From RESTful to EVENTful

Mike Amundsen and Ronnie Mitra

From RESTful to EVENTful

Mike Amundsen and Ronnie Mitra

This book is for sale at http://leanpub.com/fromrestfultoeventful

This version was published on 2021-02-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface (Mike)

(3 pgs)

Who Should Read This Book?

TK

What's Covered?

TK

What's Not Covered?

TK

Acknowledgements

TK

1. Why EVENTful? (Mike)

(10 pgs)

Highlights

Increased business demand for real-time interactions along with advances in technology are leading organizations to update their existing RESTful API platforms with EVENTful approaches. These EVENTful APIs patterns can also improve a company's ability to successfully execute on digital transformation initiatives, improve the quality and reusability of their API platform, and provide additional ways to observe and evaluate the business value of the overall API program.

From RESTful to EVENTful

A trend we have been noticing, as we talk to organizations large and small about software architecture and design, is that companies which previously worked to standardize on using only *RESTful* API patterns are now starting to incorporate another style of APIs; ones that we are calling *EVENTful* APIs. These are APIs that rely on interaction patterns different than REST's client-server and call-response model. Instead, EVENTful APIs support a publish-subscribe style interaction where any service can *publish* one or more channels (sometimes called *topics*) to which any number of other services or clients can *subscribe*. Subscribers receive data that is pushed to them any time a publisher has new data that matches the registered topic. Because new messages are sent as soon as publishers have new data, this type of API call is initiated by domain events (users logging, services writing, updating, or removing data, etc.) instead of being initiated by a client somewhere sending a request to a server.

We'll cover a more detailed definition of EVENTful and different ways you can implement event APIs in Chapter 2.

For close to twenty years, the common standard of APIs on the web has been summed up in one word: "RESTful." This notion of how APIs should be designed and implemented stems from Roy Fielding's now (in)famous 2001 PhD dissertation[1] which contained a single chapter devoted to his recommendations for softare that runs on a network. That chapter was titled "Representational State Transfer" or REST. For all sorts of reasons, Fielding's dissertation has been adopted, debated,

[1] https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

and (to hear Fielding's own telling) miss-understood by thousands of software developers over the intervening years.

It is important to point out that REST is not the only API style available to organizations. Among the more popular API patterns companies have used over the years are *remote procedure calls* such as Google's gRPC, *distributed objects* like the SOAP services, and *remote data services* like GraphQL. All these patterns can be implemented using the IETF's HTTP[2] protocol. There has also been a very successful event-driven API pattern based another protocol called MQTT[3] that has been around just about as long as HTTP.

This leads to a number of questions: If MQTT and the other options to REST have been around for so long and so many options exist, why are we seeing more EVENTful implementations now? What's changed? Are we seeing an increase in EVENTful services due to technological advancements? Changes in business? New market forces? It turns out all of these factors have contributed to the rise in EVENTful APIs in companies larges and small. And, from what we've been seeing around the world, this increase of event-driven APIs is bringing with it both opportunities and challenges.

So, before we jump into the technical details of EVENTful APIs and how you can design and implement them, let's take a moment to review the technical and business aspects of this change in the API landscape.

Why Businesses Need EVENTful Systems

You can read just about any business blog or article today and find some reference to the need for "real-time" information about your business. This focus on instant feedback is driving much of the push for digital transformation within organizations today. But getting more data faster is just a means to and end. And that "end" is the steady monitoring and improvement of your company's ability to deliver value to customers.

To meet this challenge, a big part of the digital transformation journey depends on making your IT operation more responsive to small changes in the health and welfare of systems running within the company. And it also means paying attention to customer and partner demands for more up-to-date information on project status and product shipments. There are a number of businsess-related motivations driving the increased reliance on EVENTful software systems at work here and we'll focus on four of them in this chapter:

Market Forces
: The changing market, consumer demand, and operational efficiencies are all key drivers in the increase of real-time or EVENTful APIs in all types of companies, large and small.

Adding Speed
: EVENTful APIs can add speed to your runtime products and to your internal processes.

[2] https://tools.ietf.org/html/rfc7230
[3] https://mqtt.org/

Increasing Connections
: More connections means more data points and that means more opportunity to observe and improve data flows between internal systems and systems outside the company's network.

Expanding Reach
: Adding EVENTful APIs can expand an organizations reach at both the public API and private API levels.

Market Forces

One of the reasons for the rise of EVENTful software systems can be attributed to forces in the API market itself. As more devices get added to the internet, there is bound to be a similar increase the number of APIs needed to keep them connected. And many of the recent devices that people want to connect are "headless" devices such as home alarm systems, heating and air conditioning platforms, and general building monitoring systems. And, with the increase in robots and other autonomous vehicles, there are a whole new set of APIs for monitoring and operating these devices. Finally, watches, exercise equipment, and even clothing are getting added to the API ecosystem and need to be connected, updated, and monitored.

This growing market for API-connected devices is also driving consumer expectaions. It's a virtuous (or not, depending on your point of view) cycle that pushes device makers to API-enable their products and encourages consumers to connect them together at a higher and higher rate.

Finally, in some cases, these devices may be costly to install, maintain, and operate. Or they may bew charged with monitoring costly machinery themselves. Waiting for hours or even minutes to check to see if a piece of equipment can be idled off or needs to be fired up can result in unwanted added costs or, even worse, lost revenue. The need for near real-time response and reaction to ever-chainging conditions is one of the key drivers for automation in general and these products need real-time APIs in order to meet customer demand as well as hold a lid on operating costs.

The changing market, consumer demand, and operational efficiencies are all key drivers in the increase of real-time or EVENTful APIs in all types of companies, large and small.

Adding Speed

EVENTful APIs also make it possible to add a new level of *spee* to your company's IT infrsstructure. By establishing real-time information flows, your organization and can consume and analyze information at a faster rate than traditional RESTful systems. In fact, a common problem for those adding EVENTful APIs to their API landscape is the challenge of what to do with all the new data your systems are generating. Ironically, trying to consume too much data can actually slow the organization's reaction time, too.

But the speed of information flow is just one of the benefits of adopting an EVENTful approach to APIs. Another one is the ability to monigtor and manage your own internal processes used to build and support existing APIs. Using real-time logging of the number of builds your company does, the

number of tests you're running, etc. can give you a real time picture of your own company and how it is doing in meeting stated goals and guidelines.

And, as we'll see in the next section on the technology aspects of adopting EVENTful APIs, implementing more real time APIs can lead to a more de-coupled and agile platform upon which you can build your IT solutions and products. By it's very nature, EVENTful API platforms rely on generalized message-passing through opaque routers. It is also relatively easy to add new messages and filters once a system is up and running. This, in turn, can lead to faster start-up times for MVP (minimum viable product) style projects that can adapt and grow over time based on consumption patterns learned from production use.

EVENTful APIs can add speed to your runtime products and to your internal processes.

Increasing Connections

Another force that is driving companies to add more EVENTful endpoints to their system is the need to *increase connections* between both components within the organization's IT infrastucture and between customers and partners. While comes of this work to increase connections is strategic – a way to expand and strengthen business relationships – much of the focus on connections is part of the inevitable tide of more connected, more reactive internal systems.

One example of this drive for internal, reactive connections is the rising use of in-house status dashboards to monitor both business and infrastructure health. It is more and more common today to walk into the main lobby of a company and see a giant screen displaying the real-time count of connected customers, the location of shipped goods, and/or the number of completed business transactions within the last hour. This display is a way for divisions to monitor their business in near real-time. No need for daily, weekly, and monthly printed reports delivered via in-person meetings. Instead, anyone in the organization can call up the company dashboard and view the latest stats on the business and compare that to the company's stated objects and key results (OKRs[4]).

But business monitoring is just part the real-time data story. As we'll see (below) when we talk about the technology side drivers for EVENTful systems, another reason for increased connections is the ever-growing number of 'connect-able' components inside the IT shop. Most platforms offer ways to automatically "plug-in" to monitoring endpoints and stream that internal performance data straight to the same kinds of dashboards that are displaying business metrics. But this time the screens are displaying the number of connected components around the world, the speed of transactions, error-rates, and more. These displays rely upon gigabites of real-time data streaming from every connected component and gateway within the company's network in order to present a "traffic report" on how data is moving through the system. This allows managers to essentially predict and correct network congestion and roadblocks the way transportation and logistics managers ensure the safe and timely delivery of their physical goods.

More connections means more data points and that means more opportunity to observe and improve data flows between internal systems and systems outside the company's network.

[4] https://en.wikipedia.org/wiki/OKR

Expanding Reach

Another important element in all this is that many of the devices we've listed here depend upon a real-time interaction experience. As people jog, they want to see their steps increase in near real time. As vehicles become autonmous, they need to "see" and react to new information on the road in real time. For many product and organizatsions, real-time, EVENTul APIs are the "price of entry" into the markets they want to reach.

There are also cases where an organization's internal systems can benefit greatly from EVENTful APIs. For some products a RESTful approach for public APIs is still more than adequate to serve the market, but within that same company's firewall, they may be able to take advantage of EVENTful APIs in order to react to partner and consumer demands more quickly, more smoothly, and more accurately.

EVENTful APIs can also improve a company's ability to reach across deparment boundaries and improve internal efficiencies betweewn off-the-shelf products as well as in-house built systems.

Adding EVENTful APIs can expand an organizations reach at both the public API and private API levels.

Why Now? Technology is Changing

As we discuess at the start of this chapter, the need for real-time communications at the business level has been around in one form or another for over twenty years. What's different now is that the technology required to support real-time messaging has finally caught up and become widely available at low cost. Advances in hardware and software have played their role with the heavy lifting going to the software side. Architectural models that favor small-sized messages, asynchronous programming patterns, and virtualized servers have all had a major role in making it easier to design and build EVENTful systems.

Three key drivers on the technology side will be highlighted here including:

The Rise of Asynchronous and Reactive
: EVENTful systems both rely upon and are driven by the use of reactive data production and asynchronous data-consumption patterns.

The Growth of Microservices
: EVENTful software architecture can make it easier to modify an exsiting system built from small, independently-deployable services.

The Power of Serverless and Cloud Native
: Both serverless and cloud native implementation rely on EVENTful patterns to successfully support services on the network.

The Rise of Asynchronous and Reactive

One of the key drivers in the inreasing demand for EVENTful systems is the rise in the use of asynchronous an reactive style programming models. One of the best-known web frameworks for reactive programming was initially created by Jordan Walke while at Facebook[5]. Intially driven by the need to improve handheld drive user experiences, React and other similar frameworks rekindled an interest in EVENTful implementation patterns that date back to the mid-1980s when it was used to handle interactions with hardware-centric Supervisory Control and Data Acquisition (SCADA[6]) devices.

A key principle in reactive systems is that data changes flow through the system automatically – in "real time". However, as more and more data flows through the system, the ability of components within that system to immediately consume, evaluate, and respond to changes dimishes. For that reason, a second pillar of EVENTful systems is that responses are *asynchronous* – the time it takes for new data flows to result in system behavior changes can vary.

Asynchronous programming[7] is ideal for handling heavy input-output (I/O) operations, parallel processing of large amounts of data and processing long-running tasks like monitoring a package shipment or virtual project progress over time. This ability to deal with large amounts of data that might span a long period of time is the perfect match for the increased use of reactive interfaces.

EVENTful systems both rely upon and are driven by the use of reactive data production and asynchronous data-consumption patterns.

The Growth of Microservices

Another reason for the increase in EVENTful architecture is the continued success of the Microservice approach[8] for desgining and building components. Similar to Service-Oriented Architecture (SOA[~ch01-soa]), microservices emphasizes creating small, independent components focused on a single business capability (search, data filtering, customer onboarding, etc.). Early versions of this model were introduced in the 2000s and around 2012 the name "microservices" was attached to this pattern.

Using small, independently-deployable components allows software teams to update and release their components more often and with less likelihood of internal errors. But it brings with it new challenges in the form of additional inter-component network traffic and the possibility of introducing breaking changes in the processing and data models share between components. It is these last two elements (processing and model changes) that EVENTful architecture can alleviate. In EVENTful systems, the process model can vary based on how components publish and subscribe to data flows (we'll talk more about this in Chapter 3.

EVENTful software architecture can make it easier to modify an exsiting system built from small, independently-deployable services.

[5] https://en.wikipedia.org/wiki/React_(web_framework)
[6] https://en.wikipedia.org/wiki/Reactive_programming
[7] https://en.wikipedia.org/wiki/Asynchrony_(computer_programming)
[8] https://en.wikipedia.org/wiki/Microservices

The Power of Serverless and Cloud Native

Finally, another key technology element that has given rise to increased use of EVENTful designs are the growing reliance on *serverless*[9] and *cloud native*[10] runtime platforms. The two phrases have similar histories and definitions and both focus not on the software architeture (microservices) or the message model (asynchronus and reactive) but, instead, focus on the way server infrastructure is implemented and managed at runtime.

Serverless platforms don't actually run without servers, of course. But serverless platforms greatly reduce the effort needed to mount a new server, scale that server up and down in response to traffic loads, and monitor and manage that server over time. In order to support these options, serverless platforms rely on real-time event services to handle control-channel messaging like monitoring the health of server instances, tracking parallel copies of a particular service, and allowing operations staff to spin up, modify, and take down running services without the need to stop and start network connections. In essence, serverless platforms are EVENTful platforms.

The same could be said for cloud native implementations. Like the name implies, cloud native services take advantage of widely distributed, independently operating server instances hosting code, data, and workflow services – sometimes widely separated by time and distance. While serverless platforms use EVENTful patterns to communicate control-channel information, cloud native platforms also relay in EVENTful messaging to accomplish the stated goals of the hosted microservices in the cloud including passing application data between services, emitting API and UI content on demand. It is difficult to imagine the success of cloud native services without the use of real-time EVENTful messaging.

Both serverless and cloud native implementation rely on EVENTful patterns to successfully support services on the network.

Where Events Fit In

So business demands are calling for more real-time services and technology advances are making it easier and more cost effective to support EVENTful implementations. That's all good. But what is it about the EVENTful approach that makes it more valuable from both the business and technology perspective?

It turns out a well-executed EVENTful API program can be an important catalyst for enabling your organizatsions modernization and digital transformation efforts. That is because and EVENTful approach makes it easier to put data consumers – customers – first when you design your APIs. EVENTful APIs are also relatively easy to monitor and that can improve your company's ability to observe and evaluate the business value of your API platform.

And, the same holds true for the technical aspects of EVENTful services. EVENTful designs usually result in fine-grained messages and narrowly-focused data feeds. Both of these can reduce the time

[9] https://en.wikipedia.org/wiki/Serverless_computing
[10] https://github.com/cncf/toc/blob/master/DEFINITION.md

it takes to implement a new service or data feed and decrease the chance that API changes cause downtiem in your system.

Let's look at each of these aspects (business and technology) more closely.

Meeting Business Demands

Inreasing your use of EVENTful services can help your organization meet the growing demaind for real-time inteactions. Employing EVENTful design patterns helps put information consumers up-front and center. Who is consuming our services and what data do they need to solve their problems? Answering these questions can result in well-targeted EVENTful APIs that help your partners, customers, and internal teams get the right data at the right time in order to do their work.

EVENTful implementations can also help your teams improve the overall observability of your API platform. By monitoring the various real-time feeds, tracking consumption, and analyzing usage, you can gain important insight into just which APIs are making a positive contribution to your company's OKRs and to the organization's bottom line.

Finally, using EVENTful APIs can make it easier to not only share information with partners and customers but also *consume* data from other sources and incorporate that information into your own products and services. As more organizations in your market space adopt EVENTful patterns, you and your teams can take advantage of the EVENTful opportunities around you. In some cases, your use of EVENTful APIs can give your teams the edge in competing for a share of the growing real-time market.

Reducing Technical Challenges

increasing your reliance on EVENTful patterns can also help your teams overcome a handful of technical challenges on the road to digital transformation and agile implementation. This can result in a smoother on-boarding of new services, fewer disruptions when updating existing components, and greater resilience in consuming important data.

As you'll see in Chapter 3, you can take advantage of EVENTful patterns when designing your service interfaces and data messages. These patterns help you focus on small-grained changes and to express the design ways that reduces the chance of breaking existing interfaces. This small-grained, focused design approach can also lead to a programming patterns that encourage the use of small, indepedently-deployable data consumers and producers. This can not only speed the process of "getting from idea to install", it can also reduce the likelihood of breaking existing code.

Finally, the process of writing EVENTful data consumers (see Chapter 4) can lead developers to writing more general and data-driven client applications that are both efficient and runtime but also resilient. They can be less prone to crashes when data models change and more able to survive minor model changes without experience additional downtime.

Summary

In this chapter we reviewed the an increasing trend in enterprise API programs to add more real-time, EVENTful, APIs alongside their existing RESTful implementations. This trend is drive by business demands at both the internal and external level. The need to improve the company's time to market, the increased connections across organizations as well as within the IT family and the ability expand the reach of the company's digital footprint are all leading to a rise in EVENTful services.

On the technology side, improvements in asynchronous and reactive programming languages, the continued growth of microservice style components, and the power of serverless and cloud native platforms have all made the design and implementation of EVENTful architectures safer, cheapers, and easier to use.

Finally, organizations that embark on this real-time journey can find a well-designed and implemented EVENTful program can help improve both the business and technological foundation of the company and improve that enterprise's ability to compete in the ever-growing real-time API market.

In the next chapter, we'll dig deeper into just what it means to create an EVENTful platform and how tha differs from the commomn RESTful approach used today.

Additional Reading

* For more on Objectives and Key Results (OKRs), check out John Doerr's "Measure What Matters"[11]
* For more on Reactive programming you can start with the "Reactive Manifesto"[12] and this "explainer" article from "Linux Magazine"[13]
* You'll find a nice "deep dive" course on Asynchronous Programming in O'Reilly's "Getting Started with Reactive Programming—Asynchronous Java, Promises, Actors, and Reactive Streams"[14]
* See the book "Microservice Architecture"[15] for an overview of microservices in general.
* For a hands-on microservice experience, we recommend "Microserivces Up and Running"[16].
* The booklet "What Is Serverless?"[17] provides a quick overview on this important subject.
* Check out the book "Cloud Native"[18] from O'Reilly for more on th cloud native world.

[11] https://www.penguinrandomhouse.com/books/546304/measure-what-matters-by-john-doerr-foreword-by-larry-page/
[12] https://www.lightbend.com/blog/reactive-manifesto-20
[13] https://www.linux-magazine.com/Issues/2014/163/Reactive-Programming
[14] https://www.oreilly.com/learning-paths/learning-path-getting/9781492028611/
[15] https://learning.oreilly.com/library/view/microservice-architecture/9781491956328/
[16] https://learning.oreilly.com/library/view/microservices-up-and/9781492075448/
[17] https://learning.oreilly.com/library/view/what-is-serverless/9781492074915/
[18] https://www.oreilly.com/library/view/cloud-native/9781492053811/

2. The EVENTful Architecture (Ronnie)

(15 pgs)

Highlights

TK

What is EVENTful?

As a bit of review, let's talk about just what we mean when we use the word *EVENTful*, especially when juxtaposed with the more common term, *RESTful*.

While RESTful systems focused on resources, EVENTful solutions focus on actions. And, more importantly, EVENTful solutions rely on asynchronous interactions. This opens up many more possibilities for building responsive, real-time solutions. That means an EVENTful design offers services the ability to request and respond in real time and to do it in a way that provides more flexibility in the way the data is shared, displayed, and mixed across devices and platforms.

Even though EVENTful architecture is ideal for cases where real-time responses are needed, they do not result in actual "real time" systems. Especially in large, complex implememtations, publishing thousands of messages to possibly hundreds or thousands of subscribers takes time. This can result in inconsistencies in various parts of the running system.

This temporary inconsistency is a *feature* not a bug. Technically, this aspect of EVENTful systems is called *eventual consistency*. While delivering messages in an EVENTful system may take time, eventually these multiple storage centers will be consistent with each other. Eventual consistency is a feature of asynchronous systems that helps them scale better as your system and volume grows.

There are many patterns that fall under the EVENTful umbrella. In the sections to follow we'll focus on the most-used patterns here. That includes:

* Webhooks and Pub-Sub
* Event Notification (EN)
* Event-Carried State (ECS)
* Event Sourcing or Event Streaming (ES)
* Command / Query Responsibility Separation (CQRS)

Technically, CQRS is not an event-based pattern but it is very often used in the same place as other EVENTful patterns and can be a very handy way to manage the transition from a RESTful to an EVENTful architecture model.

Web Hooks and Pub-Sub

Before diving into the three classic event-driven patterns of EN, ECS, and ES, there are two patterns that have been around for decades worth mentioning: 1) webhooks [19] and 2) publish-subscribe [20]. *Webhooks* have been around since 2007. It was Jeff Lindsey who is credited with first using the term in his blog post "Web hooks to revolutionize the web" [21] where he writes: "Web hooks are essentially user defined callbacks made with HTTP POST."

Web Hooks offer a quick way to use existing RESTful infrastructure to publish near real-time alerts to preconfigured subscribers.

The ideas behind the *publish-subscribe* pattern can be traced back to a paper published in 1987 entitled "Exploiting virtual synchrony in distributed systems"[22]. In that paper, the authors state their aim is to "… provide a toolkit for distributed programming" and they describe the process of publishing a set of messages, subscribing to receive those messages, and broadcasting the published messages to the list of subscribers. This should sound familiar since this is the list of operating elements of most all EVENTful systems.

Both pub-sub and webhooks are examples of early reactive, asynchronous patterns that influenced the kinds event-driven architecture (EDA) we use today. In the next sections we'll cover the three patterns you'll most commonly see and use in your own IT shop: Event Notification, Event-Carried State, and Event Sourcing (or Event Streaming). We'll also round out our list of patterns by exploring the Command-Query Responsibility Separation (CQRS) pattern.

Event Notification (EN)

The simplest EVENTful pattern is **event notification**. Martin Fowler describes EN as something that happens "when a system sends event messages to notify other systems of a change in its domain."[23]. This is essentially like getting a "ping" when something happens (e.g. "a user updated their account").

The Event Notification (EN) approach uses short, descriptive messages optimized for use by the message consumer.

An important aspect of the EN pattern is that it is primarily a "one-way" messaging system. Messages get sent to subscribers when something happens and the sender does not expect any reply from message receivers. This one-way approach makes EN a good de-coupled pattern that is relatively easy to implement in existing systems.

[19] https://en.wikipedia.org/wiki/Webhook
[20] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
[21] https://web.archive.org/web/20180630220036/http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/
[22] https://dl.acm.org/doi/10.1145/37499.37515
[23] https://www.martinfowler.com/articles/201701-event-driven.html

EN messages are usually quite small, too. They typically have just as few fields to identify the event title, it's name (or tag), some data items related to the event such as date/time or a link to , and possibly a link that the receiver can follow if more info is needed. Below is an example of a typical EN message. This one comes from Google's Firebase platform [24]:

A typical event notification message

```
{
  "message":{
    "token":"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...",
    "notification":{
      "title":"Portugal vs. Denmark",
      "body":"great match!"
    },
    "data" : {
      "Nick" : "Mario",
      "Room" : "PortugalVSDenmark"
    }
  }
}
```

Event notifications are helpful when you want to publish brief "alerts" about what is going on in a component or service. They are usually one-way messsges that don't require a reply and can be easy to add to an existing RESTful system.

The downside of the EN pattern is that the message usually doesn't carry enough information to allow a receiver to get the full picture of what happened. For that, you need to actually carry additional data in the message. For that you need to level-up in your EVENTful patterns and employ the Event-Carried State or ECS pattern.

Event-Carried State (ECS)

A similar pattern is event-carried state or ECS. In this approach, the actual related data is "carried" along with the alert (e.g. "a user updated their record. here is the user object ..."). with ECS messages, the message is more than just a notification. The message actually contains details about what was added or changed for a particular object or resource.

The Event Carried State (ECS) approach uses complete, self-describing messages to optimize for data integrity and accuracy.

One of the key advantages of the ECS approach is that, by carrying details of the data that was added/changed, it can reduce traffic on the network. This is different than using the EN approach

[24] https://firebase.google.com/docs/cloud-messaging/concept-options

(see above). Of course, by adding more information in the message, you also increase the size of messages and run the risk of carrying around data that few recipients really want or need.

Below is an example of an ECS message. This one comes from Amazon'e AWS platform [25].

A typical event carried state message

```
{
  "id": "6f87d04b-9f74-4f04-a780-7acf4b0a9b38",
  "detail-type": "AWS Console Sign In via CloudTrail",
  "source": "aws.signin",
  "account": "123456789012",
  "time": "2016-01-05T18:21:27Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
      "eventVersion": "1.02",
      "userIdentity": {
            "type": "Root",
            "principalId": "123456789012",
            "arn": "arn:aws:iam::123456789012:root",
            "accountId": "123456789012"
            },
      "eventTime": "2016-01-05T18:21:27Z",
      "eventSource": "signin.amazonaws.com",
      "eventName": "ConsoleLogin",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "0.0.0.0",
      "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.\
36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36",
      "requestParameters": null,
      "responseElements": {
            "ConsoleLogin": "Success"
            },
      "additionalEventData": {
            "LoginTo": "https://console.aws.amazon.com/console/home?state=hashArgs%2\
3&isauthcode=true",
            "MobileVersion": "No",
            "MFAUsed": "No" },
      "eventID": "324731c0-64b3-4421-b552-dfc3c27df4f6",
      "eventType": "AwsConsoleSignIn"
      }
}
```

[25] https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/EventTypes.html#macie_event_type

The ECS message pattern has some important implications for data storage services. First, when you are using ECS messages broadcast to multiple sources – and each of those sources will be storing some or all the data in the message – you introduce the possibility of inconsistency in data storage. This happens when Storage System A (SSA) processes and stores the information for the ECS message before Storage System B (SSB). When someone reads from SSA they may not get the same results as when they read from SSB. This inconsistency may only last for a few milliseconds but, in a high-traffic system that sends out thousands of ECS messages, the likelihood of an inconsistent read increases rapidly.

Eventual Consistency

The challenge of synchronizing data storage at multiple locations is common in all EVENTful systems and is called the **Eventual Consistency**[a] problem. Most storage systems built for EVENTful use have conflict and consistency algorithms built into the platform so you rarely need to do anything special when you build your EVENTful data store. However, it is important to be aware of it and learn how to hangle cases where consistency may cause a problem (TK – ronnie, will you deal with this in the data section below?)

[a] https://en.wikipedia.org/wiki/Eventual_consistency

Second, in systems that rely on a single source of truth or system of record (SOR) data storage pattern, the ECS record needs to have all the possibly relevant data in order to ensure the data storage is kept up-to-date. This might mean carrying the same data in subsequent update messages even if that data hasn't changed. Including this "unchanged data" can be important when the data storage system needs to validate the integrity of the information before saving and processing it for future use.

If you want to continue to support data-writing in your EVENTful implementations and you also want to reduce the size of message payloads, you'd be better off using another pattern of EVENTful messaging: Event Streaming.

Event Streaming/Sourcing (ES)

The pattern most people associated with EVENTful design today is sometimes called event sourcing or event streaming (ES). In the ES world, every event is expressed as a transaction that is shipped to anyone interested and is also recorded in a kind of "ledger" that holds all the event transactions. In the case of the user information we've been discussing, there would be a transaction that indicates the change of the data in each of the user record fields. This might actually be expressed as multiple transactions. One of the unique aspects of ES is that most all transactions that change state can be "reversed" with another transaction. This is often equated with basic accounting ledgers where debits can cancel out credits in the ledger.

The Event Streaming (ES) pattern uses small, discreet messages designed to carry just the information that changed in order to optimize for near-realtime updates of the targeted data stores.

In the ES approach, the primary goal is to design a message model is compact and yet still expressive. Both provider and subscriber applications are coded, or bound, to the message design itself. One of the challenges of creating a scalable and stable ES-style system is to carefully design the message(s) that will be sent back and forth within the system. A common tactic is to adopt a pattern where each objet your system ("product", "customer", "warehouse", etc.) becomes a message (see below).

Object-centric event-sourcing customer message

```
{
  // TK: make this better (mamund)
  "id":"CjYGiSipOE",
  "created_at":"2019-12-20T11:27:52-05:00",
  "updated_at":"2019-12-25T18:24:57-05:00",
  "givenName":"...",
  "familyName": "...",
  "email" : "...",
  "streetAddress1" : "...",
  "city":"...",
  "stateProvince":"...",
  "postalCode":"...",
  "countryCode:"..."
}
```

In each of the examples above, you can see the 'shape' of each object. While it is possible to design a "product message" and then design a "customer message" and then a "warehouse" message and so forth, this is not always the most effective way to implement an ES-style system. Instead, it can be better to design a single message that can be used by all parties to carry whatever information they wish. The results in a more generic message that, while a bit harder to humans_ to read, can be much more useful over time for machines to deal with.

Below is the same customer information displayed in a more generic event-source style message:

Generic event-sourcing customer message

```
{
  // TK: make this better (mamund)
  "id":"CjYGiSipOE",
  "created_at":"2019-12-20T11:27:52-05:00",
  "updated_at":"2019-12-25T18:24:57-05:00",
  "givenName":"...",
  "familyName": "...",
  "email" : "...",
  "streetAddress1" : "...",
  "city":"...",
  "stateProvince":"...",
  "postalCode":"...",
  "countryCode:"..."
}
```

Another thing to keep in mind when using the ES-style approach, is that small, flat messages are easier to turn into transactions than larger, deeply-nested messages. In fact, messages that only contain the data that was changed (e.g. the familyName of a customer) are much easier to "reverse" with another transaction if that is ever needed. This can be an advantage when you have large, complex records that change often.

As you might expoet, there is a downside to this fine-grained approach to tracking data changes. A system that just ships small bits of data around is not optimised for storing data in a way that is easy to query. Typically, ea small transaction needs to be written to a more strongly-typed data model. For example, changing the familyName of a customer may be one small transaction, but that transaction must be shipped to some system of reaord responsible for maintaining customer records and the one record's familyName needs to be written to durable storage (TK durable?).

Optimizing ES-style systems for writing data may mean you have a lot more work to do when you want to permanently store and later retreive that same data. Be sure you implement a system that can meet your SLA for reading data, too!

ES-style systems require you to carefully weigh the costs and advantages writing data quickly vs. reading data quickly. One way to tackle this problem is to adopt a hybrid approach to event-driven architecture by implementing a system that clearly separates querying data from writing data.

CQRS

TK: layout CQRS as a common "middle-way" for those shifting to EVENTful systems.

**TK Bertrand Meyer Command-Query Separation (https://en.wikipedia.org/wiki/Command%E2%80%93query_-separation) also (https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction) **

**TK Greg Young CQRS naming (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf) CQRS and Event Sourcing

TK clean up, expand, (re)move eventual consistency stuff

Messages vs. Resources

Throughout this review of what it is that makes up EVENTful architecture you'll find a common theme: *messages*. One of the key elements of EVENTful architecture is the design, transport, and handling of the messages passed from producers to consumers. While RESTful architecture focuses a great deal on addressable *rsources* in order to power the system, EVENTful architecture places a great deal of emphsis on the messages themselves.

This focus messages changes to leverage in the architecture. In EVENTful systems, addressable locations are *message brokers* and these brokers are known to producers and consumers. The brokers themsevles are really just 'switchboards' for recieving and delivering opaque messages as needed. That makes is relatively easy to change brokers without adversely affecting producers and consumers. That is in contrast to RESTful systems where changing the address of resources in the system can break an existing service or application.

At the same time, EVENTful systems depend a great deal on the ability of consumers to understand the messages reeived from brokers. Changing the shape and/or content of an EVENTful message runs the risk of breaking the services that depend upon the message consumers. When you consider that EVENTful solutions might continue to run for months or possibly years, managing the messages formats becomes very important for the long term health and stability of your EVENTful systems.

This is one of the reasons that we focus on messages as an essential part of yuour EVENTful implementaton.

Now that we have a handle on the common message patterns for EVENTful systems and their importance in EVENTful systems, the next challenge is to outline the basic platform elements – the infrastructure needed in order to implement and operate your EVENTful systems.

EVENTful Infrastructure

TK

Data Storage

TK

Message Brokers

TK

Testing Platform

TK

Security Tooling

TK

Build Pipeline

TK

EVENTful Mindset

TK

Actors, Not Consumers

TK

Publish-Subscribe, Not Client-Server

TK

Events, Not Resources

TK

Other Considerations

TK

Summary

TK

Additional Reading

* If you want to dig deeper into the world of Webhooks, we recommend you check out "Webhooks – The Definitive Guide" at https://requestbin.com/blog/working-with-webhooks/.
* The book "Software Architect's Handbook" by Joseph Ingeno has a nice section on the various EVENTful message patterns. You can find it here: https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/
* Bertrand Meyer's 1988 book "Object-Oriented Software Construction" (https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction) covers the origins and details of the concept of Command-Query Separation.
* Greg Young published a PDF file entitled "CQRS Documents by Greg Young" in 2010 (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf). It is there that Young introduces CQRS and Event Sourcing.

3. What You Need to Know (Ronnie)

(15 pgs)

Highlights

Designing any API-enabled system takes a reasonable amount of technology "know-how". You need to understand at least the basics of networks, coding and integration tools to put together a solution that works with composable parts. But, in most cases its not enough for an API-enabled system to just "work". It also needs to work *well* - and that requires a special kind of design skill and experience.

Working well means that the software goes beyond doing what it's supposed to. It also needs to help your team or company with their strategic goals. For example, a business that wants to cut down on costs will want software that requires less people to maintain. A product team that wants to make lots of changes very fast will want software that is designed for fast and frequent release with low change costs. A company that wants to monetise data will want software that makes it easy to collect, tag and manage all the data in the system.

Professional software designers, developers and architects need to be good at making software that fits those kinds of needs. Just about anyone can learn how to implement a simple API - but, not everyone can make APIs that are highly maintainable, work easily together and don't break when the unexpected happens. That turns out to be the hard part.

Thankfully, over the years the software architecture industry has developed a number of techniques, philosophies, patterns, methods and styles to make this job a little bit easier. The eventful system is one of those styles and it can help you build software that works well. But, like any software style there are trade-offs to consider.

An eventful system isn't the answer to every software integration problem that you'll face. It has a set of characteristics that make it well-suited for a broad number of scenarios, but it certainly isn't a silver bullet. You need to understand the characteristics of an eventful system and how they relate to the problems you are solving and the type of software that you want to build.

There are three factors that are the most important to consider for an eventful system: coupling, distance and latency. We'll explore each of these factors and how they impact eventful architectures.

TK need segue into this new section

Dissecting the Eventful System

In Chapter 2, we explored a number of different Eventful architecture styles and patterns. As we saw, there are lots of different ways of implementing an Eventful, asynchronous messaging system. That

complexity can make it difficult for us to come up with a good, overarching way of designing them. The good news is that there are some fundamental parts of an Eventful system that are consistent across all those variations. If we can understand those parts, we'll be able to use them to explore the factors of coupling, distance and latency universally.

> The parts of the system that we'll be describing here actually apply to RESTful and synchronous interactions as well. We'll compare the Eventful version and synchronous version of each of them throughout this chapter.

There are five universal parts of an Eventful system that are most important to consider: producers & consumers, the network, the message, the infrastructure and the people. Let's take a look at each of them in turn.

The Network

A key characteristic for the kinds APIs we're talking about in this book is that they use a network to communicate. That probably feels obvious. But, its an important distinction for us to understand. We need to acknowledge that the network exists and that it has an impact on the way we are designing our software. With the modern tooling, systems and layers of abstrasction that we have today, its easy to forget that the network is there.

A classic example of the importance of networks for API design are the fallacies of distributed computing[26]:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogenous

> These fallacies have evolved and extended by different authors over the years. But, Peter Deutsch is usually credited as the primary author.

The fallacies listed above highlight the things that we often assume to be true when we design solutions, but actually aren't. For example, consider the software architecture diagram in [fig-1]:

[26] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

TK fig-1: simple box and line arch diagram

On paper, this architecture paints a picture of a system that is neatly componentised into boxes that can pass event messages to each other. We could implement these components with identical messaging components and build a pretty good system. But, what if we mapped these components to a physical network architecture like [fig-2]?

TKf fig-2: box and line arch with network boundaries, some components at a great distance, others in a mobile app.

Suddenly, the realities of the network become apparent. Components that are physically far apart will need to account for latency in their design. Components that are deployed on mobile devices will need to account for a lack of network reliability and constantly changing geographic positions. Overall, the system is comprised of many different networks and sub-networks which each of their own characteristics and security profiles.

The network has a big impact on how we think about the Eventful design in terms of messaging time and message size. We'll see examples of this when we dive in to the aspects of time and space later in this chapter. When you design an Eventful system it's crucial that you have an idea of how the system will be deployed and realised from a network perspective.

In a RESTful system we almost always use TCP/IP networks and the HTTP application protocol. These are both really well-suited for the synchronous, request-reply context of a RESTful interaction. But, as you start to build EVENTful systems you can start to consider other network options. For example, the UDP network protocol is optimised for broadcasting communications to many components. The MQTT application protocol is optimised for sending lots of small messages between physical devices. The

> Be careful of confusing the synchronicity of a network or application protocol with the synchronicity of your Eventful system. It's absolutely possible to deploy an event system over HTTP - just as its absolutely possible to deploy a request/reply system on an asynchronous network.

The network has a big impact on how components communicate from a transport perspective. Now, let's take a look at that thing we are sending and receiving in the network - the message.

The Message

RESTful and Eventful systems have one very important thing in common: they are oriented around message based communication. Messages are units of communication. In the RESTful world, messages are categorised as requests and responses. When we use the HTTP protocol, a message contains a representation of a server's resource. That representation may be an expression of the state of a resource at a certain point of time (e.g. the number of widgets) or a target state of a resource (e.g. a new label for a widget).

For more on representations and resources, take a look at…. TK

Eventful systems also use messages as a form of communication. But, instead of requests and responses, the messages are *events*. Unlike requests and responses which are meant to be paired, event messages exist on their own as an island. While a request is a solicitation for something to happen (e.g. "what is the current time?") an event just communicates a fact without solicitation (e.g. "at the tone, the time will be 10:35").

In practice, the message turns out to be an incredibly important part of an EVENTful design. Just like in a RESTful system, an event message contains a representation of a time-bound state. Understanding how to create, parse and use these representations is a key element of an EVENTful system design. But, surprinsingly there is very little advice on how to do that in the industry. We'll address the importance of message design later in this chapter when we talk about coupling. Later, in the book we'll cover some design techniques and good practices to help you design better event messages.

With the importance of the message established, lets move on to understanding the components that create and use messages: the producers and consumers.

Producers and Consumers

TK - getting a bit stuck here now. In a reuest-reply interaction, a requestor both produces request messages and consumes resopnse messages. But, don't want to make this so cmplicated. Need to think about it a bit more.

In system there are two types of actors: producers and message consumers. As you'd expect, message producers create (or produce) messages and message consumers use (or consumer) messages. The distinction between producers and consumers is a useful one because it denotes a dependency relationship. Message consumers will need to parse the messages that producers create. Therefore, there is a dependency on the producers to create messages that consumers will understand.

In a RESTful system, the consumer-provider relationship is fairly easy to understand. Systems that send requests are

TK intro the concept of message producers and message consumers.

Tk The REST/sync version

TK The async version and significance of this difference

Messaging Infrastructure

In the technology world, infrastructure serves as a platform to support the software that we need to build. For example, an infrastructure of computers, disk arrays and operating systems allows a team to focus on engineering software that depends on those components. An infrastructure of container

registries, Cloud services and container orchestration allows a team to write and ship software as containers. We usually establish an infrastructure platform so that our teams can reduce the scope, effort and focus of their work by taking advantage of a shared set of tools and processes.

There is almost always some form of infrastructure in the message based integration world as well. When we design a system that connects software together, we take advantage of both hardware and software tools and systems to reduce the work we need to do to communicate in a message-based way. For example, in a RESTful system, we'd typically use the HTTP application protocol, the TCP/IP network protocol and a series of ethernet and fibre-optic cables to aid communication. A good example of this kind of model is the OSI model which describes the layers of a network-based communication stack.

But, we often add additional message infrastructure within the "application" layer of the stack to make communication easier, more consistent and safer. For example, in a RESTful system, implementers often incorporate message-based routing, access control, threat protection and message translation as a core infrastructure component. This way, teams can deliver safer, dynamic API-enabled applications without implementing that logic themselves. Instead, they focus on application logic and deploy their applications behind a set of messaging infrastructure components that ensure a minimum set of entry criteria is met before a message consumer can request a response.

When it comes to an EventFUL system, there is often a need for a special set of messaging infrastructure tools.

TK Messaging infra. compare and contrast sync and async
TK highlight how important this is for an async interaction

People

Tk highilght that we need to design for people, not just systems. Catalog the types of people.

Coupling

In a software system, we say two components have a high degree of coupling when they are highly dependent on each other. High coupling means that a change to one software component will necessitate changes to another component that is dependent on it. This concept of coupling in software design has been with us for a long time. The term coupling comes from Larry Constantine and his work on structured design in the late 1960s and early 1970s. It's been a long time since Constantine first identified the costs of coupling, but years later software designers still find themselves battling to reduce coupling costs.

[TK sidenote - Larry Constantine, structured design]

That's because high degrees of coupling makes software changes more expensive and harder to execute. Coupling increases the amount of change work we need to do - a change to one component requires a team to change other dependent components to avoid breaking the system. In a complex

software system, changes to a component with high coupling can create a massive wave of change, as each component's change triggers a corresponding change in another dependent module. In practice, this becomes a change coordination nightmare as changes need to be managed across multiple teams. In systems with high degrees of coupling, the pace of change can grind to a halt.

In Pursuit of Looser Coupling

It's no wonder that software designers prioritise patterns, principles and technologies that make it easier to reduce software coupling. In fact, the goal of looser coupling between components has heavily influenced the way we design and build software. This focus has lead to object oriented programming patterns like Abstract Factory, Facade, Visitor and Iterator to reduce the code changes we need to make within our applications.

Similarly, there's been a long history of reducing coupling for API based software architectures. Corba, XML, SOAP, REST and microservices integrations have all been introduced and popularised largely because they offer the promise of "looser coupled" systems. Each new wave of an integration style brings the promise of finally solving the "coupling problem". When the circumstances are right, this leads to a wave of mass adoption and sadly it often results in discrediting of an existing integration style. This is almost always followed closely by a wave of discontentment as software designers realise that the coupling hasn't disappeared.

Within this narrative, its easy to feel like the situation is hopeless. But, don't be fooled! As an industry, we've made steady progress towards better architectural patterns that fit the technology of the day. Our ability to change software quickly has vastly improved thanks to the hard work and innovative ideas of many people. The real danger is in viewing coupling as a simple, binary property that is either "loose" or "tight". The reality is a bit more complex. In fact, in their paper on the facts of coupling, Erik Wilde and Cesare Pautasso list twelve different types of coupling that can exist, including discovery, binding, evolution and models.

As the event-driven, asynchronous style of integration gains popularity, we're hearing the refrain of a silver bullet for coupling again. Asynchronous communication is being held up as an example of an integration style that does "loose coupling" properly. There are in fact, some aspects of an Eventful style that reduce software coupling. But, if you're building an Eventful system it's important that you understand what aspects of loose coupling you get for free and which ones you'll need to work hard to get.

The Loosely Coupled Eventful System

In an API interaction, coupling is natural and unavoidable. When one component is providing a service or producing an output, other components will naturally form a dependency on it. But, the goal for most software designers is to reduce that dependency so that the components are loosely coupled. That means that a change to one of the components that provides a service or produces data should have as small an impact as possible on the other dependent components.

As we discussed in chapter [X] (Tk - callback to Mike's chapter on differint styles/patterns), Eventful architectures can take a number of different forms. We also learned that unlike synchronous interactions, Eventful communication goes in a single direction.

is an issue when a change in

////
Event-based architectures and asynchrounous APIs are often touted as being "loosely coupled". That's true to an extent, but be careful - it may not be de-coupled in a way that is useful to your needs. As we described earlier, de-coupling software components helps us reduce the amount of effort we need to spend on writing
and maintianing code. So, does a decoupled asynchronous API help us do that?
////

(TK - this should back reference thee eventful API styles that Mike is writing)
As we described in chapter [x], an asynchronous API has some unique interaction characteristics. Instead of a request-reply model, asynchronous APIs often use a publish-subscribe model, in which multiple subscribers consume asynchronous event messages. We also described the important role of infrastructure and how it further separates the event publisher from the consumer.

It's this separation of the event message publisher from the event message consumer that people often focus on. By implementing components that can publish event messages to an eventing infrastructure, we can "de-couple" the event publisher form the consumer. The developers who write the event publishing component don't need to coordinate with event consumers in order for their messages to be picked up and received. They only need to know how to send event messages into the infrastructure - the rest is up to someone else to figure out.

This is quite different from the RESTful request-reply world. In that world, an API team needs to publish a network address so that consumers can transmit messages to them. The actual network address might be purely logical. That is, the address may point to a complicated nest of API gateways, network proxies, load balancers and content-delivery networks. But, from the client's perspective this network address represents the location of the API. The client and server become bound together by that network location.

But, it turns out that this kind of network address coupling (or de-coupling) has limited value. Its true that in an event-based system, components don't need to know each other's physical network addresses. However, that location detail has very little impact on the code that our teams write. There is another kind of coupling that still exists in an asynchronous API and it drives the vast majority of work, brittleness and costs: message coupling

Message Coupling

Components need to bind at the network level in order to send messages to each other. when it comes to asynchronous APIs, that binding can often happen between a component and the infrastructure (e.g. the location of a local Kafka server). But, most of the code we write for transmission is pretty generic. So, changing the network location of a component isn't a big change burden.

However, the work of parsing, understanding and acting on the *contents* of a message is a much bigger challenge. The code we write to consume a message needs to understand the data structure of a message, the semantics (or meaning) of the data it finds and the actions it should take based on the data it has consumed. That's really the majority of the domain-specific code we end up writing. We call this message coupling and its one of the most challenging parts of an integration to de-couple.

Reducing message coupling needs to be come a primary goal for your asynchronous solution design if you are trying to reduce change costs. In fact, this is true of any software integration - if you want faster and cheaper change in a message-based integration, you'll need to focus on de-coupling your components at the messaging level. We'll describe some techniques for doing that when we dive into design in chapter.

Distance

When we design systems on paper, its easy to forget about the physical realities (and constraints) of the systems that we are building. Physical distance turns out to be a big factor when it comes to asynchronous and event based designs. With today's technology, as the distance between a message producer and message consumer grows, the time it takes for messages to be communicated will grow as well. Another side-effect of distance is that the reliability of the network decreaes as well.

Our designs need to accomodate for the distance between components. A lot of the network protocols we use today already do that for us. For example, TCP/IP is designed to be optimised for reliability of transmission between two parties rather than for low-latency communications. The TCP/IP model of communication is actually pretty well-suited for the way HTTP and RESTful APIs work. That's not really an accident. [TK need to validate and extend this point - assume TBL/Fielding designed for TCP/IP comms]

But, out asynchronous interactions aren't really well suited for the TCP/IP interaction model. As we've mentinoned a few times, today's event based models are very often pub-sub based. That requires either a specialised protocol like UDP or requires specialised middleware or infrastructure that publishers and consumers can use.

There's also another challenge that comes from using asynchronous APIs - event messages tend to be smaller. Small messages drive a special type of messaging characteristic called "chattiness" that can turn out to be a problem.

Eventful Chattiness

"Chattiness" is a collaquial way of describing the amount of messages that are needed to support a message based interaction. When a component is "chatty" it transmits lots of messages on the network. In a RESTful architecture chatiness manifested itself as multiple request-reply messages to achieve an interaction goal. In an EVENTful system, chattiness usually means that a components produces lots of messages within a short time frame.

It's difficult to avoid chattiness when you're designing an asynchronous API. That's because event-based design often means that you're publishing events as they happen. In most systems, there are lots of things happening. That means that you'll inevitably be publishing lots of events.

Normally, this isn't a big deal. In an eventful system, infrastructure is usually tuned to support lots of producers publishing lots of messages. Consumers are designed to filter messages and only respond to what they need, so the having to wade through lots of messages isn't really an issue.

But, chattiness becomes a problem when we add the distance factor. That's because [tk ... wait a second. Not sure this is true. need to think about it]

Local vs Internet

TK

Hiding distance in the infrastructure

TK kafka model (and others)

Latency (Time)

So far, we've looked at eventful systems through the lens of code changes (through coupling) and through the lens of message transmission as a factor of distance. These factors have helped us see how an eventful system impacts two software components that need to interact. But, the final characteristic we'll look at is a bit broader in nature - we need to consider how an integration architecture is impacted by the time it takes for work to be completed.

In software engineering, latency is a measure of how long something takes from the point of input to the point that output is received. For example, in a software application,

Message Latency

TK impacted by the distance and chattiness factors.

Transaction Latency

TK long-running tasks

Design Latency

TK need to remember what this is.

(runtime latency, build-time latency, design-time latency?)

What does a good Event-System look like?

TK this section ties all the concepts together from a quality perspective. Can be inspred by Fielding's chapter 2 and chapter 3

TK (mamund) one thing I really like about Fielding's system properties narrative is that he calls out how his constraints promote the desired properties. wonder if we have the same oppty here? these are the properties. are there constraints we recommend somewhere, too?

Changeability

TK - descrive changeability (modifiability, evolveability, extensbility) in therms of concepts in this chapter

Maintainability

TK - describe maintainability

Observability

Reliability and Scalability

TK - describe reliability and scalability in terms of concepts in this chapter

Usability

TK - describe usability in terms of concepts in this chapter

Comparing the Eventful and REStful systems

TK aspirational
TK maybe transalte fielding's table and do the columns for eventful

Summary

TK

Additional Reading

TK

4. Designing EVENTful systems (Mike)

(20 pgs)

Highlights

TK

Common Patterns

TK

Filtering

TK

Parsing

TK

Acting

TK

Context

TK

Design Process

TK

Uncover the Story

TK

(output: Story Doc)

Identify the Actors

TK

(output: Persona List)

Describe the Interactions

TK

(output: workflow Diagram)

Document the Data/Events

TK

(output: Domain Vocabulary)

Define the Interface

TK

(output: AsyncAPI/proto3 Docs)

Summary

TK

Additional Reading

TK

5. Customer Onboarding Example (Mike)

(15 pgs)

(would prob. kill this or try to weave in)

Highlights

TK

Design

TK

Story

TK

Actors

TK

Workflow

TK

Vocabulary

TK

Definition

TK

Implementation

TK

Kafka Storage

TK

MQTT Interface

TK

Test/Build Pipeline

TK

Summary

TK

Additional Reading

TK

6. Parting Thoughts (Ronnie)

(7 pgs)

Highlights

tK

Where to Start

TK

Dealing with “Correctness”

TK

Sorting Out the Moving Parts

TK

Reactive Architecture

TK

Consumer-First Design

TK

Real-World Constraints

TK

Summary

TK

Additional Reading

TK

Appendix A: Onboarding Example Install/Setup

(5 pgs)

Heroku

TK

NodeJS/NPM

TK

Jest Testing Tool

* https://jestjs.io/en/

Build Pipeline

TK

Appendix B: Design Assets

(10 pgs)

Onboarding Event Story

TK

Onboarding Persona List

TK

Onboarding Workflow Diagram

TK

Onboarding Domain Vocabulary

TK

Onboarding Interface Definitions

TK

Appendix C: Code Examples

(?? pgs)

Kafka Storage

TK

Onboarding Services

TK

Onboarding Clients

TK