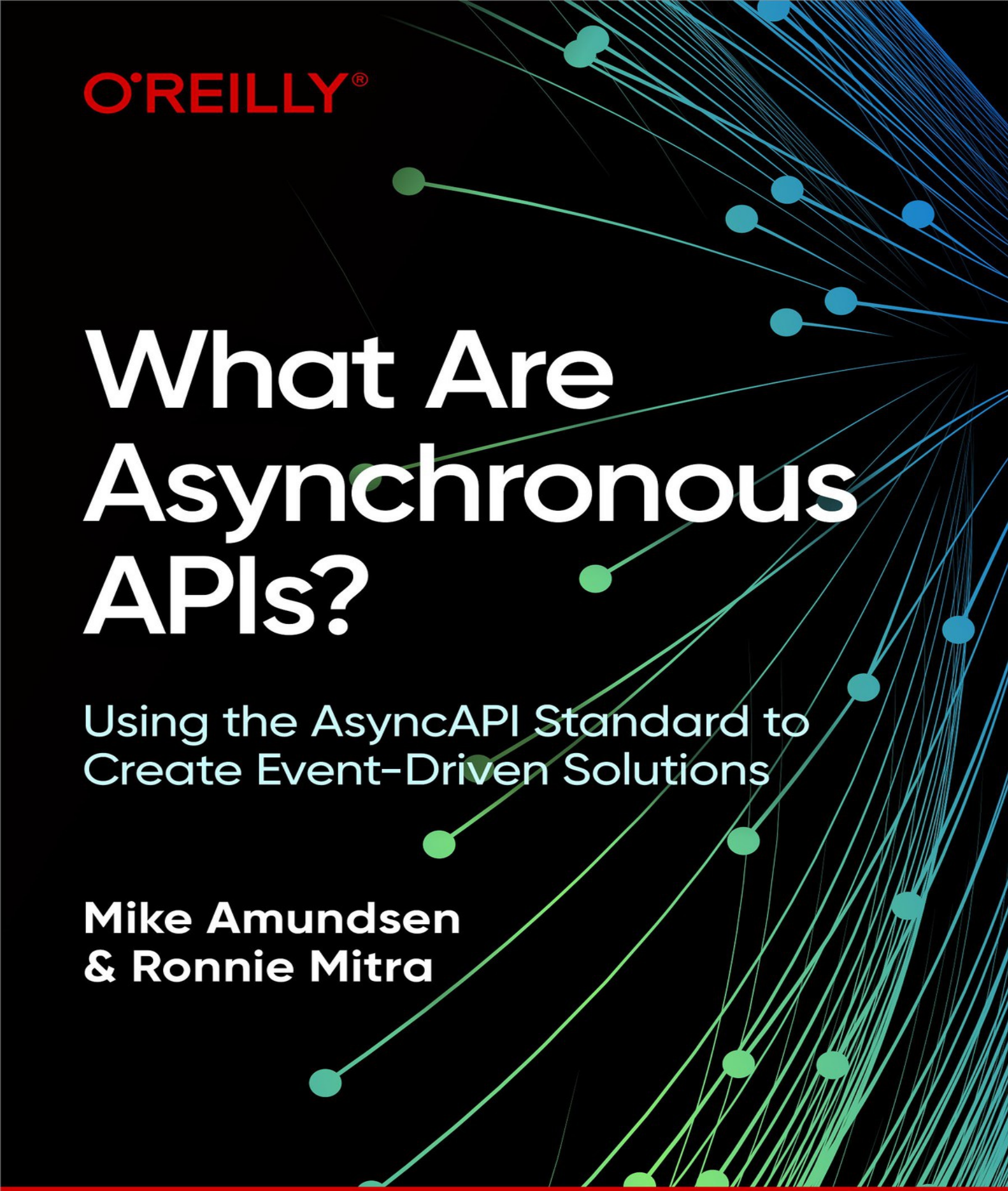

What Are Asynchronous APIs?

Using the AsyncAPI Standard to Create Event-Driven Solutions

Mike Amundsen
& Ronnie Mitra

REPORT

What Are Asynchronous APIs?

Using the AsyncAPI Standard to Create Event-Driven Solutions

Mike Amundsen and Ronnie Mitra

Beijing • Boston • Farnham • Sebastopol • Tokyo

What Are Asynchronous APIs?

by Mike Amundsen and Ronnie Mitra

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: Louise Corrigan
- Development Editor: Rita Fernando
- Production Editor: Clare Laylock
- Copyeditor: Piper Editorial Consulting, LLC
- Proofreader: Amnet Systems, LLC
- Interior Designer: David Futato
- Cover Designer: Randy Comer
- Illustrator: Kate Dullea

- July 2023: First Edition

Revision History for the First Edition

- 2023-07-26: First Release

978-1-098-14746-4

[LSI]

Preface

Welcome to *What Are Asynchronous APIs?*

As the use of APIs continues to grow, one particular style of APIs—those that are asynchronous—are showing a noticeable increase. According to a survey conducted by the research firm Coleman-Parkes, "Event-driven architecture is in widespread use for 72% of global businesses" today. This translates to approximately 11 million developers based on a total of 19.1 million API developers sourced from Nordic API's 2020 survey "APIs Have Taken Over Software Development".

This report gives you all the tools you need to successfully introduce the powerful asynchronous API pattern to your organization. More and more applications today offer "instant feedback," push notifications, and streaming data. A key component of these systems is support for asynchronous APIs—API calls that don't rely on the RESTful client-server model but, instead, support servers that push notifications and data directly to the client as soon as it is available. This push-style interaction is at the heart of asynchronous APIs.

Who Should Read This Report

This report is intended to address the needs of a wide range of readers who are looking for a clear, concise introduction to the world of asynchronous APIs. That includes architects, programmers, system designers, operations staff, and decision makers. Readers should have a basic understanding of API technologies and tooling but do not need to be proficient in order to understand the material.

This report will be handy for anyone who wants to understand the asynchronous interaction pattern for APIs, including how it is different from traditional patterns and the types of use cases it unlocks. We'll help you learn:

- How to design sustainable, safe, high-performance asynchronous APIs

- The role of the AsyncAPI specification and how to use it
- How to understand the trade-offs of an event-driven system versus others

Whether you are an application designer, frontline developer, software architect, or working in a supportive tech leadership role, this report can help you understand the basics of asynchronous APIs and how you can use them to meet your organization's goals.

What's Covered in This Report

The topic of asynchronous APIs is pretty wide, encompassing design, build, deployment, and runtime management challenges. To keep this report short, we'll be focusing on the basics here and, where appropriate, we'll recommend other books, blogs, and videos where you can get more in-depth coverage of the topics.

In this report, you will learn:

- What asynchronous APIs are and why they are important
- What the AsyncAPI definition format is and how it is used to design asynchronous services
- What common patterns in asynchronous solutions are
- What the trade-offs are when introducing asynchronous APIs into your infrastructure

What's Not Covered in This Report

Even though we'll be showing you screenshots and code snippets from a sample asynchronous API project throughout this report, we won't have the time or space to get into the technical details of coding and deploying asynchronous APIs or how to monitor them once they are up and running in production. We'll point you to other resources for that and provide portions of our running sample via a public GitHub repository for your use.

Many of our readers may be working in established enterprises, and these types of companies will usually have their own set of guidelines for API design and development, their own tool

sets, approved tooling, and their own CI/CD pipelines for moving async API projects from idea to running code. As you read this report, keep in mind that your own organization may have a different set of guidelines and processes than the ones we hint at within this short report.

Above all, we encourage you to use this report as an introduction and exploration of the world of asynchronous APIs. Hopefully, it will spark your interest, point you in the right direction, and give you helpful ideas on how you can join the many companies around the world who are taking advantage of this powerful style of APIs.

Acknowledgments

We want to thank everyone who read early copies of this report and provided valuable feedback. We especially want to thank Fran Mendez and Lukasz Dynowski for their help in improving the text. Thanks also to two great O’Reilly editors, Rita Fernando and Louise Corrigan, for all their assistance in bringing this report to life.

Chapter 1. Understanding Asynchronous APIs

To start, this chapter will cover the basics of Application Programming Interfaces (APIs) in general and specifically asynchronous APIs. We'll touch on the difference between synchronous protocols like Hypertext Transfer Protocol (HTTP) and asynchronous protocols like MQ Telemetry Transport (MQTT), and we'll introduce the idea of them representing either RESTful (HTTP) or EVENTful (MQTT) types of implementations.

We'll then focus on EVENTful message design and cover the three main ways messages are designed: notifications, objects, and streams. We'll also explore the way messages can represent actions that happened in the past (events) and actions that are meant to happen sometime in the future (commands).

With this as a foundation, we'll have all that we need to move on to Chapter 2, where we explore the AsyncAPI format and how you can use that to document your EVENTful system design.

But first, let's cover the basics of async APIs.

Reviewing the Basics of Asynchronous APIs

Asynchronous APIs have been around for a long time. In fact, one of the common protocols for implementing async APIs, MQTT, was first created in 1999. That is only a few years after the most common synchronous API protocol, HTTP, was first released (1991). As we'll see shortly, the main difference between RESTful and EVENTful systems is the way data (e.g., messages) are passed between machines. We'll see examples of both of these in the following sections.

EVENTFUL?

Throughout this report, you'll see us use the word *EVENTful* to refer to asynchronous APIs. This term is meant to represent the range of protocols and styles that fall under the names *asynchronous*, *message-based*, *event-driven*, and so forth. To help bolster the use of this term,

we offer the following “backronym” to describe the EVENT style in general: Efficient, Versatile, Nonblocking, and Timely.[1]

But first, it can be helpful to review the notion of APIs in general.

Exploring APIs’ Roots

American software engineer and technology author Joshua Bloch has proposed that the credit for the concept of programming interfaces goes to the authors of the 1951 book *The Preparation of Programs for an Electronic Digital Computer*. That would make APIs at least 70 years old!

A convenient reference point for the more recent history of modern APIs—ones that expose functionality via the web—can be traced to Jesse James Garrett’s 2005 article describing the *Ajax* pattern (Asynchronous JavaScript and XML). This was a pattern that made it possible to write scripts for browsers that would make additional calls to other web resources (in the early days returned in XML format) in order to improve the browser user experience.

TIP

It is worth noting that the first element of the Ajax pattern is the use of *asynchronous* requests. Ajax was used to connect with RESTful APIs initially, but soon people got the idea that services could be designed to support asynchronous APIs, too.

Fifteen years ago, the initial use of APIs in browsers (via Ajax) assumed the use of asynchronous interactions. However, today most API calls are still designed and implemented using another pattern—the synchronous REST (REpresentational State Transfer) pattern defined by Chapter 5 of Roy Fielding’s PhD dissertation a few years earlier (2000). However, the use of asynchronous API implementations has grown steadily over the years and is now just as prevalent as synchronous APIs.

So it makes sense to be clear about the difference between these two powerful and popular ways

to build API-based services.

RESTful Versus EVENTful

Probably the biggest difference between common synchronous web APIs (RESTful) and event-driven ones (EVENTful) is the way data is passed between clients and servers. In RESTful systems, clients initiate requests and servers respond to those requests. In EVENTful systems, clients typically receive messages that were "pushed" to them by servers.[2] And the messages that servers send to clients match filters established by the client applications.

However, clients can *send* messages to servers, too. And the messages clients send to the server are usually sent along (by that server) to other clients. This blurs the line between server and client when it comes to asynchronous APIs. Essentially, RESTful systems maintain a strict client/consumer and server/producer model while EVENTful systems support a model where any client application can act as both a message producer and a message consumer.

Let's look at some examples.

RESTful Messaging

In a system that relies on RESTful messaging, you can design and implement a server that has access to all of the data on completed orders—data that is updated each time a new order is fulfilled. Whenever a client requests the list of completed orders, the server would be able to reply with a response that has all the information on the day's completed orders.

Let's assume you are designing APIs for a company called Sportasua, which sells custom high-tech sports shoes. Your task is to architect a small system that keeps the home office, and all the stores in the field, up-to-date on the number of shoes manufactured each day.

Following our model from earlier, we can deploy a RESTful server that has access to the `completedShoes` data that exposes a request action that returns the list of completed shoes for the day. In this case, client applications can send a request to the server via a web address

(`http://sportasua.example.com/shoes-made`) and get a response back that contains the details on a day's list of completed shoes.

At the HTTP level, the response could look something like this:

```
200 OK HTTP/1.1
Content-Length: XXX
Content-Type: application/json
Date: Tue, 04 Apr 2023 17:38:41 GMT

{
  "total_made" : 23,
  "shoe_list" : [
    {
      "shoe_id" : "q1w2e3r4",
      "customer_id" : "j_smith",
      "store_id" : "y67ut5",
      "store_name" : "South Jersey Feet",
      "date_time" : "2023-03-13 12:00:00"
    },
    {
      "shoe_id" : "p0o9i8u7",
      "customer_id" : "m_dingle",
      "store_id" : "r45tef",
      "store_name" : "East North Winglet Footwear",
      "date_time" : "2023-03-13 13:00:00"
    },
    ...
  ]
}
```

Now, any client application (for example, the client app running at the "South Jersey Feet" store) can initiate a request to the home office server and get an immediate response. Although not covered here, there is probably an option that allows the store to send a request that returns only

information about the day's production for that particular store. To get the status of all the orders for just the "South Jersey Feet" store, they might send an API request like this:

```
http://sportasua.example.com/shoes-made/?store_id=q1w2e3r4
```

This works well when all the information you need is in a single location (the home office server) and you can build and run client applications that know where the server is (`http://sportasua.example.com`) and are able to make repeated requests throughout the day to get updates. And it is important that you run the client application frequently if you want the most recent information on the shoe production for the day. You won't know how things are going unless you "ask the server." In this case, even if the custom order for `j_smith` was completed early in the day, the local store will not know that until they send a request to the server for an updated list. Of course, if it is not a very busy day at the factory, you might be making lots of requests that keep returning the same data over and over again.

However, there is a way to design a system that notifies stores immediately when their order is complete—and *only* when an order is complete. That's where event-driven, asynchronous APIs come into play.

EVENTful Messaging

It turns out, while RESTful systems can be relatively easy to design and deploy, they can be rather inefficient when it comes to getting the most up-to-date available data. As mentioned earlier (see "RESTful Messaging"), you may need to schedule frequent requests for data (even if the data has not changed) and you might have to deal with gaps in reporting (when several pieces of data were generated at the source since the last time you made a request).

However, when you are relying on an EVENTful system—one that sends out responses each time the data is updated—without waiting for a client application to make a request—you can get up-to-date information without wasting time (and resources) by making repeated requests from a client app. That's because event-driven systems are designed to send data messages whenever

the data changes. The data exchange is driven by the events at the source (in our case, the shoe factory), not the client applications themselves.

Now, we can redesign our Sportasua example that updates everyone whenever a shoe order is completed. We will implement a server (at the factory) that generates a message each time a shoe is manufactured. And we'll send that message to everyone as soon as we get the data.

Figure 1-1 shows the EVENTful diagram.

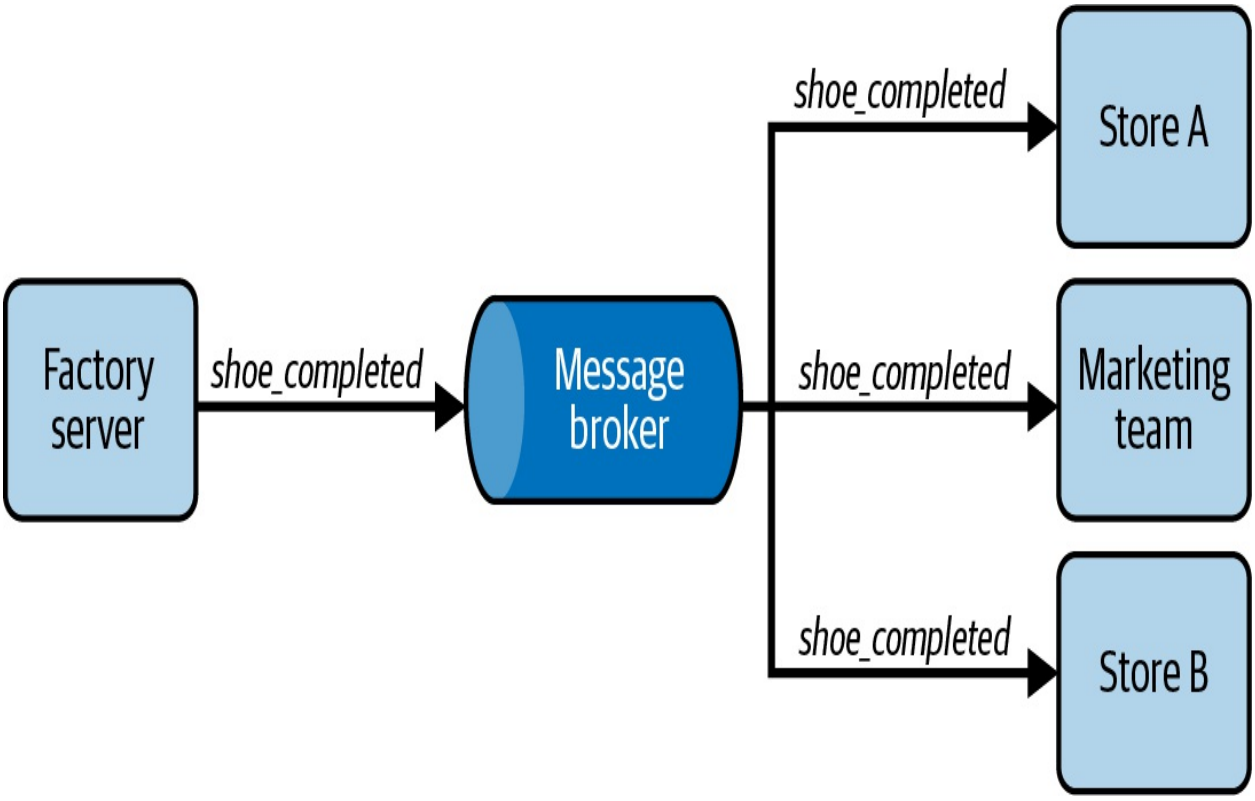

Figure 1-1. Sending/receiving `shoe_completed` event message

Note that the diagram shows lots of possible recipients of the messages. The retail stores, the marketing team at the home office, and even the customer can be notified when the event occurs. Now, it is not just the stores that get the data.

And the actual JSON payload in each `shoe_completed` message is pretty much the same as our RESTful example:

```
{
  "metadata" : {
    "message_source" : "factory_1",
    "event-type" : "shoe_completed"
  },
  "message" : {
    "shoe_id" : "q1w2e3r4",
    "customer_id" : "j_smith",
    "store_id" : "y67ut5",
    "store_name" : "South Jersey Feet",
    "date_time" : "2023-03-13 12:00:00"
  }
}
```

You'll notice that, in the example, the message sent by the factory to all the recipients contains both metadata (`message_source` and `event-type`) as well as the actual data related to the event (`shoe_id` , `customer_id` , `store_id` , `store_name` , `date_time`). The actual contents of each message in an EVENTful system may vary (just as it does in a RESTful system), but you get the idea here.

Also, it is important to point out that only one shoe record is returned here, not a list of them. Each time a new shoe order is completed, a new message will be sent to recipients. If one of those applications receiving the message wants to keep track of *all* messages sent in a day, that application will need to remember all the day's messages.

TIP

RESTful systems generate messages when *clients ask for data*. EVENTful systems generate messages when *the data changes*.

The example message we just saw is typically referred to as an "object message." This is not the only type of message EVENTful systems can generate. And we'll cover that next.

Types of EVENTful Messages

The types of messages that can be sent in EVENTful systems vary. For this short report, we'll describe three of them (notifications, event-carried state, and event-source). The names used here are not standardized, and the exact contents of each type are also not locked in. However, the notion of different types of messages is very common and can be quite helpful when you are designing an event-driven implementation.

NOTE

To learn more, check out the excellent article "What Do You Mean by 'Event-Driven'?" by Martin Fowler.

Each type has advantages and drawbacks. Selecting the right message type for each event in your design can go a long way toward creating a stable, reliable, and user-friendly EVENTful system.

Notifications

The simplest type of EVENTful message is the *notification* type. This message typically contains a minimal amount of information (sometimes only a text message) and often includes a pointer to a RESTful resource where you can find more information about the event that triggered the notification.

Continuing with our Sportasua example, we can design a notification message that is sent each time a customer logs into your website to check on the status of their online order. The notification message might look like this:

```
{
  "metadata" : {
    "event-type" : "user_login",
    "message_source" : "website",
  },
  "message" : {
```

```
      "user_id" : "j_smith",
      "date_time" : "2023-03-13 12:00:00",
      "details" :
        "https://sportasua.example.com/notifications/p1o2i3u"
    }
  }
```

Now, when a user logs into the site, this notification can alert different parts of the organization (sales, marketing, security) and, if appropriate, take some follow-up action. If they wish, they can follow the `details` link to find all the related data for this event.

There are lots of cases where a notification message can be helpful; here are some examples where sending a simple notification can be useful. You can broadcast a simple notification whenever:

- A new shoe order is completed online
- A customer makes a payment on their account
- A person walks through the door at one of your store locations
- A shoe order has been shipped from the factory

TIP

Notification messages are handy when you want to know what just happened but may not need all the details right away.

Event-Carried State

Notification messages are fine for cases where you just need the bare minimum of information to appear in a kind of *alert* status. But what about cases where you want to actually pass a block of data in order to write it to a data store or update an existing data collection? For that, you can use the Event-Carried State Transfer (ECS) type messages.

For example, you might want to create an ECS message that contains details from the

`customer` record, that customer's most recent `order` record, and some added details about the progress of the order through your system. By using an ECS approach, you can combine data from multiple sources without requiring the recipient to know anything about what is stored where in your system. For example:

```
{
  "metadata" : {
    "event-type" : "order_summary",
    "message_source" : "storage",
    "date_time" : "2023-03-13 14:13:12"
  },
  "message" : {
    "customer" : {
      "user_id" : "j_smith",
      "user_email" : "js@example.org",
      "user_voice" : "123-456-7890"
    },
    "order" : {
      "order_id" : "p0o9i8",
      "order_date" : "2023-03-11 13:13:13",
      "order_type" : "Executive Trainer",
      "order_size" : "12D"
    },
    "progress" : {
      "tracking_id" : "a1w2d3",
      "start_date" : "2023-03-12 10:11:12",
      "stage" : "cutting",
      "status" : "working"
    }
  }
}
```

TIP

The ECS pattern works for cases where you need to send a large collection of properties and want to carry all the related information in a single call.

Event-Source

The last type of message we'll cover here is *Event-Source* (ES). Event-Source messages are usually very small and are designed to include just the data elements that are important in a particular event or action. They are essentially transactions of data changes. Each time a data point is modified, that information can be streamed out to recipients. Event-Source type messages are sometimes also called *Delta* messages.

Event-Source data is usually stored in a kind of "event ledger." This ledger contains an auditable record of all the changes to any data objects in storage over some period of time. In fact, this ledger can be used to "replay" the transaction stream from any previous point in time. In this way, you can think of streaming data as a record of the "data flow timeline" for your services.

As an example in our Sportasua shoe company, the assembly line where all the custom shoe orders are fulfilled may be full of automation steps that record the progress of things in the shop: actions like cutting the sole, selecting the fabric, adding the laces, dying the material, and so forth. In our shop, completing each step might generate a series of small messages:

```
...
{"order_id" : "p0o9i8", "stage" : "cutting" ,
  "status":"started", "time" : "2023-03-13 10:11:12"}
{"order_id" : "p0o9i8", "stage" : "cutting",
  "status":"completed", "time" : "2023-03-13 10:14:13"}
{"order_id" : "p0o9i8", "stage" : "welt", "status":"started",
  "time" : "2023-03-13 10:14:15"}
{"order_id" : "p0o9i8", "stage" : "welt", "status":"completed",
  "time" : "2023-03-13 10:16:15"}
...
```

TIP

The streaming message approach works well when you have lots of events to keep track of and you want to create a log of all those events for later reference.

Events and Commands

While messages are an important element of a successful asynchronous API program, they are just the start of a fully functional EVENTful platform. Messages are sent under two key circumstances along a timeline. *Command* messages are sent when something *will* happen (e.g., write this customer data to storage), and *event* messages are sent when something *has* happened (e.g., a customer record has been written to storage).

There are also times when event messages are sent without being caused by a command. For example, when a shoebox on our Sportasua assembly line has crossed an assembly line sensor on its way to shipping, that might trigger an event message (e.g., “order p0o9i8 has been sent to shipping”). Or an event message might be triggered when a customer walks into the shop. In a way, you might consider these external actions (walking into the shop, passing an assembly line sensor) as commands, too.

When you are designing your EVENTful API system, you need to identify the events you wish to keep track of and also define the commands you want to initiate.

Identifying Events

In EVENTful architectures such as asynchronous APIs, it is important to identify all the activities that occur within your target domain. Once you identify the important activities, these can be documented as *events* in the system. These events are then exposed as part of the implementation of your API.

Examples of events include things such as:

- When a customer placed an order
- When a customer logged in to your website
- When a payment or shipment was made

You can also identify activities on the manufacturing floor of your company. Examples of these might be:

- When a custom shoe order first appeared on the factory conveyor
- When that shoe moved from station to station along the assembly line
- When that shoe was finally boxed up and ready for shipment

You can also identify events that indicate there is some problem in your processes. For example:

- When a customer payment was rejected by your bank
- When a customer's shoe was reported as defective
- When a customer canceled their shoe order

TIP

Notice that events are always described as happening in the past. The above examples use words such as *placed*, *logged in*, and *was* quite a bit. It is a good habit to always write your event descriptions as something that has already happened.

As you identify important events, you can design a message for each of them (see "Types of EVENTful Messages"). Some events might need a notification style event message (e.g., customer logged in to your website). Others will need an object style message (e.g., when a new order was placed in the system), and still other events might need a streaming style message (e.g., when a shoe order passes through stations on the factory floor).

You can put together a simple table that will list the key information about each activity including `eventName`, `messageType`, the `conditions` that cause that event to occur, as well as an example `message` and any other `notes`. This can be collected in a spreadsheet as

part of your design process and shared with designers, developers, and architects as needed.

Here's an example event list for our Sportasua example:

1. *Purchase made:* A customer purchases a pair of shoes through the Sportasua website.
2. *Bid selected:* One of Sportasua supplier team's bids has been selected to fulfill the purchase order.
3. *Order fulfilled:* The purchased item has been shipped for delivery to the customer.

Once you have identified your list of important events that happened in the past, you can move on to the next step—defining the *commands* that represent actions that will occur in the future.

Defining Commands

As mentioned earlier, In EVENTful systems, *events* tell you what has already happened in the system and *commands* are a way to tell parts of the system to do something *in the future*. That future might be what we think of as "right now." For example, a command message that changes the status of an order from `unpaid` to `paid` might be recorded almost instantly. However, if there is quite a bit of payment processing going on at the moment, the command might take a few seconds or more to complete. And, if there was a problem verifying payment with associated banks, the status change for some orders might take an hour or even longer.

NOTE

In EVENTful systems, we can't be sure *when* some future event will happen, but we can be sure it *will* happen. And when it does happen, there will likely be an *event* that tells us the action has occurred.

Commands are actions you can define and initiate whenever you want something to occur. For example, if you want to write a new order to the data storage system, you can define a command (and associated message) to do that. If you'd like to make it possible for customers to modify or cancel their order, you can define a command for those actions, too. When designing your EVENTful system, you need to define all the commands you need in order to support the actions

you want to allow.

The process for defining commands is similar to the process for identifying events. You can create a table or spreadsheet that records the `commandName`, `messageType`, an example `message`, and any other `notes`. This is another document that you can share with team members working to implement your EVENTful system.

Here's an example command list for our Sportasua example:

1. *Submit bids:* Sportasua supplier teams should bid to fulfill an open purchase order.
2. *Fulfill order:* An instruction to a supplier team to fulfill an order that they have bid on.

Once you have created your list of commands, you have completed the process of creating all three elements of an EVENTful system: messages, events, and commands.

Summary

In this chapter, we've covered the basics of an EVENTful API system, one that is designed to pass *messages* between machines based on *events* that happened in the past and *commands* that will happen in the future. You learned that there are three types of messages (notifications, Event-Carried State, and Event-Source) and when it is appropriate to use each one. You also learned how to identify and define events and commands and record them in a spreadsheet to make it easy for others on the team to know what is needed to build your EVENTful system.

In the next chapter, we'll go through the steps needed to create a complete design document using the AsyncAPI specification.

1 We asked OpenAI's ChatGPT to help us generate our backronym.

2 There are also pull-based EVENTful implementations such as Kafka.

Chapter 2. Designing APIs with AsyncAPI

In this chapter, we'll get hands-on with the work of designing an asynchronous API for the ordering system of a shoe store. We'll introduce and use the AsyncAPI description language to define the specification of our API. By the end of the chapter, we'll have designed a complete asynchronous API. Let's get started by taking a closer look at the AsyncAPI format.

Why AsyncAPI?

APIs enable software to perform jobs and access data by providing an interface. Well-designed APIs make it easier for developers to write code that uses that interface. Good API design includes using words that developers understand and describing functions that fit the developer's jobs to be done. But, along with these design concerns, designers need a way to clearly describe what their API does in a meaningful, easy-to-consume manner. That's where API *description languages* help.

Description languages are standardized formats for describing the important parts of an API. For many years, the Open API Specification (OAS) has been the universal description language for HTTP and RESTful APIs. Thanks to OAS, an API designer can express their design choices for URIs, HTTP methods, and message schemas in a single, standardized file. That OAS API description can then be automatically imported and used in a rich ecosystem of tools for discovery, documentation, development, testing, and release.

But, OAS isn't a great fit for the EVENTful style of APIs we described in the previous chapter. As of version 3.0, one of the biggest limitations of OAS for event-based integration is that it is very tightly bound to the HTTP protocol. When you describe an API in OAS, you are expected to use URIs, HTTP methods, and HTTP status codes. This works very well for RESTful web APIs, but as we discovered in Chapter 1, event implementations are often run in local networks and HTTP is only one of the protocols that are commonly chosen.

In 2017, Fran Méndez encountered this limitation firsthand while trying to use OAS to describe

an event-driven interface he was working on. But, instead of just solving the problem with a bespoke documentation solution and moving on, Fran did something that vastly improved the lives of future EVENTful developers. Fran forked the popular Open API Specification and modified it to support asynchronous style APIs, calling the new format AsyncAPI. Since then, the AsyncAPI description standard has rapidly gained adoption and has become the "de-facto" standard for event-driven interfaces. As an EVENTful API designer, it's worthwhile investing time to learn how to use AsyncAPI to describe your own event-driven interfaces. That's exactly what we will do in this chapter.

NOTE

To help bring our tour of the AsyncAPI specification to life, we'll use an example from our shoe store application throughout this chapter. You can find the full source for this AsyncAPI description at *https://oreil.ly/dz_2F*.

Writing an AsyncAPI Document

An AsyncAPI definition describes the messages, interactions, network locations, and communication details that an application requires to participate in one of these patterns. For example, AsyncAPI provides semantics that let you describe "publish and subscribe" and notification patterns, as well as protocol and event server details. It provides a concrete language for describing your EVENTful APIs so that other teams can more easily consume and use them without having to contact you directly.

Much like Open API Specification documents, AsyncAPI documents are machine readable and can be written in either the JSON or YAML format. This machine readability means that an AsyncAPI file can be parsed by tools for automation and easier integration. For example, today there are tools that can parse AsyncAPI files and generate documentation, generate code, and even generate insights about the quality of the EVENTful interface.

AsyncAPI documents have a well-defined structure of nested "objects." There are five objects in particular that you should aim to understand: *AsyncAPI* (the root object), *info*, *channels*,

operation, and *message*. We'll learn more about these objects and what they are for as we design our API. In addition to the basic structure, learning how to use AsyncAPI's *components* area to define an object once and use them again and again in your specification is a good time saver. The structure and semantics of AsyncAPI are governed by a specification. In this chapter, we'll be using version 2.6, which is the latest official release as of June 2023, shown in Figure 2-1.

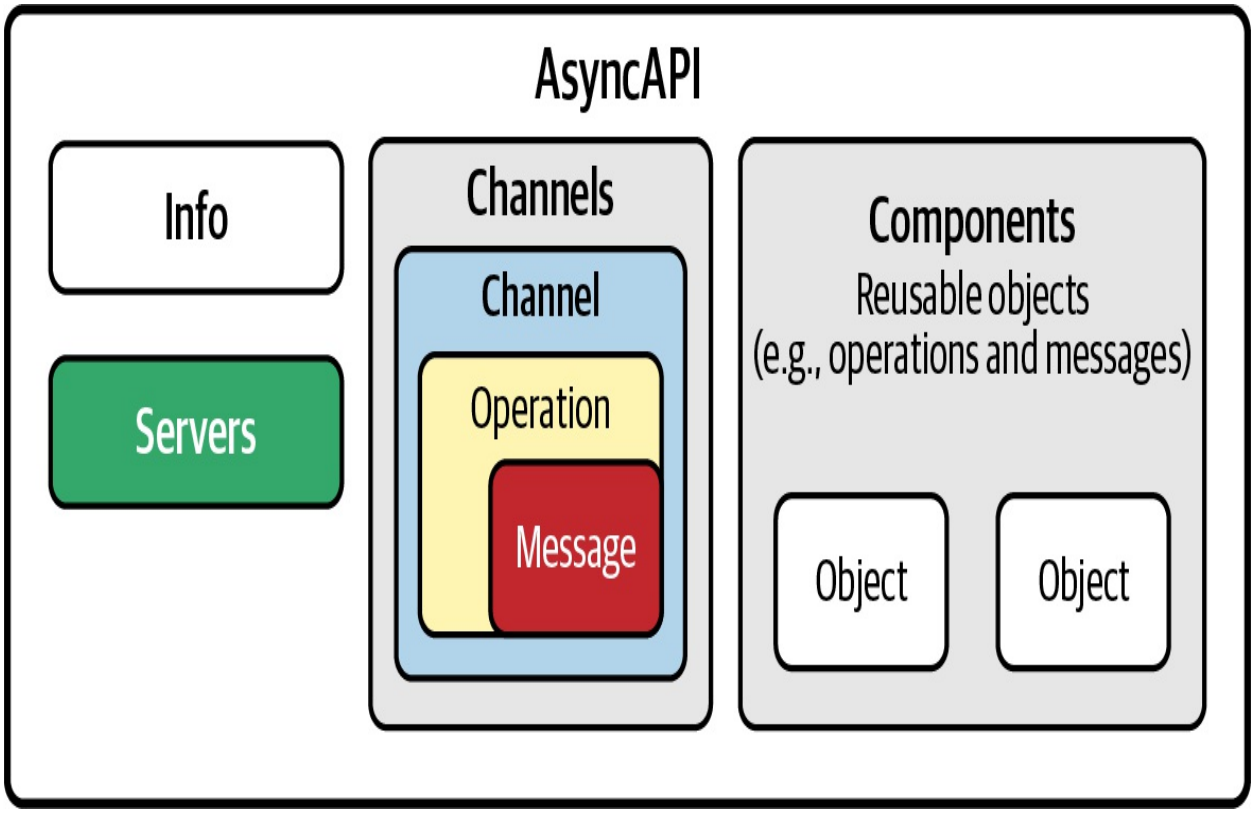

Figure 2-1. Basic structure of an AsyncAPI 2.6 document

TIP

To learn more about the AsyncAPI specification and its parts, we recommend that you start with the excellent AsyncAPI documentation at *https://oreil.ly/33KDG*.

Now that we know the basic structure of an AsyncAPI document, we can start designing one for ourselves. But, first we need to introduce the problem space that we want to design an API for: the asynchronous shoe ordering interface.

The Shoe Ordering EVENTful API

In Chapter 1, we introduced the fictional Sportasua company, which we used to highlight the differences between REST and EVENTful messages. Now, we'll design an asynchronous API design for Sportasua that addresses its shoe ordering requirement. Sportasua uses many suppliers around the world with multiple fulfillment centers that can ship shoes to customers. Our goal is to design an ordering system that takes advantage of the decentralized nature of EVENTful systems. In particular, we want low coupling between the systems that supply the shoes for delivery and the systems that handle the sales orders.

In that spirit, we're going to design an order fulfillment process that lets suppliers bid on sales orders. From a customer perspective, the mobile and web applications for buying shoes will still feel simple, synchronous, and immediate. But, behind the scenes we are going to introduce a more complicated following sequence of asynchronous events. Our target flow will be as follows:

1. *Purchase requested:* A customer purchases a pair of shoes through the Sportasua website.
2. *Respond to customer:* The website notifies the customer that the purchase is complete with the promise that shipping details will be sent shortly.
3. *Start purchasing process:* The web application publishes a `purchase order` event to all Sportasua supplier systems.
4. *Bid to fulfill:* All Sportasua supplier applications receive the `purchase order` event and *may* publish a `sales bid` command if they can source the shoes.
5. *Compare bids:* A fulfillment application receives all of the `sales bid` events and compares them.
6. *Select bid and notify:* The fulfillment application selects a bid and publishes an `order fulfilled` event, notifying the suppliers that a bid has been chosen and should be fulfilled.
7. *Fulfill order:* The customer receives shipping and tracking information via email from the fulfillment system.

This flow of events allows Sportasua to find the best supplier for each purchase at runtime. In a

traditional synchronous RESTful system, we might have to call each supplier individually to compare prices and determine the most efficient supply chain. But thanks to this asynchronous EVENTful design, suppliers can act in parallel and the network of suppliers can grow or shrink without us having to write new code.

But, to make this type of decentralized, event-driven system work, all of our applications will need to be able to communicate with each other in a standardized way. Although we can add and remove suppliers, they will still need to know how to read and write events. Specifically, all of our applications must have a shared understanding of:

- The protocol and network locations to use for event communication (e.g., a Kafka server)
- The location of event messages in the event system (e.g., a Kafka topic)
- The structure of event messages that need to be parsed or created (e.g., a `purchase order` event)

That's why the AsyncAPI language is so useful. We'll use AsyncAPI to describe these shared definitions so that our software development and operations teams will be able to easily understand how to interact in our new EVENTful system.

With the target flow for our ordering process in hand, let's define who will use the interface and what we think they'll want to achieve.

Defining the Scope of Our API

Experienced API designers know that the first step toward a good API design is to define who will use it and what they will use it for. That knowledge will help you make better decisions about functionality, naming conventions, standards, and message structures. To keep things simple, we've already done the work of defining the needs of the Sportasua shoe ordering API for you, as shown in Table 2-1.

Table 2-1. Jobs to be done in our API

Teams	Applications	Jobs to be done
Mobile and web	Mobile app, website	Place order, check on order status, receive updates on status
Sales system	Receive purchase orders and bid on them	Fulfillment

There is a fairly wide scope of needs to fulfill in our shoe ordering interface. One of the classic challenges in API design is to decide how to group functions together. For example, should there be one API that performs all ordering-related tasks? Or do we split them up into a fulfillment API and a sales API?

In the EVENTful world, functional grouping is largely determined by your grouping perspective: domain-centric or application-centric.

Domain-Centric

When you take a domain-centric API perspective, you describe all of the related events within a single logical boundary (for example, sales order fulfillment). But, you describe the events without tying those descriptions to a particular application or system. For example, we could describe our sales order fulfillment API in terms of the events for order placement and order bidding generally, without any reference to the web applications, sales systems, or shipping systems that we expect to use them. This is useful for situations where we want to build a centralized, shared, domain-based view of our world, with less dependence on the software systems that need to operate within it.

Application-Centric

In an application-centric perspective, we define the API in terms of the events that a specific software application might produce or consume. Unlike a domain-centric API, the application-

centric API is bound by the activities of a single application (or application archetype). For example, we could define an API just for our sales system that describes the purchase order sales events it knows how to process. That description could then be given to the web and fulfillment teams so that they could interoperate together. This way of describing an EVENTful API is useful when you want to give teams the autonomy to design and own their own interfaces, potentially allowing similar domain-based interactions to be described by different events.

Choosing which approach to take is really up to you. But, we're going to take the application-centric design approach for our API. That's mostly because the AsyncAPI and OpenAPI specification it's based on have a strong application-centric perspective built into the language. That means, we'll define our API in terms of the events that a sales system application will consume or produce during the order fulfillment process. In fact, we'll call our API the "Sales Application Fulfillment API."

With our scope now established, we can start writing the AsyncAPI document that describes our design. As we mentioned earlier in this chapter, you can define an AsyncAPI using either the JSON or YAML format. We'll use YAML in our examples in this chapter because it is an ideal format for human readers.

TIP

If you want to get hands-on with AsyncAPI development as we walk through these sections, the AsyncAPI team provides an online editor that you can use.

Our first step will be to define the version of the AsyncAPI specification we are using by populating a required `asyncapi` field. We'll also define the metadata for our API design using an object called `info`, as shown in the following example:

```
asyncapi: '2.6.0'
info:
  title: Shoe Sales System  Order Fulfillment API
  version: '1.0.0'
```

```
  description: |
    Describes purchase order and supply chain events for the
     Sales System.

  Events: Purchase order created, supplier selected,
   Purchase order fulfilled
  Commands: Bid on purchase orders
```

In this example, you can see that we've identified the AsyncAPI version as 2.6.0. We've also populated the `info` object with the scope and purpose of the API. These metadata details are useful for people reading the AsyncAPI document and can be used by document generation tools and API portals. We've used a very simple example here, but the more descriptive you can be in this section, the better your EVENTful reference documentation will become.

Now that we have the context and scope of our API defined, we can move onto the next big decision: choosing our transport protocol and system.

Identifying the EVENTful Transport

One of the big differences between a RESTful API and an EVENTful API is the variety of transport protocols and messaging software (or "middleware") that you need to communicate to developers who plan to use your API. While a RESTful API is assumed to use HTTP, an EVENTful API could use HTTP, MQTT, Kafka, or any other of a number of other transport schemes. To help capture and communicate this information, the AsyncAPI specification gives us a `servers` object that we can use to describe the transport mechanism for our API.

For our example's purposes, let's imagine that Sportasua has decided to use a Kafka event infrastructure because it fits their needs for high-throughput, scalable messaging. Now that we know we are using Kafka, we can add the connection details to a `servers` object after the `info` object as shown in this example:

```
  asyncapi: '2.6.0'
```

```
info: .... info details here...

servers:
  dev:
    url: test.mykafkacluster.org:8092
    protocol: kafka-secure
    description: Test broker
  production:
    url: mykafkacluster.org:8092
    protocol: kafka-secure
    description: Production broker
```

Notice that AsyncAPI lets us define multiple brokers in named environments. This means that we can use the same AsyncAPI specification throughout the development lifecycle. Note that in a full-fledged description, we'd likely include secure connection details, which we've omitted here for brevity.

With the Kafka server details defined, let's move on to defining the event messages that we plan to send through Kafka to support our sales and ordering process.

Documenting Our Messages

In an EVENTful system, the way you describe an event message is pretty important. In fact, the message schema is one of the biggest coupling points for your system. Producer systems need to "know" how an event is structured so that code can be written to serialize these messages. Consumer systems need to "know" the same structure so that their code can parse the event messages correctly and do something useful based on their contents.

For our example, we'll define messaging components for two key event messages:

1. *Purchase order event message:* The event message our supplier system expects to receive when a new purchase order event has occurred

2. *Fulfillment event message:* The event message our supplier system will transmit when it tries to bid to fulfill a purchase order

When you design an event message, it's important that you think about the activities that you expect your software applications to perform with the data from the message. For example, when we design the purchase order event message, it makes sense to include a `deadline` field so that our supplier software system knows how long it has to prepare a bid.

In the AsyncAPI specification, we can use the `messages` object to define the schema of the event messages for the API. AsyncAPI supports the use of a JSON Schema-like description language that allows us to specify the structure, properties, and format rules for our event messages. By defining the event messages in this way, we can make it easier for producers and consumers to read and write event messages programmatically.

NOTE

JSON Schema is a standardized way of describing structure and data types for JSON objects, using the JSON language. You can read more about JSON Schema at *https://oreil.ly/WRgs3*.

Earlier in this chapter, we mentioned that AsyncAPI gives us a way of componentizing parts of our API description so that we can re-use them. This is especially useful for message and data definitions because things like dates and addresses are so often used and reused.

To take advantage of this, we'll define our message objects inside of the `components` object of the AsyncAPI document. The `components` object is like a library of AsyncAPI objects we can create so that we can reference them to use again and again. Here is an example of the message payloads for our API's events specified within the `components` object:

```
asyncapi: "2.6.0"
info: ... info details ...
servers: ... server details ...
components:
```

```
schemas:
...
   orderReceivedPayload:
    description: A sales order event that can be used to
      trigger sourcing and procurement processes
    type: object
    properties:
      salesOrder:
        $ref: "#/components/schemas/salesOrder"
        description: A sales order that contains the
          customer's identifier, the items they have ordered,
          payment status, and shipping details.
      sourcingPreferences:
        type: object
        properties:
          preferredSupplierIds:
            description: A list of supplier IDs that have been
              identified as preferred suppliers for this order
            type: array
            items:
              type: string
          deadline:
            description: The deadline that suppliers need to
              meet before a sourcing decision is made
            type: string
            format: date-time

  sourcingBidPayload:
    description: A bid request made by a supplier as part of
      the sourcing process
    type: object
    properties:
      orderId:
        type: string
      supplierId:
        type: string
```

```
          bidExpiry:
            type: string
            format: date-time
          bidPrice:
              $ref: "#/components/schemas/money"
          conditions:
              type: string

  ...
```

As you can see from this snippet, we've defined two payload components: `orderReceivedPayload` and `sourcingBidPayload`. We haven't yet defined how these payloads will be used, but by defining them as components, we can make that decision later. The payloads have been defined using a structure that is inspired by JSON Schema and is fairly easy to understand. For example, we can see that the order received message will be represented as a JSON object and will have `salesOrder` and `sourcingPreferences` properties. But, notice that the `salesOrder` property contains a slightly cryptic `$ref` property within it.

The `$ref` instruction lets AsyncAPI know that we are referencing another component. When this document is parsed, the parser will replace this `$ref` line with the actual schema that it is referring to. In this case, we are referring to a schema definition for the sales order. To save printing space, we've omitted the details of the `salesOrder` schema that is being referred to here. The same is true for the `money` schema that is referred to in the `sourcingBidPayload` object. To complete the document, you can copy and paste these schemas from our finished example specification or you can try creating the schemas yourself as an exercise.

Now that we have our payload schemas defined, the final step is to define the messages themselves. A `message` component is just an abstraction in AsyncAPI that stores data schemas. In our example, we'll create two `message` components: `orderReceived` and `bidOnOrder`, as follows within the `components` section:

```
  asyncapi: "2.6.0"
```

```
  info: ... info details ...
  servers: ... server details ...
  components:
    messages:
        orderReceived:
          name: orderReceived
          summary: Sales order details for an order that has been
            accepted by the frontend application
          contentType: application/json
          payload:
            $ref: "#/components/schemas/orderReceivedPayload"

        bidOnOrder:
          name: bidOnOrder
          summary: A supplier bid message to fulfill an order
            awaiting fulfillment
          contentType: application/json
          payload:
            $ref: "#/components/schemas/sourcingBidPayload"
    schemas:
      orderReceivedPayload: ...
      sourcingBidPayload: ...
```

The `name` and `summary` properties of our message definitions are primarily used as documentation fields, helping our developers to understand what these messages are for. The `contentType` property is also a documentation field that indicates the message format of the actual message. The `payload` property contains the actual schema for an event message. In our example, we've used the `$ref` shortcut to reference a payload structure defined within the `schemas` object of the AsyncAPI document.

NOTE

To save space, we haven't included the payload examples in the text of this report. But, you can view our example payloads in the reference example.

Defining the Channel

We have now defined Kafka connection details and the schemas for our event messages. But, we haven't specified where or how an application can use our Kafka server to publish or subscribe to these events. We can define those details in AsyncAPI's mandatory `channels` object. In AsyncAPI, a channel defines the location for events within the system as well as the *operation* that an application should use to produce or consume the events. The *operation* describes the type of interaction that is allowed in the channel we've described.

TIP

This language of channels is changing in the upcoming version 3.0 of the AsyncAPI specification, including a redefinition of the publish and subscribe operations.

For example, in our Kafka-based event-driven architecture, we will create two new topics to store events messages for orders and for bids. Each of these topics will be defined as a *channel* in our AsyncAPI document. In addition, we'll use the keywords *publish* and *subscribe* to describe the *operations* that will be supported in the Kafka topics we've defined.

The following example shows how we would define this in an AsyncAPI document:

```
asyncapi: '2.6.0'
info: .... info details ...
servers: ... server details ...

channels:
  shoesales/orders:
    subscribe:
      message:
        $ref: '#/components/messages/orderReceived'
  shoesales/bids:
    publish:
```

```
        message:
          $ref: '#/components/messages/bidOnOrder'

  components: ... component details ...
```

In this snippet, we've identified two separate Kafka topics for our event messages. At runtime, our web sales application will know it can publish `orderReceived` messages to the `shoesales/orders` topic on the Kafka server. Our channels definition also lets our systems know that `bidOnOrder` messages should be published to the `shoesales/bids` topic. With these channels defined, we now have a concrete definition for our AsyncAPI design, ready to be published in an API documentation portal or for the generation of application code.

Summary

In this chapter, we designed an async API for an example order fulfillment process for our fictional Sportasua store. We started by identifying the scope for our API based on the teams that would use it and their jobs to be done. We introduced the AsyncAPI description language and started to use it to document our API design. Then, we defined the data, messages, and channel binding for the API design using the AsyncAPI format.

Now that you've had an introduction to AsyncAPI-based design, you'll be able to use it to describe your own asynchronous APIs. In the next chapter, we'll see what it takes to run asynchronous APIs at scale within a software team and run an EVENTful architecture successfully.

Chapter 3. Becoming an Asynchronous Company

Earlier in this report, we learned the fundamentals of EVENTful architecture and introduced the AsyncAPI format. Now, in this last section of the report, we'll dive into the habits and capabilities you'll need in order to run a robust EVENTful API architecture at scale. We'll take a closer look at "event-oriented" thinking, discover the tools that you'll need, and look at the unique challenges of change in an EVENTful world.

To get started, let's focus on what it takes to incorporate asynchronous APIs into an organization that already understands synchronous RESTful APIs.

Integrating Async APIs into Your Company

Unless you are starting a new company from scratch, there is a good chance that you are already working with RESTful APIs. Incorporating asynchronous APIs into an existing organization can be challenging. That's because the EVENTful style can be jarring for people who are used to the mental model, tools, and operating models from the RESTful world.

To help with this transition, we've outlined a few key areas to tackle when you move from RESTful to EVENTful: event-oriented thinking, combining RESTful and EVENTful styles, and team enablement.

Pivoting to Event-Oriented Thinking

If your team is already building and running RESTful APIs, you'll need to change the way you think about API design. That's because event-oriented systems need a reactive mindset rather than an imperative one. In practice, this means that your teams will need to "unlearn" two foundational RESTful API interactions: request-reply and client-server thinking.

Event thinking: Event-react, not request-reply

In the RESTful world, we design APIs while thinking about the data or services that clients may want to access. In this context, the client initiates every interaction and we design APIs that can serve their needs effectively. For example, when designing a shoe ordering system, the RESTful API designer asks, “What order-taking operations do we need to provide for the client?” and “How should we respond to requests to create a new order?”

But, in EVENTful systems, the events describe things that have already happened. Instead of thinking in terms of what a client may want to accomplish, we need to think about what events have already occurred. For example, faced with the same ordering system design problem, an EVENTful API designer asks, “What types of orders might the system have created?” and “What can we do after an order has been initiated?”

This shift of focus from future tense to past tense can be jarring for RESTful API designers, but it is the key to EVENTful architecture thinking. Event messages reflect things that have already happened and they cannot be changed. Another way of saying this is that events are immutable.

Event thinking: Publish to many, not client to server

In RESTful architecture, clients and servers communicate synchronously. The RESTful API designer creates an interface for a server to host, and they think about individual client systems that will connect to the server. Each request-reply conversation involves a single client and a single server, and the server has almost perfect knowledge and control over the messaging interaction. Once the connection is terminated, the messages cease to exist, unless the client or server decide to persist them.

But, in most EVENTful architectures, this imperative, client-server style of communication doesn’t exist. Instead, event messages are thrown out into an “ether” (often hosted by middleware), from which any authorized message consumer can retrieve event messages and react. In the EVENTful world, a single unique event message can be consumed by multiple consumers. In some systems, an event message can be consumed days, months, or even years after it has been emitted.

Much of this behavior depends on the middleware and infrastructure of the underlying EVENTful architecture. But, the key mental shift for designers is to consider event messages as abstract resources that can be consumed by multiple systems, rather than bespoke responses for a single consumer.

Complement RESTful with EVENTful

The technology industry is driven by innovative thinking and new techniques. For most software architects and engineers, there is a thrill to discovering a new and better way of doing something. This often translates into refactoring, modernization, and transformation projects with the goal of replacing the old with the new.

Be wary of completely replacing your RESTful systems with the EVENTful style of asynchronous interaction we've described in this report. While it is certainly possible to build an entirely event-oriented software system, in practice it makes sense to use both. In fact, we believe that there are many situations where a RESTful style of API design makes more sense than an EVENTful one.

For example, performing query interactions over a network is much easier to support with a RESTful design than an EVENTful one. On the other hand, data synchronization operations are more easily implemented with an EVENTful design. We highlighted some of these characteristics and examples in Chapter 1.

We encourage organizations to embed async thinking into existing RESTful design teams. That means that if you have a Center of Enablement (CoE) or design authority for APIs, you should be able to extend their scope to include asynchronous design as well. But, beyond design and governance, you'll likely need to invest in a team of specialized talent to manage the complexities of EVENTful middleware, networking, and architecture. For example, Kafka requires special expertise to gain the right balance of availability, data distribution, and performance. This type of infrastructure design often warrants a specialized operations team.

As you work with asynchronous APIs, you'll develop your own preferences for how and when

you want to use them in your organization. But, most importantly, if you can define your EVENTful versus RESTful decision logic clearly, you'll help a lot of people in your organization. That's because the cognitive load of deciding when and how something should be asynchronous can really slow down the API design process. By giving your teams clear guidance, you'll help them focus on more important challenges in their design work.

Introduce Tools as Asynchronous Enablers

Over the last 15 years, there has been an explosion of tools for the RESTful API ecosystem. Now, as asynchronous APIs gain popularity, we are seeing a similar growth and evolution in the tooling system. An important first task for any asynchronous company is to choose the right set of tools that can drive the right set of outputs. Asynchronous API tools usually align with the stages and focus areas of software development, such as design, engineering, testing, governance, security, and observation.

The challenge with tooling is that you may end up with too many of them. Tools are becoming increasingly specialized but can cause your teams big problems if they don't work well together. Thankfully, specification formats like OpenAPI and AsyncAPI create a common, standardized language that helps tools integrate with each other. As the AsyncAPI specification continues to gain popularity, we expect more tool developers to adopt it as a universal language for describing asynchronous APIs. This will in turn accelerate the growth of the EVENTful tooling ecosystem. We expect the EVENTful tooling landscape to evolve dramatically over the coming years.

A key to enabling success for teams at scale will be to provide some guidance or standardization of the tools, languages, and frameworks that should be used. Organizations will have different tolerances for standardization or prescriptiveness of tooling choice, but we recommend that the best and most suitable tools are highlighted for your teams to use.

In the next section, we'll introduce some of the key areas where tools can enable your work.

Platform Design Considerations

When the tools, infrastructure, and teams that enable asynchronous work are combined, they become the “platform” for an EVENTful architecture. A good internal platform helps your teams accomplish their individual product and project needs while also serving the needs of the system and organization as a whole. Platform design is a deep and complex topic that is beyond the scope of this AsyncAPI report, but we wanted to highlight some of the key decision and enablement areas you need to focus on.

Choose the right event transports

We’ve highlighted the importance of the transport protocol and event infrastructure a few times in this report. Choosing the right transport for your EVENTful system is certainly one of the biggest design decisions you’ll need to make. Generally, you should consider the access model, durability, and geographic scale of your asynchronous APIs when making this decision.

The access model of an EVENTful system is defined by the organizational structure and relationship of consumers and providers. For example, an asynchronous API may be used by consumers who do not work for your organization and have software deployed in a network outside of your own boundaries. This distinction between private and public APIs is important. The transport mechanism for an API used by internal systems in a private network will usually be different from a public API that is streaming events to clients. For example, event broker middleware products like IBM’s MQ are not typically deployed on the public web. Instead, you’ll probably use an HTTP-based transport to stream or distribute events.

The durability of events in your interactions will also have a big influence on the type of transport that you use. Some EVENTful patterns allow for event messages to be very ephemeral. That is, they exist only for a short time, and once they are consumed, they disappear. An example of this is a notification pattern in which a message provider streams notification events as they happen to interested consumers. But, some EVENTful use cases rely on events being persisted and available for longer periods of time. An extreme example of this is when events form a transaction log that can be used to rebuild the entire state of an application. These events must be stored, in order, for as long as possible. The durability requirements of events in your system will shape the requirements of the event servers you need.

Finally, the geographic scale and performance requirements of your system will have an impact on the tools and servers you choose. Apache Kafka has gained in popularity because it can be used in situations where you need to keep durable event logs synchronized across large instances. But, it requires specialized expertise to configure it in a way that balances performance, scale, and safety.

Standardize event integration patterns

Beyond the server and transport architecture, you'll need to consider how the applications in your EVENTful system can coordinate together to implement business or application processes. For example, in Chapter 2 we defined an order fulfillment process from the perspective of a supplier software component. But, we did not codify how all of the applications within the system could work together to fulfill an order. This is sometimes referred to as event orchestration, event coordination, or process management. In practice, deciding where this logic lives is a key decision you'll need to make early on.

When we say you need to decide where the logic is located, we mean that you must decide which components in your system own the logical decision process at runtime. For example, when a shoe sales order is published and consumed by a supplier software system, how does it know that the next logical step is to publish a supplier bid event? Does the supplier software own that logic? Or are we implementing a centralized component that can orchestrate these flows? This type of decision is typically related to the structure of your organization and the ownership of these orchestration decisions.

By centralizing the orchestration or process flow, you'll make it easier to wire events together into coordinated activities. But, the trade-off is that your centralized orchestration system will become more complex and over time may be difficult to maintain at a reasonable rate of change. Centralizing the logic also requires a good level of coordination between the teams involved. Good tools can help here.

Conversely, if you keep the logic within your application components, you'll be able to give

teams more autonomy in how they react and act when events are received. This can work very well when you have a connected set of teams working together closely. But, it can be a challenge at scale. Some organizations start with a decentralized logic model and move to a central orchestration model over time.

Implement enabling tools before growth

Improving the internal developer experience and providing “the right tools for the right job” have become a key principle in modern engineering companies. In the API world, this is doubly important as so much of the value of APIs comes from their ease of use, ease of maintenance, and ease of change. We expect the tools landscape for EVENTful architectures to change dramatically over the next five years, especially as disruptive technologies like generative AI start to change the way we work. In practice, implementation teams will usually find ways to solve engineering and testing challenges. As these solutions develop, you can standardize the best ones and scale them across your organization. But, based on our experience, there are two system-level problem spaces that you should address at the outset: event design and event discovery.

Event Design Tooling

Designing the right events, application roles, and system boundaries for an event-driven architecture is often challenging. That’s because it can be difficult to gauge what the right decisions should be. We recommend that you introduce standardized design practices for asynchronous design early in your move to EVENTful architecture. For example, Alberto Brandolini’s EventStorming technique is a simple, paper-based process that will help your teams iterate an EVENTful design in a structured way. Adam Dymitruk’s Event Modeling method provides you with a standardized way of documenting the flow of commands, queries, and events. And of course, the AsyncAPI specification gives your teams a standardized way of documenting their API designs.

Event Discovery Tooling

For as long as there has been APIs, there has been a problem of finding the right API to do a specific job. This problem persists in the EVENTful world—as adoption of asynchronous APIs increases in your organization, you'll soon find that implementers are unable to discover the events that could help them. This discovery need happens at design time, when teams are building software, as well as at run time, when software is responding to events. For incumbent organizations, the main challenge here is that API catalog, portals, and discovery tools usually already exist, but they lack the ability to serve metadata that is unique to events. Some API management vendors are starting to incorporate EVENTful APIs in their turnkey portal solutions. Other vendors are focusing on the event space and will sell you specialized asynchronous API portals. You'll need to make a decision on whether to implement a holistic, wide-ranging portal or provide a set of specialized catalogues. The good news is that the switching costs are not usually too high, so you can always change your mind down the road and migrate your API catalogue data into a better solution when it arises. We'll highlight an example of an event discovery approach later in this chapter when we talk about asset catalogues.

Preserve performance, quality, and scale

One of the most important aspects of designing an EVENTful platform well is ensuring that it remains resilient, performant, and functional. The spirit of this type of design is very similar to RESTful API system design, but there are a few nuances in the EVENTful world that must be considered. In our experience, the three most important areas to address are infrastructure optimization, testing coverage, and culture.

The nonfunctional aspects of an EVENTful system will largely depend on the quality of the transport protocol and middleware systems that support it. This is true of any API-based integration, but unlike the RESTful domain where the practice of tuning HTTP protocols and servers are extremely mature, tuning an EVENTful infrastructure can require some specialization. Make sure that you have a team that can tune your brokers and servers so that the platform can move messages around with minimal latency while meeting your scaling and durability requirements. This often requires a very good understanding of the underlying protocols and server technology.

The asynchronous APIs that make up your system need to be tested regularly. You can apply good testing practice to your EVENTful APIs just as you would for RESTful APIs and software applications. For most organizations, this means employing automated testing, unit testing, and performance and security testing. One of the challenges teams face in practice is that automated testing of events can be difficult because of a lack of existing tools and frameworks. Some teams have had success "wrapping" their event integrations with RESTful interfaces to make automated functional testing easier.

Finally, just like any complex software system, an EVENTful system benefits greatly from a test-driven, DevOps and/or Site Reliability Engineering culture. Driving out variation, adopting an everything-as-code mentality, and unifying "dev" and "ops" will all go a long way toward the resilience and scalability of the platform as a whole.

Maintaining and Modifying Async APIs

Experience shows that it is unlikely that your first attempt to identify all the events and commands in your system will be the last. Almost always, as the system is implemented, new events are identified ("How did we miss that one?") and new commands are needed ("I didn't know we did that ourselves"). Although you can avoid lots of surprises by using techniques such as EventStorming (see "Event Design Tooling"), you will still probably end up, at some point, wanting to add new events/commands or modify the ones you already have defined.

Adding Elements

The good news is that it is relatively easy to *add* new messages, events, and commands without disrupting other parts of the existing system. Simply defining a new message model doesn't force any clients to consume that message. And identifying additional events that broadcast those new messages works without causing any clients to break. The same is true for creating new commands to make some change in the future.

Often you can define new messages and events and wait for messaging clients to start consuming

them. However, there may be cases where you want to make sure some new events are not ignored by client applications. That means you need to coordinate releases a bit, but you can at least add the events before the clients start to subscribe to them.

Handling Modifications

The trick comes when you want to try to *modify* an existing message emitted by an existing event or command. That can cause problems. Message consumers might not be prepared to handle the change ("Hey, who added this URL to the message?"). This is especially true if you want to try to take away some part of the messages ("Where did the `hatsize` field go?"). So what's the solution?

You can limit the "breakage factor" of your message changes by creating *new* messages and never modifying *existing* messages. For example, you might decide to modify the existing `shoeDetail` message by moving some fields and renaming others.

For instance, in the `shoeDetail` message in the following example, it might be decided to collect the shoe attributes under a single subsection (`shoe`) and rename the `cobbler` field to `bootmaker` :

```
{
    "context": {
      "source" : "shop_floor",
      "type" : "shoeDetail"
    },
    "event": {
      "shoe_id" : "p0o9i8u7",
      "customer_id" : "jsmith",
      "size" : "12.5",
      "model" : "Executive Trainer",
      "color" : "maroon",
      "status" : "welt_station",
      "cobbler" : "mgeppetto"
```

```
    }
}
```

Now here's what the new message will look like:

```
{
    "context": {
      "source" : "shop_floor",
      "type" : "shoeDetail_2"
    },
    "event": {
      "shoe" : {
        "shoe_id" : "p0o9i8u7",
        "size" : "12.5",
        "model" : "Executive Trainer",
        "color" : "maroon"
      },
      "customer_id" : "jsmith",
      "status" : "welt_station",
      "boot_maker" : "mgeppetto"
    }
}
```

Instead of modifying the existing `shoeDetail` message, you can create a new message (with the rather uninspired name `shoeDetail_2`). Now you can code any components that emit the `shoeDetail` event message to *also* emit the `shoeDetail_2` message. Any applications that are consuming the `shoeDetail` message will continue to work as before. And, when they are ready, they can update to consume the new `shoeDetail_2` message.

Maintaining an Asset Catalog

As you can imagine, as a system matures, more and more messages, events, and commands will be added to the system. It is important to diligently document these system elements to avoid

confusion and misunderstandings in the future. The best way to do that is to maintain an *asset catalog*.

The asset catalog is a list of all designed messages, identified events, and defined commands. It is also a good idea to track topic filters, message brokers, and applications sending/receiving messages. This becomes the reference guide everyone can use to see what messages have been already created and which events and/or commands emit those messages. To start, you can maintain a simple document similar to the ones we discussed in "Identifying Events" and "Defining Commands".

This asset catalog works well for small collections but can get a bit hard to maintain as your system grows over time. Another handy approach is to create a more detailed, searchable document that makes it easier to navigate from messages to events to commands and so forth in a single interface.

A great open source software (OSS) tool for this kind of approach is EventCatalog.

Figure 3-1 shows an example of this.

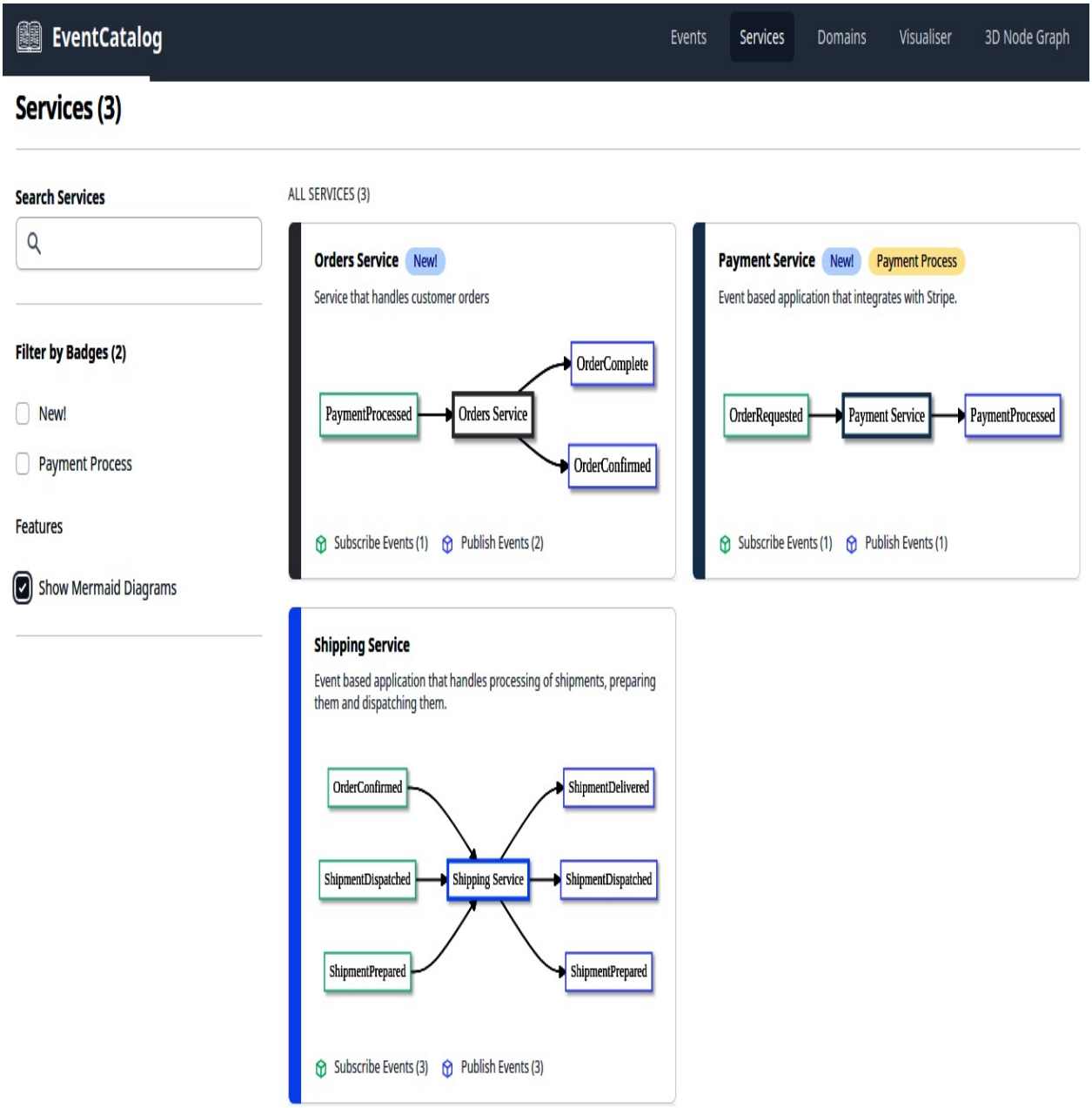

Figure 3-1. Services view from the OSS tool EventCatalog

However, this kind of asset catalog takes a bit more work to build and maintain. We recommend you start with the simple document and, if and when you need it, you can move to a more involved version of the asset catalog.

NOTE

Note that the asset catalog is not the same thing as the AsyncAPI document. The catalog is a free-standing list of

EVENTful assets (messages, events, and commands). The AsyncAPI document is a design document that details how some of those assets are arranged into a solution. You'll need to maintain updated versions of both kinds of documentation throughout the life of your system.

Overall, the key to successfully managing your EVENTful system over time is to maintain an accurate and detailed asset catalog.

Summary

In this chapter, we've explored some of the design and governance details of supporting a robust EVENTful API architecture. We reviewed how to properly integrate asynchronous APIs into an existing infrastructure (see "Integrating Async APIs into Your Company") by adopting a kind of "event-oriented" thinking and introducing tooling as an enabler.

We also covered the tasks of deploying EVENTful systems (see "Platform Design Considerations") by focusing on important platform tools and other details. Finally, we reviewed some details on how to successfully maintain and modify your EVENTful systems (see "Maintaining and Modifying Async APIs") with the least amount of disruption and downtime.

All these elements along with the basics covered in Chapter 1 and the use of the AsyncAPI format in Chapter 2 can help determine how you can best take advantage of the growing strength and flexibility of EVENTful systems powered by asynchronous APIs.

About the Authors

An internationally known author and speaker, **Mike Amundsen** consults with organizations around the world on network architecture, web development, and the intersection of technology and society.

Ronnie Mitra is a digital transformation leader at Publicis Sapient, enabling companies of all sizes to take advantage of emerging and disruptive technologies.