, the most outer definition, is merely stating that we are defining a contract:

```
Contract.make{

}
```

The two entities are the `request` and the response so let us first check the `request` we have defined in the contract. In this request it is shown what the

Consumer should send to the Producer. The `response` is next and shows what the Producer should respond with

As we look at the contract as a whole what is being said is, When a PUT request is sent to /sessions/session-id-1 which has a header with Content-Type that is equal to application/json and the body of the request has a name property that matches “An Introduction to Contracts” and a description that equals “A beginners guide to Contracts” a Response with the status 204 is returned.

We now have a full definition of a contract and to reiterate this is a definition of an interaction and NOT behavior. The contract is stating that the Producer has to return a 204 status code (when successful), it does not care how the producer does this in the background. If this were a legal contract and you are told to pay a fine, the court does not care how you procure the money (hopefully legally) so long as you pay the fine issued. This is a defined interaction, the court will get the money, and for an API the consumer will get the defined response.

One thing you may have noticed is hardcoded values such as the id of session-id-1 enforced by the contract. This works fine as from the definition we can easily tell what the contract is intended todo, however, we can make the contract more generic by using regular expressions for defined properties. Here we see the same contract in a more generic manor.

```
// (12) Response entity
response {
  // (13) The status from producer should be NO CONTENT, 204
  status NO_CONTENT()
}
```

As can be seen in the annotated code the values can be generated or inferred by the testing framework. For a Consumer, When a PUT request is sent to `/sessions/{ID}` which has a header with Content-Type that is equal to application/json and the body of the request has a name property that is a non blank string and a description that is a non blank string a Response with the status 204 is returned. For a Producer, When a PUT request is sent to

`/sessions/session-id-1` which has a header with Content-Type that is equal to application/json and the body of the request has a name property that is a non blank string and a description that is a non blank string a Response with the status 204 is returned It is not suggested that all contracts be as generic as possible, hardcoded values are not bad, however, generic values can be useful for the generated contract tests.

As this contract is something that the producer (Schedule service) should fulfill it is possible to generate tests from this contract. This is logical as, this is an interaction that is entirely defined. The generated test from this contract is the following:

```
@SuppressWarnings("rawtypes")
public class ContractVerifierTest extends BaseContract {

  @Test
  public void validate_modify_session() throws Exception {
    // given:
    WebTestClientRequestSpecification request = given()
        .header("Content-Type", "application/json")
        .body("{\"name\":\"KJSKASVJSGXRLXYKVSXY\","
          + "\"description\":\"LXYIOGALHTTSXMUCRIPY\","
          + "\"speakerIds\":[\"FQJEPDMOAPIMDUPGFCKZ\"]}");

    // when:
    WebTestClientResponse response = given().spec(request)
        .put("/sessions/session-id-1");

    // then:
    assertThat(response.statusCode()).isEqualTo(204);
  }

}
```

This generated test is testing the interaction against our producer. A real request is sent to our service against the defined endpoint and with random non blank strings for the name and description and speaker ids, just as defined in the contract. This should emphasize why it may not always be desired to have a more generic contract as the values are used for the generated test.

Eagle eyed readers may notice, it may be noticed the `extends BaseContract` in the test. This is required in the setup of Spring Cloud Contracts to define how to configure the producer so it can receive the request sent by the generated test. This makes sense as a contract can create a test, though it has no knowledge of how to start the service.

```
@SpringBootTest
public abstract class BaseContract {

  @Autowired
  private ApplicationContext context;

  @BeforeEach
  void setUp() {
    RestAssuredWebTestClient.applicationContextSetup(context);
  }
}
```

This base contract is an implementation detail to this specific framework for this generated test and to complete the story. The finer points of this framework will not be discussed any further.

These generated tests must be fulfilled by the producer as part of the build to ensure that no contracts fail. This is key when developing an API to ensure that defined interactions continue to be fulfilled and that the values and properties do not change by mistake or by accident.

The final part of this story is the generated stub service that can be used by the consumer for testing.

So what is generated?

```
{
  "id" : "bfd7dd8b-1551-4fc8-b530-231613a201a4",
  "request" : {
    "urlPathPattern" : "/sessions/[a-zA-Z0-9-]+",
    "method" : "PUT",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    },
```

```
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['name'] =~ /^\\s*\\S[\\S\\s]*/)]"
    }, {
      "matchesJsonPath" : "$[?(@.['description'] =~ /^\\s*\\S[\\S\\s]*/)]"
    }, {
      "matchesJsonPath" : "$.['speakerIds'][?(@ =~ /^\\s*\\S[\\S\\s]*/)]"
    } ]
  },
  "response" : {
    "status" : 204,
    "transformers" : [ "response-template" ]
  },
  "uuid" : "bfd7dd8b-1551-4fc8-b530-231613a201a4"
}
```

A mapping file is created that can be used by WireMock. These stub servers are extremely valuable to stand up as individual processes not just for testing but to run alongside the actual service to be able to have a working system whule the actual service is still being built.

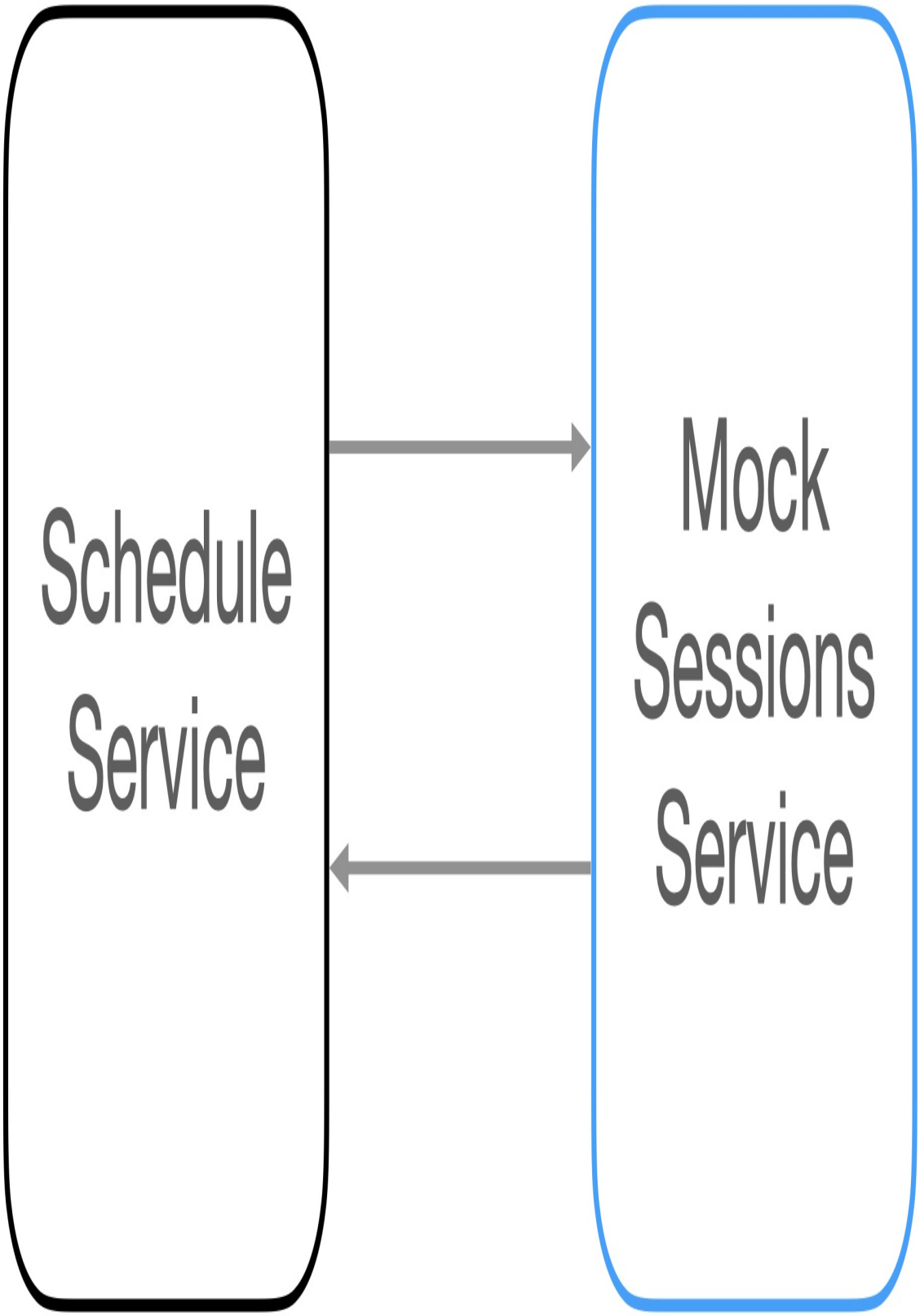

Schedule Service
Mock Sessions Service

Figure 3-8. Schedule and mock Sessions service

> **TIP**
>
> The Authors have used these generated stub servers to run demos for stakeholders while the producer had not been deployed.

When used for testing, as it is herer, we can see a bad request is made. In this example a request is made with the speaker id property being a single value and not an array against this stub service. Again a snippet is shown for brevity to see the whole picture.

```
@ExtendWith(SpringExtension.class)
@AutoConfigureStubRunner(
    repositoryRoot =
        "stubs://file://location/of/sessions-service/build/stubs",
    ids =
        "com.masteringapi:sessions-service:+:stubs",
    stubsMode =
        StubRunnerProperties.StubsMode.REMOTE
)
class TestSessionsServiceShould {

  @StubRunnerPort("sessions-service")
  int producerPort;

  @Test
  void connect_to_put() {

    final WebTestClient webTestClient = WebTestClient.bindToServer()
      .baseUrl("http://localhost:" + producerPort)
      .build();
    webTestClient
      .put()
      .uri("/sessions/session-id-1")
      .contentType(MediaType.APPLICATION_JSON)
      .bodyValue("{"
        + "\"name\": \"A name\","
        + "\"description\": \"A description\","
        + "\"speakerIds\": [\"singular-speaker-id\"],"
        + "}")
      .exchange()
```

```
      .expectStatus().isNoContent();
  }
}
```

```
-------------------------------------------------------------------------------
| Closest stub              | Request                                         |
-------------------------------------------------------------------------------
                            |
PUT                         | PUT
/sessions/[a-zA-Z0-9-]+     | /sessions/session-id-1
                            |
Content-Type [matches] :    | Content-Type:
  application/json.*        |  application/json
                            |
$[?(@.['name']              |
    =~ /^\s*\S[\S\s]*/)]    | {"name:" "A name","description:" "A   <<<<<
                            |  Body does not match
                            | description","speakerIds:" "singular-speaker-id",}
$[?(@.['description']       |
    =~ /^\s*\S[\S\s]*/)]    | {"name:" "A name","description:" "A   <<<<<
                            | Body does not match
                            | description","speakerIds:" "singular-speaker-id",}
$.['speakerIds'][?(@        |
    =~ /^\s*\S[\S\s]*/)]    | {"name:" "A name","description:" "A   <<<<<
                            | Body does not match
                            | description","speakerIds:" "singular-speaker-id",}
                            |
-------------------------------------------------------------------------------
-
```

API Contracts are valuable for both Consumers and Producers. By defining an interaction both usable tests and stub servers can be generated to validate integrations.

Using contracts promotes Acceptance Test Driven Development (ATDD). ATDD is where different teams can collaborate and come to an agreement on a set of criteria that will solve a problem. Here the problem being solved is the API interaction.

NOTE

It is tempting to use contracts for scenario tests. e.g. first perform a create session, second, use the get session to check the behavior is correct in the auto generated tests. This should be avoided.

Frameworks do support this though also discourage it. A producer should verify behavior in component and unit tests and not in API Contracts, they are not designed for behavior testing but for testing API interactions.

API Contracts Development Methodologies

Now that we understood how contracts work and the value that they provide to the consumer and the producer, we need to know how to use them as part of the development process.

There are two main Contract Development Methodologies:

- Producer Contract testing
- Consumer Driven Contracts

Each methodology has a specific purpose but can be used in conjunction with each other. A similar example is that of the Classicist and Mockist mentioned in unit testing, both methodologies can be used together.

Producer Contracts

Producer contract testing is when a producer defines its own contracts. If starting out on an API program and wanting to introduce contracts to a project then this is a great place to start. New contracts can be created for the producer to ensure that that service fulfils the interaction criteria and will continue to fulfil them. If Consumers are complaining that a producers API is breaking on new releases then introducing contracts can help with this issue.

The other common reason for using Producer contract testing is when there is a large audience for your API. This is defined as an API that is being used outside your immediate organization and by unknown users, i.e external third parties. When developing an API that has a wide public audience it will need to maintain its integrity and though it is something that is updated and improved, immediate feedback and individual feedback will not be applied. A concrete example int the real world is the Microsoft Graph API, it can be used to look at the users registered in an Active Directory Tenant. A consumer may find it preferable to have an additional for preferred name on

the Users endpoint of the API. This is something that can be messaged to Microsoft as a suggestion, however, this is not likely to be changed and if the suggestion was seen as a good idea it would unlikely happen quickly. This is something that would have to be weighed up and considered. Is this something that will be useful for others, is it backwards compatible change, how does this change the interactions across the rest of the service?

Here we take the same approach as the example given, with our Schedule API. We do not want consumers making suggested changes to the API that only benefits them. It is external facing and needs to be very solid, most importantly the structure of this API can not suddenly change. Breaking changes involve versioning which be be read about in "API Versioning".

Consumer Driven Contracts

Consumer Driven Contracts (CDC) is when a consumer drives out the functionality that they wish to see. Consumers submit contracts to the producer for new API functionality and the producer will choose to accept or reject the contract.

CDC is very much an interactive and social process. The owners of the applications that are consumers and producers should be within reach. Recall the contract we defined earlier in "API Contracts", where the Schedules Service is consuming from the Sessions Service. A new API interaction is desired and a contract is then defined. This contract is submitted as a pull request into the Sessions Service. The maintainers of the Sessions Service can then look at the PR and start reviewing this new interaction. At this point a discussion takes place about this new functionality to ensure that this is something that the Sessions service should and will fulfill and that the contract is correct. While this contract is to be a PUT request, a discussion can take place if this should be in fact a PATCH request. This is where a good part of the value of contracts comes from, this discussion for both parties about what the problem is, and using a contract to assert that this is what the two parties accept and agree to as a solution. Once the contract is agreed, the producer (Sessions service) accepts the contract as part of the project and can start fulfilling it.

As we've seen, contracts are required to pass as part of build. Therefore the next release of the Sessions service must fulfill this contract. From contracts the stubs can be generated which the consumer can use for local testing. Both services can develop independently of each other just by using contracts.

Contracts methodology overview

These methodologies should hopefully give an overview of how to use contracts as part of the development process. This should not be taken as gospel as variations do exist on the exact steps. For example, the consumer when writing the contract should also create a basic implementation of the producer code to fulfil the contract. In another example, the consumer should tdd the functionality they require and then create the contract before submitting the pull request. The exact process that is put in place may vary by team. By understanding the core concepts and patterns of CDC the exact process that is put in place is an implementation detail.

If starting out on a journey to add contracts it should be noted that there is a cost to it. This cost is the setup time to incorporate contracts into a project and also the cost of writing the contracts. It is worth looking at tooling that can create contracts for you based on an OpenAPI Specification. At the time of writing there are a few projects that are available, though none are actively maintained so it is difficult to recommend any.

API Contracts Storage and Publishing

Having seen how contracts work and methodologies of incorporating them to the development process the next question becomes where are contracts stored and how should they be published.

There are a few options for storing and publishing contracts and these again depend on the setup that is available to you and your organization.

Most commonly Contracts are stored along side the Producer code in version control (e.g., git). They can then be published alongside the Producer build into a artifact repository such as Artifactory. The contracts are then easily available for the Producer as part of the auto generated tests. Consumers that use the service can retrieve the contracts to generate stubs and can also

submit new contracts for the project. The Producer has control over the contracts in the project and can ensure that undesired changes aren't made or additional contracts are added. The downside to this approach is that in a large organization if can be difficult to find all the API services that use contracts.

Another option is to store all the contracts in a centralized location. A single location for all contracts is very useful as it also serves as a location to see other API interactions that are available. This central location would typically be a git repository, though if well organized could also be a folder structure. The downside to this approach is that unless organized and setup correctly it is possible and likely that contracts get pushed into a module that the producer has no intention on fulfilling.

Yet another option for storing contracts is to use a broker. The PACT contract framework has a broker product that can be used as a central location to host contracts. A broker can show all contracts that have been validated by the producer as the producer will publish those contracts have been fulfilled. A broker can also see who is using a contract to produce a network diagram, integrate with CI/CD pipelines and perform even more valuable information. With the pact broker a producer will be able to pull in submitted contracts to fulfill, it does not pull them in automatically. A broker seems to solve the problems of the previous suggestions, however, there is a setup cost and only the PACT framework has the PACT Broker product.

There are positives and negatives to each approach of storing contracts and it is tempting to think that a Broker is essential to an API Contract Testing program. When starting out in the world of contracts, following the standard of keeping contracts alongside a producer is great approach. It is possible to move to a broker later and should be noted that many users of contracts never use a broker and stick to the basics of contracts alongside the Producer.

API Contract Frameworks

We've explained what contracts are, how to use them as part of a project development and how they are stored and distributed. The next step is to choose a contract framework.

The good news is that there are multiple options that are available to choose from. When choosing it is valuable to see the languages that are supported, documentation and most importantly is it going to continue to be developed by the project maintainers. It is always possible to write your own, however, these frameworks handle setups that are common in industry.

As of writing the most popular API contract testing frameworks are:

- Pact.io
- Spring Cloud Contracts

Both frameworks support a range of languages though Pact has a large amount of native support in languages, while Spring Cloud Contracts uses a Docker container to support non-jvm languages.

Pact at its core embraces Consumer Driven Contracts and has been written for this purpose, Spring Cloud Contracts began more aimed at Producer Contract Testing. Both frameworks support both testing methodologies.

As Pact firmly embraces Consumer Driven Contracts, the consumer can define the contract which is a very nice way to develop an API.

It should be noted Spring Cloud Contracts does have partial support for the PACT Broker as it can read the PACT specification.

There is no clear cut rule about which framework is better, unfortunately as is always the case in software the answer is, it depends.

Please note that each contract framework has its own suggestions about publishing and storing contracts. Frameworks can support multiple different ways of storing and publishing so it is important to read the documentation to make an informed decision.

Checklist: Should we adopt API Contracts

Table 3-1. API Contracts Checklist

Decision	Do we need to API Contracts?

Discussion Points	Do we have APIs that keep breaking compatibility? Are producers making API changes that they shouldn't? Do we need to enforce consistency for our APIs Are consumers finding it hard to integrate with our APIs? Are consumers using real instances of services for tests. Do the teams work closely together and is there dialogue between producers and consumers. Are the resources available to introduce contracts?
Recommendations	Contracts should be added for external facing APIs that are being used by customers. Breaking compatibility and APIs for a paying customer is not acceptable. This is a simple and effective starting point. Contracts are not a silver bullet, teams need to work together for a successful contracts program. There is a cost involved in setting up contracts and maintaining them. In the long term Contracts can allow consumers to integrate more rapidly when a successful Contracts program has been established.

Introduction to Testcontainers

It is time to turn our attention to the Attendees Service that uses gRPC that we wish to integrate with and test this boundary. Here the Attendees Service team has created a simple docker image that contains a basic version of the Attendees Service that is a glorified stub. The Stub Attendees Service can be used to confirm that the Schedule service can make requests and receive a result. As this is gRPC a lot of this is generated for us, the proto files are effectively our contracts as they define the interactions and message formats. To learn more please read "Alternative API Formats". The important part here is testing across this boundary and TestContainers can help launch this stub service to verify this interaction.

Testcontainers is a library that integrates with your testing framework to orchestrate containers. Testcontainers orchestrates these ephemeral instances, in our case the stub Attendee Service, can be brought up and down for lifetime of our test.

Lets look at our integration test between the Schedule service and the Stub

Attendees Service. The strategy will be to start the container, read the connection details to the Stub Attendees Service and then connect to it for the test. Here is shown a the test setup for test containers

```
@SpringBootTest(
    classes = TestAttendeesServiceShould.DefaultConfig.class
)
class TestAttendeesServiceShould {

  private static GenericContainer container = new GenericContainer(
        DockerImageName
          .parse("docker.io/library/attendees-server:0.0.1-SNAPSHOT")
      )
      .withExposedPorts(9090);

  @DynamicPropertySource
  static void grpcClientProperties(DynamicPropertyRegistry registry) {
    container.start();
    registry.add("grpc.client.attendeesService.address",
        () -> "static://" + container.getHost()
        + ":" + container.getFirstMappedPort().toString()
    );
    registry.add("grpc.client.attendeesService.negotiationType",
        () -> "PLAINTEXT"
    );
  }

  @Autowired
  AttendeesService attendeesService;

  @BeforeEach
  void setUp() {
    Assertions.assertTrue(container.isRunning());
  }
```

Let us describe some of the setup of this test:

- The service is using a plaintext connection.
- The Spring annotation `@SpringBootTest` is used to help setup the testing.
- The `AttendeesService` is autowired in, this is the class undertest.

- The `@DynamicPropertySource` contains the configuration of the connection to the Attendee Service. This is required as when the container is started, it needs to be interrogating so information about the container can be supplied as property values. This is not the most idiomatic way to perform this task, however, this highlights very explicitly what is happening to the reader.

With all the setup clarified we can now look at our tests. A validation that the test container has started is performed in the `@BeforeEach` section of the test. This is not required, however, has been left in for brevity and also for sanity to confirm the container is running.

Here we see a test that validates that it is possible to get a list of attendees. Again it is important to note that this is a Stub Attendees Service and the response to this service has been hard coded to return the name of “James”

```
@Test
void return_an_attendee() {
  final AttendeeResponse attendees = attendeesService.getAttendees();
  final Attendee jamesAttendee = attendees.getAttendees(0);
  Assertions.assertEquals("James", jamesAttendee.getGivenName());
}
```

This is a sample of integration tests that are validating across a boundary. Again it is important to acknowledge that just one boundary integration is being validated and not any other components of the service.

Other dockerized services

The Testcontainers setup has shown a boundary being tested using docker images, and this same setup has been used in the Schedules Service to test a database connection . The options for testing integration boundaries for a database would be mocking out the Database, use an inMemory Database e.g. H2 or run a local version of the database using Testcontainers.

Testcontainers is a really powerful tool and should be considered when testing boundaries between any external services. Other common external boundaries that benefit from using Testcontainers include Kafka, Redis,

NGINX.

Service testing Summary

We have covered a lot on service testing which includes testing your module and external modules. In case of integration testing we looked at integrating with a RESTful API, gRPC service and suggested how to test a database. All this is being performed without leaving the development environment. With service testing we can clearly see how the Schedules service we are building is able to work with its ingress and egress points, what we do not know is how it works as a whole when the other parts of the Conference System are deployed. Next we look at the UI testing portion of the test pyramid to help us answer this question.

> **NOTE**
>
> It is important to reiterate that though Service testing has been the largest section of this chapter Unit tests should still be the foundation of any module being developed. APIs have a lot of options for testing at a service level and we've presented these options so you can learn what might work best for your use case.

UI Testing

UI tests for an API seems a bit of an oxymoron. What we are designing is an API for other applications to call our system. If it is a UI that calls our system, fine. If it is a server side service or a cURL request that calls our system that is fine as well. The common misconception is that UI testing involves some sort of Web UI like a Single Page Application (SPA). However, in reality the UI can be cURL, POSTman or a SPA app.

So, what exactly is UI testing? The UI part of the the testing pyramid can mean a few things. There are many interpretations—for some this is testing with a Web UI with a partially stubbed out backend service and ensuring the UI works, for others it means End to End testing the whole system works and the other meaning is to run set of critical stories end to end driven through the

UI. In reality it is some or all of these and it is up to you to understand what it is that you need. Across these examples of a UI test, a general observation is that multiple things are interacting together and all of them sound complex. This is why these tests are at the top of the testing pyramid. They are difficult to create and maintain and also take time to run.

The return for these tests is verifying the interactions from your UI and having validation that from an outside perspective the service is running correctly.

Having a full UI-driven test is really important for confidence and certainty that your module is working as expected. Let us reiterate the point, UI tests are complicated and should not be the dominant test of your API. It is incorrect to assume their is more value testing the system as a whole than the parts that make up the system. This is a cheap investment fallacy, please have a look at the excellently written article about this topic by Steve Smith.

WIth UI tests, we are looking to test is the module that is being built and is working correctly in its domain. When it deploys it should work with the other services that it calls and executes with and works as part of the system. With our UI test we look at putting our newly developed or updated service into the mix with real instances of the rest of the system as we are looking to ensure that we have the correct behavior. It is okay to stub out some entities of a system as this may be an external provider such as AWS S3 and you really do not want to run your tests and have to rely on an external entity where the network can go down.

When starting to UI test, define the boundary of your service and what should be tested. If we look at the Sessions service we can see that the service is being triggered through the Schedule service. This is a good point to ask, is a test for the UI required? Or does a Kafka cluster (which is not used, in this instance) need to be spun up? The answer to both questions is no. There is no benefit to this. Being able to define your boundary is useful for not trying to launch everything at once and make it more difficult to test. It could be said that in this case that it is fine to bring everything up as this could be done with Docker Compose and be run on most peoples laptop. However, in

industry where you have 100’s or thousands of applications this does not make sense, especially when half of them may not talk to your system.

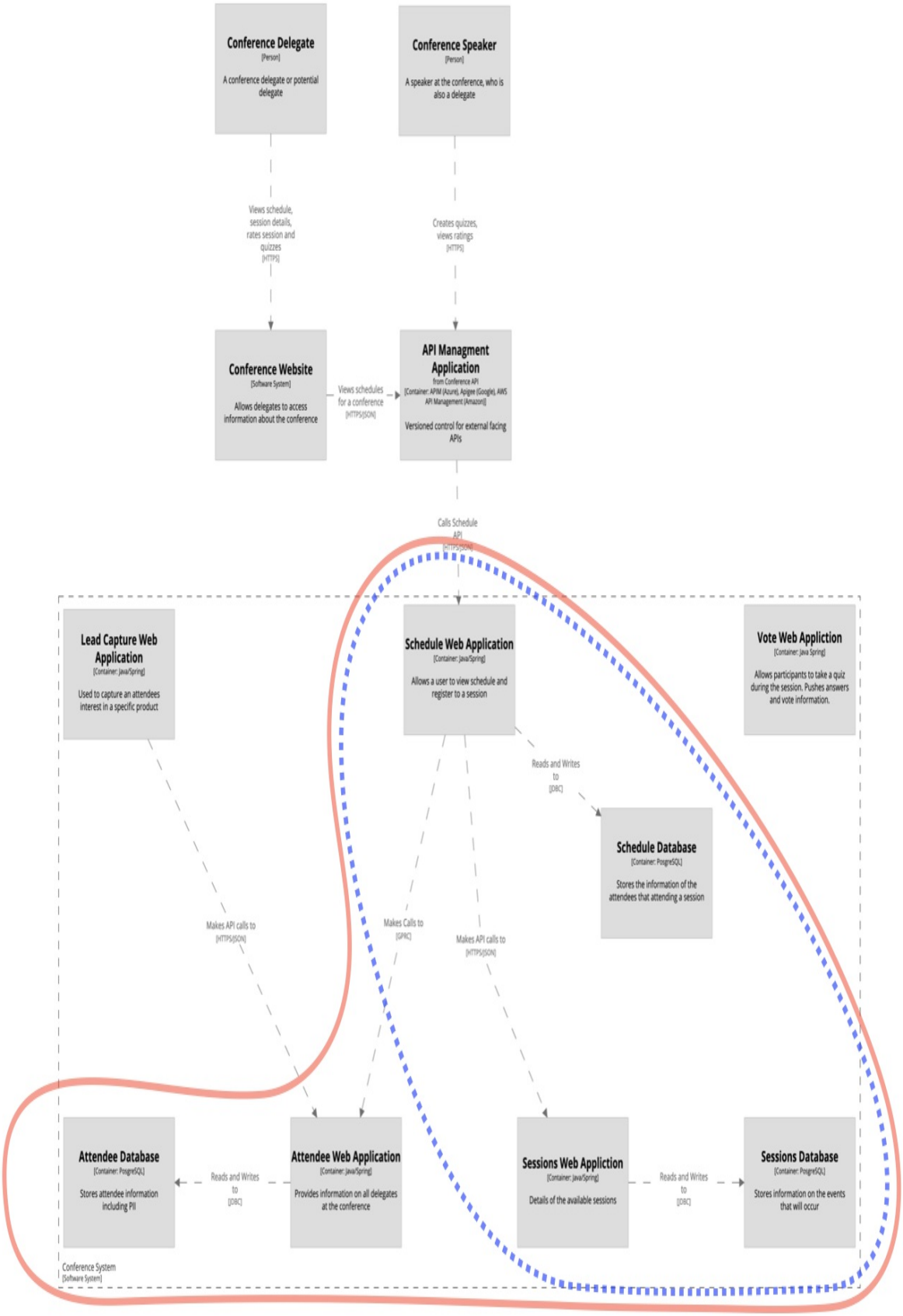

Conference Delegate
[Person]
A conference delegate or potential delegate
Conference Speaker
[Person]
A speaker at the conference, who is also a delegate
Views schedule, session details, rates session and quizzes
[HTTPS]
Creates quizzes, views ratings
[HTTPS]
Conference Website
[Software System]
Allows delegates to access information about the conference
Views schedules for a conference
[HTTPS/JSON]
API Managment Application
from Conference API
[Container: APIM (Azure), Apigee (Google), AWS API Management (Amazon)]
Versioned control for external facing APIs
Calls Schedule API
[HTTPS/JSON]
Lead Capture Web Application
[Container: Java/Spring]
Used to capture an attendees interest in a specific product
Schedule Web Application
[Container: Java/Spring]
Allows a user to view schedule and register to a session
Vote Web Appliction
[Container: Java Spring]
Allows participants to take a quiz during the session. Pushes answers and vote information.
Reads and Writes to
[JDBC]
Schedule Database
[Container: PosgreSQL]
Stores the information of the attendees that attending a session
Makes API calls to
[HTTPS/JSON]
Makes Calls to
[GPRC]
Makes API calls to
[HTTPS/JSON]
Attendee Database
[Container: PosgreSQL]
Stores attendee information including PII
Reads and Writes to
[JDBC]
Attendee Web Application
[Container: Java/Spring]
Provides information on all delegates at the conference
Sessions Web Appliction
[Container: Java/Spring]
Details of the available sessions
Reads and Writes to
[JDBC]
Sessions Database
[Container: PosgreSQL]
Stores information on the events that will occur
Conference System
[Software System]

Figure 3-9. Test boundaries for the Schedule service

The visual of our conference system in Figure 3-9 shows the boundaries that we expect to to use as a valid UI test. The two boundaries are:

- The dotted line of the Sessions and Schedule Service with the associated databases
- The solid line, which contains the Attendees Service on top of all the services in the dotted line.

How should we test the Sessions Service and the Schedule service? The Sessions service is consumed by a WebUI (conference website) through the API Management layer and so it makes sense to add some UI tests to verify this behavior.

In our case here we have a WebApp that is directly in our control. This WebUI is part of our overall domain and so it is easy for us to test using a tool like Selenium. Selenium is a tool that drives a browser and can run tests such as entering data, executing button clicks and generally mimicking a users journey thorough a WebApp. As was mentioned there is a cost to these types of tests, especially with setting up. Is there a way to avoid a UI test, in this instance?

Subcutaneous Tests

Subcutaneous tests are tests that run just below the UI. An example is that instead of filling out a form and pressing the submit button on the UI. For a subcutaneous test the call would be made directly from a web client, e.g., cURL. Martin Fowler discusses using these types of tests “when you want to test end-to-end behavior, but it’s difficult to test through the UI itself”.

Using this technique is a great way to reduce the cost of setting up a UI driven test and still be able to get the full functionality of testing through your system.

If Subcutaneous tests are so good then why do they not replace UI driven tests? There are things that a UI may need or use that are not required when

just running against the API directly via a web client. This includes things such as CORS or CSRF. When running tests by driving a test through the UI the requests and stories will be created using realistic payloads. This is sometimes forgotten with subcutaneous tests as values such as name will be set to “d” instead of realistic value such as “Daniel”. Realistic payloads provide value as it is too common to find cases where a test passes in development, but then in production a consumer sends a payload that is far bigger then was tested with and it breaks some validation rule.

> **WARNING**
>
> If you are building an external facing API and you have multiple third parties that are consuming it, don’t try and copy the third party UI to replicate how their UI works. Doing so will mean that huge amounts of time will be spent trying to replicate something out of your domain.

Behavior Driven Development

Behavior Driven Development (BDD) is a development methodology where a behavior is defined that a system should fulfil. Like TDD, in BDD a test is written first and then the application will fulfil the behavior. Behavior should drive the implementation.

BDD has been discussed in this UI Section as normally BDD is done at a top level on the system or, groupings of components or services.

A common misconception is that BDD is that it is about creating automated test stories Test stories are tests in a defined ordering that look a bit like a narrative, thus these are known as stories. However, BDD is actually about communication between developers, business users and QA/testers and discussing how a service should behave. A common language between the parties should be used so that no software tools are required.

BDD is about writing tests as user stories. These user stories consist of scenarios.

```
TITLE: The Schedule being used by an Attendee
```

```
Scenario 1:
Given an Attendee requests a Schedule
When they are registered for the day
Then a schedule will be returned to them

Scenario 2:
Given an Attendee requests a Schedule
When there is no conference scheduled on that  day
Then no schedule is returned
```

These scenarios can be documented and ran through by a tester. These user stories should be viewed as living documentation. If the tester runs through the user stories and they fail then either the service is broken or something has changed and the story needs to updated.

Think of the user stories as the business requirements that are being verified.

As mentioned a common language should be used between all the parties, this makes it possible to create or use an already well made DSL (Domain Specific Language). A side-effect is that tools need be developed to read this DSL and only then it is possible to have automated tests defining how a service should behave. An example of tool for BDD is cucumber where a DSL is used to then generate automated tests.

UI test summary

UI Testing is a good value add at the top of the pyramid to give us confidence in our system. The cost to build and maintain these tests is greater than the other tests, but this should not discourage you from using them. As long as these tests are concentrated on core journeys of the user and not edge cases. UI tests are not for just testing a UI as in most UI frameworks it is possible to add unit tests to a WebApp such as Jasmine for JavaScript.

Testing during the building of your application

When building your application you have the opportunity to verify your module is backwards compatible and has a valid OpenApi Specification. If the module you are changing is being driven by specification first or code

first the OpenApi spec can be obtained and verified. It is possible then to add check to a GitHub Action, an Jenkins build pipeline or any other build tool.

We covered Open API diff tools and what they provide in Chapter 2 "OpenAPI Specification and Versioning".

Can you test too much?

Testing gives us confidence and though it is recommended more time be spent on writing tests over business logic there is a point where too much testing can be done. If too much time is spent working on tests then the module will never be delivered. A balance is always needed to have a good test coverage and provide confidence. Being smart about what should be tested and what is irrelevant, such as creating tests that duplicate scenarios is a waste of resources.

An Architect must be able to recognize where the boundary is for excessive testing for an API/module/application, as its value for customers and the business is only realized when running in production.

Summary

In this chapter we have covered the core requirements to test an API, including what should be tested and where time should be dedicated. Key takeaways are as follows:

- By sticking to the fundamentals of the Unit Testing and using TDD for the core of the application.
- Performing service tests on your component and isolating the integrations to validate incoming and outgoing traffic.
- Then finally using UI tests to validate your modules all integrate together and verifying this using journeys of core interactions.

This should deliver a solid product for your customers.

While we've given you lots of information, ideas and techniques for building a quality API, this is by no means an exhaustive list of tools available. We encourage you to do some research on frameworks and libraries that you may want to use, to ensure you are making an informed decision.

However, no matter how much testing is done upfront, nothing is as good as seeing how an application actually runs in production. To learn more about testing in production refer next to Chapter 5.

SUGGESTED READING MATERIAL ON SOFTWARE CRAFT

Software craft is about encouraging high quality development and to continuously learn and improve, like a craft such as basket weaving.

- The Software Craftsman: Professionalism, Pragmatism, Pride
- The Pragmatic Programmer: From journeyman to master
- Manifesto for Software craftsmanship

---

1 The authors' friend owns Titan Mouthguards. One of the authors was on the receiving end of hearing about the arduous process for testing the integrity of the product. A mouthguard must use simulations, stress testing and other such integrity tests. No one wants a mouthguard where the only/majority of testing takes place in on the Rugby field!

2 The Classicist is also known as the Chicago School and Mockist as the the London School.

Chapter 4. API Gateways

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be Chapter 4 of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at feedback@mastering-api.com.

Now that you have a good understanding of the life of an API, the protocols involved, and how to begin testing, we can turn our attention to platforms and tooling that are responsible for delivering APIs to end users in production. An API gateway is a critical part of any modern technology stack, sitting at the "edge" of systems and acting as a management tool that mediates between a client and a collection of backend services.

In this chapter you will learn about the "what," "where," and "why" of API gateways, and also explore the history of the API gateway and other edge technologies. In order to build upon the success of previous generations of engineers and developers, it is important that you understand the evolution of this technology and examine how this relates to important architectural concepts like coupling and cohesion.

You will also explore the taxonomy of API gateways, and learn how these fit into the bigger picture of system architecture and deployment models. You will revisit your earlier exploration of ingress ("north-south") traffic and service-to-service ("east-west") traffic, and explore the technologies that can be used to manage each traffic type. You will also explore the challenges and

potential antipatterns when mixing traffic patterns and technologies.

Building on all of the topics above, you will conclude the chapter by learning how to select an appropriate API gateway based on your requirements, constraints, and use cases.

What Is an API Gateway?

In a nutshell, an API gateway is a management tool that sits at the edge of a system between a client and a collection of backend services and acts as a single point of entry for a defined group of APIs. The client can be an end-user application or device, such as a single page web application or a mobile app, or another internal system or third-party application or system.

As introduced in Chapter 1, an API gateway is implemented with two high-level fundamental components, a control plane and data plane. The control plane is where operators interact with the gateway and define routes, policies, and required telemetry, and the data plane is the location where all of the work specified in the control plane occurs; where the network packets are routed, the policies enforced, and telemetry emitted.

What Functionality Does an API Gateway Provide?

At a network level an API gateway typically acts as a reverse proxy to accept all of the API requests from a client, calls and aggregates the various application-level backend services (and potentially external services) required to fulfill them, and returns the appropriate result. An API gateway provides cross-cutting requirements such as user authentication, request rate limiting, timeouts/retries, and observability. Many API gateways provide additional features that enable developers to manage the lifecycle of an API, assist with the onboarding and management of developers using the APIs (such as providing a developer portal and related account administration and access control), and provide enterprise governance.

Where Is an API Gateway Deployed?

An API gateway is typically deployed at the edge of a system, but the definition of “system” in this case can be quite flexible. For startups and many small-medium businesses (SMBs) an API gateway will often be deployed at the edge of the data center or cloud. In these situations there may only be a single API gateway (deployed and running via multiple instances for high availability) that acts as the front door for the entire back end estate, and the API gateway will provide all of the edge functionality discussed in this chapter via this single component.

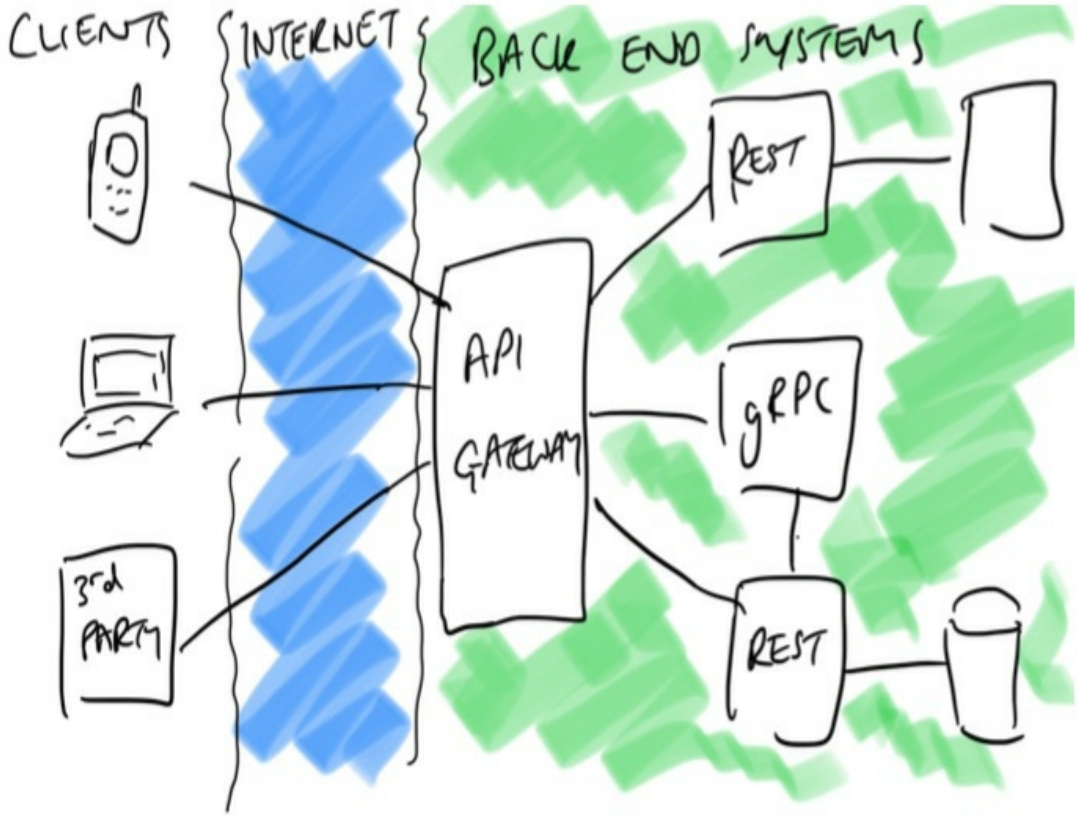


Figure 4-1. A typical startup/SMB API gateway deployment

For large organizations and enterprises an API gateway will typically be deployed in multiple locations, often as part of the initial edge stack at the perimeter of a data center, and additional gateways may be deployed as part of each product, line of business, or organizational department.

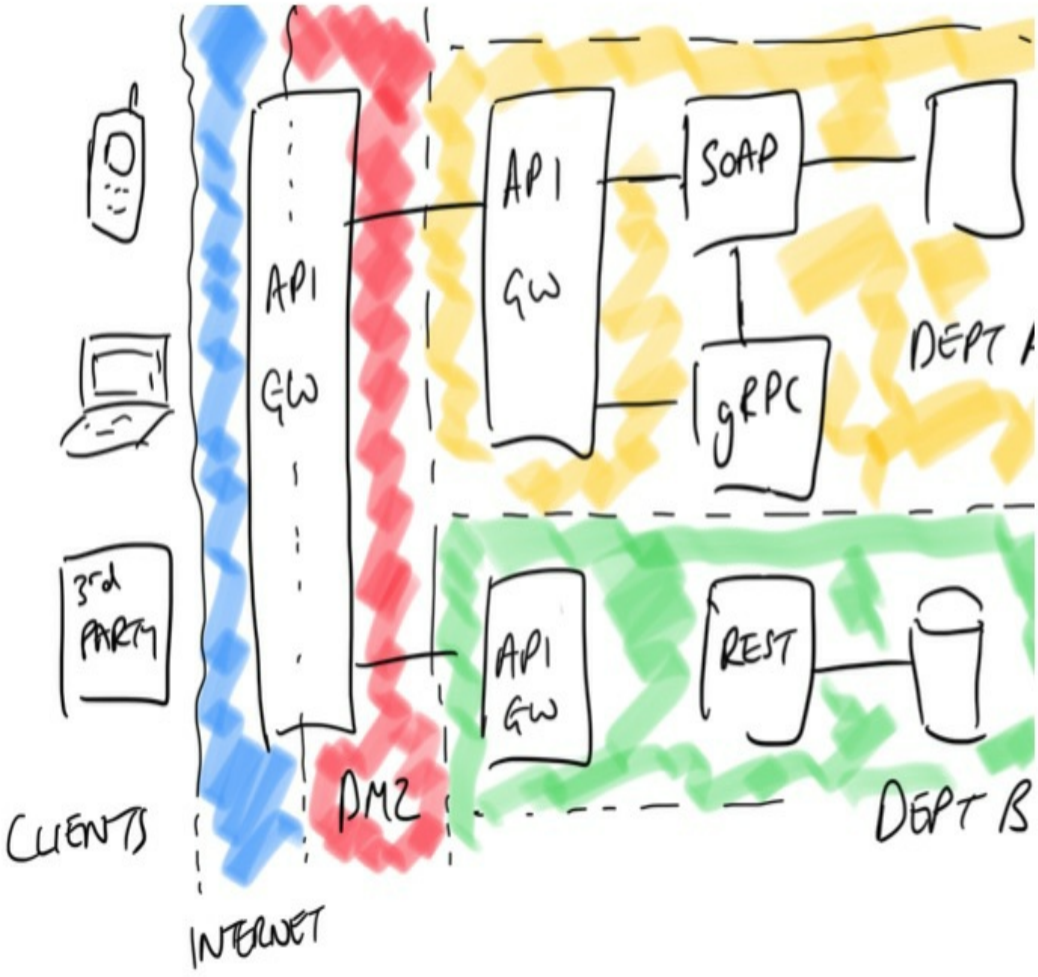


Figure 4-2. A typical large/enterprise API gateway deployment

As you will learn later in this chapter, the definition and exact functionality offered within an API gateway isn't always consistent across implementations, and so the diagrams above should be thought of as more conceptual rather than logical.

How Does an API Gateway Integrate with a Typical Edge Stack?

There is typically many components deployed at the edge of an API-based system. This is where the clients and users first interact with the backend, and hence many cross-cutting concerns are best addressed here. Therefore, a modern "edge stack" provides a range of functionality that meets essential

cross functional requirements for API-based applications. In some edge stacks each piece of functionality is provided by a separately deployed and operated component, and in others the functionality and/or components are combined. You will learn more about the individual requirements in the next section of the chapter, but for the moment the diagram below should highlight the key layers of a modern edge stack.

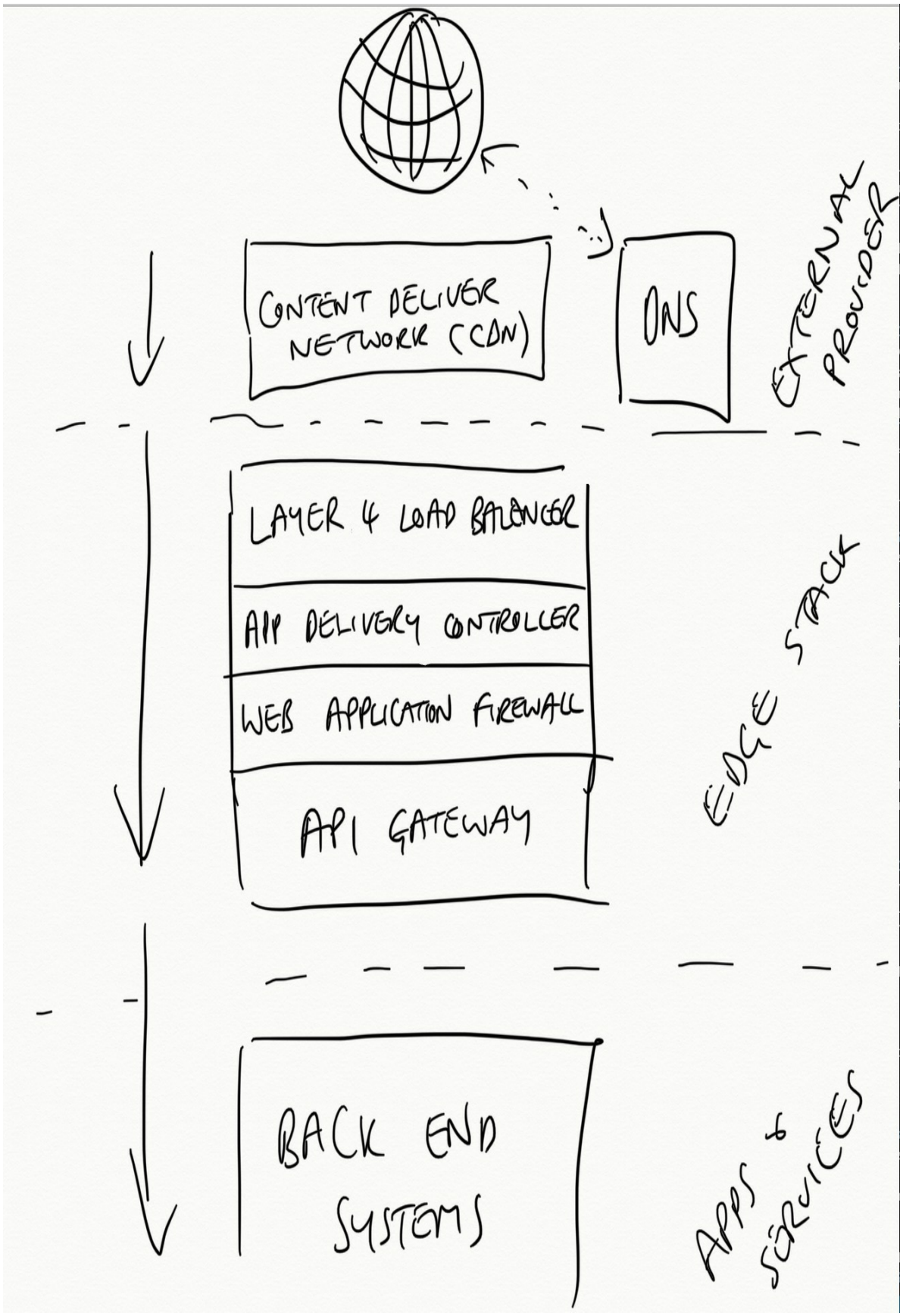

CONTENT DELIVER NETWORK (CDN)
DNS
EXTERNAL PROVIDER
LAYER 4 LOAD BALENCER
APP DELIVERY CONTROLLER
WEB APPLICATION FIREWALL
API GATEWAY
EDGE STACK
BACK END SYSTEMS
APPS & SERVICES

Figure 4-3. A modern edge stack

Now that you have a good idea about the "what" and "where" of an API gateway, let's now look at why an organization would use an API gateway.

Why Use an API Gateway?

Establishing clearly the business goals, technical challenges, and organizational constraints before choosing a solution that involves a piece of technology is a big part of the modern software architect's role. It can be tempting to deploy technology for technology's sake, or add layer after layer of technology to a system—who hasn't heard the cliche that every problem in software can be solved with another layer of indirection. It is therefore vital that you understand the goals, benefits, and drawbacks of modern API gateway technology, and can clearly articulate they "why" of this component to both technical and non technical stakeholders. This section of the chapter will provide you with an overview of the key problems that an API gateway can address, such as:

- Reduce Coupling: Adapter / Facade Between Front Ends and Back Ends
- Simplify Consumption: Aggregating / Translating Back End Services
- Protect APIs from Overuse and Abuse: Threat Detection and Mitigation
- Understand How APIs Are Being Consumed: Observability
- Manage APIs as Products: API Lifecycle Management
- Monetize APIs: Account Management, Billing, and Payment

Reduce Coupling: Adapter / Facade Between Front Ends and Back Ends

Two fundamental concepts that every software architect learns about early in their career are coupling and cohesion. You are taught that systems that are designed to exhibit loose coupling and high cohesion will be easier to understand, maintain and modify. Loose coupling allows different implementations to be swapped in easily, and internals to be modified without experiencing unintended cascading effects on surrounding modules or systems. High cohesion promotes understandabilty—i.e., all code in a module or system supports a central purpose—and reliability and reusability. In our experience, APIs are often the locations in a system in which the architectural theory meets the reality; an API is quite literally and figuratively an interface that other engineers integrate with.

BOUNDED CONTEXT, CONTRACTS, AND COUPLING

You've already learned in earlier chapters that care should be taken with design of your API, as this is effectively a contract with your consumers. Like many tools, an API gateway can be used for good and evil: you can use functionality in the gateway to provide a more business-focused and cohesive API than the backend components may otherwise expose, but you can also add additional layers of indirection and complexity. Learning more about domain-driven design can provide you with useful knowledge about exposing systems, bounded contexts, and aggregates with the appropriate level of abstraction.

An API gateway can act as a single entry point and an adapter or facade (two classic Gang of Four design patterns), and hence promote loose coupling and cohesion. Clients integrate with the API exposed at the gateway, which, providing the agreed upon contract is maintained, allows components at the backend to change location, architecture, and implementation (language, framework, etc.) with minimal impact.

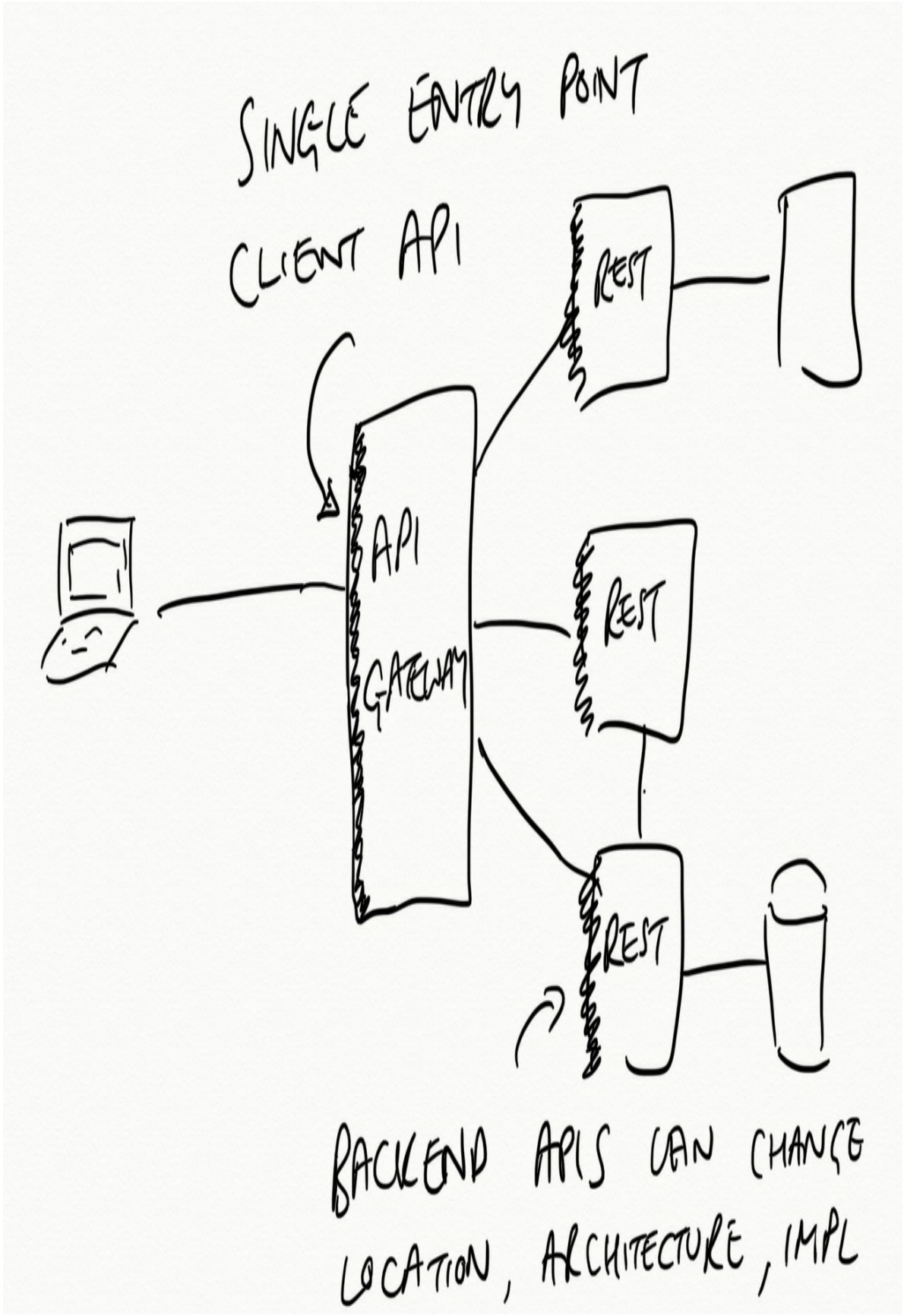

SINGLE ENTRY POINT
CLIENT API
REST
API
GATEWAY
REST
REST
BACKEND APIS CAN CHANGE
LOCATION, ARCHITECTURE, IMPL

Figure 4-4. An API gateway providing a facade between front ends and back ends

BACKENDS FOR FRONTENDS (BFF): SINGLE POINT OF ENTRY?

A popular architectural pattern for exposing APIs or back end functionality across multiple devices (or "experiences", as this is often referred to) is the "backend for frontend (BFF)" pattern, popularized by Phil Calçado and the SoundCloud team. The BFF pattern can be implemented with in one of two styles: firstly, with one API gateway acting as a single point of entry that provides cross-cutting concerns, and which acts as an adapter to multiple BFF back ends; and secondly, with the use of multiple API gateways, each one providing cross-cutting concerns that are coupled to a specific BFF back end (e.g. "iOS API gateway and BFF," "Web API gateway," etc.). The first approach is commonly seen in enterprise organizations with strict governance requirements at the edge, where all traffic must be funneled through a single point of entry. The second is often used within smaller organizations when the cross-cutting concerns are different for each back end, e.g. differing authentication, protocols, and resiliency requirements.

Simplify Consumption: Aggregating / Translating Back End Services

Building on the discussion of coupling in the previous section, it is often the case that the API you want to expose to the front end systems is different than the current interface provided by a back end or composition of backend systems. For example, you may want to aggregate the APIs of several back end services that are owned by multiple owners into a single client-facing API in order to simplify the mental model for front end engineers, streamline data management, or hide the back end architecture. GraphQL is often used for exactly these reasons.

ORCHESTRATING CONCURRENT API CALLS

A popular simplification approach implemented in API gateways is orchestrating concurrent backend API calls. This is where the gateway orchestrates and coordinates the concurrent calling of multiple independent backend APIs. Typically you want to call multiple independent and non-coupled APIs in parallel rather than sequentially in order to save time when gathering results for the client. Providing this in the gateway removes the need to independently implement this functionality in each of the clients.

It is also common within an enterprise context that some protocol translation will be required. For example, you may have several “heritage” (money making) systems that provide SOAP-based APIs, but you only want to expose REST-like APIs to clients. Or your legacy systems may only support HTTP/1.1, but clients require HTTP/2 connections. Some organizations may implement all internal service APIs via gRPC and Protobuf, but want to expose external APIs using HTTP and JSON. The list goes on; the key point here is that some level of aggregation and translation is often required to meet externals requirement or provide further loose coupling between systems.

An API gateway can provide this aggregation and translation functionality. Implementations vary and can be as simple as exposing a single route and composing together (“mashing”) the responses from the associated multiple internal system, or providing a protocol upgrade from HTTP/1.1 to HTTP/2, all the way through to mapping individual elements from an internal API to a completely new external format and protocol.

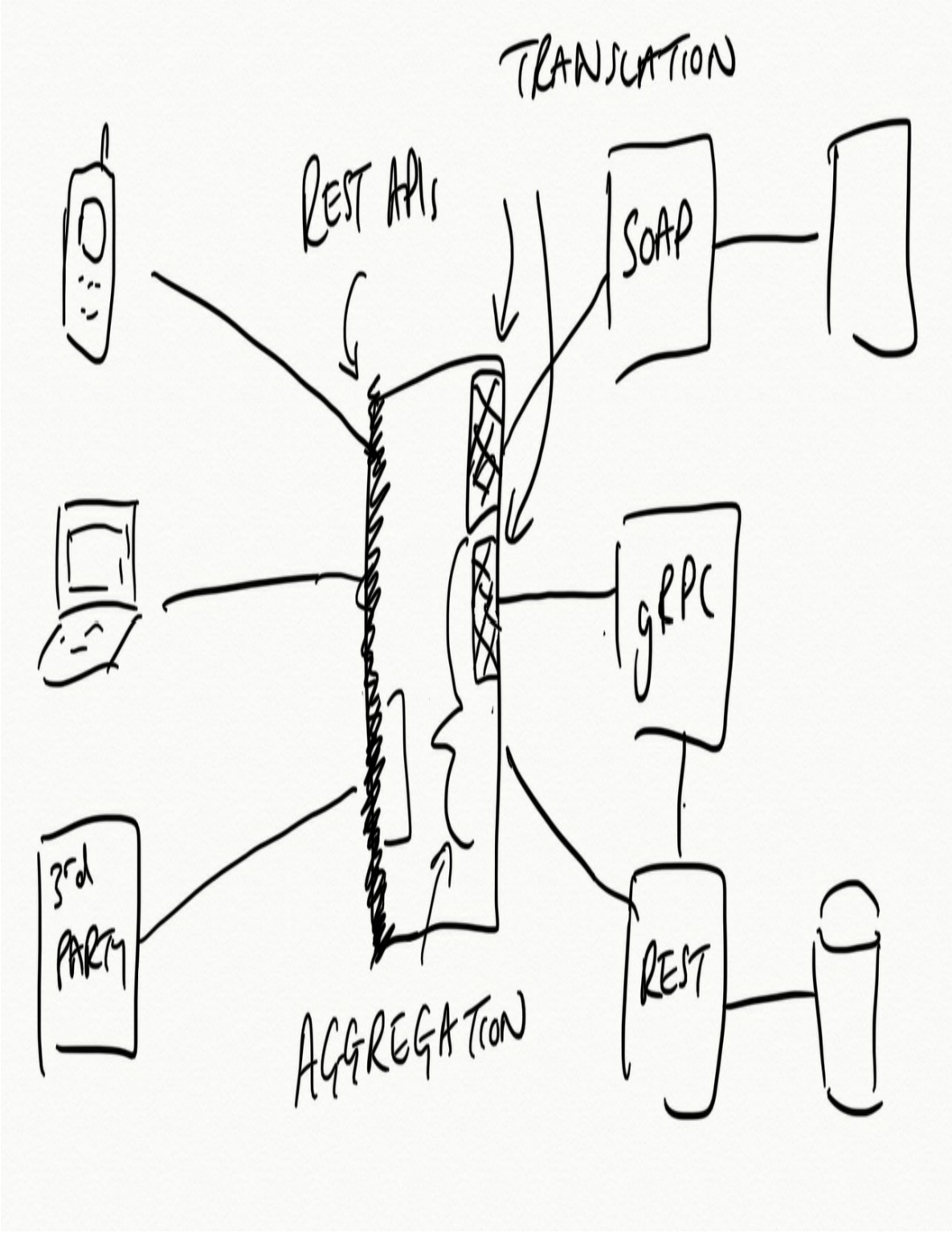


Figure 4-5. An API gateway providing aggregation and translation

Protect APIs from Overuse and Abuse: Threat Detection

and Mitigation

The edge of a system is where your users first interact with your applications. It is also often the the point where bad actors and hackers first encounter your systems. Although the vast majority of enterprise organizations will have multiple security-focused layers to their edge stack, such as a content delivery network (CDN) and web application firewall (WAF), and even a perimeter network and dedicated demilitarised zone (DMZ), for many smaller organizations the API gateway can be the first line of defense. For this reason many API gateways include security-focused functionality, such as TLS termination, authentication/authorization, IP allow/deny lists, WAFs (either inbuilt or via external integration), and rate limiting and load shedding.

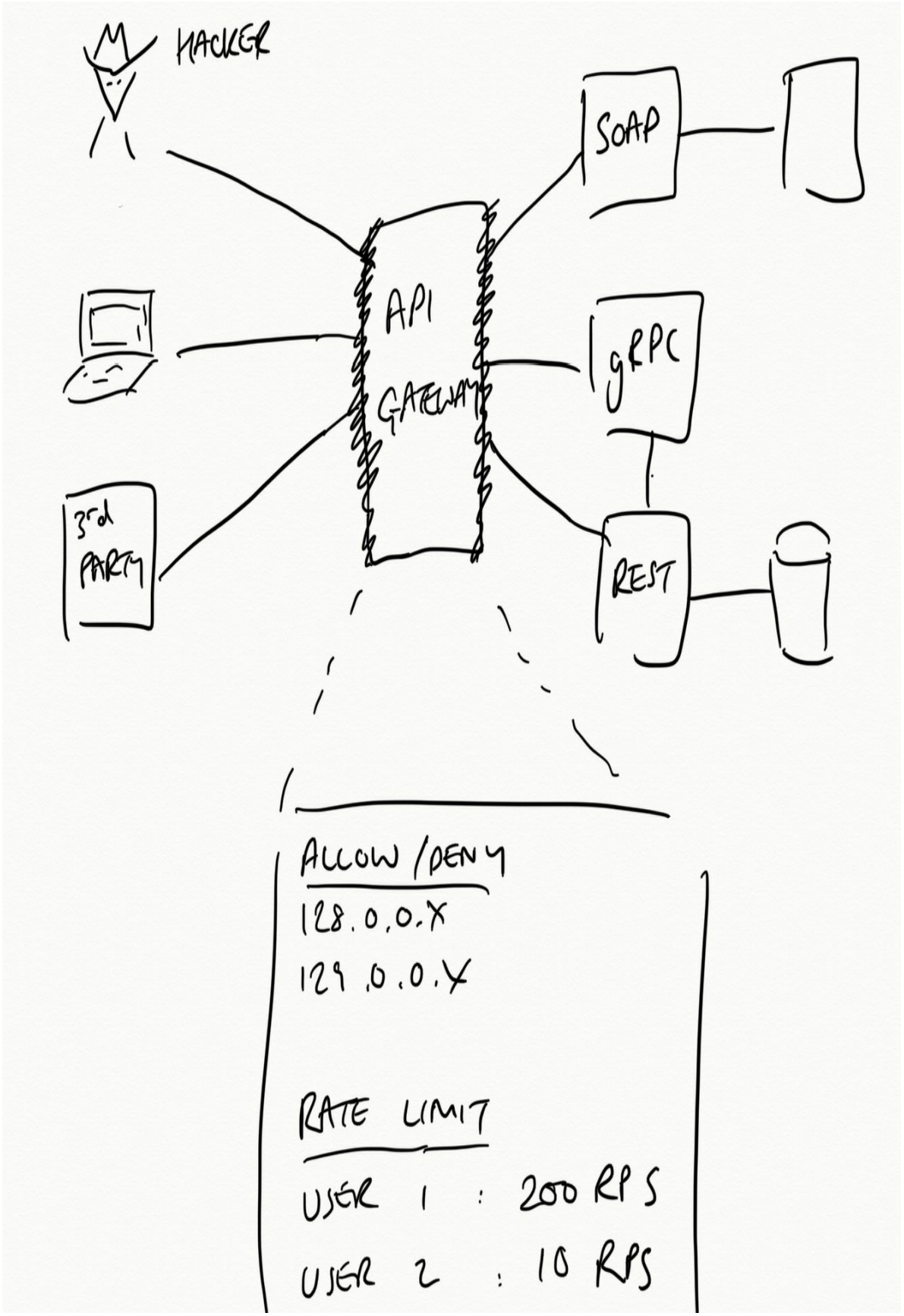

HACKER
SOAP
API
GATEWAY
gRPC
3rd
PARTY
REST
ALLOW / DENY
128.0.0.X
129.0.0.X
RATE LIMIT
USER 1 : 200 RPS
USER 2 : 10 RPS

Figure 4-6. API gateway overuse and abuse

A big part of this functionality is the capability to detect API abuse, either accidental or deliberate, and for this you wil need to implement a comprehensive observability strategy.

Understand How APIs Are Being Consumed: Observability

Understanding how systems and applications are performing is vitally important for ensuring business goals are being met and that customer requirements are being satisfied. It is increasingly common to measure business objectives via key performance indicators (KPIs), such as customer conversion, revenue per hour, stream starts per second etc. Infrastructure and platforms are typically observed through the lens of service level indicators (SLIs), such as latency, errors, queue depth, etc. As the vast majority (if not all) of user requests flow through the edge of a system, this is a vital point for observability. It is a ideal location to capture top-line metrics, such as the number of errors, throughput, and latency, and it is also a key location for identifying and annotating requests (potentially with application-specific metadata) that flow throughout the system further upstream. Correlation identifiers (such as OpenZipkin b3 headers) are typically injected into a request via the API gateway and are then propagated by each upstream service. These identifiers can then be used to correlate log entries and request traces across services and systems.

Although the emitting and collecting of observability data is important at the system-level, you will also need to think carefully how to process, analyse, and interpret this data into actionable information that can then be used to drive decision making. Creating dashboards for visual display and manipulation, and also defining alerts are vital for a successful observability strategy. Cindy Sridharan's O'Reilly book Distributed Systems Observability is a great primer for this topic.

Manage APIs as Products: API Lifecycle Management

Modern APIs are often designed, built, and run as products that are consumed by both internal and third-parties, and must be managed as such. Many large organizations see APIs as a critical and strategic component within their business, and as such will create a API program strategy, and set clear goals, constraints, and resources. With a strategy set, the day-to-day tactical approach is often focused on application lifecycle management. Full lifecycle API management spans the entire lifespan of an API that begins at the planning stage and ends when an API is retired. Engineers interact with an associated API gateway, either directly or indirectly, within many of these stages, and all user traffic flows through the gateway. For these reasons, choosing an appropriate API gateway is a critical decision.

There are multiple definitions for key API lifecycle stages. The Swagger and SmartBear communities define the five key steps as: planning and designing the API, developing the API, testing the API, deploying the API, and retiring the API.

The 3Scale and Red Hat teams define thirteen steps:

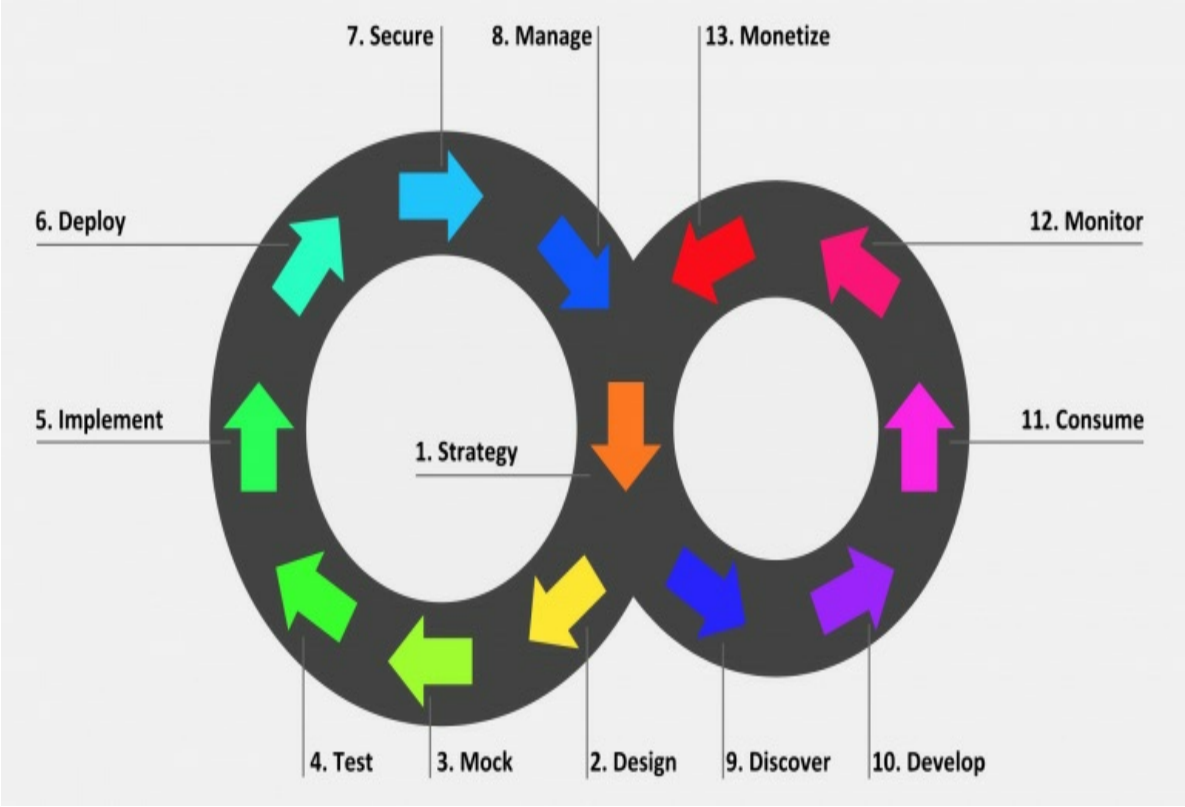


Figure 4-7. The 3Scale and Red Hat teams approach to full API lifecycle management

The Axway team strike a good balance with 3 key components—create, control, and consume—and 10 top stages of an API lifecycle:

Building

Designing and building your API.

Testing

Verifying functionality, performance, and security expectations.

Publishing

Exposing your APIs to developers.

Securing

Mitigating security risks and concerns.

Managing

Maintaining and managing APIs to ensure they are functional, up to date, and meeting business requirements.

Onboarding

Enabling developers to quickly learn how to consume the APIs exposed. For example, offering OpenAPI or ASyncAPI documentation, and providing a portal and sandbox.

Analyzing

Enabling observability, and analyzing monitoring data to understand usage and detect issues.

Promoting

Advertising APIs to developers, for example, listing in an API Marketplace.

Monetizing

Enabling the charging for and collection of revenue for use of an API. We cover this aspect of API lifecycle management as a separate stages in the next section.

Retirement

Supporting the deprecation and removal of APIs, which happens for a variety of reasons including, business priority shifts, technology changes, and security concerns.

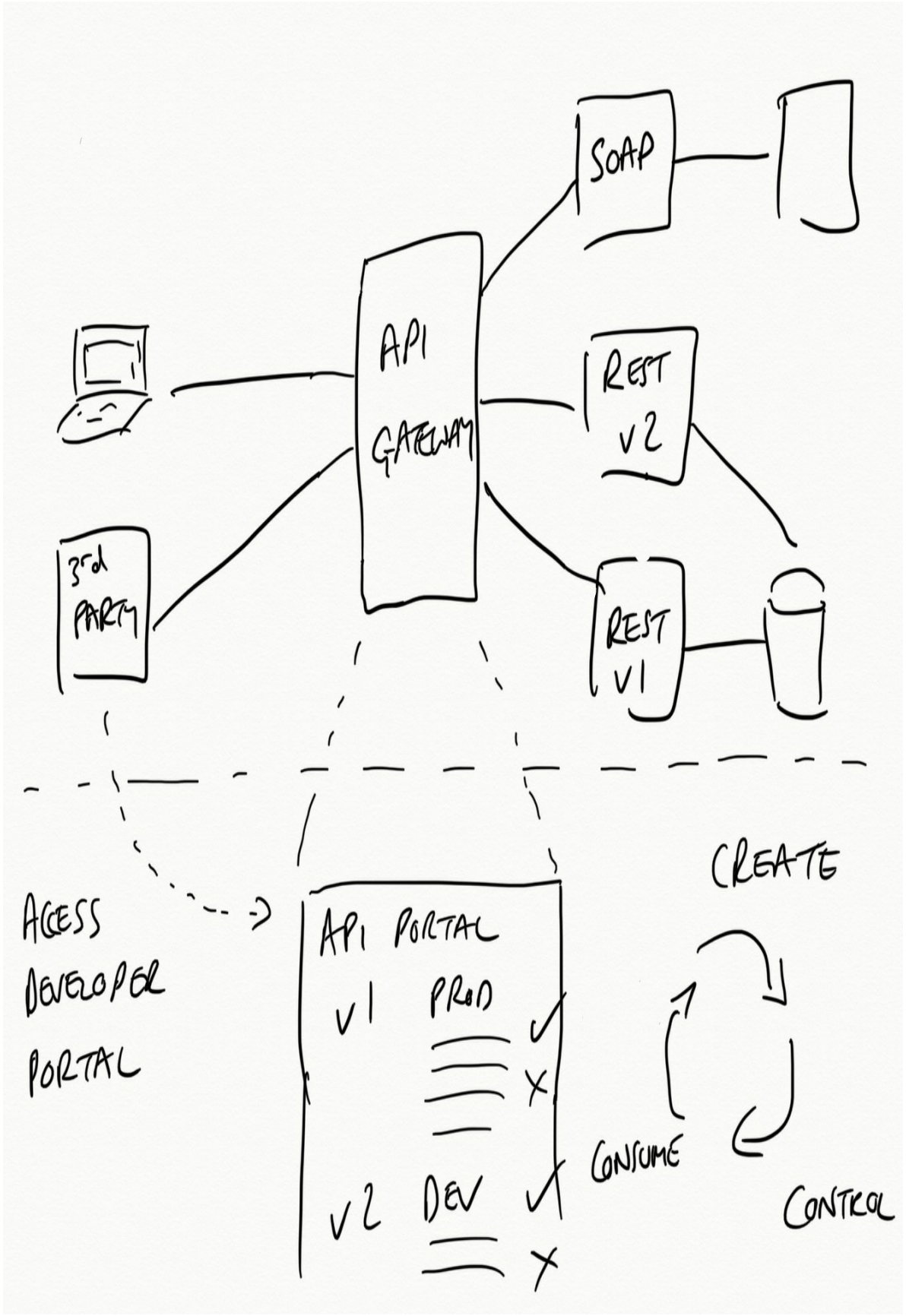

SOAP
API
GATEWAY
REST
v2
3rd
PARTY
REST
v1
ACCESS
DEVELOPER
PORTAL
API PORTAL
v1 PROD
v2 DEV
CREATE
CONSUME
CONTROL

Figure 4-8. API gateway lifecycle management

Monetize APIs: Account Management, Billing, and Payment

The topic of billing monetized APIs is closely related to API lifecycle management. The APIs being exposed to customers typically have to be designed as a product, and offered via a developer portal that also includes account management and payment options. Many of the enterprise API gateways include monetization, such as Apigee Edge and 3Scale. These payment portals often integrate with payment solutions, such as PayPal or Stripe, and enable the configuration of developer plans, rate limits, and other API consumption options.

A Modern History of API Gateways

Now that you have a good understanding of the "what", "where", and "why" of API gateways, it is time to take a glance backwards through history before looking forward to current API gateway technology. As Mark Twain was alleged to have said, "history doesn't repeat itself, but it often rhymes", and anyone who has worked in technology for more than a few year will definitely appreciate the relevance this quote has to the general approach seen in the industry. Architecture style and patterns repeat in various "cycles" throughout the history of software development, as do operational approaches. There is typically progress made between these cycles, but it is also easy to miss the teachings that history has to offer.

This is why it is important that you understand the historical context of API gateways and traffic management at the edge of systems. By looking backwards we can build on firm foundations, understand fundamental requirements, and also try to avoid repeating the same mistakes.

Late 1990s: Hardware Load Balancers

The concept of World Wide Web (www) was proposed by Tim Berners-Lee

in the late 1980s, but this didn't enter the consciousness of the general public until the mid 1990s, where the initial hype culminated in the dotcom boom and bust of the late 90s. This "Web 1.0" period drove the evolution of web browsers (Netscape Navigator was launched late 1994), the web server (Apache Web Server was released in 1995), and hardware load balancers (F5 was founded in 1996). The Web 1.0 experience consisted of users visiting web sites via making HTTP requests using their browser, and the entire HTML document for each target page being returned in the response. Dynamic aspects of a website were implemented via Common Gateway Interface (CGI) in combination with scripts written in languages like Perl or C. This was arguably the first incantation of what we would call "Function as a Service (FaaS)" today.

As an increasing number of users accessed each website this strained the underlying web servers. The requirement to architect systems support increased load and implement fault tolerance was required. Hardware load balancers were deployed at the edge of the data center, with the goal of allowing infrastructure engineers, networking specialists, and sysadmins to spread user requests over a number of web server instances. These early load balancer implementations typically supported basic health checks, and if a web server failed or began responding with increased latency then user requests could be routed elsewhere accordingly.

HARDWARE LOAD BALANCERS: KEY POINTS

Target users

Hardware engineers, networking specialists, sysadmins

Primary goals

High availability, resilience, and scalability

Benefits

Automated fail-over

Challenges

Specialist knowledge required to update or modify configuration

Early 2000s: Software Load Balancers

As the Web overcame the early business stumbles from the dotcom bust, the demand for supporting a range of activities, such as users sharing content, ecommerce and online shopping, and businesses collaborating and integrating systems, continued to increase. In reaction, web-based software architectures began to take a number of forms. Smaller organizations were building on their early work with CGI and were also creating monolithic applications in the emerging web-friendly languages such asJava and .NET. Larger enterprises began to embrace modularization (taking their cues from David Parnas' work in the 1970s), and Service Oriented Architecture (SOA) and the associated "Web Service" specifications (WS-) enjoyed a brief moment in the sun.

The requirements for high availability and scalability of web sites were increasing, and the expense and inflexibility of early hardware load balancers was beginning to become a constraining factor. Enter software load balancers, with HAProxy being launched in 2001 and NGINX in 2002. The target users were still operations teams, but the skills required meant that sysadmins comfortable with configuring software-based web servers were increasingly happy to take responsibility for what used to be a hardware concern.

SOFTWARE LOAD BALANCERS: STILL GOING STRONG

Although they have both evolved from initial launches, NGINX and HAProxy are still widely in use, and they are still very useful for small organizations and simple API gateway use cases (both also offer commercial variants more suitable for enterprise deployment). The rise of cloud (and virtualization) cemented the role of software load balancers, and we recommend learning about the basics of this technology.

This time frame also saw the rise of other edge technologies that still required specialized hardware implementation. Content Delivery Networks (CDNs), primarily driven by the Telco organizations, began to be increasingly adopted in order to offload requests from origin web servers. Web Application Firewalls (WAFs) also began to see increasing adoption, first implemented using specialized hardware, and later via software. The open source ModSecurity project, and the integration of this with the Apache Web Server, drove mass adoption of WAFs.

SOFTWARE LOAD BALANCERS: KEY POINTS

Target users

Sysadmins, networking specialists

Primary goals

High availability, resilience, and scalability

Benefits

Reduced operational cost (compared with hardware), increased flexibility (early implementation of infrastructure as code)

Challenges

Reduced performance (versus dedicated hardware), specialist knowledge required to update or modify configuration

Mid 2000s: Application Delivery Controllers

The mid 2000s continued to see the increasing pervasiveness of the web in everyday life. The emergence of Internet capable phones only accelerated this, with BlackBerry initially leading the field, and everything kicking into a higher gear with the launch of the first iPhone in 2007. The PC-based web browser was still the de facto method of accessing the www, and the mid

2000s saw the emergence of “Web 2.0”, triggered primarily by the widespread adoption in browsers of the XMLHttpRequest API and the corresponding technique named “asynchronous JavaScript and XML (AJAX)”. At the time this technology was revolutionary. The asynchronous nature of the API meant that no longer did an entire HTML page have to be returned, parsed, and the display completed refreshed with each request. By decoupling the data interchange layer from the presentation layer, AJAX allowed web pages to change content dynamically without the need to reload the entire page.

All of these changes placed new demands on web servers and load balancers, for yet again handling more load, but also supporting more secure (SSL) traffic, increasingly large (media rich) data payloads, and different priority requests. This led to the emergence of a new technology named Application Delivery Controllers (ADCs). As these were initially implemented using specialized hardware this led to the existing networking players like F5 Networks, Citrix, Cisco dominating the market. ADCs provided support for compression, caching, connection multiplexing, traffic shaping, and SSL offload, combined with load balancing. The target users were once again infrastructure engineers, networking specialists, and sysadmins.

THE BENEFITS, AND COSTS, OF SPECIALIZATION

By the mid 2000s nearly all of the components of modern edge stacks were widely adopted across the industry. The benefits of the separation of concerns were becoming clear (e.g. each edge technology had a clear and focused purpose), but this was increasingly the siloing between teams. If a developer wanted to expose a new application within a large organization this typically meant many separate meetings with the CDN vendors, the load balancing teams, the InfoSec and WAF teams, and the web/application server team. Movements like DevOps emerged, partly driven by a motivation to remove the friction imposed by these silos. If you still have a large number of layers in your edge stack and are migrating to the cloud or a new platform, now is the time to potentially think about the tradeoffs with multiple layers and specialist teams.

APPLICATION DELIVERY CONTROLLERS: KEY POINTS

Target users

Hardware engineers, networking specialists, sysadmins

Primary goals

Application acceleration: compression, caching, SSL/TLS offload, traffic shaping; and high availability

Benefits

Clear separation of concerns, and reduced burden on web/application servers

Challenges

Introduced another layer (and potentially another dedicated team) into the edge stack

Early 2010s: First Generation API Gateways

The late 2000s and early 2010s saw the emergence of the API economy and associated technologies. Organizations like Twilio were disrupting telecommunications, with their founder, Jeff Lawson, pitching that “We have taken the entire messy and complex world of telephony and reduced it to five API calls.” The Google Maps API was enabling innovative user experiences, and Stripe was enabling organizations to easily charge for access to services. Founded in late 2007, Mashape was one of the early pioneers in attempting to create an API marketplace for developers. Although this exact vision didn’t pan out (arguably it was ahead of it’s time, looking now to the rise of “no code"/"low code” solutions), a byproduct of the Mashape business model was the creation of the Kong API Gateway, built upon OpenResty and the open source NGINX implementation. Other implementations included WSO2 with Cloud Services Gateway, Sonoa Systems with Apigee, and Red Ht with 3Scale Connect.

These were the first edge technologies that were targeted at developers in addition to platform teams and sysadmins. A big focus was on managing the

software development lifecycle (SDLC) of an API and providing system integration functionality, such as endpoints and protocol connectors, and translation modules. Due to the range of functionality offered, the vast majority of first generation API gateways were implemented in software. Developer portals sprang up in many products, which allowed engineers to document and share their APIs in a structured way. These portals also provided access controls and user/developer account management, and publishing controls and analytics. The theory was that this would enable the easy monetization of APIs, and the management of "APIs as a product".

During this evolution of developer interaction at the edge there was increasing focus on the application layer (layer 7) of the OSI Networking model. The previous generations of edge technologies often focused on IP addresses and ports, which primarily operate at the transport layer (layer 4) of the OSI model. Allowing developers to make routing decisions in an API gateway based on HTTP metadata such as path-based routing or header-based routing provided the opportunity for richer functionality.

There was also an emerging trend towards creating smaller service-based architectures, and some organizations were extracting single-purpose standalone applications from their existing monolithic code bases. Some of these monoliths acted as an API gateway, or provided API gateway-like functionality, such as routing and authentication. With the first generation of API gateways it was often the case that both functional and cross-functional concerns, such as routing, security, and resilience, were performed both at the edge and also within the applications and services.

WATCH FOR RELEASE COUPLING WHEN MIGRATING AWAY FROM A MONOLITH

Extracting standalone services from a monolithic application and having the monolith acting as a simple gateway can be a useful migration pattern towards adopting a microservices architecture. However, beware of the costs related to coupling between the application gateway and services that are introduced with this pattern. For example, although the newly extracted services can be deployed on demand, their release is often dependent on the release cadence of the monolith, as all traffic must be routed through this component. And if the monolith has an outage, then so do all of your services operating behind it.

FIRST GENERATION API GATEWAYS: KEY POINTS

Target users

Platform engineers, sysadmins, and developers

Primary goals

API (software development) lifecycle management, monetization of APIs, dynamic (layer 7) routing

Benefits

Developers gained access to the edge, and enabled innovation in integrating systems/APIs

Challenges

Introduced another layer and another persona (developer) into the management of the edge stack

2015 Onwards: Second Generation API Gateways

The mid 2010s saw the rise of the next generation of modular and service-oriented architectures, with the concept of “microservices” firmly entering the zeitgeist by 2015. This was largely thanks to “unicorn” organizations like Netflix, AWS, and Spotify sharing their experiences of working with these architectural patterns. In addition to back end systems being decomposed into more numerous and smaller services, developers were also adopting container technologies based on Linux LXC. Docker was released in March of 2013, and Kubernetes followed hot on it’s heels with a v1.0 release in July of 2015. This shift in architectural style and changing runtimes drove new requirements at the edge. Netflix released their bespoke JVM-based API gateway, Zuul, in mid 2013. Zuul supported service discovery for dynamic back end services and also allowed Groovy scripts to be injected at runtime in

order to dynamically modify behaviour. This gateway also consolidated many cross cutting concerns into a single edge component, such as authentication, testing (canary releases), rate limiting and load shedding, and observability. Zuul was a revolutionary API gateway in the microservices space, and it has since involved into a second version, and Spring Cloud Gateway has been built on top of this.

WATCH FOR COUPLING OF BUSINESS LOGIC IN THE GATEWAY

When using Zuul it was all too easy to accidentally highly couple the gateway with a service by including related business logic in both a Groovy script (or scripts) and the service. This meant that the deployment of a new version of a service often required modifications to the code running in the gateway. The pinnacle of this bad coupling can be seen when a microservice team decides to reuse an existing deployed Groovy script for their service, and then at an some arbitrary time in the future the script is modified by the original owning team in a incompatible way. This can quickly lead to confusion as to why things are broken, and also to whack-a-mole type fixes. This danger wasn't unique to Zuul, and nearly all proxies and many gateways allowed the injection of plugins or dynamic behaviour, but this was often only accessible to operations teams or written in obscure (or unpopular) languages. The use of Groovy in Zuul made this very easy for application developers to implement.

With the increasing adoption of Kubernetes and the open source release of the Envoy Proxy in 2016 by the Lyft team, many API gateways were created around this technology, including Ambassador, Contour, and Gloo. This drove further innovation across the API gateway space, with Kong mirroring functionality offered by the next generation of gateways, and other gateways being launched, such as Traefik, Tyk, and others. Increasingly, many of the Kubernetes Ingress Controllers called themselves "API gateways", regardless of the functionality they offered, and this led to some confusion for end users in this space.

API GATEWAYS, EDGE PROXIES, AND INGRESS CONTROLLERS

As Christian Posta noted in his blog post API Gateways Are Going Through an Identity Crisis, there is some confusion around what an API gateway is in relation to edge technologies emerging from

the cloud space. Generally speaking, in this context an API gateway enables some form of management of APIs, ranging from simple adaptor-style functionality operating at the application layer (OSI layer 7) that provides fundamental cross-cutting concerns, all the way to full lifecycle API management. Edge proxies are more general purpose traffic proxies or reverse proxies that operate at the network and transport layers (OSI layers 3 and 4 respectively) and provide basic cross-cutting concerns, and tend not to offer API-specific functionality. Ingress controllers are a Kubernetes-specific technology that controls what traffic enters a cluster, and how this traffic is handled.

The target users for the second generation of API gateways was largely the same as the cohort for the first generation, but with a clearer separation of concerns and a stronger focus on developer self-service. The move from first to second generation of API gateways saw increased consolidation of both functional and cross-functional requirements being implemented in the gateway. Although it became widely accepted that microservices should be built around the idea espoused by James Lewis and Martin Fowler of "smart endpoints and dumb pipes", the uptake of polyglot language stacks mean that "microservice gateways" emerged (more detail in the next section) that offered cross-cutting functionality in a language-agnostic way.

SECOND GENERATION API GATEWAYS: KEY POINTS

Target users

Platform engineers, sysadmins, and developers

Primary goals

API lifecycle management, dynamic (layer 7) routing, developer-focused self-service control

Benefits

Focus on self-service and consolidation of cross-functional concerns enabled fullcycle development teams to iterate rapidly

Challenges

Lack of clarity about what functionality an API gateway should offer in the cloud native and Kubernetes space

Coda (2017 Onwards): Service Mesh and/or API Gateway?

In early 2017 there was increasing buzz about the use of "service meshes", a new communication framework for microservice-based systems. William Morgan, CEO of Buoyant and ex-Twitter engineer, is largely credited with coining the term. The early service meshes exclusively targeted the east-west, service-to-service communication requirements. As Phil Calçado writes in Pattern: Service Mesh, service meshes evolved from early in-process microservice communication and resilience libraries, such as Netflix's Ribbon, Eureka, and Hystrix. Buoyant's Linkerd was the first service mesh to be hosted by the CNCF, but Google's Istio captured the attention of many engineers (largely thanks to Google's impressive marketing machine).

Although there is a lot of functional overlap between service meshes and API gateways, and much of underlying proxy (data plane) technology is identical, the use cases and control plane implementations are at the moment quite different. Several service meshes offer ingress and gateway-like functionality, for example, Istio Gateway and Consul Ingress Gateways. However, this functionality is typically not as feature-rich or as mature as that currently offered by existing API gateways. This is why the majority of modern API gateways integrate effectively with at least one service mesh.

START AT THE EDGE AND WORK INWARDS

Every production system that exposes an API will require an API gateway, but not necessarily a service mesh. Many existing systems, and also simple greenfield applications, are implemented as a shallow stack of services, and therefore the added operational complexity of deploying and running a service mesh does not provide a high enough level of benefit. For this reason it is typically the case that we recommend engineers "start at the edge and work inwards", i.e. select and deploy an API gateway, and only when the number of services (and their interactions) grows consider selecting a service mesh.

Current API Gateway Taxonomy

As can be the case with terminology in the software development industry, there often isn't an exact agreement on what defines or classifies an API gateway. There is broad agreement in regards to the functionality this technology should provide, but different segments of the industry have different requirements, and hence different views, for an API gateway. This has led to several sub-types of API gateway emerging and being discussed. In this section of the chapter you will explore the emerging taxonomy of API gateways, and learn about their respective use cases, strengths, and weaknesses.

Traditional Enterprise Gateways

The traditional enterprise API gateway is typically aimed at the use case of exposing and managing business-focused APIs. This gateway is also often integrated with a full API lifecycle management solution, as this is an essential requirement when releasing, operating, and monetizing APIs at scale. The majority of gateways in this space may offer an open source edition, but there is typically a strong usage bias towards the open core/commercial version of the gateway.

These gateways typically require the deployment and operation of dependent services, such as data stores. These external dependencies have to be run with high availability to maintain the correct operation of the gateway, and this must be factored into running costs and DR/BC plans.

Micro/Microservices Gateways

The primary use case of microservices API gateway, or micro API gateway, is to route ingress traffic to backend APIs and services. There is typically not much provided for the management of an API's lifecycle. These types of gateway are often available fully-featured as open source or are offered as a lightweight version of a traditional enterprise gateway.

The tend to be deployed and operated as standalone components, and often make use of the underlying platform (e.g., Kubernetes) for the management of any state. As microservices gateways are typically built using modern proxy technology like Envoy, the integration capabilities with service meshes (especially those built using the same proxy technology) is typically good.

Service Mesh Gateways

The ingress or API gateway included with a service mesh is typically designed to provide only the core functionality of routing external traffic into the mesh. For this reason they often lack some of the typical enterprise features, such as comprehensive integration with authentication and identity provider solutions, and also integration with other security features, such as a WAF.

The service gateway typically manages state using the capabilities of the mesh itself or underlying deployment fabric (e.g. Kubernetes). This type of gateway is also implicitly coupled with the associated service mesh (and operational requirements), and so if you are not yet planning to deploy a service mesh, then this is most likely not a good first choice of API gateway.

Comparing API Gateway Types

The table below highlights the difference between the two most widely deployed API gateway types across six important criteria.

Table 4-1. Comparison of Enterprise, Microservices, and Service Mesh API gateway

Use case	Traditional Enterprise API gateway	Microservices API gateway	Service Mesh Gateway
Primary Purpose	Expose, compose, and manage internal business APIs	Expose and observe internal business services	Expose internal services within the mesh
Publishing Functionality	API management team or service team registers/updates gateway via	API Service team registers/updates gateway via declarative	Service team registers/updates mesh and gateway

	admin	code as part of the deployment process	via declarative code as part of the deployment process
Monitoring	Admin and operations focused e.g. meter API calls per consumer, report errors (e.g. internal 5XX).	Developer focused e.g. latency, traffic, errors, saturation	Platform focused e.g. utilization, saturation, errors
Handling and Debugging Issues	L7 error-handling (e.g. custom error page). Run gateway/API with additional logging. Troubleshoot issue in staging environment	L7 error-handling (e.g. custom error page, failover, or payload). Configure more detailed monitoring. Enable traffic shadowing and/or canarying to verify issue	L7 error-handling (e.g. custom error page or payload). Configure more detailed monitoring. Debug via specialized tooling e.g. Squash or Telepresence
Testing	Operate multiple environments for QA, Staging, and Production. Automated integration testing, and gated API deployment. Use client-driven API versioning for compatibility and stability (e.g. semver)	Facilitate canary routing for dynamic testing (taking care with data mutation side effects in live environments). Use developer-driven service versioning for upgrade management	Facilitate canary routing for dynamic testing
Local Development	Deploy gateway locally (via installation script, Vagrant or Docker), and attempt to mitigate infrastructure differences with production. Use language-specific gateway mocking and stubbing frameworks	Deploy gateway locally via service orchestration platform (e.g. container, or Kubernetes)	Deploy service mesh locally via service orchestration platform (e.g. Kubernetes)

Deploying API Gateways: Understanding and Managing Failure

Regardless of the deployment pattern and number of gateways involved

within a system, an API gateway is typically on the critical path of many, if not all, user requests entering into your system. An outage of a gateway deployed at the edge typically results in the unavailability of the entire system. And an outage of a gateway deployed further upstream typically results in the unavailability of some core subsystem. For this reason the topics of understanding and managing failure of an API gateway are vitally important to learn.

API Gateway as a Single Point of Failure

The first essential step in identifying single points of failure in a system is to ensure that you have an accurate understanding of the current system. This is typically accomplished by investigation and the continual update of associated documentation and diagrams. Assembling a diagram that traces a user-initiated request for each core journey or use case, all the way from ingress to data store or external system and back to egress, that shows all the core components involved can be extremely eye opening. This is especially the case in large organizations, where ownership can be unclear and core technologies can accidentally become abandoned.

In a typical web-based system, the first obvious single point of failure is typically DNS. Although this is often externally managed, there is no escaping the fact that if this fails, then your site will be unavailable. The next single points of failure will typically then be the global and regional layer 4 load balancers, and depending on the deployment location and configuration, the security edge components, such as the firewall or WAF.

CHALLENGE ASSUMPTIONS WITH SECURITY SINGLE POINTS OF FAILURE

Depending on the product, deployment, and configuration, some security components may “fail open”, i.e. if the component fails then traffic will simply be passed through to upstream components or the backend. For some scenarios where availability is the most important goal this is desired, but for others (e.g. financial or government systems), this is most likely not. Be sure to challenge assumptions in your current security configuration.

After these core edge components, the next layer is typically the API gateway. The more functionality you are relying on within the gateway, the bigger the risk involved and bigger the impact of an outage. As an API gateway if often involved in a software release the configuration is also continually being updated. It is critical to be able to detect and resolve issues, and mitigate any risks.

Detecting Problems

The first stage in detecting issues is ensuring that you are collecting and have access to appropriate signals from your monitoring system, i.e. data from metrics, logs, and traces. If you are new to this concept, then we recommend learning more about Brendan Gregg's utilization, saturization, and errors (USE) method, Tom Wilkie's rate, errors, and duration (RED) method, and Google's four golden signals of monitoring. In particular, the Google Site Reliability Engineering (SRE) book, is highly recommended. A key takeaway from the book is that for any critical system you should ensure a team owns it and is accountable for any issues. The Google team talk about the need to define service level objectives (SLOs) SLOs, which can be codified into service level agreements (SLAs) for both internal and external customers.

SYNTHETIC MONITORING: ONE OF THE MOST VALUABLE DETECTION TECHNIQUES

One of the more effective ways of detecting user-facing issues, particularly in a rapidly evolving system, is to use synthetic monitoring (also called semantic monitoring and synthetic transactions). This technique consists of continually running a subset of an application's business-focused automated tests against the production system in the same manner a user would make them. The user-facing results are then pushed into a monitoring service, which triggers alerts in case of failures.

Resolving Incidents and Issues

First and foremost, each API gateway operating within your system needs an owner that is accountable if anything goes wrong with the component. As an API gateway is on the critical path of requests, some portion of this owning

team should be on-call as appropriate (this may be 24/7/365). The on-call team will then face the tricky task of fixing the issue as rapidly as possible, but also learning enough (or locating and quarantining systems and configuration) to learn what went wrong. After each incident you should strive to conduct blameless a post mortem, and document and share all of your learning. Not only can this information be used to trigger remediate action to prevent this issue reoccurring, but this knowledge can be very useful for engineers learning the system and for external teams dealing with similar technologies or challenges. If you are new to this space then the Learning from Incidents website is a fantastic jumping off point.

Mitigating Risks

Any component that is on the critical path for handling user requests should be made as highly available as is practical in relation to cost and operational complexity. Software architects and technical leaders deal with tradeoffs; this type is one of the most challenging. In the world of API gateways, high availability typically starts with running multiple instances. With on-premise/co-lo instances this translates into operating multiple (redundant) hardware appliances, ideally spread across separate locations. In the cloud, this translates into designing and running the API gateway instance in multiple availability zones/data centers and regions. If a (global) load balancer is deployed in front of the API gateway instances, then this must be configured appropriately with health checks and failover processes that must be test regularly. This is especially important if the API gateways instances run in active/passive or leader/node modes of operation.

LOAD BALANCING CHALLENGES

You must ensure that your load balancer to API gateway failover process meets all of your requirements in relation to continuity of service. Common problems experienced during failover events include:

- User client state management issues, such as backend state not being migrated correctly, which causes the failure of sticky sessions
- Poor performance, as client are not redirected based on geographical considerations e.g.

European users being redirected to the US west coast when an east coast data center is available

- Unintentional cascading failure, such as a faulty leader election component that results in deadlock, which causes all backend systems to become unavailable

Care should be taken with any high availability strategy to ensure that dependent components are also included. For example, many enterprise API gateways rely on a data store to store configuration and state in order function correctly. This must also be run in a HA configuration, and it must be available to all active and passive components. It is becoming increasingly common to split the deployment and operation of the control plane and data plan, but you will need to understand what happens if the control plane has an outage over a considerable period of time.

Often the biggest period of change with an API gateway is during configuration update, for example, a release of a new API or service. Progressive delivery techniques should be used, such as rolling upgrades, localized rollouts, blue/green deployments, and canary releasing. These allow fast feedback and incremental testing in a production environment, and also limit the blast radius of any outage.

Finally, you can put in place reactive fallback measures to handle failure. Examples of this include building functionality to serve a degraded version of the website from static servers, caches, or via a CDN.

API Gateway Antipatterns: Mixing Traffic Patterns

Although it can be tempting to want to find a single solution to manage all traffic flowing into and within a system (“one gateway to rule them all”), history has not been kind to this approach. There are many examples of technology that became overly coupled with the underlying application and added increasing friction to the software delivery process being a key culprit. There are several antipatterns you should always try and avoid.

API Gateway Loopback: “Service Mesh Lite”

As with all antipatterns, the implementation of this pattern often begins with good intentions. When an organization has only a few services this typically doesn’t warrant the installation of a service mesh. However, a subset of service mesh functionality is often required, particularly service discovery. An easy implementation is to route all traffic through the edge or API gateway, which maintains the official directory of all service locations. At this stage the pattern looks somewhat like a “hub and spoke” networking diagram. The challenges present themselves in two forms: Firstly, when all of the service-to-service traffic is leaving the network before reentering via the gateway this can present performance, security, and cost concerns (cloud vendors often charge for egress). Secondly, this pattern doesn’t scale beyond a handful of services, as the gateway becomes overloaded and a bottleneck, and it becomes a true single point of failure.

API Gateway as an ESB: Egregious Spaghetti Boxes?

The vast majority of API gateways support the extension of their out-of-the-box functionality via the creation of plugins or modules. NGINX supported Lua modules, which OpenResty and Kong capitalised on. Envoy Proxy originally supported extensions in C, and now WebAssembly filters. And we’ve already discussed how the original implementation of Netflix’s Zuul API gateway “2015 Onwards: Second Generation API Gateways”, supported extension via Groovy scripts. Many of the use cases realised by these plugins are extremely useful, such as authn/z, filtering, and logging. However, it can be tempting to put business logic into these plugins, which is a way to highly-couple your gateway with your service or application. This leads to a potentially fragile system, where a change in a single plugin ripples throughout the organization, or additional friction during release where the target service and plugin have to be deployed in lockstep.

Turtles (API Gateways) All the Way Down

If one API gateway is good, more must be better, right? It is common to find

multiple API gateways deployed within the context of large organization, often in a hierarchical fashion, or in an attempt to segment networks or departments. The intentions are typically good: either for providing encapsulation for internal lines of business, or for a separation of concerns with each gateway (e.g. “this is the transport security gateway, this is the auth gateway, this is the logging gateway…”). The antipatterns rears its head when the cost of change is too high, e.g. you have to coordinate with a large number of gateway teams to release a simple service upgrade, there are understandability issues (“who owns the tracing functionality?”), or performance is impacted as every network hop naturally incurs a cost.

Selecting an API Gateway

Now that you learned about the functionality provided by an API gateway, the history of the technology, and how an API gateway fits into to the overall system architecture, next is the $64,000 question: how do you select an API gateway to included in your stack?

Identifying Requirements

One of the first steps with any new software delivery or infrastructure project is identifying the related requirements. This may appear obvious, but it is all too easy to get distracted by shiny technology, magical marketing, or good sales documentation!

You can look back to “Why Use an API Gateway?”, to explore in more detail the high-level requirements you should be considering during the selection process:

- Reducing coupling between front ends and back ends
- Simplifying client consumption of APIs by aggregating and/or translating back end services
- Protecting APIs from overuse and abuse, via threat detection and mitigation

- Understanding how APIs are being consumed and how the underlying systems are performing
- Managing APIs as products i.e. API Lifecycle Management requirements
- Monetizing APIs, including the needs for account management, billing, and payment

It is important to ask question both focused on current pain points and also your future roadmap. For example, do you need to simply expose REST-based backend APIs with a thin layer of load balancing and security, or do you need comprehensive API aggregation and translation functionality combined with enterprise-grade security?

Exploring Constraints: Team, Technologies, and Roadmap

In addition to identifying requirements it is also essential that you identify your organization's constraints. Broadly speaking, we have found there are three key areas that are well worth exploring when choosing your API gateway: team structure and dynamics, existing technology investments, and future roadmap.

The much discussed Conway's law, stating that organizations design systems mirror their own communication structure, has almost become a cliche. But there is truth to this. As explored by Matthew Skelton and Manuel Pais in Team Topologies your organizational structure will constrain your solution space. For example, if your organization has a separate InfoSec team, there is a good chance that you will end up with a separate security gateway.

Another well discussed concept is the notion of the sunk cost fallacy, but in the real world we see this ignored time and time again. Existing technology deployed often constrains future decisions. You will need to investigate the history associated with many of the big ticket technology decision making within an organization, such as API gateway choice, in order to avoid messy politics. For example, if a platform team has already invested millions of

dollars into creating a bespoke API gateway, there is a good chance they will be reluctant to adopt another gateway, even if this is a better fit for the current problem space.

Finally, you will need to identify and explore your organization's roadmap. Often this will constrain the solution space. For example, some organizations are banning the use of certain cloud vendors or technology vendors, for reasons related to competition or ethical concerns. Other times, you will find that the roadmap indicates the leadership is "all-in" on a specific technology, say, Java, and so this may restrict the deployment of a gateway that doesn't run on a JVM.

Build Versus Buy

A common discussion when selecting an API gateway is the "build versus buy" dilemma. This is not unique to this component of a software system, but the functionality offered via an API gateway does lead to some engineers gravitating to this that they could build this "better" than existing vendors, or that their organization is somehow "special", and would benefit from a custom implementation. In general, we believe that the API gateway component is sufficiently well-established that it typically best to adopt an open source implementation or commercial solution rather than build your own. Presenting the case for build versus buy with software delivery technology could take an entire book, and so in this section we only want to highlight some common challenges:

- Underestimating the Total Cost of Ownership (TCO): Many engineers discount the cost of engineering a solution, the continued maintenance costs, and the ongoing operational costs.
- Not thinking about opportunity cost: Unless you are a cloud or platform vendor, it is highly unlikely that a custom API gateway will provide you with a competitive advantage. You can delivery more value to your customers by building some functionality closer to your overall value proposition

- Not being aware of current technical solutions of products. Both the open source and commercial platform component space moves fast, and it can be challenging to keep up to date. This, however, is a core part of the role of being a technical leader.

Radars, Quadrants, and Trend Reports

Although you should always perform your own experiments and proof of concept work, we recommend keeping up to date with technology trends via reputable IT press. This type of content can be especially useful when you are struggling with a problem or have identified a solution and are in need of a specific piece of technology that many vendors offer.

We recommend the following sources of information for learning more about the state of the art of technology within the API gateway space:

- ThoughtWorks Technology Radar
- Gartner Magic Quadrant for Full Life Cycle API Management
- Cloud Native Computing Foundation (CNCF) Tech Radar
- InfoQ Trends Reports

Several organizations and individuals also publish periodic API gateway comparison spreadsheets, and these can be useful for simple “paper evaluations” in order to shortlist products to experiment with. It should go without saying that you will need to check for bias across these comparisons (vendors frequently sponsor such work), and also ensure the publication date is relatively recent. Enterprise API gateways solutions do not change much from month to month, but this is not true in the cloud native and Kubernetes space.

API Gateway: A Type 1 Decision

Jeff Bezos, the CEO of Amazon, is famous for many things, and one of them is his discussion of Type 1 decisions and Type 2 decisions. Type 1 decisions are not easily reversible, and you have to be very careful making them. Type

2 decisions are easy to change: “like walking through a door — if you don’t like the decision, you can always go back.” Usually this concept is presented in relation to confusing the two, and using Type 1 processes on Type 2 decisions: “The end result of this is slowness, unthoughtful risk aversion, failure to experiment sufficiently, and consequently diminished invention. We’ll have to figure out how to fight that tendency.” However, in the majority of cases—especially within a large enterprise context—choosing an API gateway is very much a Type 1 decision. Ensure your organization acts accordingly!

Checklist: Selecting an API Gateway

Table 4-2. Selecting an API Gateway Checklist

Decision	How should we approach selecting an API gateway for our organization?
Discussion Points	Have we identified and prioritized all of our requirements associated with selecting an API gateway?
	Have we identified current technology solutions that have been deployed in this space within the organization?
	Do we know all of our team and organizational constraints?
	Have we explored our future roadmap in relation to this decision?
	Have we honestly calculated the “build versus buy” costs?
	Have we explored the current technology landscape and are we aware of all of the available solutions?
	Have we consulted and informed all involved stakeholders in our analysis and decision making?
Recommendations	Focus particularly on your requirement to reduce API/system coupling, simplify consumption, protect APIs from overuse and abuse, understand how APIs are being consumed, manage APIs as products, and monetize APIs
	Key questions to ask include: is there are existing API gateway in use? Has a collection of technologies been assembled to provide similar functionality (e.g. hardware load balancer combined with a monolithic app that performs authentication and application-level routing)? How many components currently make up your edge stack (e.g. WAF, LB, edge cache, etc.)
	Focus on technology skill levels within your team, availability of people to work

on a API gateway project, and available resources and budget etc

It is important to identify all planning changes, new features, and current goals that could impact traffic management and the other functionality that an API gateway provides

Calculate the total cost of ownership (TCO) of all of the current API gateway-like implementations and potential future solutions.

Consult with well known analysts, trend reports, and product reviews in order to understand all of the current solutions available.

Selecting and deploying an API gateway will impact many teams and individuals. Be sure to consult with the developers, QA, the architecture review board, the platform team, InfoSec etc.

Summary

In this chapter you have learned what an API gateway is, and also explored the historical context that led the to evolution of the features currently provided by this essential component in any web-based software stack. You have explored the current taxonomy of API gateways and their deployment models, which has equipped you to think about how to manage potential single points of failure in an architecture where all user traffic is routed through an edge gateway. Building on the concepts of managing traffic at the (ingress) edge of systems, you have also learned about service-to-service communication and how to avoid antipatterns such as deploying an API gateway as a less-functional enterprise service bus (ESB).

The combination of all of this knowledge has equipped you with the key thinking points, constraints, and requirements necessary in order to make an effective choice when selecting an API gateway for your current use cases. As with most decisions a software architect or technical leader has to make, there is no distinct correct answer, but there often can be quite a few bad solutions to avoid.

Now that you have explored the functionality that API gateways provide during the runtime operation of systems, the next important concept to learn about is how to test APIs and API gateways in production.

About the Authors

James GoughJ has worked extensively with financial systems and is the architectural lead for Client APIs at Morgan Stanley. He has a very pragmatic approach to building software and has been responsible for building API Gateways and applications to support a large enterprise API transformation.

Daniel Bryant works as a product architect at Datawire. His technical expertise focuses on DevOps tooling, cloud/container platforms, and microservice implementations. Daniel is a Java Champion, and contributes to several open source projects. He also writes for InfoQ, O'Reilly, and TheNewStack, and regularly presents at international conferences such as OSCON, QCon, and JavaOne. In his copious amounts of free time he enjoys running, reading, and traveling.

Matthew Auburn has worked for Morgan Stanley on a variety of financial systems. Before working at Morgan Stanley he has built a variety of mobile and web applications. Matthew's Masters degree primarily focused on security and this has fed into working in the security space for building APIs.

a. Introduction to REST
    i. The Richardson Maturity Model
b. REST API Standards and Structure
    i. Collections and Pagination
    ii. Filtering Collections
    iii. Updating Data
    iv. Error Handling
    v. Checklist: Choosing an API Standard
c. Specifying REST APIs
    i. OpenAPI Specification Structure
    ii. Visualizing OpenAPI Specifications
d. Practical Application of OpenAPI Specifications
    i. Code Generation
    ii. OpenAPI Validation
    iii. Examples and Mocking
    iv. Detecting Changes
e. API Versioning
    i. Semantic Versioning
    ii. API Lifecycle
    iii. OpenAPI Specification and Versioning
f. Alternative API Formats
    i. Remote Procedure Call (RPC)