10 Event-driven architecture with Spring Cloud Stream

This chapter covers

- Understanding event-driven architecture processing and its relevance
- Using Spring Cloud Stream to simplify event processing
- Configuring Spring Cloud Stream
- Publishing messages with Spring Cloud Stream and Kafka
- Consuming messages with Spring Cloud Stream and Kafka
- Implementing distributed caching with Spring Cloud Stream, Kafka, and Redis

Human beings are always in a state of motion as they interact with their environment. Typically, their conversations aren't synchronous, linear, or as narrowly defined as a request-response model. It's something more like message-driven, where we're constantly sending and receiving messages from the things around us. As we receive messages, we react to those messages, often interrupting the primary task that we're working on.

This chapter is about how to design and implement our Spring-based microservices to communicate with other microservices using asynchronous messages. Using asynchronous messages to communicate between applications isn't new. What's new is the concept of using messages to communicate events representing changes in state. This concept is called event-driven architecture (EDA). It's also known as message-driven architecture (MDA). What an EDA-based approach allows us to do is to build highly decoupled systems that can react to changes without being tightly coupled to specific libraries or services. When combined with microservices, EDA allows us to quickly add new functionality to our application by merely having the service listen to the stream of events (messages) being emitted by our application.

The Spring Cloud project makes it trivial to build message-based solutions through the Spring Cloud Stream subproject. Spring Cloud Stream allows us to easily implement message publication and consumption while shielding our services from the implementation details associated with the underlying messaging platform.

10.1 The case for messaging, EDA, and microservices

Why is messaging important in building microservice-based applications? To answer that question, let's start with an example. For this, we'll use the two services that we've used throughout the book: our licensing and organization services.

Let's imagine that after these services are deployed to production, we find that the licensing service calls are taking an exceedingly long time when looking up information from the organization service. When we look at the usage patterns of the organization data, we find that the organization data rarely changes and that most of the data reads from the organization service are done by the primary key of the organization record. If we could cache the reads for the organization data without having to incur the cost of accessing a database, we could significantly improve the response time of the licensing service calls. To implement a caching solution, we need to consider the following three core requirements:

1. *Cached data needs to be consistent across all instances of the licensing service.* This means that we can't cache the data locally within the licensing service because we want to guarantee that the same organization data is read regardless of the service instance hitting it.
2. *We cannot cache the organization data within the memory of the container hosting the licensing service.* The run-time container hosting our service is often restricted in size and can obtain data using different access patterns. A local cache can introduce complexity because we have to guarantee our local cache is in sync with all of the other services in the cluster.
3. *When an organization record changes via an update or delete, we want the licensing service to recognize that there has been a state change in the organization service.* The licensing service should then invalidate any cached data it has for that specific organization and evict it from the cache.

Let's look at two approaches to implement these requirements. The first approach will implement the previously stated requirements using a synchronous request-response model. When the organization state changes, the licensing and organization services will communicate back and forth via their REST endpoints.

For the second approach, the organization service will emit an asynchronous event (message) to communicate that its organization data has changed. The organization service will then publish a message to a queue, which will indicate that an organization record was updated or deleted—a change in state. The licensing service will listen with an intermediary (message broker or queue) to determine if an organization event occurred, and if so, clear the organization data from its cache.

10.1.1 Using a synchronous request-response approach to communicate state change

For our organization data cache, we're going to use Redis (https://redis.io/), a distributed key-value store used as a database, cache, or message broker. Figure 10.1 provides a high-level overview of how to build a caching solution using a traditional synchronous request-response programming model such as Redis.

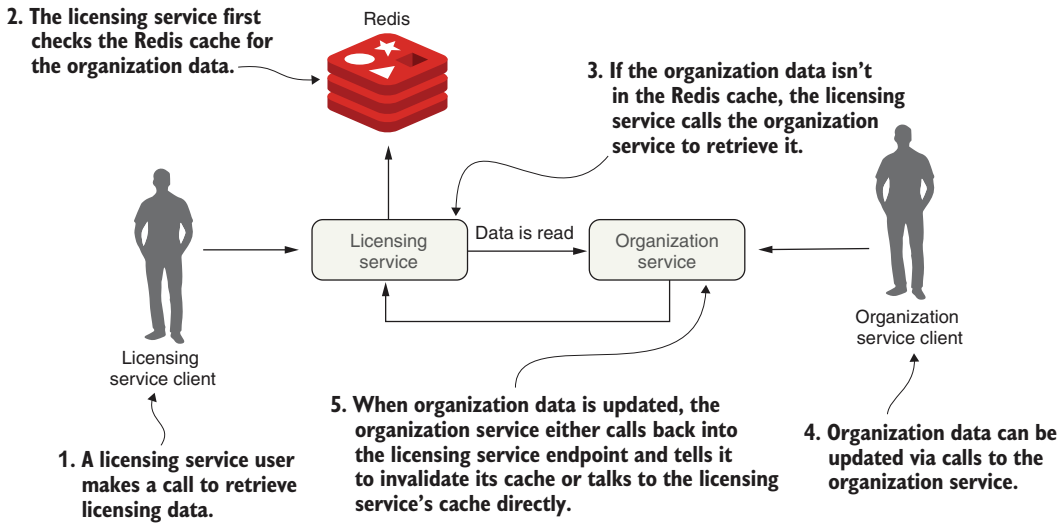

Figure 10.1 In a synchronous request-response model, tightly coupled services introduce complexity and brittleness.

In figure 10.1, when a user calls the licensing service, the licensing service will need to look up the organization data. To do so, the licensing service will first retrieve the desired organization by its ID from a Redis cluster. If the licensing service can't find the organization data, it will then call the organization service using a REST-based endpoint, storing the data returned in Redis before returning the organization data back to the user.

If someone updates or deletes the organization record using the organization service's REST endpoint, the organization service will need to call an endpoint exposed on the licensing service and tell it to invalidate the organization data in its cache. In figure 10.1, if we look at where the organization service calls back into the licensing service to tell it to invalidate the Redis cache, we can see at least three problems:

- The organization and licensing services are tightly coupled. This coupling introduces brittleness between the services.
- If the licensing service endpoint for invalidating the cache changes, the organization service has to change. This approach is inflexible.
- We can't add new consumers of the organization data without modifying the code on the organization service to verify that it calls the licensing service to let it know about any changes.

TIGHT COUPLING BETWEEN SERVICES

To retrieve data, the licensing service is dependent on the organization service. However, by having the organization service directly communicate back to the licensing service when an organization record is updated or deleted, we've introduced coupling from the organization service back to the licensing service (figure 10.1). For the data in the Redis cache to be invalidated, the organization service either needs an exposed endpoint on the licensing service that can be called to invalidate its Redis cache, or the organization service needs to talk directly to the Redis server owned by the licensing service to clear the data in it.

Having the organization service talk to Redis has its own problems because we're talking to a data store owned directly by another service. In a microservice environment, this a big no-no. While one can argue that the organization data rightly belongs to the organization service, the licensing service uses it in a specific context and could potentially transform the data or build business rules around it. If the organization service talks directly to the Redis service, it can accidentally break rules implemented by the team owning the licensing service.

BRITTLENESS BETWEEN THE SERVICES

The tight coupling between the licensing service and the organization service also introduced brittleness between the two services. If the licensing service is down or running slowly, the organization service can be impacted since the organization service is now communicating directly with the licensing service. Again, if the organization service talks directly to the licensing service's Redis data store, we create a dependency between the organization service and Redis. In this scenario, any problem with the shared Redis server now has the potential to take down both services.

INFLEXIBILITY IN ADDING NEW CONSUMERS TO CHANGES IN THE ORGANIZATION SERVICE

With the model in figure 10.1, if we had another service that was interested in the organization data changes, we'd need to add another call from the organization service to the other service. This means a code change and redeployment of the organization service, which can introduce a state of inflexibility in our code.

If we use the synchronous, request-response model for communicating state change, we start to see a web-like pattern of dependency between our core services in our application and other services. The centers of these webs become our major points of failure within our application.

Another kind of coupling

While messaging adds a layer of indirection between our services, we can introduce tight coupling between two services. Later in this chapter, we'll send messages between the organization and licensing services. These messages are going to be serialized and deserialized to a Java object using JSON formatting for the message. Changes to the structure of the JSON message can cause problems when converting back and forth to Java if the two services don't gracefully handle different versions of the same message type.

JSON doesn't natively support versioning. However, we can use Apache Avro (https://avro.apache.org/) if we need that. Avro is a binary protocol that has versioning built into it. Spring Cloud Stream supports Apache Avro as a messaging protocol. Unfortunately, using Avro is outside the scope of this book, but we did want to make you aware that it helps if you truly need to consider message versioning.

10.1.2 Using messaging to communicate state changes between services

We're going to inject a topic between the licensing and organization service with a messaging approach. The messaging system won't be used to read data from the organization service but will instead be used by the organization service to publish any state changes within the data managed by the organization service when these occur. Figure 10.2 demonstrates this approach.

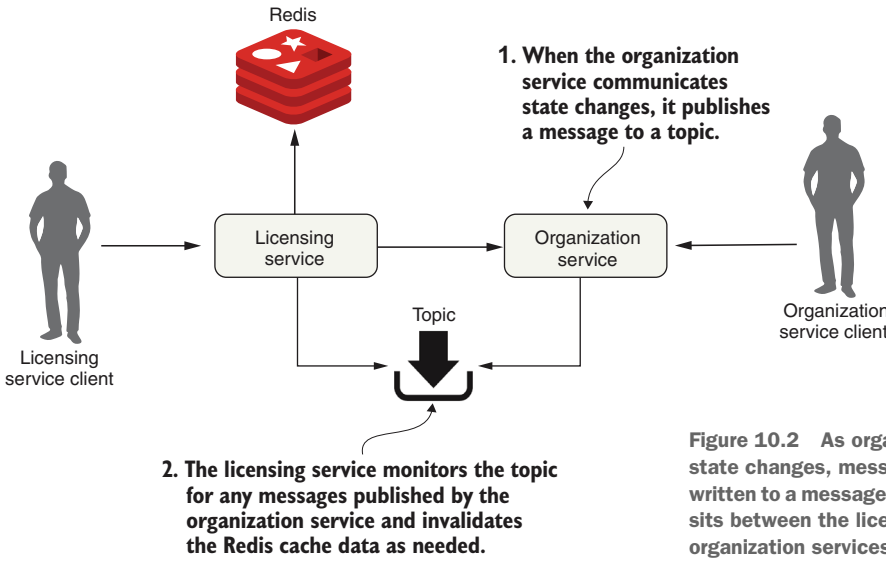

Figure 10.2 As organization state changes, messages are written to a message queue that sits between the licensing and organization services.

In the model in figure 10.2, when organization data changes, the organization service publishes a message to a topic. The licensing service monitors the topic for messages, and when a message arrives, it clears the appropriate organization record from the Redis cache. When it comes to communicating state, the message queue acts as an intermediary between the licensing and organization services. This approach offers four benefits: loose coupling, durability, scalability, and flexibility. We'll look at each in the following sections.

LOOSE COUPLING

A microservices application can be composed of dozens of small and distributed services that interact with each other and the data managed by one another. As we saw with the synchronous design proposed earlier, a synchronous HTTP response creates a hard dependency between the licensing and organization services. We can't eliminate these dependencies completely, but we can try to minimize dependencies by only exposing endpoints that directly manage the data owned by the service.

A messaging approach allows us to decouple the two services because, when it comes to communicating state changes, neither service knows about each other. When the organization service needs to publish a state change, it writes a message to a queue. The licensing service only knows that it gets a message; it has no idea who has published the message.

DURABILITY

The presence of the queue allows us to guarantee that a message will be delivered even if the consumer of the service is down. For example, the organization service can keep publishing messages even if the licensing service is unavailable. The messages are stored in the queue and stay there until the licensing service becomes available. Conversely, with the combination of a cache and the queue approach, if the organization service is down, the licensing service can degrade gracefully because at least part of the organization data will be in its cache. Sometimes old data is better than no data.

SCALABILITY

Because messages are stored in a queue, the sender of the message doesn't have to wait for a response from the message consumer. The sender can go on its way and continue working. Likewise, if a consumer reading a message from the queue isn't processing messages fast enough, it's a trivial task to spin up more consumers and have them process the messages. This scalability approach fits well within a microservices model.

One of the things we've emphasized throughout this book is that it should be trivial to spin up new instances of a microservice. The additional microservice can then become another service to process the message queue. This is an example of scaling horizontally.

Traditional scaling mechanisms for reading messages in a queue involved increasing the number of threads that a message consumer could process at one time. Unfortunately, with this approach, we were ultimately limited by the number of CPUs

available to the message consumer. A microservice model doesn't have this limitation because we can scale by increasing the number of machines hosting the service consuming the messages.

Flexibility

The sender of a message has no idea who is going to consume it. This means we can easily add new message consumers (and new functionality) without impacting the original sending service. This is an extremely powerful concept because new functionality can be added to an application without having to touch existing services. Instead, the new code can listen for events being published and react to them accordingly.

10.1.3 Downsides of a messaging architecture

Like any architectural model, a message-based architecture has trade-offs. A message-based architecture can be complicated and requires the development team to pay close attention to several key things, including message-handling semantics, message visibility, and message choreography. Let's look at these in more detail.

Message-handling semantics

Using messages in a microservice-based application requires more than understanding how to publish and consume messages. It requires that we understand how our application will behave based on the order in which messages are consumed and what happens if a message is processed out of order. For example, if we have strict requirements that all orders from a single customer must be processed in the order they are received, we'll need to set up and structure our message handling differently than if every message can be consumed independently of one another.

It also means that if we're using messaging to enforce strict state transitions of our data, we need to think about designing our applications to take into consideration scenarios where a message throws an exception or an error is processed out of order. If a message fails, do we retry processing the error or do we let it fail? How do we handle future messages related to that customer if one of the customer's messages fails? These are important questions to think through.

Message visibility

Using messages in our microservices often means a mix of synchronous service calls and asynchronous service processing. The asynchronous nature of messages means they might not be received or processed in close proximity to when the message is published or consumed. Also, having things like correlation IDs for tracking a user's transactions across web service invocations and messages is critical to understanding and debugging what's going on in our application. As you may remember from chapter 8, a correlation ID is a unique number that's generated at the start of a user's transaction and passed along with every service call. It should also be passed along with every message that's published and consumed.

MESSAGE CHOREOGRAPHY

As alluded to in the section on message visibility, a message-based application makes it more difficult to reason through its business logic because its code is no longer processed in a linear fashion with a simple block request-response model. Instead, debugging message-based applications can involve wading through the logs of several different services, where user transactions can be executed out of order and at different times.

> **NOTE** Messaging can be complicated but powerful. In the previous sections, we didn't mean to scare you away from using messaging in your applications. Instead, our goal was to highlight that using messaging in your services requires forethought. A positive side of messaging is that businesses themselves work asynchronously, so in the end, we are modeling our business more closely.

10.2 Introducing Spring Cloud Stream

Spring Cloud makes it easy to integrate messaging into our Spring-based microservices. It does this through the Spring Cloud Stream project (https://spring.io/projects/spring-cloud-stream), which is an annotation-driven framework that allows us to easily build message publishers and consumers in our Spring applications.

Spring Cloud Stream also allows us to abstract away the implementation details of the messaging platform that we're using. We can use multiple message platforms with Spring Cloud Stream, including the Apache Kafka project and RabbitMQ, and the platform's implementation-specific details are kept out of the application code. The implementation of message publication and consumption in your application is done through platform-neutral Spring interfaces.

> **NOTE** For this chapter, we'll use a message bus called Kafka (https://kafka.apache.org/). Kafka is a highly performant message bus that allows us to asynchronously send streams of messages from one application to one or more other applications. Written in Java, Kafka is the de facto message bus for many cloud-based applications because it's highly reliable and scalable. Spring Cloud Stream also supports the use of RabbitMQ as a message bus.

To understand Spring Cloud Stream, let's begin with a discussion of its architecture and familiarize ourselves with some terminology. The new terminology can be somewhat overwhelming if you've never worked with a message-based platform before, so let's begin our discussion by looking at the Spring Cloud Stream architecture through the lens of two services communicating via messaging. One service is the message publisher, and one service is the message consumer. Figure 10.3 shows how Spring Cloud Stream is used to facilitate this message passing.

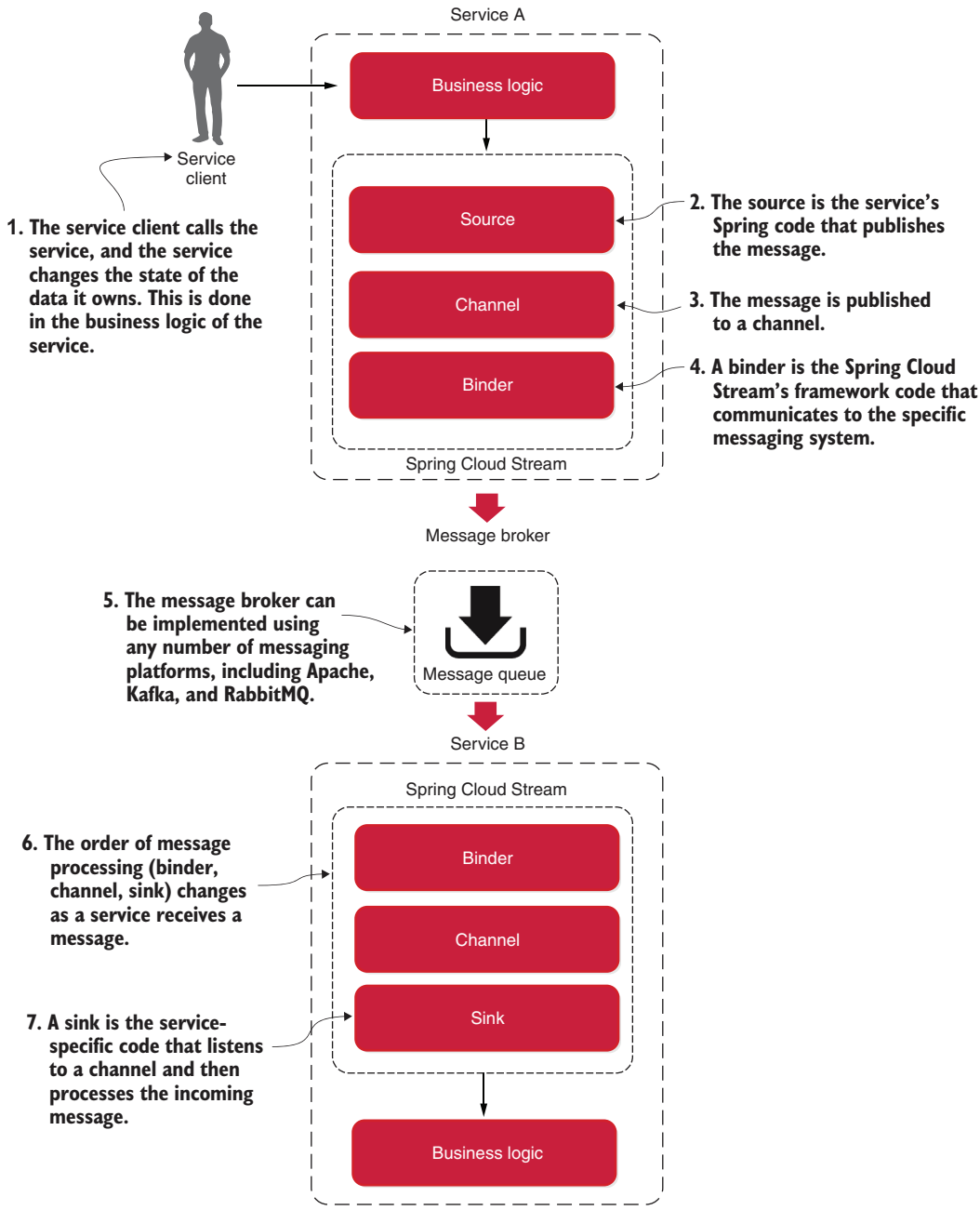

Figure 10.3 As a message is published and consumed, it flows through a series of Spring Cloud Stream components that abstract away the underlying messaging platform.

With Spring Cloud, four components are involved in publishing and consuming a message:

- Source
- Channel
- Binder
- Sink

When a service gets ready to publish a message, it will publish the message using a *source*. A source is a Spring-annotated interface that takes a Plain Old Java Object (POJO), which represents the message to be published. The source takes the message, serializes it (the default serialization is JSON), and publishes the message to a channel.

A *channel* is an abstraction over the queue that's going to hold the message after it's published by a message producer or consumed by a message consumer. In other words, we can describe a channel as a queue that sends and receives messages. A channel name is always associated with a target queue name, but that queue name is never directly exposed to the code. Instead, the channel name is used in the code, which means that we can switch the queues the channel reads or writes from by changing the application's configuration, not the application's code.

The *binder* is part of the Spring Cloud Stream framework. It's the Spring code that talks to a specific message platform. The binder part of the Spring Cloud Stream framework allows us to work with messages without having to be exposed to platform-specific libraries and APIs for publishing and consuming messages.

In Spring Cloud Stream, when a service receives a message from a queue, it does it through a *sink*. A sink listens to a channel for incoming messages and deserializes the message back into a POJO object. From there, the message can be processed by the business logic of the Spring service.

10.3 Writing a simple message producer and consumer

Now that we've walked through the essential components in Spring Cloud Stream, let's look at a simple Spring Cloud Stream example. For the first example, we'll pass a message from our organization service to our licensing service, which prints a log message to the console. In addition, because we'll only have one Spring Cloud Stream source (the message producer) and a sink (message consumer) in this example, we'll start the example with a few simple Spring Cloud shortcuts. This will make setting up the source in the organization service and in a sink in the licensing service trivial. Figure 10.4 highlights the message producer and builds on the general Spring Cloud Stream architecture from figure 10.3.

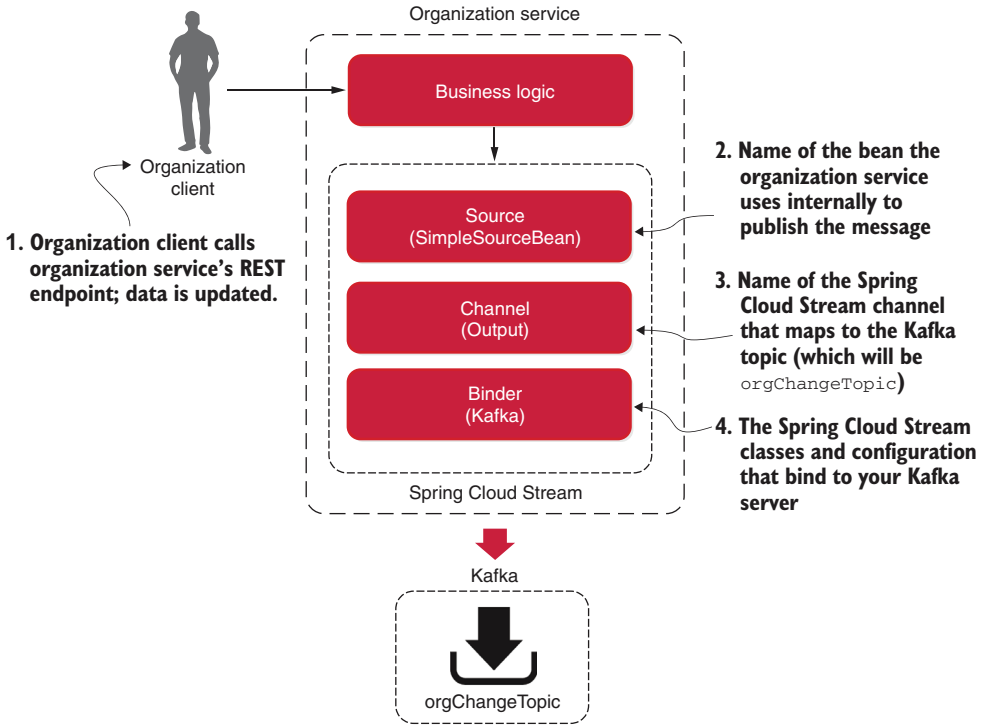

Figure 10.4 When the organization service data changes, it publishes a message to `orgChangeTopic`.

10.3.1 Configuring Apache Kafka and Redis in Docker

In this section, we'll explain how to add the Kafka and Redis services to our Docker environment for our message producer. To achieve this, let's start by adding the code shown in the following listing to our docker-compose.yml file.

Listing 10.1 Adding Kafka and Redis services to docker-compose.yml

```
//Parts of docker-compose.yml removed for conciseness
...

zookeeper:
    image: wurstmeister/zookeeper:latest
    ports:
      - 2181:2181
    networks:
      backend:
        aliases:
          - "zookeeper"
  kafkaserver:
    image: wurstmeister/kafka:latest
    ports:
      - 9092:9092
```

```
    environment:
      - KAFKA_ADVERTISED_HOST_NAME=kafka
      - KAFKA_ADVERTISED_PORT=9092
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_CREATE_TOPICS=dresses:1:1,ratings:1:1
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    depends_on:
      - zookeeper
    networks:
      backend:
        aliases:
          - "kafka"
  redisserver:
    image: redis:alpine
    ports:
      - 6379:6379
    networks:
      backend:
        aliases:
          - "redis"
```

10.3.2 Writing the message producer in the organization service

To focus on how to use topics in our architecture, we'll begin by modifying the organization service so that every time organization data is added, updated, or deleted, the organization service will publish a message to a Kafka topic, indicating that an organization change event has occurred. The published message will include the organization ID associated with the change event and what action occurred (add, update, or delete).

The first thing we need to do is set up our Maven dependencies in the organization service's Maven pom.xml file. You'll find the pom.xml file in the root directory for the organization service. In the pom.xml, we need to add two dependencies, one for the core Spring Cloud Stream libraries and the other for the Spring Cloud Stream Kafka libraries:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

NOTE We're using Docker to run all the examples. If you want to run this locally, you need to have Apache Kafka installed on your computer. If you are using Docker, you can find the up-to-date docker-compose.yml file with the Kafka and Zookeeper containers here: https://github.com/ihuaylupo/manning-smia/tree/master/chapter10/docker.

Remember to execute the services by executing the following commands in the root directory where the parent pom.xml is located:

```
mvn clean package dockerfile:build && docker-compose
            -f docker/docker-compose.yml up
```

Once we've defined the Maven dependencies, we need to tell our application that it's going to bind to a Spring Cloud Stream message broker. We can do this by annotating the organization service's bootstrap class, `OrganizationServiceApplication`, with `@EnableBinding`. You'll find the code for this class in /organization-service/src/main/java/com/optimagrowth/organization/OrganizationServiceApplication.java. For your convenience, the following listing shows the `OrganizationServiceApplication.java` source code.

Listing 10.2 The annotated `OrganizationServiceApplication.java` class

```
package com.optimagrowth.organization;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import
     org.springframework.security.oauth2.config.annotation.web.configuration.
     EnableResourceServer;

@SpringBootApplication
@RefreshScope
@EnableResourceServer
@EnableBinding(Source.class)
public class OrganizationServiceApplication {

   public static void main(String[] args) {
      SpringApplication.run(OrganizationServiceApplication.class, args);
   }
}
```

Tells Spring Cloud Stream to bind the application to a message broker

In listing 10.2, the `@EnableBinding` annotation tells Spring Cloud Stream that we want to bind the service to a message broker. The use of `Source.class` in `@EnableBinding` tells Spring Cloud Stream that this service will communicate with the message broker via a set of channels defined in the `Source` class. Remember, channels sit above a message queue. Spring Cloud Stream has a default set of channels that can be configured to speak to a message broker.

At this point, we haven't told Spring Cloud Stream what message broker we want the organization service to bind to. We'll get to that shortly. Now, we can go ahead and implement the code that publishes a message. The first step is to change the `UserContext`

class in /organization-service/src/main/java/com/optimagrowth/organization/utils/UserContext.java. This change will make our variables thread local. The following listing shows the code for the `ThreadLocal` class.

Listing 10.3 Making our `UserContext` variables `ThreadLocal`

```
package com.optimagrowth.organization.utils;
//Imports removed for conciseness

@Component
public class UserContext {
    public static final String CORRELATION_ID = "tmx-correlation-id";
    public static final String AUTH_TOKEN     = "Authorization";
    public static final String USER_ID        = "tmx-user-id";
    public static final String ORG_ID         = "tmx-org-id";

    private static final ThreadLocal<String> correlationId =
             new ThreadLocal<String>();
    private static final ThreadLocal<String> authToken =
             new ThreadLocal<String>();
    private static final ThreadLocal<String> userId =
             new ThreadLocal<String>();
    private static final ThreadLocal<String> orgId =
             new ThreadLocal<String>();

    public static HttpHeaders getHttpHeaders(){
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.set(CORRELATION_ID, getCorrelationId());

        return httpHeaders;
    }
}
```

Defining our variables as ThreadLocal lets us store data individually for the current thread. The information set here can only be read by the thread that set the value.

The next step is to create the logic to publish the message. You'll find the code to publish our message in the /organization-service/src/main/java/com/optimagrowth/organization/events/source/SimpleSourceBean.java class file. The following listing shows the code for this class.

Listing 10.4 Publishing a message to the message broker

```
package com.optimagrowth.organization.events.source;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

import com.optimagrowth.organization.events.model.OrganizationChangeModel;
import com.optimagrowth.organization.utils.UserContext;
```

```
@Component
public class SimpleSourceBean {
    private Source source;

    private static final Logger logger =
       LoggerFactory.getLogger(SimpleSourceBean.class);

    public SimpleSourceBean(Source source){
        this.source = source;
    }

    public void publishOrganizationChange(ActionEnum action,
                                    String organizationId){
       logger.debug("Sending Kafka message {} for Organization Id: {}",
                    action, organizationId);
        OrganizationChangeModel change =  new OrganizationChangeModel(
                OrganizationChangeModel.class.getTypeName(),
                action.toString(),
                organizationId,
                UserContext.getCorrelationId());

        source.output().send(MessageBuilder
                           .withPayload(change)
                           .build());
    }
}
```

Injects a Source interface implementation for use by the service

Publishes a Java POJO message

Sends the message from a channel defined in the Source class

In listing 10.4, we injected the Spring Cloud `Source` class into our code. Remember, all communication to a specific message topic occurs through a Spring Cloud Stream construct called a *channel*, which is represented by a Java interface class. In the listing, we used the `Source` interface, which exposes a single method called `output()`.

The `Source` interface is convenient to use when our service only needs to publish to a single channel. The `output()` method returns a class of type `MessageChannel`. With this type, we'll send messages to the message broker. (Later in this chapter, we'll show you how to expose multiple messaging channels using a custom interface.) The `ActionEnum` passed by the parameters in the `output()` method contains the following actions:

```
public enum ActionEnum {
   GET,
   CREATED,
   UPDATED,
   DELETED
}
```

The actual publication of the message occurs in the `publishOrganizationChange()` method. This method builds a Java POJO called `OrganizationChangeModel`. The following listing shows the code for this POJO.

Listing 10.5 Publishing the `OrganizationChangeModel` object

```
package com.optimagrowth.organization.events.model;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Getter @Setter @ToString
public class OrganizationChangeModel {
    private String type;
    private String action;
    private String organizationId;
    private String correlationId;

    public OrganizationChangeModel(String type,
            String action, String organizationId,
            String correlationId) {
        this.type = type;
        this.action = action;
        this.organizationId = organizationId;
        this.correlationId = correlationId;
    }
}
```

The `OrganizationChangeModel` class declares three data elements:

- `action`—This is the action that triggered the event. We've included the `action` element in the message to give the message consumer more context on how it should process an event.
- `organizationId`—This is the organization ID associated with the event.
- `correlationId`—This is the correlation ID of the service call that triggered the event. We should always include a correlation ID in our events as it helps greatly with tracking and debugging the flow of messages through our services.

If we go back to the `SimpleSourceBean` class, we can see that when we're ready to publish the message, we can use the `send()` method on the `MessageChannel` class returned from the `source.output()` method like this:

```
source.output().send(MessageBuilder.withPayload(change).build());
```

The `send()` method takes a Spring `Message` class. We use a Spring helper class, called `MessageBuilder`, to take the contents of the `OrganizationChangeModel` class and convert it to a Spring `Message` class. This is all the code we need to send a message. However, at this point, everything should feel a little bit like magic because we haven't seen how to bind our organization service to a specific message queue, let alone the actual message broker. This is all done through configuration.

Listing 10.6 shows the configuration that maps our service's Spring Cloud Stream `Source` to a Kafka message broker and a message topic. This configuration information can be localized in our Spring Cloud Config entry for the organization service.

NOTE For this example, we use the classpath repository on the Spring Cloud Config. The configuration for the organization service is found in /configserver/src/main/resources/config/organization-service.properties.

Listing 10.6 The Spring Cloud Stream configuration for publishing a message

Names the message queue (or topic) that writes the messages

```
#Parts omitted for conciseness

spring.cloud.stream.bindings.output.destination=
  orgChangeTopic
spring.cloud.stream.bindings.output.content-type=
  application/json
spring.cloud.stream.kafka.binder.zkNodes=
  localhost
spring.cloud.stream.kafka.binder.brokers=
  localhost
```

Provides (hints) the message type that's sent and received (in this case, JSON)

These properties provide the network location of Kafka and Zookeeper.

NOTE Apache Zookeeper is used to maintain configuration and name data. It also provides flexible synchronization in distributed systems. Apache Kafka acts like a centralized service that keeps track of the Kafka cluster nodes and topics configuration.

The configuration in code listing 10.6 looks dense, but it's quite straightforward. The `spring.cloud.stream.bindings` is the start of the configuration needed for our service to publish to a Spring Cloud Stream message broker. The configuration property `spring.cloud.stream.bindings.output` in the listing maps the `source.output()` channel in listing 10.4 to the `orgChangeTopic` on the message broker we're going to communicate with. It also tells Spring Cloud Stream that messages sent to this topic should be serialized as JSON. Spring Cloud Stream can serialize messages in multiple formats including JSON, XML, and the Apache Foundation's Avro format (https://avro.apache.org/).

Now that we have the code that will publish a message via Spring Cloud Stream and the configuration to tell Spring Cloud Stream to use Kafka as a message broker, let's look at where the publication of the message in our organization service actually occurs. The class `OrganizationService` will do this for us. You'll find the code for this class in /organization-service/src/main/java/com/optimagrowth/organization/service/OrganizationService.java. The following listing shows the code for this class.

Listing 10.7 Publishing a message in the organization service

```
package com.optimagrowth.organization.service;
//Imports removed for conciseness
@Service
public class OrganizationService {

    private static final Logger logger =
     LoggerFactory.getLogger(OrganizationService.class);

    @Autowired
    private OrganizationRepository repository;

    @Autowired
    SimpleSourceBean simpleSourceBean;

    public Organization create(Organization organization){
        organization.setId( UUID.randomUUID().toString());
        organization = repository.save(organization);
        simpleSourceBean.publishOrganizationChange(
            ActionEnum.CREATED,
            organization.getId());
        return organization;

    }

   //Rest of the code removed for conciseness
}
```

Uses autowiring to inject the SimpleSourceBean into the organization service

For each method in the service that changes organization data, calls simpleSourceBean.publishOrgChange()

What data should I put in the message?

One of the most common questions we get from teams when they're first embarking on their message journey is, exactly how much data should I put in the message? Our answer is, it depends on your application.

As you may have noticed, in all our examples, we only return the organization ID of the organization record that's changed. We never put a copy of the data changes in the message. Also, we use messages based on system events to tell other services that the data state has changed, and we always force the other services to go back to the master (the service that owns the data) to retrieve a new copy of the data. This approach is costlier in terms of execution time, but it guarantees we always have the latest copy of the data. But a slight chance exists that the data we work with could change right after we've read it from the source system. That's much less likely to occur, however, than if we blindly consume the information right off the queue.

Our advice is to think carefully about how much data you're passing around. Sooner or later, you'll run into a situation where the data passed is "stale." It could be because a problem caused it to sit in the message queue too long, or a previous message containing data failed and the data you're passing in the message now represents data in an inconsistent state. This might be because your application relied on the message's state rather than on the actual state in the underlying data store. If you're going to pass state in your message, make sure to include a date-time stamp or version number so that the service that's consuming the data can inspect the data passed to it and ensure that it's not older than the copy of the data it already has.

10.3.3 Writing the message consumer in the licensing service

At this point, we've modified the organization service to publish a message to Kafka every time the organization service changes data. Any interested service can react without having to be explicitly called by the organization service. It also means we can easily add new functionality that can react to the changes in the organization service by listening to messages coming into the message queue.

Let's now switch directions and look at how a service can consume a message using Spring Cloud Stream. For this example, the licensing service will consume the message published by the organization service. To begin, we need to add our Spring Cloud Stream dependencies to the licensing services pom.xml file. You'll find this file in the licensing-service root directory of the source code for the book. Figure 10.5 shows where the licensing service fits into the Spring Cloud architecture, first shown in figure 10.3. Similar to the organization-service pom.xml file that you saw earlier, we'll add the following two dependency entries to the .pom file:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

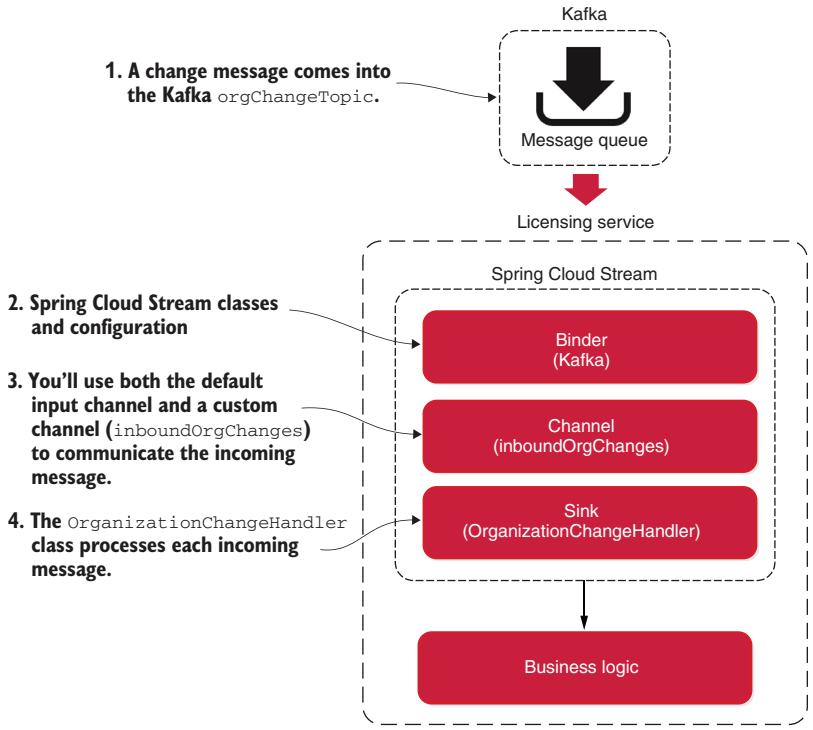

Figure 10.5 When a message comes into the Kafka `orgChangeTopic`, the licensing service responds.

Next, we'll tell the licensing service that it needs to use Spring Cloud Stream to bind to a message broker. Like the organization service, we'll annotate the licensing service's bootstrap class, `LicenseServiceApplication`, with the `@EnableBinding` annotation. You can find the code for this class in the /licensing-service/src/main/java/com/optimagrowth/license/LicenseServiceApplication.java file. The difference between the licensing service and the organization service is the value we'll pass to `@EnableBinding`, which the following listing shows.

Listing 10.8 Consuming a message using Spring Cloud Stream

Tells the service to the use the channels defined in the Sink interface to listen for incoming messages

```
package com.optimagrowth.license;
//Imports and some annotations removed for conciseness

@EnableBinding(Sink.class)
public class LicenseServiceApplication {

   @StreamListener(Sink.INPUT)
   public void loggerSink(OrganizationChangeModel orgChange) {
      logger.debug("Received an {} event for organization id {}",
       orgChange.getAction(), orgChange.getOrganizationId());
   }

   //Rest of the code omitted for conciseness
}
```

Executes this method each time a message is received from the input channel

Because the licensing service is the message consumer, we'll pass `@EnableBinding` the value `Sink.class`. This tells Spring Cloud Stream to bind to a message broker using the default Spring `Sink` interface. Similar to the `Source` interface (described in section 10.3.1), Spring Cloud Stream exposes a default channel on the `Sink` interface. This `Sink` interface channel is called `input` and is used to listen for incoming messages.

Once we've defined that we want to listen for messages via the `@EnableBinding` annotation, we can write the code to process a message coming from the `Sink` input channel. To do this, we'll use the Spring Cloud Stream `@StreamListener` annotation. This annotation tells Spring Cloud Stream to execute the `loggerSink()` method when receiving a message from the input channel. Spring Cloud Stream automatically deserializes the incoming message to a Java POJO called `OrganizationChangeModel`.

Again, the licensing service's configuration implements the mapping of the message broker's topic to the input channel. Its configuration is shown in listing 10.9 and can be found in the Spring Cloud config repository for this example in the /configserver/src/main/resources/config/licensing-service.properties file.

Listing 10.9 Mapping the licensing service to a message topic in Kafka

```
#Some properties removed for conciseness

spring.cloud.stream.bindings.input.destination=
  orgChangeTopic
```

Maps the input channel to the orgChangeTopic queue

```
spring.cloud.stream.bindings.input.content-type=
  application/json
spring.cloud.stream.bindings.input.group=
  licensingGroup
spring.cloud.stream.kafka.binder.zkNodes=
  localhost
spring.cloud.stream.kafka.binder.brokers=
  localhost
```

Process semantics once per service

The configuration in this listing looks like the configuration for the organization service. It has, however, two key differences. First, we now have an input channel defined with the `spring.cloud.stream.bindings` property. This value maps to the `Sink.INPUT` channel defined in the code from listing 10.8. This property maps the input channel to the `orgChangeTopic`. Second, we see the introduction of a new property called `spring.cloud.stream.bindings.input.group`. The `group` property defines the name of the consumer group that will consume the message.

The concept of a consumer group is this: we can have multiple services with each service having multiple instances listening to the same message queue. We want each unique service to process a copy of a message, but we only want one service instance within a group of service instances to consume and process a message. The `group` property identifies the consumer group that the service belongs to.

As long as all the service instances have the same group name, Spring Cloud Stream and the underlying message broker will guarantee that only one copy of the message will be consumed by a service instance belonging to that group. In the case of our licensing service, we'll call the `group` property value `licensingGroup`. Figure 10.6 illustrates how this consumer group helps to enforce consumption once the semantics for a consumed message across multiple services is satisfied.

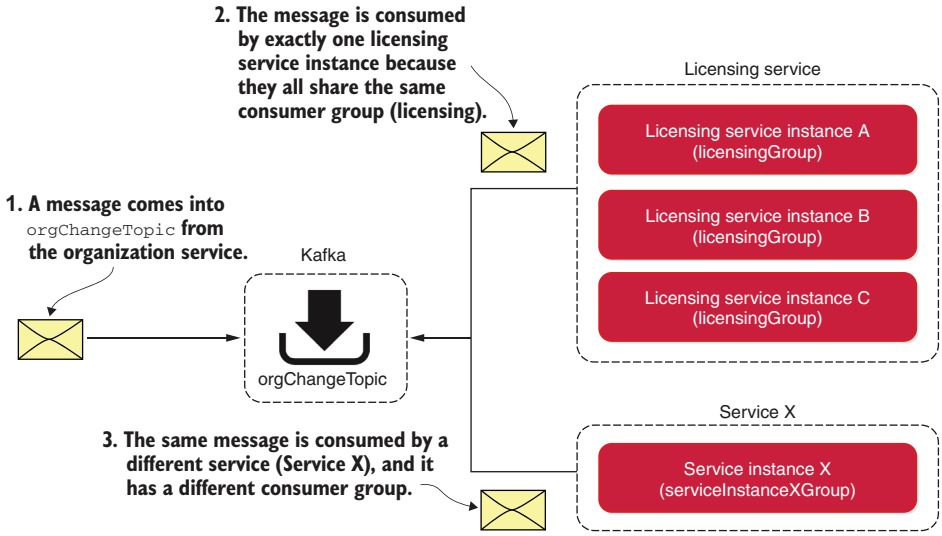

Figure 10.6 The consumer group guarantees that a message is only processed once by a group of service instances.

10.3.4 Seeing the message service in action

At this point, the organization service publishes a message to the `orgChangeTopic` each time a record is added, updated, or deleted, and the licensing service receives the message with the same topic. Next, we'll see this code in action by creating an organization service record and watching the console to see the corresponding log message from the licensing service.

To create the organization service record, we're going to issue a POST on the organization service. We'll use the `http://localhost:8072/organization/v1/organization/` endpoint and send the following body on the POST call to the endpoint. Figure 10.7 shows the returned output from this POST call:

```
{
    "name":"Ostock",
    "contactName":"Illary Huaylupo",
    "contactEmail":"illaryhs@gmail.com",
    "contactPhone":"888888888"
}
```

NOTE Remember that we first needed to implement the authentication to retrieve the tokens and pass the access token via the authorization header as a Bearer Token. We discussed this in the previous chapter. If you didn't follow the code samples there, download the code from http://github.com/ihuaylupo/manning-smia/tree/master/chapter9. We then pointed our code to the Spring Cloud Gateway. That's why the endpoint has the 8072 port and the /organization/v1/organization path instead of just /v1/organization.

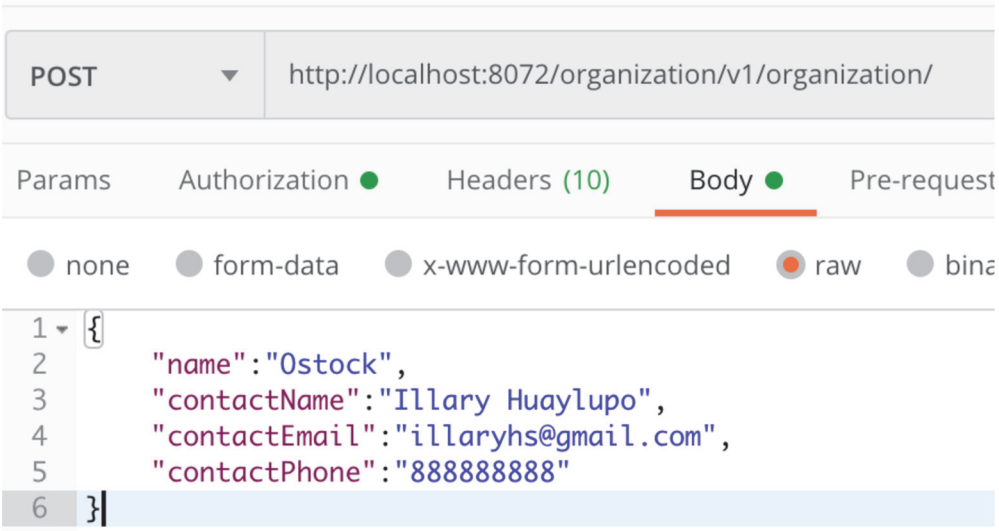

Figure 10.7 Creating a new organization service record using the organization service

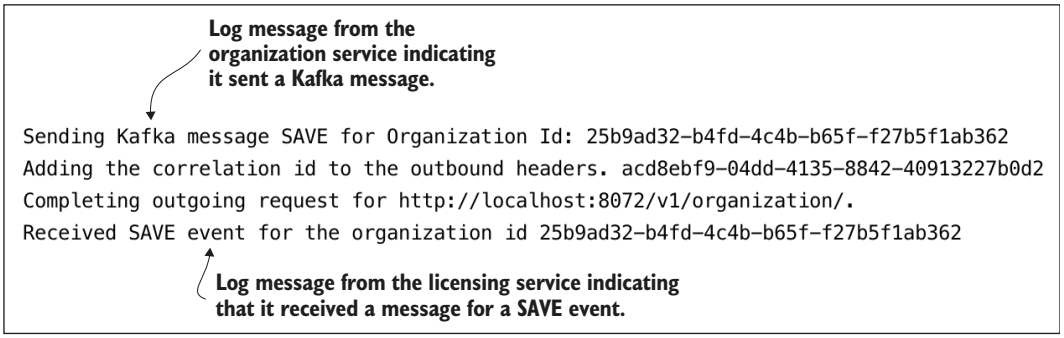

Figure 10.8 The console shows the message from the organization service being sent and then received.

Once we make the organization service call, we should see the output shown in figure 10.8 in the console window running the services.

Now we have two services communicating with each other using messages. Spring Cloud Stream acts as the middleman for these services. From a messaging perspective, the services know nothing about each other. They're using a messaging broker to communicate as an intermediary and Spring Cloud Stream as an abstraction layer over the messaging broker.

10.4 A Spring Cloud Stream use case: Distributed caching

We now have two services communicating with messaging, but we're not really doing anything with the messages. Next, we'll build the distributed caching example we discussed earlier in the chapter. For that, the licensing service will always check a distributed Redis cache for the organization data associated with a particular license. If the organization data exists in the cache, we'll return the data from the cache. If it doesn't, we'll call the organization service and cache the results of the call in a Redis hash.

When the data is updated in the organization service, the organization service will issue a message to Kafka. The licensing service will pick up the message and issue a DELETE against Redis to clear the cache.

Cloud caching and messaging

Using Redis as a distributed cache is relevant to microservices development in the cloud. You can use Redis to

- *Improve performance when looking up commonly held data.* Using a cache, you can significantly improve the performance of several key services by avoiding reads to the database.
- *Reduce the load (and cost) on the database tables holding your data.* Accessing data in a database can be a costly undertaking. With every read, you make a chargeable event. Using a Redis server, you can implement reads by a primary key rather than by accessing a database, which is significantly more cost effective.

(continued)

- *Increase resiliency so your services can degrade gracefully if your primary data store or database is having performance problems.* Depending on how much data you keep in your cache, a caching solution can help reduce the number of errors you might get from accessing your data store.

Redis is far more than a caching solution. It can, however, fill that role if you need a distributed cache.

10.4.1 Using Redis to cache lookups

In this section, we'll begin by setting up the licensing service to use Redis. Fortunately, Spring Data makes it simple to introduce Redis into our licensing service. To use Redis in the licensing service, we need to do the following:

1. Configure the licensing service to include the Spring Data Redis dependencies.
2. Construct a database connection to Redis.
3. Define the Spring Data Redis repositories that our code will use to interact with a Redis hash.
4. Use Redis and the licensing service to store and read organization data.

CONFIGURING THE LICENSING SERVICE WITH SPRING DATA REDIS DEPENDENCIES

The first thing we need to do is include the `spring-data-redis` dependencies, along with `jedis` into the licensing service's pom.xml file. The next listing shows these dependencies.

Listing 10.10 Adding the Spring Redis dependencies

```
//Some code removed for conciseness
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-redis</artifactId>
</dependency>

<dependency>
   <groupId>redis.clients</groupId>
   <artifactId>jedis</artifactId>
   <type>jar</type>
</dependency>
```

CONSTRUCTING THE DATABASE CONNECTION TO A REDIS SERVER

Now that we have the dependencies in Maven, we need to establish a connection to our Redis server. Spring uses the Jedis open source project (https://github.com/xetorthio/jedis) to communicate with a Redis server. To communicate with a specific Redis instance, we'll expose a `JedisConnectionFactory` class as a Spring bean. You'll find the source for this class in /licensing-service/src/main/java/com/optimagrowth/license/LicenseServiceApplication.java.

Once we have a connection to Redis, we'll use that connection to create a Spring `RedisTemplate` object. The Spring Data repository classes that we'll implement shortly use the `RedisTemplate` object to execute the queries and saves of organization service data to our Redis service. The following listing shows this code.

Listing 10.11 Establishing how our licensing service communicates with Redis

```
package com.optimagrowth.license;

import org.springframework.data.redis.connection.RedisPassword;
import org.springframework.data.redis.connection.
       RedisStandaloneConfiguration;
import org.springframework.data.redis.connection.jedis.
       JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

//Most of the imports and annotations removed for conciseness

@SpringBootApplication
@EnableBinding(Sink.class)
public class LicenseServiceApplication {

   @Autowired
   private ServiceConfig serviceConfig;

   //All other methods in the class have been removed for conciseness

   @Bean
   JedisConnectionFactory jedisConnectionFactory() {
       String hostname = serviceConfig.getRedisServer();
       int port = Integer.parseInt(serviceConfig.getRedisPort());
       RedisStandaloneConfiguration redisStandaloneConfiguration
           = new RedisStandaloneConfiguration(hostname, port);
    return new JedisConnectionFactory(redisStandaloneConfiguration);
   }

   @Bean
   public RedisTemplate<String, Object> redisTemplate() {
       RedisTemplate<String, Object> template = new RedisTemplate<>();
       template.setConnectionFactory(jedisConnectionFactory());
       return template;
   }

   //Rest of the code removed for conciseness
}
```

Sets up the database connection to the Redis server

Creates a RedisTemplate to carry out actions for our Redis server

The foundational work for setting up the licensing service to communicate with Redis is complete. Let's now move on to writing the logic that will get, add, update, and delete data.

`ServiceConfig` is a simple class that contains the logic to retrieve the custom parameters that we'll define in the configuration file for the licensing service; in this

particular scenario, the Redis host and port. The following listing shows the code for this class.

Listing 10.12 Setting up the `ServiceConfig` class with Redis data

```
package com.optimagrowth.license.config;
//Imports removed for conciseness

@Component @Getter
public class ServiceConfig{

//Some code removed for conciseness

   @Value("${redis.server}")
      private String redisServer="";

   @Value("${redis.port}")
      private String redisPort="";
}
```

The Spring Cloud Config service repository defines the following host and port for the Redis server in the /configserver/src/main/resources/config/licensing-service.properties file:

```
redis.server = localhost
redis.port = 6379
```

> **NOTE** We use Docker to run all the examples. If you want to run this example locally, you need to install Redis on your computer. But if you use Docker, you'll find the docker-compose.yml file up to date with the Redis container at the following link: https://github.com/ihuaylupo/manning-smia/tree/master/chapter10/docker.

DEFINING THE SPRING DATA REDIS REPOSITORIES

Redis is a key-value data store that acts like a big, distributed, in-memory HashMap. In the simplest case, it stores and looks up data with a key. It doesn't have any sophisticated query language to retrieve data. Its simplicity is its strength, and one of the reasons why so many developers have adopted it for use in their projects.

Because we're using Spring Data to access our Redis store, we need to define a repository class. As may you remember from early on in the first chapters, Spring Data uses user-defined repository classes to provide a simple mechanism for a Java class to access our Postgres database without having to write low-level SQL queries. For the licensing service, we'll define two files for our Redis repository. The first file is a Java interface that will be injected into any of the licensing service classes that need to access Redis. The following listing shows the /licensing-service/src/main/java/com/optimagrowth/license/repository/OrganizationRedis-Repository.java interface.

Listing 10.13 `OrganizationRedisRepository` defines methods used to call Redis

```
package com.optimagrowth.license.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.optimagrowth.license.model.Organization;

@Repository
public interface OrganizationRedisRepository extends
     CrudRepository<Organization,String>{
}
```

By extending from `CrudRepository`, `OrganizationRedisRepository` contains all the CRUD (Create, Read, Update, Delete) logic used for storing and retrieving data from Redis (in this case). The second file is the model we'll use for our repository. This class is a POJO containing the data that we'll store in our Redis cache. The /licensing-service/src/main/java/com/optimagrowth/license/model/Organization.java class is shown in the following listing.

Listing 10.14 Organization model for Redis hash

```
package com.optimagrowth.license.model;

import org.springframework.data.redis.core.RedisHash;
import org.springframework.hateoas.RepresentationModel;
import javax.persistence.Id;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Getter @Setter @ToString
@RedisHash("organization")
public class Organization extends RepresentationModel<Organization> {

    @Id
    String id;
    String name;
    String contactName;
    String contactEmail;
    String contactPhone;
}
```

Sets the name of the hash in the Redis server where the organization data is stored

One important thing to note from the code in listing 10.14 is that a Redis server can contain multiple hashes and data structures within it. We therefore need to tell Redis the name of the data structure we want to perform the operation against in every interaction with Redis.

USING REDIS AND THE LICENSING SERVICE TO STORE AND READ ORGANIZATION DATA

Now that we have the code in place to perform operations with Redis, we can modify our licensing service so that every time the licensing service needs the organization data, it checks the Redis cache before calling the organization service. You'll find the logic for doing this in the `OrganizationRestTemplateClient` class in the class file /service/src/main/java/com/optimagrowth/license/service/client/OrganizationRestTemplateClient.java. The next listing shows this class.

Listing 10.15 Implementing cache logic with `OrganizationRestTemplateClient`

```
package com.optimagrowth.license.service.client;
//Imports removed for conciseness

@Component
public class OrganizationRestTemplateClient {
@Autowired
RestTemplate restTemplate;
@Autowired
OrganizationRedisRepository redisRepository;
private static final Logger logger =
     LoggerFactory.getLogger(OrganizationRestTemplateClient.class);

private Organization checkRedisCache(String organizationId) {
    try {
       return redisRepository
                .findById(organizationId)
                .orElse(null);
    }catch (Exception ex){
       logger.error("Error encountered while trying to retrieve
        organization{} check Redis Cache.  Exception {}",
        organizationId, ex);
       return null;
    }
}
private void cacheOrganizationObject(Organization organization) {
    try {
       redisRepository.save(organization);
    }catch (Exception ex){
       logger.error("Unable to cache organization {} in
        Redis. Exception {}",
        organization.getId(), ex);
    }
}
public Organization getOrganization(String organizationId){
    logger.debug("In Licensing Service.getOrganization: {}",
        UserContext.getCorrelationId());

    Organization organization = checkRedisCache(organizationId);
    if (organization != null){
      logger.debug("I have successfully retrieved an organization
        {} from the redis cache: {}", organizationId,
        organization);
```

Autowires OrganizationRedisRepository in OrganizationRestTemplateClient

Tries to retrieve an Organization class with its organization ID from Redis

Saves the organization in Redis

If you can't retrieve data from Redis, calls the organization service to retrieve the data from the source database to later save it in Redis

```
      return organization;
    }
    logger.debug("Unable to locate organization from the
        redis cache: {}.",organizationId);
    ResponseEntity<Organization> restExchange =
       restTemplate.exchange(
        "http://gateway:8072/organization/v1/organization/
        {organizationId}",HttpMethod.GET,
        null, Organization.class, organizationId);

    organization = restExchange.getBody();
    if (organization != null) {
      cacheOrganizationObject(organization);
    }
      return restExchange.getBody();
   }
}
```

The `getOrganization()` method is where the call to the organization service takes place. Before we make the actual REST call, we need to retrieve the `Organization` object associated with the call from Redis using the `checkRedisCache()` method.

If the organization object in question is not in Redis, the code returns a `null` value. If a `null` value is returned from the `checkRedisCache()` method, the code invokes the organization service's REST endpoint to retrieve the desired organization record. If the organization service returns an organization, the returned organization object is cached using the `cacheOrganizationObject()` method.

> **NOTE** Pay close attention to exception handling when interacting with the cache. To increase resiliency, we never let the entire call fail if we cannot communicate with the Redis server. Instead, we log the exception and let the call through to the organization service. In this particular case, caching is meant to help improve performance, and the absence of the caching server shouldn't impact the success of the call.

With the Redis caching code in place, using Postman, we can select the licensing service to view the logging messages. If we make two back-to-back GET requests on the licensing service endpoint, `http://localhost:8072/license/v1/organization e839ee96-28de-4f67-bb79-870ca89743a0/license/279709ff-e6d5-4a54-8b55-a5c37542025b`, we'll see the following two output statements in our log:

```
licensingservice_1       | DEBUG 1 --- [nio-8080-exec-4]
c.o.l.s.c.OrganizationRestTemplateClient : Unable to locate organization from
the redis cache: e839ee96-28de-4f67-bb79-870ca89743a0.

licensingservice_1       | DEBUG 1 --- [nio-8080-exec-7]
c.o.l.s.c.OrganizationRestTemplateClient : I have successfully retrieved an
organization e839ee96-28de-4f67-bb79-870ca89743a0 from the redis cache:
Organization(id=e839ee96-28de-4f67-bb79-870ca89743a0, name=Ostock,
contactName=Illary Huaylupo, contactEmail=illaryhs@gmail.com,
contactPhone=888888888)
```

The first output from the console shows the first time we tried to access the licensing service endpoint for the organization e839ee96-28de-4f67-bb79-870ca89743a0. The licensing service checked the Redis cache and couldn't find the organization record it was looking for. The code then called the organization service to retrieve the data. The next output shows that when you entered the licensing service endpoint a second time, the organization record is now cached.

10.4.2 Defining custom channels

Previously we built our messaging integration between the licensing and organization services to use the default output and input channels that are packaged with the `Source` and `Sink` interfaces in Spring Cloud Stream. However, if we want to define more than one channel for our application, or we want to customize the names of our channels, we can define our own interface and expose as many input and output channels as our application needs.

To create a custom channel, we'll call `inboundOrgChanges` in the licensing service. You can define the channel with the `CustomChannels` interface found in /licensing-service/src/main/java/com/optimagrowth/license/events/CustomChannels.java, as shown in the following listing.

Listing 10.16 Defining a custom input channel for the licensing service

```
package com.optimagrowth.license.service.client;
//Imports removed for conciseness
package com.optimagrowth.license.events;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

public interface CustomChannels {

    @Input("inboundOrgChanges")
    SubscribableChannel orgs();

}
```

Names the channel

Returns a SubscribableChannel class for each channel exposed by @Input

The key takeaway from listing 10.16 is that for each custom input channel we want to expose, we define a method with `@Input` that returns a `SubscribableChannel` class. We then use `@OutputChannel` before the method that will be called if we want to define output channels for publishing messages. In the case of an output channel, the defined method returns a `MessageChannel` class instead of the `SubscribableChannel` class used with the input channel. Here's the call to `@OutputChannel`:

```
@OutputChannel("outboundOrg")
MessageChannel outboundOrg();
```

Now that we have a custom input channel, we need to modify two more things to use it in the licensing service. First, we need to modify the licensing service to map the Kafka

topic's custom input channel name in the licensing configuration file. The following listing shows this change.

Listing 10.17 Modifying the licensing service to use our custom input channel

```
//Parts removed for conciseness
spring.cloud.stream.bindings.inboundOrgChanges.destination=
    orgChangeTopic
spring.cloud.stream.bindings.inboundOrgChanges.content-type=
    application/json
spring.cloud.stream.bindings.inboundOrgChanges.group=
     licensingGroup
spring.cloud.stream.kafka.binder.zkNodes=
    localhost
spring.cloud.stream.kafka.binder.brokers=
    localhost
```

Next, we need to inject the `CustomChannels` interface previously defined into a class that's going to use it to process messages. For the distributed caching example, we've moved the code for handling an incoming message to the `OrganizationChangeHandler` licensing service class. You'll find the source for this class in /licensing-service/src/main/java/com/optima-growth/license/events/handler/OrganizationChangeHandler.java.

The following listing shows the message handling code that we'll use with the `inboundOrgChanges` channel we just defined.

Listing 10.18 Processing an organization change with the new custom channel

```
package com.optimagrowth.license.events.handler;
//Imports removed for conciseness

@EnableBinding(CustomChannels.class)
public class OrganizationChangeHandler {

private static final Logger logger =
    LoggerFactory.getLogger(OrganizationChangeHandler.class);

private OrganizationRedisRepository
           organizationRedisRepository;

@StreamListener("inboundOrgChanges")
public void loggerSink(
        OrganizationChangeModel organization) {

    logger.debug("Received a message of type " +
          organization.getType());
    logger.debug("Received a message with an event {} from the
         organization service for the organization id {} ",
         organization.getType(), organization.getType());
    }
}
```

Moves the @EnableBindings out of Application.java and into OrganizationChangeHandler. This time instead of using the Sink class, we use CustomChannels as the parameter to pass.

Injects OrganizationRedisRepository into OrganizationChangeHandler to allow for CRUD operations

Inspects the action that the data undertakes and then reacts accordingly

The utility class doing all the metadata extraction work

Now, let's create an organization and then find it. We can do this using the following two endpoints. (Figure 10.9 shows the console output of these calls with the `OrganizationChangeHandler` class.)

```
http://localhost:8072/organization/v1/organization/
http://localhost:8072/organization/v1/organization/d989f37d-9a59-4b59-b276-
     2c79005ea0d9
```

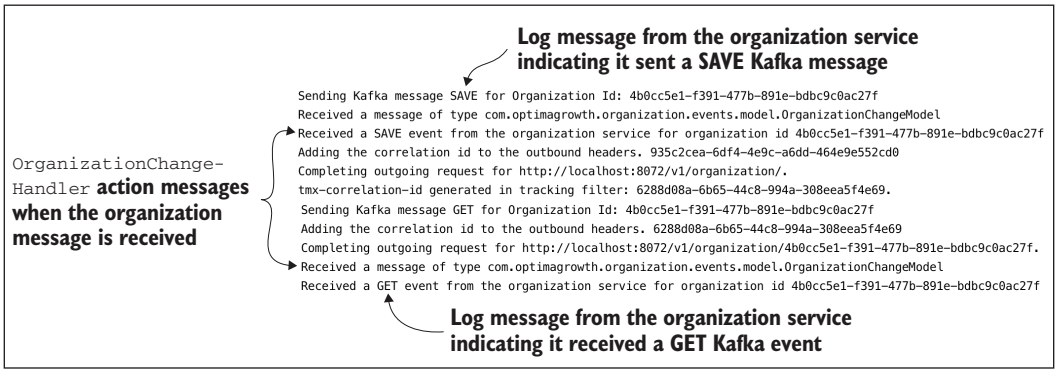

Figure 10.9 The console shows the message from the organization service that was sent and then received.

Now that we know how to use Spring Cloud Stream and Redis, let's continue with the next chapter, where we will see several techniques and technologies to create a distributed tracing using Spring Cloud.

Summary

- Asynchronous communication with messaging is a critical part of microservices architecture.
- Using messaging within our applications allows our services to scale and become more fault tolerant.
- Spring Cloud Stream simplifies the production and consumption of messages by using simple annotations and abstracting away platform-specific details of the underlying message platform.
- A Spring Cloud Stream message source is an annotated Java method that's used to publish messages to a message broker's queue.
- A Spring Cloud Stream message sink is an annotated Java method that receives messages from a message broker's queue.
- Redis is a key-value store that you can use as both a database and a cache.