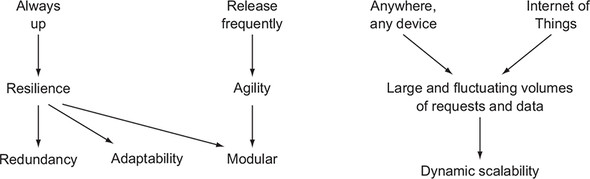
**Chapter 12. Cloud-native data: Breaking the data monolith**

*This chapter covers*

* Why every microservice needs a cache
* Using events to populate local data stores/caches
* Using messaging in event-driven systems
* The difference between messaging and events
* The event log and event sourcing

Remember how I defined *cloud-native* in [chapter 1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01)? There I did a lightweight analysis that took us from high-level requirements for modern software to a set of four characteristics: cloud-native software is redundant, adaptable, modular, and dynamically scalable ([figure 12.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig01)). And for the bulk of the book, you’ve studied these characteristics in the context of the services and interactions that make up our software. But recall that the third entity in the mental model that I also laid out in [chapter 1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01) was *data*. The characteristics of cloud-native software apply just as much to the data layer.

**Figure 12.1. User requirements for software drive our development toward cloud-native architectural and management tenets. For cloud-native data, we’ll turn our focus to modularity and the autonomy that comes from it.**



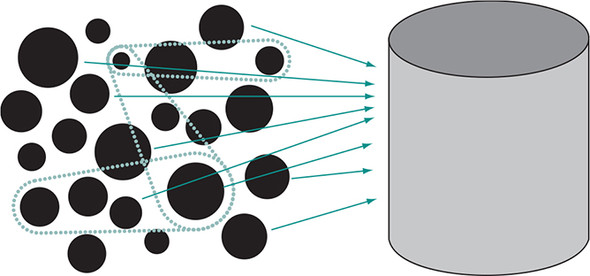
Take, for example, redundancy. Although the value of having multiple copies of data has been long understood, the patterns for achieving it in the past were often bolted on, and sometimes came through operational practices. For example, in active/passive deployment topologies, the active node is serving all read and write traffic, while the passive node is being updated with writes behind the scenes. If something happens to the active node, the whole system could fail over to what had been the passive node. Modern cloud-native data services (recall that in [section 5.4.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05lev2sec3), I talked about the special types of services that are the stateful ones) have redundancy built deeply into their multinode designs and employ patterns such as Paxos and leader/follower to drive the consistency and availability characteristics they deliver.

For scalability, we’ve also seen significant changes in the patterns that rule the day. Traditional databases were most often scaled vertically, by providing larger and larger hosts and storage devices to serve expanding needs. But as you’ve seen throughout the text, with cloud-native horizontal scalability rules, most modern databases such as Cassandra, MongoDB, and Couchbase have this model designed into the core of their systems. As data volumes increase, new nodes can be joined to the database cluster, and existing data and requests will be redistributed across all of the old and new nodes.

Although we could study a great deal about all four characteristics shown at the bottom of [figure 12.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig01), in this final chapter I place the bulk of our attention on *modularity*. Clearly, having many individual (micro)services brought together to form our software is central to cloud-native architectures. But the name *microservice* is a bit misleading (hence, I’ve used it only sparingly throughout the text). It encourages us to focus too much on the size of the service, rather than on the most valuable thing it brings: autonomy. When done right, microservices can be built by independent teams and can be independently managed (scaled). Further, they’re the primary entity against which many of the other cloud-native patterns are applied (circuit breakers, service discovery) and they’re at the endpoints of cloud-native interactions.

We used to have monolithic applications that we’ve now decomposed into many individual services, and this has given us modularity. Or has it? Those monolithic applications had at the backend monolithic databases, and all too often we see new cloud-native designs that look like [figure 12.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig02). We’ve broken the compute portion of our software into individual services but have left a centralized, monolithic data tier.

**Figure 12.2. Independent microservices that share a single database are not autonomous.**



This simple diagram makes it clear that such designs offer only the illusion of modularity; a shared database creates transitive dependencies between otherwise independent services. For example, if one service wants to change a database schema, it must coordinate with all other services that share any part of that database schema. Further, with a shared database, many individual services are all competing for concurrent access, creating additional bottlenecks. Alas, our microservices aren’t all that autonomous after all.

As the chapter title suggests, ultimately, our goal is to break up that data monolith. You may already have heard it said that every microservice should own its own database, but that might sound dangerous. How will you maintain integrity of your data across dozens or hundreds of stores? How will different teams coordinate across this complex network? Just as breaking the compute monolith apart brought new challenges—challenges we’ve systematically addressed with the set of patterns presented throughout this text—breaking the data monolith also brings challenges. And those challenges will be addressed with patterns—cloud-native data patterns.

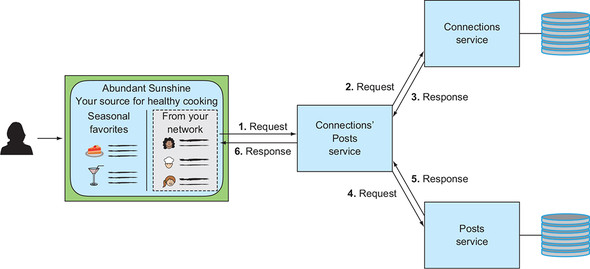
Many in the industry believe that event sourcing is the ultimate answer to these problems, and this chapter covers that topic. But just as with all of the other patterns covered in this book, it’s not all or nothing. I would like to take us on a journey that starts with basic, familiar design patterns and work our way up to event sourcing. My hope is that presenting the topic in this manner will not only deepen your understanding of the key elements of cloud-native data handling, but also provide a practical way for you to get there in steps.

I’ll start with caching, a technique that has been in use for some time and remains relevant in cloud-native software architectures; remember what happened when we added a cache as part of our fallback behaviors when studying resiliency patterns? We’ll have another look at caching here. Then I’ll do a brief review of [chapter 4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04), where we turned request/response on its head and sent events flowing through our services topology instead. I’ll enhance that event-driven design by adding an event log, and then I’ll finally introduce the concept of event sourcing. As we go through this design evolution, I’ll use an analysis of request successes and failures under different outage conditions as a measure of autonomy, and you’ll see the important role that data design plays in our cloud-native architectures.

**12.1. Every microservice needs a cache**

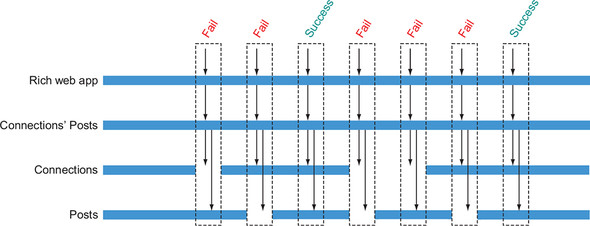
To see the value that caching brings—and it goes beyond just performance—let’s start with a design that doesn’t yet use a cache. [Figure 12.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig03) shows our usual example, in which the Connections’ Posts service aggregates content from our two other services: Connections, which manages users and who they follow, and Posts, which manages blog posts. In this diagram, we’re using the familiar request/response protocol to interact between these services.

**Figure 12.3. A simple topology with an aggregating service communicating with dependent services via request/response.**



Using this simple architecture as a baseline for the analysis that runs through the rest of the chapter, let’s look at the system’s resiliency in the face of partial outages. [Figure 12.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig04) shows the four compute components of our software: the rich web application and the three services depicted with the horizontal lines. When solid, these lines represent a time that the service is available and producing results, and breaks in those lines indicate when the service is unavailable or not functioning properly.

**Figure 12.4. Whenever one of the dependent services is unavailable or otherwise not functioning correctly, the Connections’ Posts service will also fail to generate a result.**

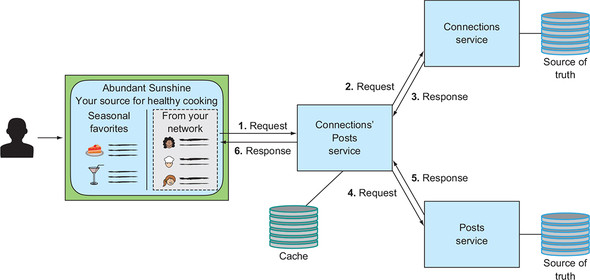


You’ll note that I haven’t depicted any downtime for the rich web application or the Connections’ Posts service. This is not to imply that they never go down. Rather, I want to focus only on the dependencies between microservices and with the request/response protocol; those interactions exist only when the aggregating service is being called and is functioning.

Coming in vertically are requests from the web application to the Connections’ Posts service, and then cascading to the dependent services. Whenever either the Connections service or the Posts service is down, the aggregating service fails to generate a result.

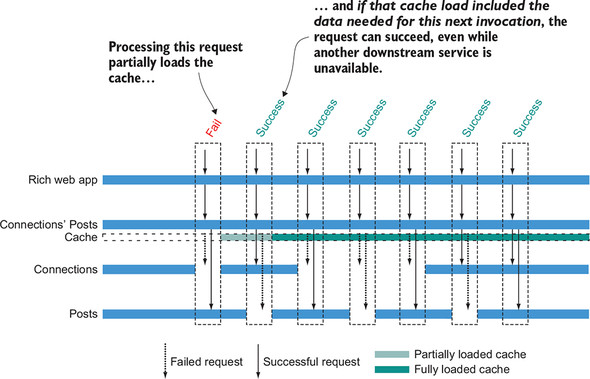
Now let’s add a cache to the Connections’ Posts service ([figure 12.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig05)). This cache may be populated in numerous ways. With *look-aside caching*, the cache client (in this case, the Connections’ Posts service) is responsible for implementing the caching protocol: when data is needed, it will check the cache for a value, and if it’s not present, will make a request to the downstream service and write the result into the cache before itself returning a result. With *read-through caching*, the Connections’ Posts service accesses only the cache, and the cache implements the logic of obtaining the value from the downstream service when necessary. Regardless of the protocol, a value is stored locally following a successful downstream request.

**Figure 12.5. A cache added to the Connections’ Posts service will store the responses from successful requests to dependent services. The source of truth for the data remains in the databases tied to each of the downstream services; the storage local to the Connections’ Posts service is not an authoritative copy.**



Let’s now look at whether our system has become more resilient with this addition. [Figure 12.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig06) adds another horizontal bar alongside that of the Connections’ Posts service. To simplify things a bit, assume that the cache is available anytime the Connections’ Posts service is. We can see that initially the request results are exactly what we previously saw: when either of the downstream services is unavailable, the aggregating service will be unable to generate a full response. But after the cache is populated, it insulates the consumer, the web app, from failures of the downstream services.

**Figure 12.6. By adding a cache to Connections’ Posts, you add a degree of autonomy to the service and achieve greater resilience as a result.**



I want to draw your attention to one subtlety in [figure 12.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig06). Compared to the scenario depicted in [figure 12.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig04), notice that you can start seeing success on incoming requests as soon as each downstream service has been reached at least once, and that successful downstream requests needn’t all come at the same time. It should be clear that the level of resilience is directly tied to our system components’ degree of autonomy.

That’s all pretty compelling, and clearly you achieved far greater levels of resilience when you made Connections’ Posts more autonomous by adding a cache. Why isn’t this enough? The answer is that everything I’ve shown here is a significant oversimplification of caching. You can see a hint of that in [figure 12.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig06), where I make the point that the second request *could* succeed *if* the first request happened to load the data needed for the second one. How do you know, for example, if a lack of any cached posts for Food52 means there are no new posts on that site, or that a successful Posts request including that site has yet to happen? And if the cache has an entry, how do you know whether it’s up to date?

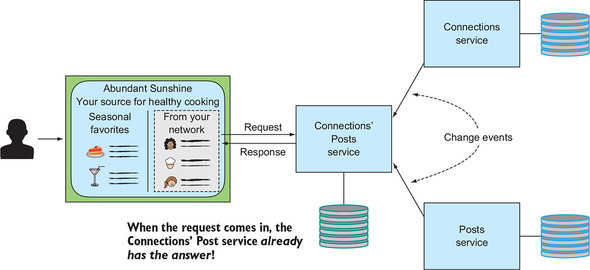
The way we’ve historically used caching, and even the motivating factors for using it, were different from our present aims for its use in microservice architectures. Often used to achieve a certain level of performance gain, the data we cached tended to be more static (website images and zip-code-to-city mappings, for example), so expiring the cache based on a timer was usually sufficient. A cache miss was a consistent indicator that it hadn’t yet been loaded, and processes were often put in place to warm or preload the cache when an app using it was started. Trying to use a cache in these microservice-centric scenarios requires that we rethink the patterns, and front and center is the concern around cache freshness. Ideally, whenever something changes in a downstream service, we’d like to have those changes reflected in our local data store as soon as possible.

Aha! You see where I’m going with this?

**12.2. Moving from request/response to event driven**

As I suggested, we began our journey to this better way of handling local storage in [chapter 4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04), when we turned request/response on its head, moving to an event-driven interaction protocol. [Figure 12.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig07) looks similar to the diagrams earlier in the chapter, but with one important distinction: the interaction between Connections’ Posts and its related services is initiated from the opposite end. Connections’ Posts does have its own local storage, reminiscent of the cache from the previous example, but it’s now updated anytime one of the downstream services broadcasts a change.

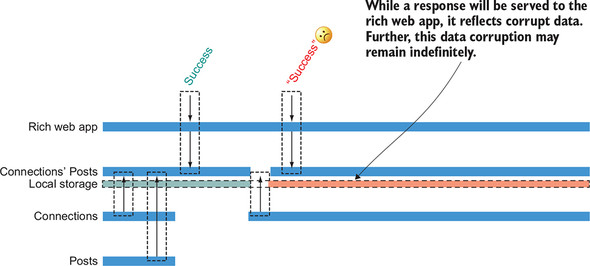
**Figure 12.7. When changes occur in downstream services, events describing those changes are sent to interested parties where their local storage is updated.**



Gone is the need to worry about cache expiry. Gone is the need to wonder whether a lack of data truly means that such data doesn’t exist. Assuming that the mechanism for delivery of downstream events is functioning (and you’ll see shortly how we assure this), the Connections’ Posts service can operate in its own bounded context and not worry about what’s happening elsewhere in the system. This is indeed a thing of beauty!

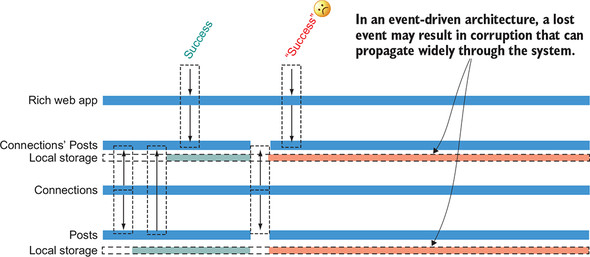
Let’s look at what this does to our system’s resilience. [Figure 12.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig08) updates our earlier diagrams. You can now see that the events coming from Posts and Connections are sent to Connections’ Posts, and when a request comes in from the web page, whether the dependent services are up or down doesn’t matter; Connections’ Posts has its own local storage and is operating autonomously. But what happens if Connections’ Posts is unavailable when one of the other services has an event to deliver? As implemented in [chapter 4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04), that event would be lost. Sure, we can use some of the other patterns we’ve studied, such as retries, to compensate for some of the failures, but in some cases those won’t work. The net result is bad.

**Figure 12.8. The event-driven approach allows the Connections’ Posts service to operate with complete autonomy, using the data in the local storage. Missed events, however, can corrupt that local storage, “successfully” returning incorrect results.**



To Connections’ Posts, everything seems right in the world. It will return a result based on the data in its local storage, exactly as it should. It has no idea that the data is now not accurate. And it gets worse. It’s not usually the case that an event is of interest to only a single party; instead, many other entities may care if a user in the system changes their username, for example. If that change event doesn’t reach many intended participants, inconsistencies can propagate widely through your system, as shown in [figure 12.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig09).

**Figure 12.9. Events are usually of interest to numerous parties, so when they’re not properly delivered, this can cause inconsistencies to propagate throughout the system.**



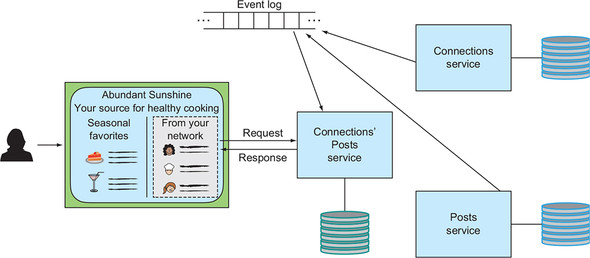
In essence, local stores, possibly many of them, are now corrupt. Even worse, they may remain so indefinitely! The event-driven approach promised to eliminate the question of cache freshness, but it has failed.

Remember that comment about the mechanism for delivering downstream events being sound? Clearly, we don’t yet have a sound system. So, let’s continue.

**12.3. The event log**

You have to eliminate the need for Connections’ Posts to be available when an event it cares about is being sent. Yes, you guessed it—you’ll use some type of asynchronous messaging system. Instead of Connections sending events directly to Connections’ Posts, it will deliver those events to a system that’s responsible for the delivery thereof. Of course, your software will then depend on that messaging system being available, but concentrating messaging semantics within a system specially designed for that purpose not only allows for consistent implementation of the patterns, but also allows you to focus resiliency efforts in the messaging fabric rather than across the vast network of interrelated services. If the messaging system is itself designed in a cloud-native way, with redundancy and dynamic scaling, it will be dependable. [Figure 12.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig10) shows the event log added to your software architecture.

**Figure 12.10. With the addition of an event log, you’ve now completely decoupled each of the three services from one another. Events are produced into and consumed from the event log, and the event log is responsible for retaining the events until they’re no longer needed.**

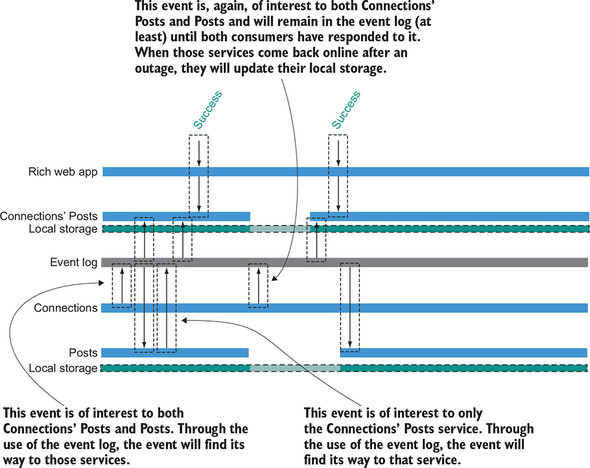


Let’s see how this impacts the resilience of the software as a whole. [Figure 12.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig11) adds another component: the event log. As the Connections and Posts services generate events, they’re sent to the log, and the parties interested in the event will pick them up and process them. Notice that the first message, sent from the Connections service, is picked up by both the Posts and Connections’ Posts services. The second event, however, generated by the Posts service, makes its way only to Connections’ Posts. And an event that’s produced when consumers are unable to immediately respond is maintained in the event log, (at least)**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fn1)**]** until all interested parties have consumed it, even if those consumers come online at vastly different times. As a result, we can see that the availability of the aggregating service is strong even while outages are occurring throughout the system.

***1***

*Stay tuned—I’ll address this phrase, “at least,” when I talk about event sourcing.*

**Figure 12.11. Events are produced into and consumed from the event log, which maintains those events as long as they’re needed. Through use of the event log, you’ve entirely decoupled the services from one another.**



One note on consistency: Previously, you saw a case where, because of a missed event, your cache could become corrupt. Arguably, even here, in the time between the production of an event and the consumption thereof, the local storage of the consumer is out of date. When I talk about event sourcing later in this chapter, I will address this concern.

|  |
| --- |
|  |

**Connecting to a messaging fabric usually happens through a client library**

Note that producing and consuming messages from code is usually done with some type of client library, and that library will often implement some of the resiliency patterns you’ve been studying. For example, if the event log can’t be reached with a first request, the client library may retry. And if the connectivity continues to fail, the library code may invoke a service-discovery protocol to find alternative endpoints to contact.

Developers needn’t be concerned with the details of the protocol. Instead, they need only specify the type of service level they need from the interaction. For example, if the implementation requires guaranteed delivery of events (the software must report an error if the message can’t be delivered), that will be specified through the client API. If, on the other hand, the implementation can tolerate some event loss, that too can be specified.

|  |
| --- |
|  |

It’s high time we looked at some code, don’t you think?

**12.3.1. Let’s see this in action: Implementing event-driven microservices**

The implementation for this chapter builds on that of [chapter 4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04), where I had challenged our natural inclination toward request/response, instead providing an event-driven solution. But there you still had the tight coupling between the services, as described in [section 12.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12lev1sec2), where the client sent events directly to consumers. [Figure 12.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig07) depicts that implementation.

What I do here is add the event log, thereby more loosely coupling the services from one another. I’m basically headed in the direction of the depiction in [figure 12.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig10), but when we get into the details, the example is slightly more complicated. Not only is the Connections’ Posts service interested in the generated events, but so too is the Posts service. This slightly more sophisticated topology allows us to explore details in a more comprehensive way.

The Posts service has the responsibility for managing blog posts—allowing for new posts to be created and for publishing events when those new posts come in. The posts it stores include the title, body, and date of the post, as well as the ID for the user who made the post. In our system, the Connections service is responsible for managing users, as well as the connections between them. In order to allow a user to change their details, username, or given name, for example, queries in your code will refer to that user only by ID. But the user ID is an implementation detail, an internal identifier that should not be leaked out through any API. Therefore, through the API, you refer to users via their username. For example, when adding a new post, you send a POST request to the Posts service with a payload that looks something like this:

{

"username":"madmax",

"title":"I love pho",

"body":"Yesterday I made my mom a beef pho that was very close to what I

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg ate in Vietnam earlier this year ..."

}

A user with the username madmax has published a post about the pho he recently cooked. When you store this in the database, however, you don’t want to store the username, because if Max later changes it, you’ll have difficulty finding his older posts. As such, you store the post with his user ID, as shown in the following output of a SQL query:

mysql> select \* from cookbookposts.post;

+----+----------------------+---------------------+-------------+---------+

| id | body | date | title | user\_id |

+----+----------------------+---------------------+-------------+---------+

| 1 | Yesterday I made... | 2018-10-30 11:56:05 | I love pho | 2 |

...

Because of this level of indirection, the Posts service needs to map user IDs to usernames. The Posts service doesn’t manage users (the Connections service does), but in order to remain decoupled from the Connections service, it must keep in its own database an up-to-date copy of some of the data managed by the Connections service; namely, the correlation of ID to username. You can see that table through the following SQL query:

mysql> select \* from cookbookposts.user;

+----+-----------+

| id | username |

+----+-----------+

| 1 | cdavisafc |

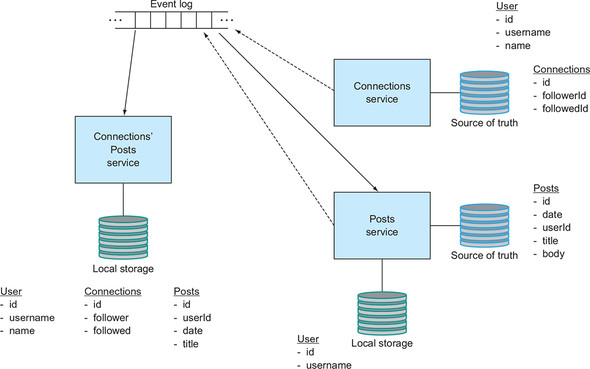
| 2 | madmax |

| 3 | gmaxdavis |

+----+-----------+

Okay, so in [figure 12.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig12), both the Connections and the Posts services are generating events, and the Posts and the Connections’ Posts services are consuming events. In that diagram, you can also see the data stores associated with each of your services. The Connections and Posts services each own data that’s stored in a source-of-truth database, and the Posts and Connections’ Posts services store copies of data owned by another service in local storage databases. These local databases are kept up to date through eventing.

**Figure 12.12. The full event and data topology of our implementation. The Posts and Connections services both “own” data (source of truth) and produce events (dotted arrows) when changes occur in that data. Posts and Connections’ Posts both store data they don’t own (local storage) and update that data by handling events (solid arrows).**



[Table 12.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12table01) sums up the roles that each service plays in the overall software architecture.

**Table 12.1. Service roles in our event-driven software architecture**

| Role | Connections service | Posts service | Connections’ Posts service |
| --- | --- | --- | --- |
| *Source of truth*—A dedicated database to store data owned by the service. | This service owns users and connections between users. | This service owns posts. |  |
| *Write and event producer*—HTTP endpoint implementations update source-of-truth databases and deliver events to Kafka. | Generates user event anytime a user is created, updated, or deleted. Generates connection event anytime a connection is created or deleted. | Generates a post event anytime a post is created. |  |
| *Event handler*—Subscribes to events and updates local storage databases accordingly. |  | Updates local storage databases when users are created, updated, or deleted. | Updates local storage when users, connections, and posts are created, updated, or deleted. |
| *Local storage*—A dedicated database to store data not owned by this service. |  | This service stores mappings of username to user ID. | This service stores user and connection data as well as post summaries. |
| *Read*—HTTP endpoint implementations that serve the domain entities for the service. | Users. Connections. | Posts. | Posts made by users followed by an individual. |

Turning now to the project, you can see the preceding details reflected in the directory and package structure. So that you can look at this yourself, let me get you set up with that project.

Assuming you’ve already cloned the repository, check out the following tag:

git checkout eventlog/0.0.1

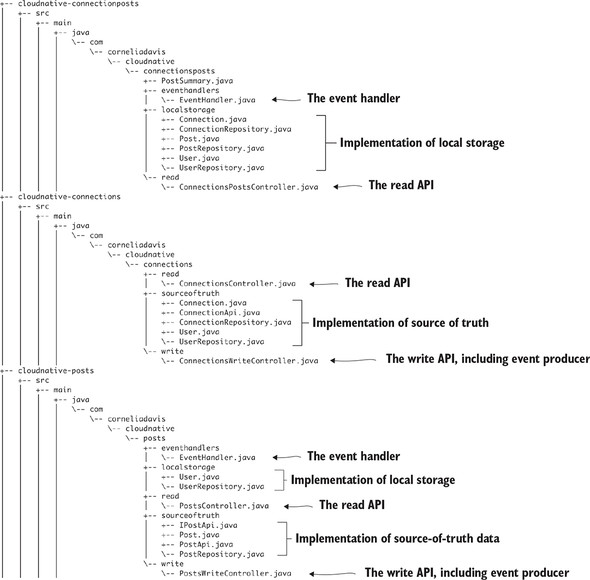
You can now change into the directory for this chapter:

cd cloudnative-eventlog

[Figure 12.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig13) shows the key parts of the directory structure of the project, as described in the preceding table:

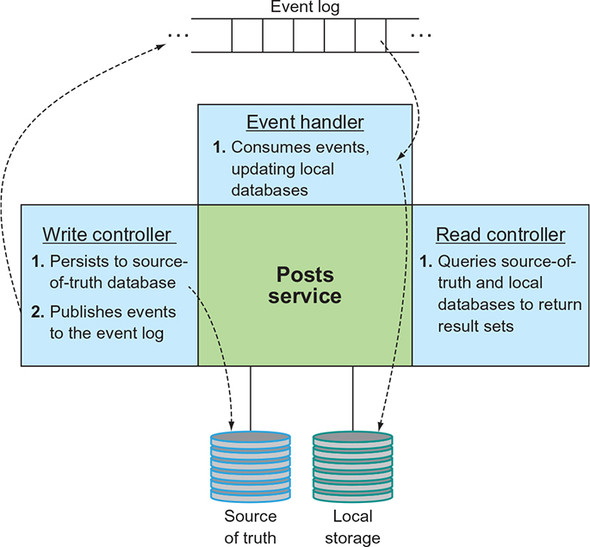
* Read APIs are implemented as controllers that are located in packages with the suffix read. All three services support read APIs.
* Any source-of-truth data is implemented with JPA classes located in a package with the suffix sourceoftruth. This package exists for both the Posts and Connections services.
* Write APIs, supporting HTTP methods such as POST, PUT, and DELETE, are implemented as controllers that are located in packages with the suffix write. In addition to persisting data into the source-of-truth database for the service, these write controllers produce events into our messaging fabric. This package exists for both the Posts and Connections services.
* Event consumers are implemented within packages with the suffix eventhandlers. This package exists for both the Posts and the Connections’ Posts services.
* The local storage for data obtained through events is implemented with JPA classes located in a package with the suffix localstorage. This package exists for both the Posts and Connections’ Posts services.

**Figure 12.13. The directory structure for our three microservices, showing the organization of event-producing and -consuming code, as well as read APIs and both “owned” data storage and “local” data storage.**



[Figure 12.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig14) offers another way to view microservices structured in this way—using the Posts service as an example. The post data (date, title, body, and author ID) is owned by the service and stored in the source-of-truth database. The user-ID-to-username mapping data is only a copy of data owned by another service and is stored in the local storage database. We have a write controller that performs two steps; it both persists source-of-truth data and delivers events into the event log. The event handler performs a single task that consumes events of interest and persists data into the local storage database. And finally, the read controller performs a single task that queries both databases to generate lists of posts.

**Figure 12.14. A service implements a write controller that’s responsible for taking changes to the first-class entities of the service, an event handler that’s responsible for maintaining a local database of nonprimary entity data that’s nevertheless part of the bounded context, and a read controller that supports querying data across these databases.**



With an understanding of this structure established, let’s now take a closer look at a few parts of the implementation, starting with the write controller for the Connections service. The code in the following listing executes when a new user is created.

**Listing 12.1. Method from ConnectionsWriteController.java**

@RequestMapping(method = RequestMethod.POST, value="/users")

public void newUser(@RequestBody User newUser,

HttpServletResponse response) {

logger.info("Have a new user with username " + newUser.getUsername());

// persist this user in our DB

userRepository.save(newUser); ***1***

// send event to Kafka

UserEvent userEvent =

new **UserEvent**("created",

newUser.getId(),

newUser.getName(),

newUser.getUsername()); ***2***

kafkaTemplate.send("user", userEvent);

}

* ***1* Stores data into source-of-truth DB**
* ***2* Delivers change event**

First, the new user is stored in your source-of-truth database, and then an event is sent to the event log (you can see from the comment that you’re using Kafka—I’ll talk a bit more about Kafka when we get to running the example). You send out a user event that captures that a user has been created with a particular ID, name, and username, and you deliver that event to a topic called user. I’ll say more about topics (and queues) in just a moment, but for now just think of these as the topics you’re familiar with from your prior message-driven implementations. A *topic* is simply a channel onto which messages are delivered and from which messages are consumed.

Let’s compare this to code from the event-driven implementation of [chapter 4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04) in [listing 12.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12ex02) (you can find this code in this cloudnative-eventlog module by looking at the repo tag eventlogstart).

**Listing 12.2. Method from ConnectionsWriteController.java**

@RequestMapping(method = RequestMethod.POST, value="/users")

public void newUser(@RequestBody User newUser,

HttpServletResponse response) {

logger.info("Have a new user with username " + newUser.getUsername());

// persist this user in our DB

userRepository.save(newUser); ***1***

// let interested parties know about this new user

// posts needs to be notified of new users

try {

RestTemplate restTemplate = new RestTemplate(); ***2***

restTemplate.postForEntity(postsControllerUrl+"/users",

newUser, String.class);

} catch (Exception e) {

// for now, do nothing – when we add the event log this known bad

// will go away

logger.info("problem sending change event to Posts");

}

// connections posts needs to be notified of new users

try {

RestTemplate restTemplate = new RestTemplate(); ***3***

restTemplate.postForEntity(connectionsPostsControllerUrl+"/users",

newUser, String.class);

} catch (Exception e) {

// for now, do nothing – when we add the event log this known bad

// will go away

logger.info("problem sending change event to ConnsPosts");

}

}

* ***1* Persists the data**
* ***2* Sends event to Posts service**
* ***3* Sends event to Connections’ Posts service**

A snippet of this code shows that after saving the user in the database, events were delivered both to the Posts and the Connections’ Posts services. With the introduction of an event log, you were able to remove this coupling—the event producer needn’t know anything about the consumers—and the code is also obviously far simpler and therefore far more maintainable.

Let’s now turn to an implementation of an event consumer. The next listing shows a portion of the event handler code for the Connections’ Posts service.

**Listing 12.3. Methods from EventHandler.java**

@KafkaListener(**topics="user"**,

groupId = "connectionspostsconsumer",

containerFactory = "kafkaListenerContainerFactory")

public void userEvent(**UserEvent** userEvent) {

logger.info("Posts UserEvent Handler processing - event: " +

userEvent.getEventType());

if (userEvent.getEventType().equals("created")) {

**// make event handler idempotent.**

**// If user already exists, do nothing**

User existingUser

= userRepository.findByUsername(userEvent.getUsername());

if (existingUser == null) {

// store record in local storage

User user = new User(userEvent.getId(), userEvent.getName(),

userEvent.getUsername());

userRepository.save(user);

logger.info("New user cached in local storage " +

user.getUsername());

userRepository.save(new User(userEvent.getId(),

userEvent.getName(),

userEvent.getUsername()));

} else

logger.info("Already existing user not cached again id " +

userEvent.getId());

} else if (userEvent.getEventType().equals("updated")) {

// ... handle updated event

}

}

@KafkaListener(**topics="connection"**,

groupId = "connectionspostsconsumer",

containerFactory = "kafkaListenerContainerFactory")

public void connectionEvent(**ConnectionEvent** connectionEvent) {

// ... handle changes to connections – who follows who.

// it is created and deleted events

}

@KafkaListener(**topics="post"**,

groupId = "connectionspostsconsumer",

containerFactory = "kafkaListenerContainerFactory")

public void postEvent(**PostEvent** postEvent) {

// ... handle changes to posts – that is, new posts.

}

There are several interesting things in this implementation:

* You can see from this, and the previous code showing the event producer, that the details of interfacing with the event log are abstracted away from the developer. Including the Spring Kafka dependency in the POM file draws in a Kafka client, allowing the developer to use a simple API to designate topics and other details so that events are easily delivered and consumed.
* The Connections’ Posts service event handler consumes messages from three topics. These topics organize change events for users, connections, and posts. Although the producer simply provided a topic name, consumers must also provide a groupId that controls how messages are consumed.
* From both this and the producer code previously shown, you can see references to event schema: UserEvent, ConnectionEvent, and PostEvent. Events published to and consumed from an event log must have a format, and both the producer and consumer must know the details.
* You can see a comment in this code about making the consumer logic for the user-created event idempotent. In a distributed system, guaranteeing exactly once delivery of events can be complex and expensive from a performance standpoint. Making services idempotent relieves some of this burden.

We will drill into each of these topics in just a moment, but first let’s run the code.

**Setting up**

One last time, I refer you to the setup instructions for running the samples in earlier chapters in this text; there are no new requirements for running the sample in this chapter.

You’ll be accessing files in the cloudnative-eventlog directory, so in your terminal window change into that directory.

And as I’ve described in previous chapters, I prebuilt Docker images and made them available in Docker Hub. If you want to build the Java source and Docker images and push them to your own image repository, I refer you to earlier chapters (the most detailed instructions are in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05)).

**Running the application**

This example can be run on a small Kubernetes cluster, so if you’d like, you may use Minikube or something similar. If you still have the examples running from the previous chapter, get those cleaned up; run the script I’ve provided as follows:

./deleteDeploymentComplete.sh all

Running this deletes all instances of the Posts, Connections, and Connections’ Posts services, as well as any MySQL, Redis, or SCCS services that are running. If you have other apps and services running in your Kubernetes cluster, you may want to clear some of those out too; just make sure you have enough capacity. In earlier chapters, deleting the services was optional, but in this case, I encourage you to delete all of them. Some of them are not used here (SCCS), and I’d like you to use a fresh MySQL instance because the database topology is fairly different.

There is a slight start-order dependency in that after creating the MySQL server, you need to create the actual databases therein. So get it and the event log running first:

./deployServices.sh

Once the MySQL database is up and running, which you can see by running kubectl get all, you’ll create the databases using the mysql CLI as follows:

mysql -h $(minikube service mysql-svc --format "{{.IP}}")

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg -P $(minikube service mysql-svc --format "{{.Port}}") -u root -p

The password is password, and after you’re in, you’ll run the following three commands:

create database cookbookconnectionsposts;

create database cookbookposts;

create database cookbookconnections;

Notice that you’re now creating databases for each of the services; until now, you always created only a single database. That means that you were implementing exactly the design depicted in [figure 12.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig02)!

Now you can start the microservices, which you do by running the following script:

./deployApps.sh

Everything is now up and running, and you can load data. But hang on. If you’ve been running the examples all along, you might note that I’ve never asked to you load data before. Why now?

The reason is that you’ve broken the database apart and have both source-of-truth databases and local storage databases, and I want you to use the very eventing that you’ve built into your services to load data into all of these data stores. And I want you to watch what happens when this data is loaded, so please stream the logs from each of your microservices, one each in three different terminal windows, with the following command (remember, you can get your pod names with a kubectl get all):

kubectl logs -f po/<name of your pod instance>

If you look at the end of the log for your Posts service, you’ll see a line that includes this:

o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[user-0]

And if you look at the end of the log for your Connections’ Posts service, you’ll see lines that include these:

o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[post-0]

o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[connection-0]

o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[user-0]

This shows that the Posts service is listening for events on the user topic, and the Connections’ Posts service is listening for events on each of the user, connection, and post topics. Ah, you see our eventing topology coming to life! Now load some data and see that flow in action. Run the following command:

./loadData.sh

If you look in that file, you’ll see that I created same sample data that you’ve been using all along. Look again at the logs.

The log for the Connections service shows exactly the entries you expect—creation of three users and connections between them:

...ConnectionsWriteController : Have a new user with username madmax

...ConnectionsWriteController : Have a new user with username gmaxdavis

...ConnectionsWriteController : Have a new connection: madmax is

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg following cdavisafc

...ConnectionsWriteController : Have a new connection: cdavisafc is

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg following madmax

...ConnectionsWriteController : Have a new connection: cdavisafc is

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg following gmaxdavis

The log output of the Posts service is a bit more interesting; you see the listener being invoked to handle three user-created events and to create three users in your local user storage:

...EventHandler : Posts UserEvent Handler processing - event: created

...EventHandler : New user cached in local storage cdavisafc

...EventHandler : Posts UserEvent Handler processing - event: created

...EventHandler : New user cached in local storage madmax

...EventHandler : Posts UserEvent Handler processing - event: created

...EventHandler : New user cached in local storage gmaxdavis

Later in the same log, you see messages that posts are being created:

...PostsWriteController : Have a new post with title Cornelia Title

...PostsWriteController : find by username output

...PostsWriteController : user username = cdavisafc id = 1

...PostsWriteController : Have a new post with title Cornelia Title2

...PostsWriteController : find by username output

...PostsWriteController : user username = cdavisafc id = 1

...PostsWriteController : Have a new post with title Glen Title

...PostsWriteController : find by username output

...PostsWriteController : user username = gmaxdavis id = 3

In the implementation for the Posts writer, I’ve included a log message that shows the lookup of the user ID from the username, the latter of which is supplied as a part of the post; of course, this lookup is happening against our local user storage database.

Finally, looking at the logs for the Connections’ Posts service, you can see what I hope you have now come to expect—messages showing the user, connection, and post events being processed and data being stored in your local databases:

...EventHandler. : Posts UserEvent Handler processing - event: created

...EventHandler. : New user cached in local storage cdavisafc

...EventHandler. : Posts UserEvent Handler processing - event: created

...EventHandler. : New user cached in local storage madmax

...EventHandler. : Posts UserEvent Handler processing - event: created

...EventHandler. : New user cached in local storage gmaxdavis

...EventHandler. : Creating a new connection in the cache:2 is following 1

...EventHandler. : Creating a new connection in the cache:1 is following 2

...EventHandler. : Creating a new connection in the cache:1 is following 3

...EventHandler. : Creating a new post in the cache with title Max Title

...EventHandler. : Creating a new post in the cache with title Cornelia

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg Title

...EventHandler. : Creating a new post in the cache with title Cornelia

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg Title2

...EventHandler. : Creating a new post in the cache with title Glen Title

...EventHandler. : Posts UserEvent Handler processing - event: created

Let’s run a few more commands to watch them in action.

When you now ask Connections’ Posts for the posts of people I follow

curl $(minikube service \

--url connectionsposts-svc)/connectionsposts/cdavisafc | jq

you see the following result:

[

{

"date": "2019-01-22T01:06:19.895+0000",

"title": "Chicken Pho",

"usersName": "Max"

},

{

"date": "2019-01-22T01:06:19.985+0000",

"title": "French Press Lattes",

"usersName": "Glen"

}

]

Let’s now change a username. With the following, change the username for cdavisafc to cdavisafcupdated:

curl -X PUT -H "Content-Type:application/json" \

--data '{"name":"Cornelia","username":"cdavisafcupdated"}' \

$(minikube service --url connections-svc)/users/cdavisafc

One more time, let’s have a look at the logs. First, notice that each of the Posts and Connections’ Posts logs has processed that event. Each log shows an entry such as this:

...: Updating user cached in local storage with username cdavisafcupdated

Now requests both to the Connections’ Posts service and the Posts service (recall that all three services implement read controllers) reflect that change:

curl $(minikube service --url \

connectionsposts-svc)/connectionsPosts/cdavisafc

curl $(minikube service --url \

connectionsposts-svc)/connectionsPosts/cdavisafcupdated

curl $(minikube service --url posts-svc)/posts

I encourage you to explore this event flow topology by issuing more commands, watching the logs, and seeing what the results yield. I’ll remind you that Posts and Connections also have read APIs, including endpoints that allow you to see each of the collections of objects they control:

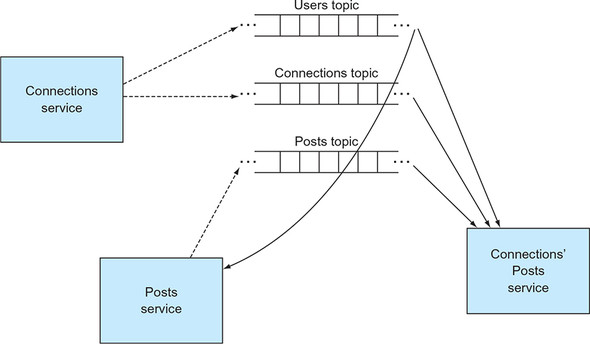
curl $(minikube service --url connections-svc)/users

curl $(minikube service --url connections-svc)/connections

curl $(minikube service --url posts-svc)/posts

[Figure 12.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig15) shows the event topology in a bit more detail. You can see the various producers, consumers, and topics, and how they relate to one another.

**Figure 12.15. The event topology defines the topics that will hold events and the producers and consumers of each. The Connections service produces users and connection events. The Posts service produces post events and consumes user events. The Connections’ Posts service consumes user, connection, and post events.**



One final comment before I move on to three more event-log architectural topics: You might have noticed the sleep command in the loadData.sh script. Yes, it’s a bit hacky, but it’s in there to avoid a race condition whereby the curl command to create a post reaches the Posts service before the events for creating the user have been processed by that Posts service. You can handle this scenario in a less hacky way, such as a simple retry from within the code, sending the event into a “retry” topic to attempt the processing again at a later time, or even failing to create the post. Analyzing these options and choosing the right one is an interesting topic for sure but is beyond the scope of this chapter. Have a look at the reference I point you to at the end of this chapter.

Okay, now go back to the three topics that I queued up for further discussion:

* Different types of messaging channels and their applicability in cloud-native software
* The event payload
* The value of idempotent services

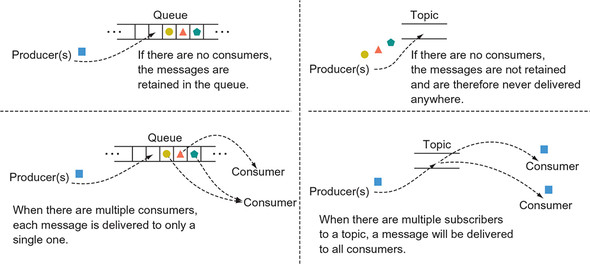
**12.3.2. What’s new with topics and queues?**

Those of you who have worked with or are familiar with the basics of the messaging systems of the past, very possibly JMS (Java Messaging Service) systems, will know about queues and topics. But in case it’s been some time ago, let me jog your memory. Both abstractions have publishers and subscribers, the actors that produce messages to and consume messages from a named channel (the topic or queue), but the way messages are handled, both by the messaging fabric and by the consumers, differs.

For *queues*, which can have multiple subscribers, a single message will be processed by only *one* of the subscribers. Further, if no subscribers are available at the time a message appears in the channel, the message will be retained until a subscriber consumes it. After it has been consumed, the message disappears.

For *topics*, which can also have multiple subscribers, a single message will be sent to *all* subscribers at the time it’s produced. The message will be processed by any number of consumers, even zero. Consumers will receive messages only if they’re attached to the topic at the time the message is produced. If a consumer isn’t connected to the topic when the message is produced, it will never see it. If there are no subscribers on a topic, messages delivered there will simply disappear. You can think of the topic as providing more of a message-routing function. [Figure 12.16](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig16) depicts both queues and topics.

**Figure 12.16. Messaging systems, largely standardized with JMS, provide support for queues and topics. Queues retain messages until each one is consumed. Topics don’t retain any messages, instead simply delivering them to all available subscribers.**



Although the newer *event log* shares some of the same concepts as those from the JMS days (brokers, producers, consumers, and even topics), the semantics around some of these are subtly different and can result in vastly different behaviors. From a developer perspective, the broker looks essentially the same as it did before; it’s the conduit to having producers and consumers connect to an event log. The implementation of a modern event-logging system such as Apache Kafka, however, is generally more cloud-native, allowing for brokers to come in and out of existence as the infrastructure changes and/or the event log cluster is scaled up and down. The role of a producer is also much the same as before; it will connect to the event log via a broker and deliver events.

It’s when it comes to topics and consumption from those topics that things change. Consumers indicate an interest in events that are published to a particular topic. But the way events are picked up when multiple consumers exist is where it gets interesting. As you’ve seen in abundance, in our cloud-native architectures, we will have many different microservices, each of which will have multiple instances deployed. The consumption patterns of the event log are optimized for this use case.

|  |
| --- |
|  |

**Note**

What we need is a semantic that allows for different microservices to be able to consume the same event, but have only one instance of each process the event.

|  |
| --- |
|  |

With Kafka, this semantic is supported with two abstractions: the *topic* and the *group ID*. Any microservice that has an interest in a particular event will create a listener for the topic that carries the event, and any new event will be delivered to *all* listeners—all microservices. The groupId is used to ensure that only one instance of a particular microservice receives a message; it will be delivered to only one of all instances sharing the same groupId.

The easiest way to understand this is in the context of our sample application.

Recall that events that capture the creation of a new user, or changes to an existing one, need to be consumed by both the Posts service and the Connections’ Posts service. Therefore, both the Posts and the Connections’ Posts services will listen on the user topic. But then, because you want only one instance of the Posts service, and only one instance of the Connections’ Posts service to process an event, you set the groupId uniformly over all Posts instances and set the groupId uniformly over all Connections’ Posts instances as shown here—from the Posts event handler:

@KafkaListener(topics="user", groupId="postsconsumer")

public void listenForUser(UserEvent userEvent) {

...

}

and from the Connections’ Posts event handler:

@KafkaListener(topics="user", groupId = "connectionspostsconsumer",

containerFactory = "kafkaListenerContainerFactory")

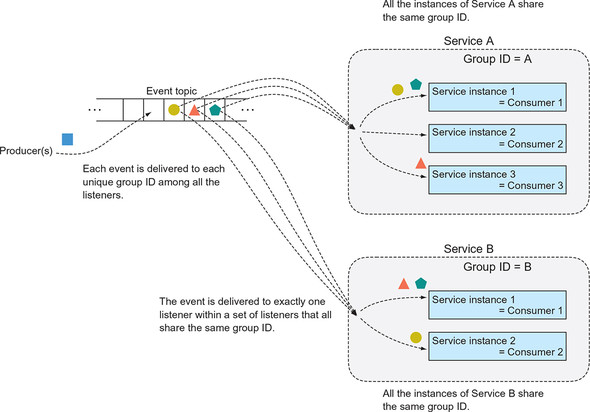
public void userEvent(UserEvent userEvent) {

...

}

In essence, an event topic is a bit of a hybrid between an old-school topic and an old-school queue; the event topic acts as a topic across consumers with different group IDs, and as a queue across consumers sharing the same group ID. [Figure 12.17](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig17) shows this event consumption topology.

**Figure 12.17. Event topics in Kafka have behaviors that are a hybrid between JMS topics and queues. Like the old-school topics, events are delivered to all unique group IDs—there are multiple subscribers. But within a set of listeners that share the same group ID, the behavior is like a queue, delivering the event to one of the listeners.**



An event log topic is different from a messaging topic in one other way, and the name provides a hint—specifically, the word “log.” Whether it’s an application log being routed to Splunk or a database write log, we generally think of logs as something in which the entries are kept around, possibly indefinitely. Preservation of the event log is a fundamental principle of our event-log-based architectures. Events are kept around for any consumers that might be interested in them, even if the consumer isn’t connected at the time of event production. You’ve already seen this in the analyses of our resilience and resultant service autonomy characteristics from earlier in the chapter; in [figure 12.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig11), you saw that the parties interested in an event can pick them sometime after they are produced.

I want to take a concept here to an extreme, an extreme that I assert isn’t actually outlandish at all. The events in the event log are kept for any consumer that might at any point in the future be interested in them—*even consumers that we haven’t even thought of and don’t yet exist!* That’s right, we want to keep events around so that when we have an idea of something we want to do (train a machine learning model to come up with suggestions for food-blogging sites you might like based on the history of users, connections, and posts from the last two years), you can develop that software and it can consume all the events of interest. Preserving the two-year history of events is supporting the innovations of today.

Yes, that means we’ll need a lot of storage, but storage costs have decreased to a point where it’s manageable, and the opportunity gained well outweighs it. There’s no question in my mind that virtually any organization should be thinking about generating and persisting these types of event logs, both for the immediate needs I’ve been describing through the chapter, as well as these forward-looking ones.

I want to be explicit about one last thing when it comes to topics, even though it has already shown up subtly in our discussions so far: each consumer maintains their own cursor into the event log/topic. A *cursor* is the position in the event log at which they have consumed preceding events. As a result, each consumer can process events on their own schedule—even those yet-to-be imagined consumers I talked about a moment ago! And here we come back to this notion of autonomy. Recall the semantics of a messaging topic—all consumers needed to be online at the very moment that a message was produced. Tight coupling!

Having consumers manage their own cursors enables interesting patterns. For example, say you lost the database for the Connections’ Posts service. Because all tables were generated from event log data, to recover the Connections’ Posts service, you need only stand up a new, empty database, set the cursor back to the beginning of the event log, and reprocess all the events. No need for mainline code that is different from failure-recovery code; the results will be predictably the same.

Or will they? As is often the answer to such open-ended questions, it depends. Our next topic has an impact.

**12.3.3. The event payload**

I want to touch upon several aspects related to event payloads here, but I’ll start with what I had in mind when I posed the predictability question a moment ago. You want to be able to re-create your state, the state of the Connections’ Posts service, for example, entirely from the messages in the event log. All the data needed to derive that state must be carried in the events—no references to outside sources are allowed.

Let me illustrate this with an example. Say when the Posts service publishes an event to a topic, in order to save space in the event log, you decide not to include the body of the blog post, but instead only the URL to the blog post. Any consumers know that this is the format of the event, so if they need to process the body of the post, perhaps to do some sentiment analysis, they can retrieve the body by following the link. Although this may work initially, what happens if the blog post is taken down? Reprocessing the event log at a later date won’t allow you to generate the same result you initially did. This brings us to the first rule.

|  |
| --- |
|  |

**Event Payload, rule #1**

An event that’s published to an event log should be described *in its entirety*.

|  |
| --- |
|  |

I know what you’re thinking: “There’s no way that I am putting high-resolution images into an event log!” And I don’t disagree—you shouldn’t. I suggest that asynchronous image processing is not a use case for the type of event-driven workflow that I’ve just described. Yes, you want some processing to occur as the result of an image showing up in a specific location on the filesystem, for example. You’ll want to land a notification of this new image in a topic (or more likely, a queue), but I suggest that what is dropped onto the topic, a URL to where the image can be found, is a message rather than an event. I assert that if the payload isn’t a complete representation of an event from which state can be derived, that it’s instead a message. Of course, it’s completely acceptable to continue using message-passing patterns even as you use event-logging patterns in other parts of your system; it’s all about choosing the right pattern for the job.

Now let’s dig a bit deeper into the details of an event—in particular, the schema. Events have structure, and it’s essential that consumers know that structure so that they can appropriately process the payload. The question, then, is who is responsible for that structure. In the early 2000s, one of the answers that was quite en vogue was the enterprise service bus (ESB). The idea was that a canonical data model would be defined and, if you will, “installed” into the ESB. Producers would then transform their message into the canonical model as a part of the on-ramp into the ESB, and consumers would similarly transform from the canonical model into their own off-ramp. ESBs had entire frameworks around the production and management of the canonical models and the transformations. It was all very hard, and very nonagile.

I’ve come to think of the ESB as a slight incremental evolution of the centralized, canonical, corporate database. Look back at [figure 12.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig02). All of the “independent” microservices in that picture aren’t very independent at all when tethered to a single, centralized database. It turns out that moving that centralization into the messaging fabric didn’t make our systems any less coupled than they were before. What we want instead is to allow producers to deliver events in the form that’s native to their domain. Consumers, who had to adapt to the canonical model in the past anyway, will adapt to the model from the producer.

|  |
| --- |
|  |

**Event Payload, rule #2**

There’s no canonical event model for the event log. Producers have control over the data format of the events they deliver, and consumers will adapt to that.

|  |
| --- |
|  |

The most observant of you will notice that the cloudnative-eventlog project has another module in it called cloudnative-eventschemas, and it looks a little bit like a centralized data model. Guilty. I’ve created this single module and used it for events across all three of our microservices only as a convenience to keep our sample project as simple as possible. In reality, the Connections, Posts, and Connections’ Posts services would have their own repositories, and each service would define the schemas for the events they generate and the schemas for the data of interest in events produced by other services.

And this leads me to the final point I want to make about event payloads: the schemas for these payloads must be managed. Producers who “own” the schemas for the events they produce must adequately describe *and version* those schemas, and the descriptions must be made available to any interested parties in a formalized way. This is why Confluent, the company providing a commercial offering based on Apache Kafka, has a schema registry.**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fn2)**]** The schema registry can then also be used to serialize and deserialize events—payloads needn’t be JSON in the event log. To sum it up, just because we aren’t centralizing schema into a single canonical model doesn’t mean we have the wild, wild west; there still needs to be order.

***2***

*The Confluent Schema Registry provides a host of services that support both creators and consumers of events:*[*http://mng.bz/GWAM*](http://mng.bz/GWAM)*.*

|  |
| --- |
|  |

**Event Payload, rule #3**

All events published to an event log must have an associated schema that may be accessed by all interested parties, and schemas must be versioned.

|  |
| --- |
|  |

The content I’ve covered here only begins to scratch the surface on the topic of event payloads, but I hope that it has adequately highlighted some of the key tenets and how they differ from those of the message-based systems you might be familiar with from the past. As with everything else in cloud-native systems, the focus is on autonomy rather than centralization, evolution rather than prediction, and adaptation rather than strictness.

**12.3.4. Idempotency**

Now, let’s go back to that comment in the event handler code in the Connections’ Posts service:

// make event handler idempotent.

For just a moment, let’s think about the flow that’s happening across our three microservices without the complications that come with them being distributed. New users, connections, and posts come into the Connections and Posts services, which in turn will deliver events to the event log. Consumers pick up these events and process them, writing entries into their respective local storage database. It’s a simple flow. But we know that plenty can go wrong as soon as we distribute the components.

For starters, a producer may experience trouble delivering an event to the event log. Say the producer sends an event to the log but doesn’t get an acknowledgement that it was received. In this case, it will try again. On the second attempt, it receives an acknowledgment, and the producer is done. But consider this: when its first attempt to write to the log wasn’t acknowledged, it’s possible that it was in fact recorded but that the acknowledgment was lost before being received by the producer. In this case, the second attempt could well have put a second event into the log.

You can avoid having duplicate log entries in various ways (most messaging fabrics support *exactly once* delivery, for example), but this can be expensive in terms of performance. As a result, *at-least-once* delivery is generally preferred, effectively shifting the responsibility for deduplication to the consumer.

And that’s where idempotency comes in. An *idempotent* operation is one that can be applied one *or more* times, yielding the same result. When creating a new record in a local database, for example, checking to see whether it was already created and doing nothing in this case makes the operation idempotent. A delete operation is usually idempotent because if you delete an object once, or more than once, in the end the state is the same—the object is deleted. Updates of entities are also usually idempotent.

If you write a consumer that’s not idempotent, this places restrictions on the way it may be used. It forces exactly once (or at-most-once) semantics on the event-delivery protocol, for example. If, on the other hand, you write a consumer that’s idempotent, it affords its use in a broader range of use cases.

|  |
| --- |
|  |

**Event Consumer, rule #1**

If at all possible, make the operations of your event consumers idempotent.

|  |
| --- |
|  |

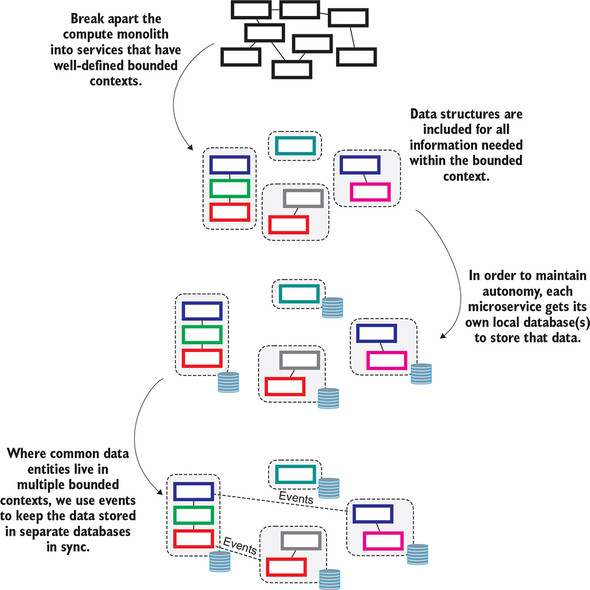
**12.4. Event sourcing**

Whew. We’ve just covered was lot—some new things to think about and a lot of new ways of applying familiar concepts. Yet we’re not quite done. Before launching into this final topic, however, I want to remind you of where we began and where we’re aiming to go in this discussion.

**12.4.1. The journey so far**

This chapter is about cloud-native *data*. At the core, we’re talking about breaking up the data monolith, because without doing this, componentizing the compute monolith gives us only the illusion of autonomy. The microservice, which has a bounded context, is a natural place for us to also draw the lines for data storage, so the first step is that every microservice will get its own database. The question then is how those local data stores are populated. One option is caching, and we see immediate gains in autonomy and resultant resilience. But caching works far better for infrequently changing content than for the often-changing content associated with my microservices. The latter entails many complexities for keeping the data up to date. This leads us to an event-driven approach, where changes are proactively propagated through the network of services that make up our software. But events can be done in a way that too tightly couples different services to one another, so we use the familiar pub/sub pattern. But rather than calling it *messaging*, we call it an *event log* because, among other important differences, the event log should be thought of as something that persists entries indefinitely, and all consumers can work their way through the log in the way that meets their needs. [Figure 12.18](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig18) depicts this progression.

**Figure 12.18. Our derivation of cloud-native data begins by breaking apart the data monolith so that the data supports the bounded contexts of the microservices. Microservices get their own databases, and where needed, data is kept consistent across the distributed data fabric through eventing.**



Despite what I believe seems a reasonable progression that addresses more and more of the challenges of distributing the data through our cloud-native software, I hope that you have at least a vague sense of discomfort. More hairy edge cases exist than the ones I talked about in the previous “[Idempotency](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12lev2sec4)” section, for example. And the brief overview on payload handling might have left you with more questions than answers. But an even bigger cloud is looming over the architecture I’ve derived thus far: *who “owns” what data?*

**12.4.2. The source of truth**

In our sample application, figuring out who owns what data is pretty simple. The Connections service owns users and connections, and the Posts service owns posts. But in a more realistic setting, the question is far more difficult to answer. Does the software running in the bank’s branch location “own” a customer record, or is it the mobile banking application? Or is it perhaps the premiere-customer web application?

At issue here is, what is the source of truth for each and every piece of data in your system? When a discrepancy exists in the email address for a customer across different apps and data stores, which is the right one?

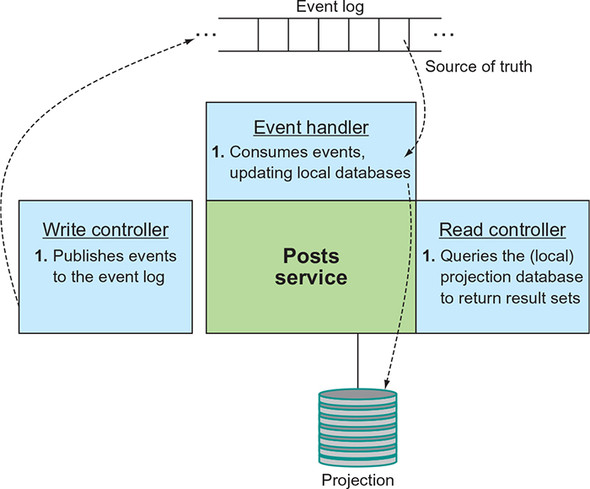
What if I were to tell you it’s *none* of the apps? And here we come to the explanation for the title of this section: *the source of truth for all data is the event store/log.*

This is what is behind the term *event sourcing*. The basic pattern for event sourcing is that all sources of record data writes are made *only* to the event store, and all other stores are simply holding projections or snapshots of state derived from the events stored in the event log. Recall [figure 12.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig14); it showed the three interfaces to the Posts service: the read controller, the write controller, and the event handler. The write controller did two things, persisted new posts in the Posts database and delivered an event to the event log for any other microservice that cares about that event. At the time, I intentionally glossed over what happens when the record is written successfully to the database. But then the event isn’t successfully delivered, or we don’t know that it has been successfully delivered, to the event log. Egad, another hairy edge case.

In a way, the problem is that a two-phase commit doesn’t work across this heterogeneous and distributed set of components; I can’t create a transaction around the database write and the delivery of the event. The solution is this: rather than coordinate across two operations, do only one thing. [Figure 12.19](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig19) updates [figure 12.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig14) to reflect exactly that:

* The write controller is responsible only for delivering events to the event log. Looking back at [figure 12.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig14), you can see that the write controller had been responsible for two things—storing data and delivering events. That it is now responsible for only one of these functions yields a more robust system.
* The event handler is responsible only for reading events and projecting those events into the local data store.
* The read controller is responsible only for returning results based on the state in the local data store.

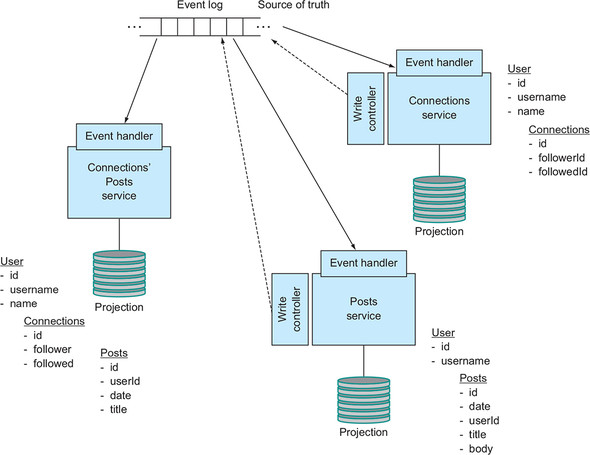
**Figure 12.19. The write controller *only* writes to the event log, which is now the source of truth. The local database for the microservice stores only projections derived from the event log data; this derivation is implemented in the event handler.**



Notice that in this diagram it appears as though some of the data in the local store is being updated it a roundabout way. The event goes into the event store and then comes out of the event store into the projection store. But rather than think of this as a deficiency, I challenge you to think of the write controller as being separate from the Posts service. That’s why I’ve drawn the box not touching the Posts service; each part of the service does only one thing, and we achieve our desired outcomes though composition. What might seem like an inefficiency is offset by a corresponding increase in the robustness of our system.

[Figure 12.20](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig20) updates the earlier event-driven [figure 12.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12fig12) to reflect the event-sourcing approach. You can see the resulting simplification in the data topology. No longer do some services have both source-of-truth databases and local/projection databases while others don’t; all services have only their projection data stores, and the means for keeping those stores up to date is also uniform. Elegant designs are more robust and dependable designs.

**Figure 12.20. The event log is the single source of truth. All microservice-local stores are projections derived from the events in the event log, by the event handlers. The microservice-local stores are only projections.**



**12.4.3. Let’s see this in action: Implementing event sourcing**

Let’s look at the code changes that take our earlier event-log-based implementation all the way to an event-*sourced* one. Check out the following tag from the Git repo:

git checkout eventlog/0.0.2

As discussed previously, we’ve simplified the set of roles that any given service might fulfill. This is reflected in [table 12.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12table02); the consistency of patterns across the different services is stark.

**Table 12.2. Service roles in our event-sourced software architecture**

| Role | Connections service | Posts service | Connections’ Posts service |
| --- | --- | --- | --- |
| *Write controller/event producer*—HTTP endpoint implementations deliver events to Kafka. | Generates user event anytime a user is created, updated, or deleted. Generates connection event anytime a connection is created or deleted. | Generates a post event anytime a post is created. |  |
| *Event handler*—Subscribes to events and updates projection databases accordingly. | Invoked anytime a user or connections event appears in the event log. | Invoked anytime a post or user event appears in the event log. | Invoked anytime a user, connection, or post event appears in the event log. |
| *Projection database*—A dedicated database to store all data of the bounded context of the service. | Stores user and connection data. | Stores a subset of user data. | Stores subsets of user, connection, and post data. |
| *Read*—HTTP endpoint implementations that serve the domain entities for the service. | Users. Connections. | Posts. | Posts made by users followed by an individual. |

Because it was the most sophisticated of all of the services, let’s study the Posts service in more detail.

First, you’ll see that all database functionality has been consolidated from the localstorage and sourceoftruth packages into a single one that I’ve named *projectionstorage*; all data is now stored in the projection store. This isn’t to say that you couldn’t have more than one projection store. You could, for example, project events into a relational database to support relational queries, and project events into a graph database to support graph queries. The emphasis in this refactor is that there’s now only projection data management.

Second, the read controller is largely unchanged, except to accommodate the data store refactoring I’ve just described.

Third, you can see in [listing 12.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12ex04) that the write controller no longer writes to local storage, but only sends the create or update event to the event log. You’ll also notice that I’ve moved the ID generation out of the database because I want to assign an ID when the event is produced, not when the projection is created.

**Listing 12.4. Method from PostsWriteController.java**

@RequestMapping(method = RequestMethod.POST, value="/posts")

public void newPost(@RequestBody PostApi newPost,

HttpServletResponse response) {

logger.info("Have a new post with title " + newPost.getTitle());

Long id = idManager.nextId();

User user = userRepository.findByUsername(newPost.getUsername());

if (user != null) {

// send out new post event

PostEvent postEvent

= new PostEvent("created", id, new Date(), user.getId(),

newPost.getTitle(), newPost.getBody());

kafkaTemplate.send("post", postEvent);

} else

logger.info("Something went awry with creating a new Post - user with

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg username "

+ newPost.getUsername() + " is not known");

}

And finally, added to the event handler is a new method for processing post events. It essentially contains the logic that you removed from the original write controller; in the next listing, you persist the post data in the projection store.

**Listing 12.5. Method from EventHandler.java**

@KafkaListener(topics="post",

groupId = "postsconsumer",

containerFactory = "kafkaListenerContainerFactory")

public void listenForPost(PostEvent postEvent) {

logger.info("PostEvent Handler processing - event: "

+ postEvent.getEventType());

if (postEvent.getEventType().equals("created")) {

Optional<Post> opt = postRepository.findById(postEvent.getId());

if (!opt.isPresent()){

logger.info("Creating a new post in the cache with title "

+ postEvent.getTitle());

Post post = new Post(postEvent.getId(),

postEvent.getDate(),

postEvent.getUserId(),

postEvent.getTitle(),

postEvent.getBody());

postRepository.save(post);

} else

logger.info("Did not create already cached post with id "

+ existingPost.getId());

}

}

Similar changes were made to the Connections service, but no changes were needed for the Connections’ Posts service because it had no source-of-record data, or write controller to start. After studying the code, I encourage you to run the deployApps.sh script to update the deployment to this latest implementation. Run through some examples to see this in action (create a user, create connections and posts, delete a connection), watch the logs, and read the resultant state by using each service’s read controllers:

curl $(minikube service --url connections-svc)/users

curl $(minikube service --url connections-svc)/connections

curl $(minikube service --url posts-svc)/posts

curl $(minikube service --url connections-svc)/connectionsPosts/<username>

This event-sourcing approach, while perhaps different from the way you might think about implementing services such as the examples in this book, is a critical part of a microservices architecture. It’s a pattern that allows you to lose a projection database and any backups thereof and still recover the state of your software by replaying the logs to generate them anew. It allows for microservices that might previously have been beholden to a deep dependency hierarchy for their functionality to operate autonomously, even in the event of network or other infrastructure failures or hiccups. It allows teams to evolve their services without needing to be in lockstep evolution with other services. This pattern is an essential tool to building change-tolerant software.

**12.5. We’re just scratching the surface**

Although we’ve covered a lot of ground in this chapter, there are still so many more things to talk about when it comes to cloud-native data:

* We haven’t talked about partitioning the event log, which is required for scale. When you have 10 million users, you’re going to need to organize the users into subsets. Do you do this by grouping user events by the first letter of their last name, or some other characteristic?
* We haven’t talked about event ordering very much, which is essential to using the event log to derive state projections. Event log technologies implement sophisticated algorithms to ensure proper ordering and may at times tell a producer that it couldn’t record an event because of certain ordering constraints not being met; the event producer must accommodate this.
* We haven’t talked about how to evolve event schemas or techniques such as schema resolution (supported by Apache Avro) that allow old events to impersonate new events.
* We haven’t talked about a practice of taking periodic snapshots of the projection data stores so that if you need to rebuild a projection store from the log, you needn’t go all the way back to the beginning of time.

The topic of cloud-native data is so involved that it deserves its own volume, and I have a recommendation for you. In 2017, Martin Kleppmann, computer science researcher and one of the originators of Apache Kafka at LinkedIn, published *Designing Data-Intensive Applications* (O’Reilly Media). I could not give this a stronger endorsement!

**Summary**

* When you give a microservice a database to store the data that it needs to fulfill its job, it realizes a significant gain in autonomy. As a result, your system as a whole will be more resilient.
* Although in many scenarios it’s far better than nothing, using caching to fill this local database is rife with challenges; caching isn’t a pattern that works well for frequently changing data.
* Proactively pushing data changes into these local data stores via events is a far better approach.
* The familiar pub/sub pattern is used as the backbone of this technique, though the entities that we produce and consume are events rather than messages.
* Making the event log the single source of truth for data, with all service-local databases holding only projections, achieves consistency in a way that works in the highly distributed, constantly changing environment that our cloud-native software runs in.