17

Clustering

This chapters covers

- Clustered HTTP
- Clustered Camel routes in active/passive and active/active modes
- Clustered messaging with JMS and Kafka
- Clustered caches
- Clustered scheduling
- Calling clustered services using the Service Call EIP

"How do I set up a highly available cluster with Camel?" That's a simple question that doesn't have a straightforward answer. You can ask the same question about a Java application server such as Apache Tomcat or WildFly and get a well-documented answer. Unlike Camel, these application servers are within a well-defined scope. They often support only a few protocols such as HTTP (Servlet) and messaging (JMS). Camel, on the other hand, speaks a lot more protocols, as you can tell by the many Camel components.

For example, a Camel application may expose a REST service and place its content into a database or filesystem. Or it may accept HL7 messages over TCP and route to a JMS broker. Or you may use Camel to stream from Kafka topics to a cloud service running on AWS. The answer to the question "How do I set up a highly available cluster with Camel?" depends on what you do with Camel.

To understand clustering support in Camel, we have to go back to the beginning. Camel was created in 2007 as a lightweight integration framework. Back then, Camel was just a set of JARs that you added to your classpath, so you could then run Camel in any JVM. Typically, you'd embed Camel into an existing application server or ESB.

At that time, any clustering was beyond the scope of Camel's core module. *Clustering* was intended to come from a third-party (such as application servers and ESBs) or from the Camel components that facilitate clustering.

Historically, Camel came from the ESB world—from Apache ServiceMix. ServiceMix would then bring support for clustering to its service bus. That clustering was based on messaging and came from Apache ActiveMQ.

Fast-forward to today, and the landscape has changed. Clustering in Camel is mainly driven by the following:

- *Camel components*—Camel components with native clustering support
- *Clustered route policies*—Running Camel routes in active/passive mode (also called *master/slave*)
- *Container-based infrastructure*—Clustering by the infrastructure (covered in chapter 18)

This chapter covers various use cases illustrating how to cluster Camel with some of the most common protocols such as HTTP, files, JMS, Kafka, and clustered caches. We'll also show you how to set up clustered Camel routes in master/slave mode, ensuring that only one route is active at any time. The last section covers the Service Call EIP, which is used for calling services in a clustered and distributed system.

Let's start with one of the simplest clustering techniques: HTTP clustering using load balancing.

17.1 Clustered HTTP

A dozen or so of the Camel components speak HTTP—such as CXF, CXF-RS, Jetty, Undertow, and Servlet, to name a few. These components can all be used in Camel to define HTTP-based services. And because HTTP is prolific, it's probably the most easily understood clustering scenario we'll cover.

HTTP has two modes: stateless and stateful. *Stateless* is by far the most commonly used and is our recommended approach. It scales well, requires no coordination between the nodes in the cluster, and is much eas-

ier to set up. You run as many instances of the HTTP service as you need, and the services can be colocated or spread across data centers. The clustering isn't provided by Camel or the HTTP service, but is done with the help of a load balancer in front of each HTTP service, as illustrated in figure 17.1.

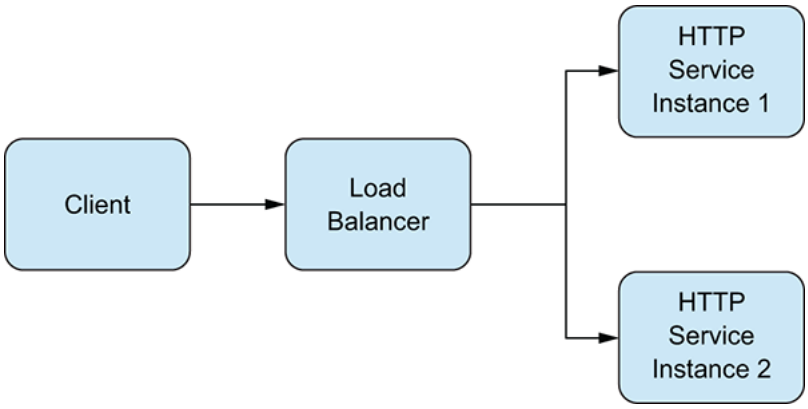

Figure 17.1 The load balancer fronts the HTTP services and directs traffic from clients to available instances in the cluster.

If any instances of the HTTP service goes down, the load balancer will redirect traffic to one of the other available instances, as illustrated in figure 17.2.

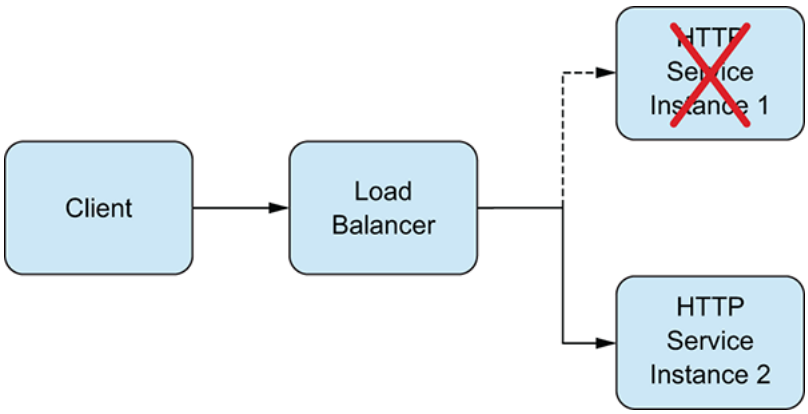

Figure 17.2 Instance 1 of the HTTP service is down, and the load balancer redirects traffic to the available instances (which, in this example, is instance 2).

Using a load balancer requires somehow knowing which instances are currently available so the load balancer can direct traffic to only services that are running. To accommodate this, you often have to implement some kind of heartbeat mechanism—for example, an HTTP endpoint with a *happy page* that returns a positive HTTP status code if the service is ready and alive, and an error code otherwise. The load balancer should be configured to periodically call those HTTP endpoints to obtain the latest status of the instances in the cluster.

Modern infrastructures such as cloud or container platforms often have this mechanism baked into their platform. Chapter 18 covers Kubernetes, which includes readiness and liveness probes as part of its service discovery and load-balancing features.

Stateful HTTP web applications have fallen out of favor in the last decade or so. By *web applications*, we mean Java web applications running on Java EE application servers. The support for clustering came out of the box from the application server. Often all you'd have to do is set up the application servers in a cluster and possibly turn on HTTP session replication.

The usefulness of the HTTP session replication is also questionable to Camel users, as it requires building web applications and storing state in the HTTP session. Camel provides better alternatives for stateful applications, which we talk about in section 17.5.

The next section takes us back in time to talk about an old technique for exchanging data using files or FTP. How do you poll files in a reliable way, using a highly available cluster? It's harder than you may think. How do you ensure that an active node is always polling files? How do you perform failover in case a node dies? These are great questions, and the answers all start with how to cluster Camel routes.

17.2 Clustered Camel routes

When consuming files using the file or FTP component, you can use a couple of strategies for a clustered setup:

- *Active/passive*—Only a single master polls.
- *Active/active*—All nodes poll concurrently.

These two modes will be covered in this section, starting with active/passive mode.

17.2.1 Active/passive mode

In active/passive mode, you have a single master instance polling for files, while all the other instances (slaves) are passive. For this strategy to work, some kind of locking mechanism must be in use to ensure that only

the node holding the lock is the master and all other nodes are on standby. Figure 17.3 illustrates active/passive mode.

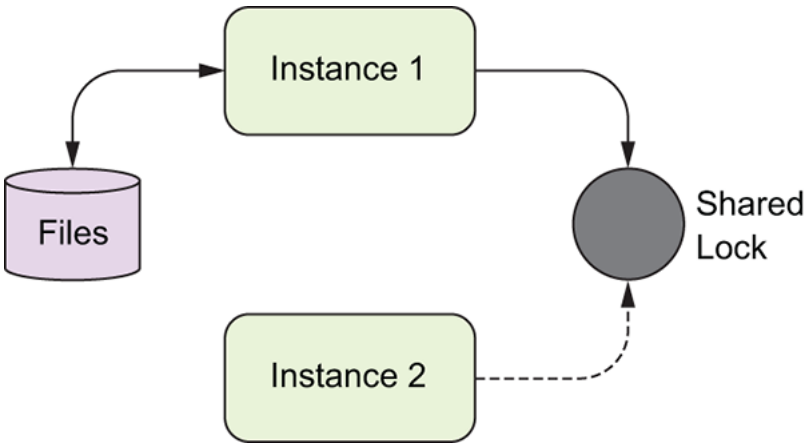

Figure 17.3 Clustering Camel instances using a shared lock in active/passive mode: instance 1 holds the lock and is the master—and therefore is the only instance that's active and polls the files. The other instances are slaves that wait to obtain the lock in order to become the master.

If the master node is shut down or unexpectedly dies, one of the slave instances will obtain the lock and become the new master. Only one node at most is actively polling for files; no concurrent consumers are racing to poll the same files.

When polling files, you often won't need the highest possible performance; for example, if you need to receive the files in batches only once a day, the cluster can handle that with a single instance, which fits perfectly with active/passive mode. But what if you're receiving a continued stream of many files and need to process these files by multiple nodes in the cluster in an active/active mode?

17.2.2 Active/active mode

In active/active mode, all nodes compete concurrently to pick up and process files. This strategy works by having some kind of clustered repository that keeps track of which files are currently being processed by the nodes in the cluster. Figure 17.4 illustrates active/active mode.

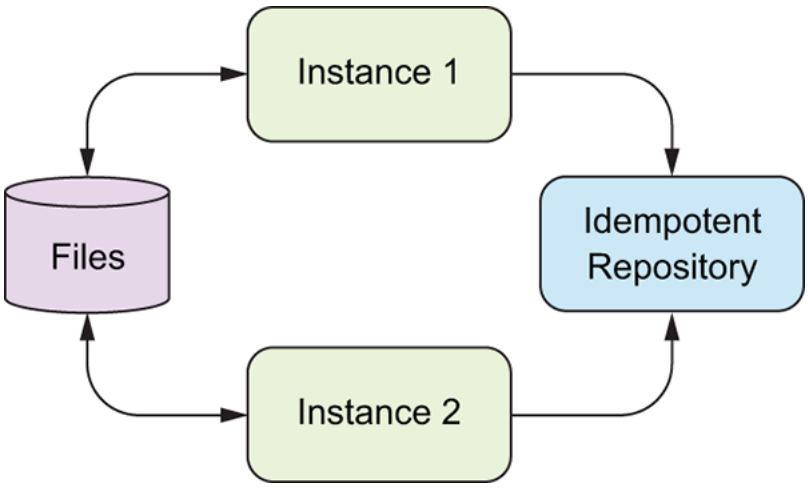

Figure 17.4 Clustering Camel instance using a distributed idempotent repository to track which files are being processed by which nodes. This ensures that a file is being picked up and processed by only one node at most.

When using active/active mode, you can't, for example, use route policy to control the clustering, because it's best suited for active/passive mode. Instead, the file and FTP components allow you to use a distributed idempotent repository, which ensures that only one node is granted access to process the file. Does that mean only one file is processed at any given time? No: all the active nodes compete concurrently to pick up and process files, so each node could pick up a different file, all of which could be processed at the same time. This arrangement yields higher performance, because you can process as many files concurrently as there are nodes.

We covered this use case previously, in chapter 12, section 12.5.3. You can read that section again and review the example if you want to practice working with active/active mode. The remainder of this section focuses on active/passive mode.

How do you set up this active/passive mode with a shared lock? You need a shared lock that supports clustering, and then you use that with a Camel route policy.

You can use any of the following Camel components that support clustered route policy: camel-consul, camel-etcd, camel-hazelcast, or camel-zookeeper. The following sections use Hazelcast, Consul, and ZooKeeper in the examples.

17.2.3 Clustered active/passive mode using Hazelcast

To represent a cluster with two nodes, we have two almost identical source files in `ServerFoo` and `ServerBar`. The following listing shows the source code for `ServerBar`.

Listing 17.1 Hazelcast route policy in master/slave mode

```
public class ServerBar {
  private Main main;

  public static void main(String[] args) throws Exception {
    ServerBar bar = new ServerBar();
    bar.boot();
  }

  public void boot() throws Exception {
    HazelcastInstance hz = Hazelcast.newHazelcastInstance();   (1)
```

(1) Creates embedded Hazelcast instance

```
    HazelcastRoutePolicy routePolicy = new HazelcastRoutePolicy(hz);   (2)
```

(2) Creates route policy

```
    routePolicy.setLockMapName("myLock");   (3)
```

(3) Configures route policy

```
    routePolicy.setLockKey("myLockKey");   (3)
    routePolicy.setLockValue("myLockValue");   (3)
    routePolicy.setTryLockTimeout(5, TimeUnit.SECONDS);   (3)
```

```
    main = new Main();
  main.bind("myPolicy", routePolicy);   ❹
```

---

❹

Registers route policy in Camel registry

```
  main.addRouteBuilder(new FileConsumerRoute("Bar", 100));   ❺
```

---

❺

Adds route and runs the application

```
  main.run();   ❺
    }
}
```

This example runs as a standalone Camel application using a `main` method. At first, you create and embed a Hazelcast instance ❶ that by default reads its configuration from the root classpath using the name hazelcast.xml. The route policy is then created ❷ and configured ❸. It's important to configure the route policy with the same lock name in all the nodes so they're using the same shared lock. The remainder of the code uses Camel `Main` to easily configure with the route policy ❹ and run as a standalone Camel application ❺.

The Camel route ❺ that uses the route policy is shown in the following listing.

Listing 17.2 Camel route using clustered Hazelcast route policy

```
public class FileConsumerRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("file:target/inbox?delete=true")
   .routePolicyRef("myPolicy")   ❶
```

❶

Configures route policy on the route

```
        .log(name + " - Received file: ${file:name}")
    .delay(delay)    ❷
```

❷

Delays processing the file so we humans have a chance to see what happens

```
        .log(name + " - Done file:     ${file:name}")
        .to("file:target/outbox");
    }
}
```

The Camel route is a file consumer that picks up files from the target/inbox directory (which is the shared directory in this example). The route is configured by the route policy with the value `myPolicy` ❶, which corresponds to the name that the route policy was given in listing 17.1. When a file is being processed, it's logged and delayed ❷, so the application runs a bit slower, and you can better see what happens.

You can try this example from the chapter17/cluster-file-hazelcast directory by running the following Maven goals for separate shells so they run at the same time (you can also run the goals from your IDE, because it's a standard Java main application—the IDE typically has a right-click menu to run Java applications):

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

When you start the second application, Hazelcast should log the cluster state, as shown here:

```
Members [2] {
  Member [localhost]:5701
```

```
    Member [localhost]:5702 this
}
```

Here you can see two members in the cluster that are linked together with TCP via ports 5701 and 5702.

If you copy a bunch of files to the target/inbox directory, only the master node will pick up and process the files. While the files are being processed, you can try to shut down or kill the master, which should trigger a failover. The slave node then becomes the master and starts processing the files.

The example can also use Infinispan or Consul instead of Hazelcast. Infinispan and Hazelcast are similar in usage. You can find an example of using Infinispan in the chapter17/cluster-file-infinispan directory, which has further instructions in the readme file for downloading and installing Infinispan (which must be done prior to running the example). The next section covers using Consul.

17.2.4 Clustered active/passive mode using Consul

HashiCorp Consul is a distributed service registry that also can be used as a clustered lock. You can use Consul in the previous example by making only a few changes. All you have to do is to add camel-consul as a dependency to the project and then use `ConsulRoutePolicy`, as shown here:

```
ConsulRoutePolicy routePolicy = new ConsulRoutePolicy();
routePolicy.setServiceName("myLock");
routePolicy.setTtl(5);
```

You can find this example with the source code in the chapter17/cluster-file-consul directory. To try the example, you need to run Consul, which can be done using Docker:

```
docker run -it --rm -p 8500:8500 consul
```

Consul is then accessible using HTTP on port 8500, which is what the camel-consul client is using. The Consul web console is also available at http://localhost:8500/ui, and you can view that while running the exam-

ple. Then you can run the two Camel applications by starting two shells and running the following Maven goals:

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

By copying files to the target/inbox directory, you should see that only one of the Camel applications will pick up and process the files. You can then try to shut down or kill one of the applications and see that Consul fails over to a new master node. From the web console, you should be able to see that Consul also provides details about the distributed lock under the services with the name `myLock`.

TIP Camel registers route policies for Hazelcast, Infinispan, and Consul in JMX, which allows you to see at runtime which node is the master and which are the slaves.

Yet another Camel component supports clustering. The zookeeper-master component uses ZooKeeper to ensure that only a single consumer in a cluster consumes from a given endpoint with automatic failover if that JVM dies.

17.2.5 Clustered active/passive mode using ZooKeeper

In principle the zookeeper-master component works the same way as Hazelcast and Consul. The ZooKeeper cluster is a distributed and highly available registry that's also capable of orchestrating nodes to conduct in master/slave mode. When your Camel applications are starting, they connect to the ZooKeeper cluster. ZooKeeper then elects one of them as the master, and all other nodes will be slaves on standby. If the master node dies or is shut down, one of the remaining slave nodes is elected as the new master.

The camel-zookeeper-master JAR provides this functionality both as a Camel component and route policy.

The source code contains two examples. The first example is located in the chapter17/cluster-zookeeper-master directory.

In the cluster, we have yet again two almost identical nodes in the source code as `ServerFoo` and `ServerBar`. The following listing shows the source code for the Camel route that is used by both `ServerFoo` and `ServerBar`.

Listing 17.3 Camel route using master component for master/slave mode

```
String url = "file:target/inbox?delete=true";    ❶
```

❶ Consumes files from a shared directory

```
from("zookeeper-master:myGroup:" + url)    ❷
```

❷ zookeeper-master component to use master/slave mode

```
    .log(name + " - Received file: ${file:name}")
    .delay(delay)
    .log(name + " - Done file:     ${file:name}")
    .to("file:target/outbox");
```

The Camel route is a simple route that consumes from a shared directory ❶. Notice that the route starts from `"zookeeper-master:myGroup:" + url` ❷. The route is consuming from the zookeeper-master component in the cluster group with the name `myGroup`. This lets the zookeeper-master component be in control of the lifecycle of the intended consumer, which is the file consumer. Only if the zookeeper-master component becomes the master will it start up the file consumer.

Before you can try this example, you need to configure the zookeeper-master component, which is done as shown here:

```
MasterComponent master = new MasterComponent();
master.setZooKeeperUrl("localhost:2181");    ❶
```

❶

Configures URL to ZooKeeper cluster(s)

```
main.bind("zookeeper-master", master);    ❶
```

❷

Registers component in Camel registry

```
```

If you're using XML DSL, you configure the component using `<bean>` style, as shown here:

```
<bean id="zookeeper-master"
       class="org.apache.camel.component.zookeepermaster.MasterComponent
  <property name="zooKeeperUrl" value="localhost:2181"/>
</bean>
```

As you run this example locally, the URL is set to `localhost:2181`. In a real production use case, you'd configure the URL to your ZooKeeper master nodes. You can separate multiple hostnames with commas:

```
keeper1:2181,keeper2:2181,keeper3:2181
```

Trying the example

You can try this example from the source code in chapter17/cluster-zookeeper-master. First you need to start ZooKeeper, which can easily be run using Docker:

```
docker run -it --rm -p 2181:2181 -d zookeeper
```

Then you can start the Foo and Bar server from two shell commands:

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

Copy a bunch of files to the target/inbox directory, where only the master node should pick up and process the files. The node that becomes the master will log accordingly:

```
INFO  Elected as master. Consumer started: file://target/inbox?delete=tr
```

While the files are being processed, you can try to shut down or kill the master (i.e. Foo, or Bar servers), which should trigger a failover. The slave node then becomes the master and starts processing the files.

NOTE The source code also contains an example using a route policy instead of a component. You can find this example in the chapter17/cluster-zookeeper-master-routepolicy directory.

What's so special about this zookeeper-master component? It works not only with files but with any Camel consumer endpoint. Because it works in master/slave mode only, the number of use cases is limited. You can use it when you must have at most one active consumer running in your cluster; that's the use case it solves.

Summary of active/passive vs. active/active mode

We've now covered two modes of clustering routes that consume files. The active/passive mode is the easiest and often best choice to process files if a single node can keep up with the volume of files. If you must process a lot of files in a streaming fashion, you may have to use active/active mode to concurrently process the files by all nodes in the cluster. This adds more complexity, because now Camel must use a distributed idempotent repository to orchestrate which files are processed by which nodes.

Camel can also be used with clustering with other protocols such as messaging. The following two sections cover two popular messaging platforms: JMS and Apache Kafka.

17.3 Clustered JMS

JMS clustering with Camel is one of the most difficult scenarios. Many forces are at play, and you have many facets to consider and customize to tailor Camel and the JMS broker and network topology to the business needs.

Most of the clustering functionality with JMS doesn't come from Apache Camel, but from the messaging broker in both its client and server software. The Camel JMS component uses the JMS API and has no concept of clustering in regards to high availability. This section uses Apache ActiveMQ as the message broker, but if you're using a different broker, what you read here is still likely relevant because most message brokers offer similar functionality; certainly all of them support clustering and claim to be highly available.

TIP You can find valuable information about Apache ActiveMQ in *ActiveMQ in Action* by Bruce Snyder et al. (Manning, 2011).

17.3.1 Client-side clustering with JMS and ActiveMQ

When using ActiveMQ, clustering starts from the client point of view, where you can easily turn on support for clustering. This is done using the failover protocol in the URL connection to the broker when configuring the connection factory to ActiveMQ. The following piece of XML illustrates this:

```
<bean id="jmsConnectionFactory"
       class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL"
             value="failover:tcp://localhost:61616,tcp://localhost:5151
</bean>
```

In this code, the connection factory has been configured using the failover protocol to connect two ActiveMQ brokers running on `tcp:localhost:616161` and `tcp:localhost:51515`. In a real production system, those broker URLs would refer to separate hostnames where the

brokers would be running, but because the code is from the book's source code, they use `localhost` to allow you to run this from your computer.

You can find this example in the chapter17/cluster-jms-client directory. Before you can run the example, you need to download and run two instances of the ActiveMQ broker on the computer. The example has step-by-step instructions in the accompanying readme file.

Upon running this example, you should notice that the client can fail over between the brokers if you shut one of them down. We've pasted output from the client's log showing what happens when we shut down the master ActiveMQ broker (running on `localhost:61616`). The client logs a series of WARN messages stating the transport error, followed by an INFO log message stating that the client was successfully able to fail over to the new master broker that runs on `localhost:51515`:

```
INFO  Successfully connected to tcp://localhost:61616
INFO  Sent message: Time is now Sun Apr 20 21:26:56 CET 2017
WARN  Transport (tcp://localhost:61616) failed,
       attempting to automatically reconnect
...
...
INFO  Successfully reconnected to tcp://localhost:51515
INFO  Sent message: Time is now Sun Apr 20 21:27:02 CET 2017
```

This is the easiest setup with ActiveMQ, but it does have its restrictions. The ActiveMQ brokers must be using a shared filesystem that supports file locks. The filesystem must be highly available as well, and therefore you must use a SAN or a hardware appliance providing this kind of functionality.

HIGHLY AVAILABLE MESSAGING IS COMPLEX

Now you get to the point where you have many facets to consider regarding how to set up a highly available messaging system that your Camel applications can use in the cluster. This topic is complex, and having a full overview takes years for even experts in the field. We advise you to spend diligent time reading the documentation from the broker vendor and other sources. A good blog entry about clustering JMS message brokers is from Josh Reagan: http://joshdreagan.github.io/2016/07/28/ha_deployments_with_fuse.

Depending on what message broker you use, you should consult its documentation to determine the specific cluster capabilities it provides. For example, Apache ActiveMQ has many features beyond JMS that make it flexible, such as virtual topics. Using virtual topics, you can better cluster, fail over, and scale out than with regular JMS topics.

In recent years, Apache Kafka has become popular. Camel also works great with Kafka, and together, they work with clustering as well.

17.4 Clustered Kafka

Apache Kafka describes itself as a distributed streaming platform that's used for building real-time data pipelines and streaming applications. This section shows you how to stream data from and to Kafka using the camel-kafka component. Kafka is distributed by nature, and you can fairly easily cluster Camel to reliably consume from Kafka with automatic failover.

This section uses a simple example to demonstrate how to cluster Camel with Kafka. You can run this example locally on your computer. The example, which is in the chapter17/cluster-kafka directory, has four players, as depicted in figure 17.5.

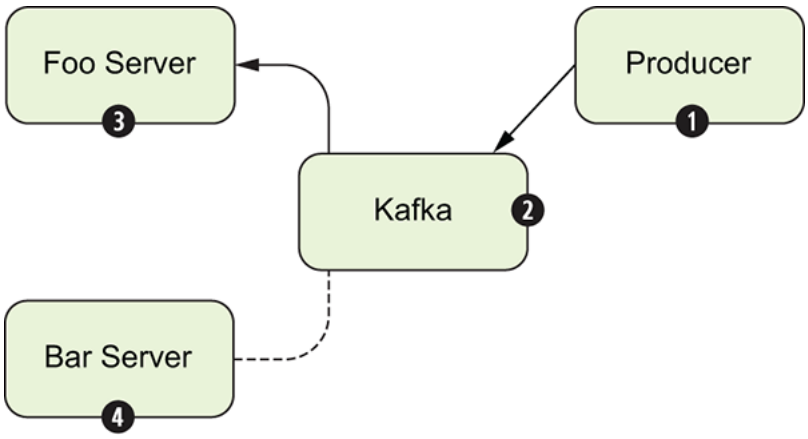

Figure 17.5 The producer ❶ continuously streams data to the Kafka broker ❷. Two Camel servers, Foo ❸ and Bar ❹, run in clustering mode to consume the events in master/slave mode.

The producer is implemented using a little Camel route that's triggered by a timer that generates a random word that's sent to a Kafka topic.

To use the camel-kafka component, you need to set up the URL to the Kafka broker:

```
KafkaComponent kafka = new KafkaComponent();
kafka.setBrokers("localhost:9092");    ❶
```

❶ The URL to the Kafka broker(s)

```
getContext().addComponent("kafka", kafka);    ❷
```

❷ Adds component to CamelContext

In this example, you run Kafka locally and therefore specify the URL as `localhost:9092` ❶. Because the example uses Java DSL, you need to add the component to `CamelContext` ❷. If you're using XML DSL, you can configure the component as follows:

```
<bean id="kafka" class="org.apache.camel.component.kafka.KafkaComponent"
  <property name="brokers" value="localhost:9092"/>
</bean>
```

TIP In a production scenario, you'd set up Kafka in a clustered setup, and therefore the URL to the brokers can contain multiple hostnames separated by commas—for example: `kafka.setBrokers ("kafka1:9092,kafka2:9092,kafka3:9092");.`

The Camel route used by the producer is only four lines of code:

```
from("timer:time?period=100")
    .bean(new WordBean())
  .to("kafka:words")  ❶
```

❶ Sends message to Kafka topic: words

```
    .to("log:words?groupInterval=1000");
```

When Camel sends a message to Kafka, you need to specify which topic to send the message to. In the example, we send a message to a topic named `words` ❶.

ABOUT KAFKA MESSAGE KEY

A Kafka message may contain a key that typically has a semantic meaning, such as a customer ID or item number. The key doesn't have to be unique. It could be specified in the endpoint URI, such as `kafka:words?key=mykey`. You can also specify the key as a header, which allows dynamic values. For example, you can use a header with the customer ID `setHeader("kafka.KEY", header("customerId"))`.

Kafka doesn't require having active consumers running, so you can start the example and let the producer send messages to Kafka. To run Kafka, download Kafka from the Apache Kafka website and run it according to its instructions. You can find details in the readme file with the source code of this example.

In figure 17.5, the two consumers are the Foo ❸ and Bar ❹ servers that you also implement using Camel, as shown in the following listing.

Listing 17.4 Consuming from Kafka using Camel with the XML DSL

```
<bean id="kafka" class="org.apache.camel.component.kafka.KafkaComponent"
  <property name="brokers" value="localhost:9092"/>    ❶
```

❶ Configures URL to the Kafka broker(s)

```
</bean>

<camelContext id="foo" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="kafka:words?groupId=mygroup"/>    ❷
```

❷ Consumes from the Kafka topic: words

```
    <log message="Foo got word ${body}"/>
  </route>
</camelContext>
```

The Kafka component must be configured with the URL to the Kafka broker, which is done using the `<bean>`-style configuration ❶. The Camel route is consuming from the Kafka topic named `words` ❷, and each message is then logged. The option `groupId=mygroup` is used to group consumers into the same group across the cluster. The group name is global across the cluster, so be sure to use the correct name.

You're now ready to run the example from the chapter17/cluster-kafka directory and start the producer and the first of the two consumers, which is the Foo server. This is done using separate command shells and executing the following Maven goals:

```
mvn compile exec:java -P producer
```

That starts the producer, which starts sending messages to Kafka. The consumer is started as follows:

```
mvn compile exec:java -P foo
```

When the consumer is up and running, you should see the receive messages being logged:

```
INFO  Foo got word #1-Dude
INFO  Foo got word #2-Rocks
INFO  Foo got word #3-Dude
INFO  Foo got word #4-Bad
INFO  Foo got word #5-Bad
INFO  Foo got word #6-Fabric8
```

You haven't yet clustered Camel with Kafka, because you have only one Camel consumer running. Let's see what happens when you start the second consumer:

```
mvn compile exec:java -P bar
```

When the second consumer starts, it joins the Kafka broker as a consumer on the topic named `words` with the group ID `mygroup`. Because a consumer is already running on that same topic with the same group ID, two things happen:

- *Kafka repartitions the connected consumers*—Because you have only two consumers and one partition, only one consumer can receive messages while the other is on standby. You have a master/slave scenario.
- *Whenever Kafka repartitions, it can decide to assign partitions to new consumers*—Either the existing or the new consumer will be assigned the partition. The one who has been assigned will be the only active consumer that receives the messages.

The following snippet shows such a situation; Kafka has reassigned the partition to the second consumer (Bar):

```
INFO  Successfully joined group mygroup with generation 22
INFO  Setting newly assigned partitions [words-0] for group mygroup
INFO  Bar got word #269-Beer
INFO  Bar got word #270-Cool
```

Kafka will also repartition when a consumer is stopped. For example, if you try to stop the active consumer, you should see the other consumer take over. You've achieved clustering Camel with Kafka in a way that works with automatic failover. Figure 17.6 illustrates this failover action.

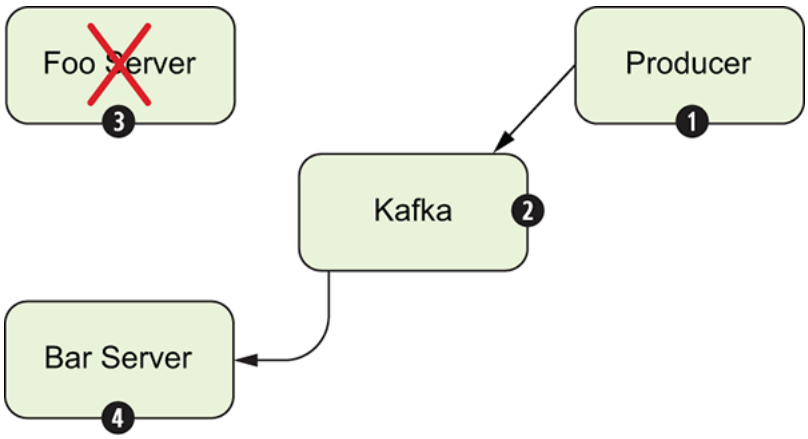

Figure 17.6 Kafka failover in action: the Foo server ❸ was the previously active consumer that received messages from the Kafka topic. But the server was stopped, and Kafka repartitioned the topic among the remaining consumers. Only the Bar server ❹ is left, which then becomes the new active consumer and starts receiving messages.

Kafka is built using a different architecture than traditional message brokers. With JMS messaging, the broker plays a central role in distributing messages fairly and reliably among consumers, and JMS clients have to worry about only sending and receiving messages. Kafka, on the other hand, is client-centric. Clients take over many of the functionalities of traditional brokers, handling duplicate messages and dealing with other kinds of errors.

Next we'll look at one of these client responsibilities that affects running Kafka with Camel. With Kafka, the client must keep track of which messages it has received, known as the *consumer offset*.

17.4.1 Kafka consumer offset

With Kafka, you may get duplicate messages, and why is that? Whenever a consumer consumes from a Kafka topic, the consumer is responsible for keeping track of which messages it has received; this record is called an *offset*. The consumer periodically commits this offset to Kafka so it can re-

sume from this offset in the event of a restart or failover. By default, camel-kafka commits this offset every 5 seconds (autocommit). The offset is also committed when Camel is stopped—for example, when you stop the Camel application gracefully.

TIP From Camel version 2.21 onward you can control consumer offset commits manually as well by using the `KafkaManualCommit` API. See the camel-kafka documentation from the Camel website for more details.

You can see this by starting and stopping the consumers in the example. You should notice that no duplicate messages occur when Camel fails over.

In the following snippet, the Foo server is stopped, and Bar is taking over:

```
INFO  Foo got word #118-Cool
INFO  Foo got word #119-Bad
INFO  Auto commitSync on stop words-Thread 0 from topic words
```

Here are logs from Bar:

```
INFO  Setting newly assigned partitions [words-0] for group mygroup
INFO  Bar got word #120-Hawt
INFO  Bar got word #121-Dude
```

Notice that the last message processed by Foo is number 119. Upon stopping Foo, the offset is committed, which is what that last line means. When Bar takes over, the offset is 119, and therefore Bar can continue where Foo left off (at the next message, 120, and so on).

Okay, this is awesome, so we're all good? Frankly, we're not, because this works reliably only when Kafka consumers are shut down gracefully. A hard stop such as a JVM crash isn't a graceful shutdown, and what happens then? Why not have fun and try this?

17.4.2 Crashing a JVM with a running Kafka consumer

What you want to do now is to let both the Foo and Bar servers run at the same time, kill the JVM that has the active Kafka consumer, and then see what happens when the other consumer is failing over.

First you start the producer so the consumers will have messages to consume from Kafka:

```
mvn compile exec:java -P producer
```

Then you start both consumers:

```
mvn compile exec:java -P foo
mvn compile exec:java -P bar
```

Now you want to kill the running consumer, which can be either the Foo or Bar server. In this example, you'll assume that the Foo server is the active consumer. To kill a JVM, you need its process ID (PID), which you can find using the Java `jps` tool, as shown here:

```
$ jps -m
13138 QuorumPeerMain config/zookeeper.properties
14835 Launcher compile exec:java -P producer
14837 Launcher compile exec:java -P foo
14839 Launcher compile exec:java -P bar
14873 Jps -m
13358 Kafka config/server.properties
```

As you can see, the process ID for the Foo server is 14837, which we can kill using the following:

```
kill -9 14837                    (using OSX or Linux)
taskkill /PID 14837 /F         (using Windows)
```

When the Foo server is killed, it stops processing, and the last log is shown here:

```
INFO  Foo got word #4640-Bad
INFO  Foo got word #4641-Hawt
```

```
Killed: 9
```

You then wait for the Bar server to fail over and start processing the messages. The failover doesn't happen immediately, as it did previously when you stopped the Foo server gracefully. Instead, the Kafka broker has to realize that the consumer has crashed. Every Kafka consumer that's part of a cluster group will use a heartbeat to tell the Kafka brokers that it's alive. But if a consumer hasn't successfully reported a heartbeat within a 10-second time-out (session time-out), the consumer is considered dead. Therefore, it will take 10 seconds or longer before Kafka repartitions the topic and assigns it to the Bar consumer. This is what happens in the following snippets of the Bar server's logs when it starts processing the messages:

```
INFO  Successfully joined group mygroup with generation 4
INFO  Setting newly assigned partitions [] for group mygroup
INFO  (Re-)joining group mygroup
INFO  Successfully joined group mygroup with generation 5
INFO  Setting newly assigned partitions [words-0] for group mygroup
INFO  Bar got word #4627-Donkey
INFO  Bar got word #4628-Camel
INFO  Bar got word #4629-Donkey
INFO  Bar got word #4630-Dude
```

As you can see, the Bar consumer was assigned the partition of the `words` topic and then started receiving the messages.

The first message received by the Bar consumer is `#4627-Donkey`, and the last message sent by the Foo server before it crashed is `#4641-Hawt`. You'll receive duplicate messages for messages 4627 through 4641. This duplication occurs because the Foo consumer performs a periodical auto-commit of the consumer offset every 5 seconds. Therefore, a new consumer that has been assigned a partition because the previous consumer died risks duplicate messages going back for 5 seconds.

APACHE KAFKA BOOKS

If you need to use Apache Kafka, we recommend studying and reading relevant Kafka material such as tutorials, videos, and books. In particular, if you come from a traditional messaging background, you'll see that Kafka does things differently than those messaging systems.

Here are two books that we recommend:

Kafka: The Definitive Guide by Neha Narkhede (O'Reilly, 2017)

Understanding Message Brokers by Jakub Korab (free ebook by O'Reilly)

That concludes the most commonly used component for clustering. The next section covers specifics about clustered caches with Camel.

17.5 Clustering caches

Writing stateful applications in a distributed and clustered system is more complex because the state must be shared to all the nodes in the cluster. This is what *clustered memory grid* (clustered cache) systems such as Hazelcast, Infinispan, or Redis are designed to solve.

Let's illustrate this with a simple example, shown in figure 17.7.

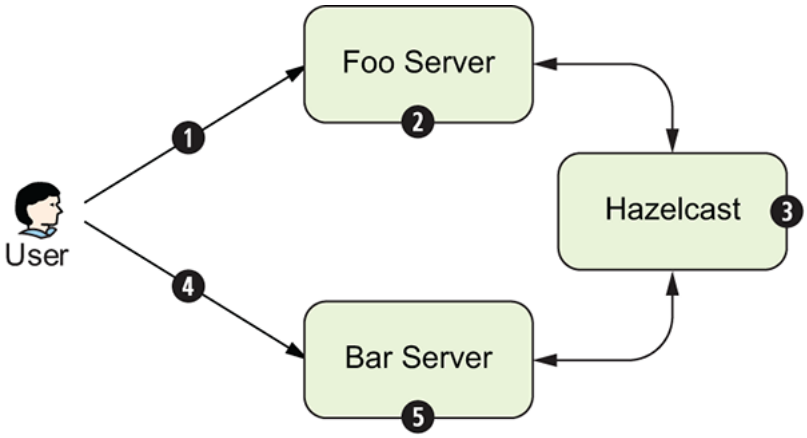

Figure 17.7 A user accesses a distributed Camel application that has been clustered and runs on two nodes (❷, ❺). The application stores state about user activity, which is made available at all times using the clustered cache provided, such as Hazelcast ❸.

The same Camel application is deployed and running on both servers ❷ ❺ in a clustered setup. This is done because the application must be made highly available in case one of the servers becomes unavailable. In this

example, either of the two servers can be accessed by the client ❶ ❹, which would require any data that must be shared between the nodes to be made available on both nodes. This is where the clustered cache from Hazelcast ❸ enters the scene and provides such functionality.

17.5.1 Clustered cache using Hazelcast

As a Camel developer, you shouldn't worry too much about how Hazelcast clustering works. Just use camel-hazelcast like any other Camel component.

In this example, the client calls the application using HTTP. The stateful data is a counter value that's incremented on each call and is returned as a response to the client. The following listing shows the source code of the Camel route implementing this.

Listing 17.5 Stateful Camel application using Hazelcast to distribute state in the Camel cluster

```
fromF("jetty:http://localhost:" + port)    ❶

    .setHeader(HazelcastConstants.OBJECT_ID, constant("myCounter"))    ❷

    .to("hazelcast:map:myCache?hazelcastInstance=#hz&defaultOperation=get")    ❷

    .process(new Processor() {    ❸
```

❶ Starts the route using Jetty as HTTP server

❷ Gets the shared data from Hazelcast

❸ Updates the shared data

```
    public void process(Exchange exchange) throws Exception {
      Integer counter = exchange.getIn().getBody(Integer.class);
      if (counter == null) {
        counter = 0;
      }
      counter++;
      exchange.getIn().setBody(counter);
    }
  })
  .setHeader(HazelcastConstants.OBJECT_ID, constant("myCounter"))    ❹
```

---

❹

Stores the shared data back to Hazelcast

```
  .to("hazelcast:map:myCache?hazelcastInstance=#hz&defaultOperation=put")    ❹

  .log(name + ": counter is now ${body}")
  .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))    ❺
```

---

❺

Prepares the HTTP response message

```
  .transform().simple("Counter is now ${body}\n");    ❺
```

The Camel route starts with the Jetty endpoint that exposes an HTTP service running on `localhost` ❶. The shared state is then accessed from the Hazelcast cluster using the camel-hazelcast component. A header must be configured with the name of the state (object ID) before calling the endpoint ❷. We've highlighted in bold the pieces that make up what Camel is asking of Hazelcast:

```
.setHeader(HazelcastConstants.OBJECT_ID, constant("myCounter"))
.to("hazelcast:map:myCache?hazelcastInstance=#hz&defaultOperation=get")
```

As you can see, you get from a Hazelcast map named `myCache` the value of the key `myCounter`.

The Camel route uses a processor ❸ to update the counter, which is then stored back into Hazelcast ❹. The last piece of the route is to prepare the response to return to the user ❺.

You can find this example in the chapter17/cluster-cache-hazelcast directory. To try this example, you need to run two JVMs, each with the Foo or Bar server, which can be started from Maven as follows:

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

From a web browser, you can call either the Foo or Bar server with the following URLs: http://localhost:8080 and http://localhost:9090. You should then be able to receive a response with an increasing counter that's distributed. If you mix the calls between the two JVMs, the counter is correctly increased by one each time. You can try to stop one of the servers and bring it back up, which should let it reconnect to the Hazelcast cluster and work again.

The perceptive reader may have spotted a problem with the example source code in listing 17.5. At first you get the current counter value ❷, which is then updated by Camel ❸ and stored back again ❹. What happens if both servers concurrently read the counter at the same time? For example, if both servers read the counter as 42, they'd both update and set the counter to 43. But the counter was supposed to be 44. What happened is called the *lost update problem*.

Storing stateful data in a map is good for many uses cases, such as tracking user activity or items in a shopping cart. But this example was purposefullly designed to make you aware of the problems with counters. Instead of using a map, you should use *clustered* counters.

Clustered Counters

Some distributed memory grid systems have support for atomic counters. For example, Hazelcast has this support, and it's on the roadmap for Infinispan 9.*x*. Using a clustered counter with camel-hazelcast is easy. The source code from listing 17.5 can be shortened to just one call to Hazelcast:

```
fromF("jetty:http://localhost:" + port)
  .setHeader(HazelcastConstants.OBJECT_ID, constant("myCounter"))
  .to("hazelcast:atomicvalue:Cache?
       hazelcastInstance=#hz&defaultOperation=increment")
  .log(name + ": counter is now ${body}")
  .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
  .transform().simple("Atomic Counter is now ${body}\n");
```

Instead of using Hazelcast `map`, you use `atomicvalue`. The counter is updated using the `increment` operation, which works safely in the cluster.

You can try this variation of the example by running the Foo and Bar servers with the following Maven goals:

```
mvn compile exec:java -Pfoo-atomic
mvn compile exec:java -Pbar-atomic
```

The examples using Hazelcast in this chapter have all been using embedded mode. This means you've colocated Hazelcast with your Camel application. In the real world, you'd use a separate Hazelcast cluster that you can manage and run independently from your Camel applications. The next section covers an example of using a separate cluster with Camel, except instead of using Hazelcast, you'll use Infinispan, and use JCache as the abstraction API between Camel and the cache.

17.5.2 Clustered cache using JCache and Infinispan

JCache is a standard that provides a common set of caching APIs that allow clients to access different caching providers using the same API standard. This should allow you to use the camel-jcache component and then plug in different Cache providers such as Hazelcast or Infinispan without having to change your Camel route. This example uses a clustered Infinispan setup. You can do this on your own by downloading Infinispan Server from http://infinispan.org. Further instructions for setting up Infinispan are provided in the source code in the chapter17/cluster-jcache directory.

When using JCache, you must specify which `JCachingProvider` you're using and its configuration. To use Infinispan, you need to use the

`org.infinispan.jcache.remote.JCachingProvider` provider, which you find in the infinispan-jcache-remote JAR file.

The Camel application is represented with almost identical source code in the `ServerFoo` and `ServerBar` classes. The following listing shows the source code for `ServerBar`.

Listing 17.6 Setting up JCache component to use Infinispan remote server

```
public class ServerBar {
  private Main main;

  public static void main(String[] args) throws Exception {
    ServerBar bar = new ServerBar();
    bar.boot();
  }

  public void boot() throws Exception {
    main = new Main();

    JCacheComponent jcache = new JCacheComponent();    (1)
```

(1) Creates Camel JCache component

```
    jcache.setCachingProvider(JCachingProvider.class.getName());    (2)
```

(2) Uses Infinispan provider

```
    jcache.setConfigurationUri("hotrod-client.properties");    (3)
```

(3) Specifies the Infinispan configuration file

```
    main.bind("jcache", jcache);    ❹
```

---

❹

Uses jcache as the component name

```
        main.addRouteBuilder(new CounterRoute("Bar", 8889));
        main.run();
    }
}
```

The camel-jcache component requires you to choose a JCache provider, such as Infinispan or Hazelcast. You specify the chosen provider and, optionally, additional configurations as options on the camel-jcache component, as shown in listing 17.6. At first the `JCacheComponent` is created ❶, and then the provider from Infinispan is configured ❷. Because you're using a remote Infinispan cluster, you must configure the provider with details such as the URLs to the remote Infinispan cluster and other relevant settings. This example uses the Infinispan Java client, which is named `hotrod`, and hence the configuration name is `hotrod-client.properties` ❸. To keep the example simple, the configuration file has only one line:

```
infinispan.client.hotrod.server_list=localhost:11222;localhost:11372
```

After the component has been configured, it's added to the Camel register using the name `jcache` ❹.

The Camel route is similar to what you've seen when using Hazelcast, such as in listing 17.5. The following code snippet shows you how to get from the cache:

```
.setHeader(JCacheConstants.KEY, constant("myCounter"))
.to("jcache:myCache?action=get")
```

And the following shows how to update the cache:

```
  .setHeader(JCacheConstants.KEY, constant("myCounter"))
  .to("jcache:myCache?action=put")
```

If you've successfully set up the Infinispan cluster (remember to create `myCache` using the Infinispan web console), you should be able to run the two Camel applications with Maven from the chapter17/cluster-jcache directory:

```
mvn compile exec:java -Pfoo
mvn compile exec:java -Pbar
```

The example works similarly to the Hazelcast examples. You call either http://localhost:8888 or http://localhost:8889 to access the servers, and the response should include the counter going up by one number for each call.

The last clustered component we want to discuss is the Quartz scheduler, used to schedule Camel routes to run clustered.

17.6 Using clustered scheduling

The Quartz component is used for scheduling Camel routes to run at certain intervals. We covered Quartz in chapter 6, section 6.7. This time, we'll show you how to use Quartz in a clustered setup. You may want to do this when you need running tasks periodically or according to a cron expression (cron was covered in section 6.7.2). Suppose you want to run a task every minute during opening hours of the business (8 a.m. to 6 p.m.), every day. Here's the cron expression for doing this:

```
0 0/1 08-18 ? * *
```

Now suppose you have a cluster of two nodes, each running a Camel application, and you want to run this job on only one node. You must run this task in master/slave mode, and you can implement that using a clustered scheduler, which means using the Quartz component.

17.6.1 Clustered scheduling using Quartz

To use Quartz in clustering, you need to do the following three tasks:

- Set up and prepare a shared database
- Configure Quartz to use the database
- Define Camel route(s) using the Quartz component

An example with the source code is located in the chapter17/cluster-quartz directory.

Setting up database for Quartz

Quartz supports most common databases, and we've chosen to use Postgres. To quickly run Postgres, you can run the following Docker command:

```
docker run -p 5432:5432 -e POSTGRES_USER=quartz -e POSTGRES_PASSWORD=qua
```

This starts Postgres and binds the network listener of Postgres on port 5432. This will allow you to log in to the database using the supplied username and password. The name of the database is the same name as the username, which is `quartz`. You then need to prepare the database to create the necessary tables that Quartz uses to store its clustered state. If you download the Quartz distribution, you can find the SQL scripts for the supported databases in the docs/dbTable directory. We've made it easy to set up the Postgres tables using Java. You can run the following Maven goal from the chapter17/cluster-quartz2 directory:

```
mvn compile exec:java -P init
```

Configuring Quartz to use the database

Quartz must be configured to use the database, which is done in the quartz.properties file. The most noteworthy configuration is shown here:

```
org.quartz.scheduler.instanceId = AUTO
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.
                                          jdbcjobstore.PostgreSQLDelegat
org.quartz.jobStore.dataSource = quartzDataSource
org.quartz.jobStore.isClustered = true
org.quartz.dataSource.quartzDataSource.driver = org.postgresql.Driver
org.quartz.dataSource.quartzDataSource.URL =
```

```
                                        jdbc:postgresql://localhost:5432/quartz
org.quartz.dataSource.quartzDataSource.user = quartz
org.quartz.dataSource.quartzDataSource.password = quartz
```

If you use a different database, you must configure the file according to the name of the JDBC driver and the URL.

Camel route using Quartz

Using the Camel Quartz component is easy—all you need to do is to configure the component with the location of the quartz.properties file:

```
QuartzComponent quartz = new QuartzComponent();
quartz.setPropertiesFile("quartz.properties");
main.bind("quartz2", quartz);
```

Then the Quartz component is used in the Camel route to trigger according to the desired cron expression:

```
from("quartz2:myGroup/myTrigger?cron=0+0/1+08-18+?+*+*")
  .log(name + " running at ${header.fireTime}");
```

You can run this example by starting two Camel applications known as Foo and Bar using the following Maven goals:

```
mvn compile exec:java -P foo
mvn compile exec:java -P bar
```

Then you should notice that only one node will ever run the task at every minute. Quartz will let only the node that holds the database lock run the task. The lock is then released after the task is complete. At the next scheduled time, both nodes will race to acquire the lock, and whichever grabs the lock runs the task. (The node that runs the task is determined randomly.) If a node crashes, the other node will be there to run the task. You have high availability as long you have at least one running node, and the database must also be running. How to make the Postgres database clustered is beyond the scope of this book. This brings us to the caveat of using Quartz in clustered mode: Quartz must use a database.

The last part of this chapter covers the newest addition (at the time of this writing) to Camel's EIP patterns: the Service Call EIP.

17.7 Calling clustered services using the Service Call EIP

The Service Call EIP is a new EIP pattern added in Camel 2.18. This pattern is used for calling remote services in a distributed system. The pattern has the following noteworthy features:

- *Location transparency*—Decouples Camel and the physical location of the services using logical names representing the services.
- *URI templating*—Allows you to template the Camel endpoint URI as the physical endpoint to use when calling the service.
- *Service discovery*—Looks up the service from a service registry of some sort to know the physical locations of the services.
- *Service filter*—Allows you to filter unwanted services (for example, blacklisted or unhealthy services).
- *Service chooser*—Allows you to choose the most appropriate service based on factors such as geographical zone, affinity, plans, canary deployments, and SLAs.
- *Load balancer*—A preconfigured Service Discovery, Filter, and Chooser intended for a specific runtime (these three features combined as one).

In a nutshell, the EIP pattern sits between your Camel application and the services running in a distributed system (cluster). The pattern hides all the complexity of keeping track of all the physical locations where the services are running and allows you to call the service by a name.

This is an oversimplification of what happens, so let's dive deep into the inners of Camel and see how this EIP works.

17.7.1 How the Service Call EIP works

Figure 17.8 depicts the algorithm in use by the Service Call EIP when a service is to be called.

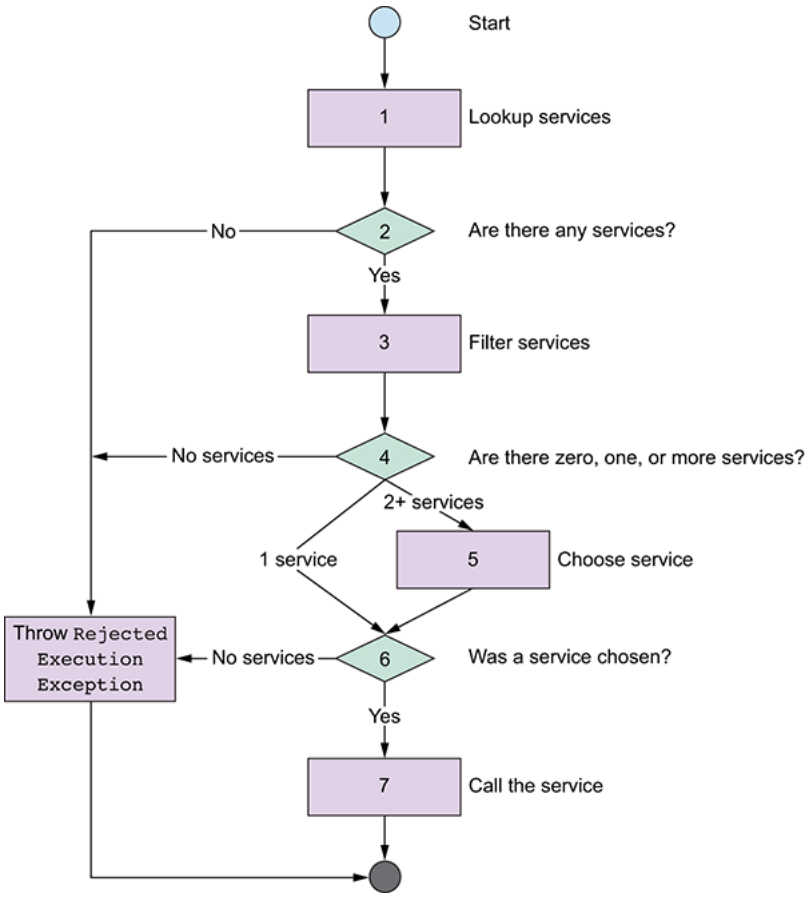

Figure 17.8 Flow chart of how the Service Call EIP works when calling a service

Here's the algorithm for the way the Service Call EIP works:

1. The logical name of the service to call is used to look up from an external registry the physical locations where the service is hosted.
2. The lookup returns a list of physical locations of the service. If the list is empty, `RejectedExecutionException` is thrown and the call is ended. If the list contains numerous services, go to step 3.
3. The services are filtered to remove unwanted services. For example, some services may be blacklisted. A typical filter would be to include only services considered healthy. This requires using health checks, which we talk about in chapter 18.
4. After the filtering, are there zero, one, or more services? If there are no services, `RejectedExecutionException` is thrown, and the call is ended. If there's only one service, go to step 7.
5. When two or more services exist after the filtering, you need to choose one. For example, if the client has called the service before, you could try to choose the same physical service again (affinity). Another criteria could be to choose the physical service located in the same data center, or in the same geographical region. Or you can use old-fashioned round-robin or randomly choose a service.

6. Was a service chosen? If not, `RejectedExecutionException` is thrown, and the call is ended. If a service was chosen, go to step 7.
7. A service has been chosen and is about to be called. When Camel calls the service, it maps the physical location of the service to a Camel endpoint URI using URI templating (it's like Camel's `<toD uri="..."/>` to call a dynamic endpoint). This allows extreme flexibility, and to use any of the Camel components as a producer calling the service.

Now it's time to take a look at how this algorithm applies in practice.

17.7.2 Service Call using static service registry

Let's start easy and use an example that has a client that calls a clustered service. The cluster is static and hosts the service on two running nodes (Foo server and Bar server). Figure 17.9 depicts this scenario.

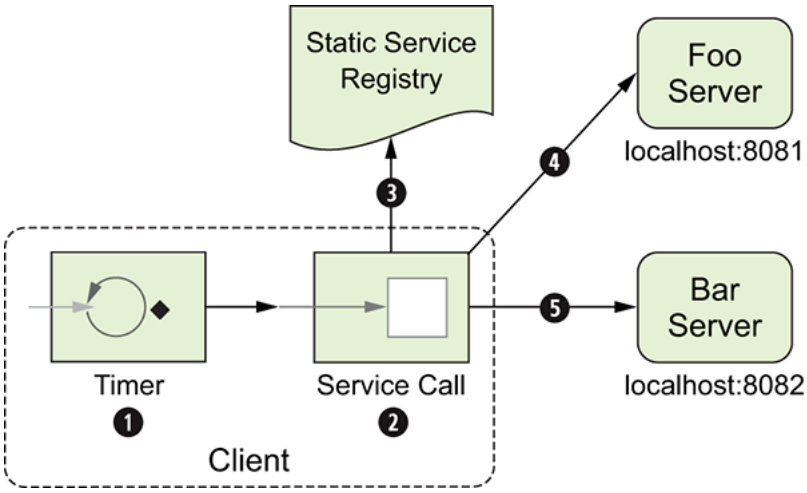

Figure 17.9 A timer ❶ triggers the Service Call EIP ❷ to call a clustered service. The physical locations of the service are looked up in the service registry ❸. The service is then called in a round-robin fashion by calling either Foo server ❹ or Bar server ❺.

The client is developed as a Camel route that starts from a timer ❶, which then calls the service ❷. The source code for such a route in Java and XML DSL is as follows:

```
from("timer:trigger")
   .serviceCall("hello-service")
```

In XML DSL:

```
<route>
  <from uri="timer:trigger"/>
```

```
    <serviceCall name="hello-service"/>
  </route>
```

There's more to this: you need to configure the Service Call EIP with a strategy for discovering and looking up the services, as well as for choosing one service among the available services (for example, according to the algorithm laid out in figure 17.8).

The example uses a static list of services ❸, which is configured using the syntax `serviceName@hostname:port,serviceName@hostname2:port2` (you separate multiple host configurations with commas). The name of the service is `hello-service`, which gives the following server configuration:

```
hello-service@localhost:8081,hello-service@localhost:8082
```

The static list of servers contains two physical locations (❹ ❺), but the Service Call EIP needs to call only one of them. Therefore, one is chosen, and it's chosen in round-robin fashion by default. Having chosen one server, the EIP constructs the Camel endpoint URI, using URI templating, to use when calling the service. We'll go over this in more detail in a moment. But first, the following listing shows you the source code for this example.

Listing 17.7 Using Service Call EIP to call a clustered service (static) in Java DSL

```
public class MyStaticRouteEmbedded extends RouteBuilder {

  public void configure() throws Exception {
    from("timer:trigger?period=2000")
   .serviceCall()    ❶
```

❶ Service Call EIP

```
    .component("http4")    ❷
```

❷

Using http4 component

```
    .name("hello-service/camel/hello")    ❸
```

❸

Configures name of service

```
    .staticServiceDiscovery()    ❹
```

❹

Static location of services

```
        .servers("hello-service@localhost:8081,"    ❹
                + "hello-service@localhost:8082")    ❹
    .end()    ❺
```

❺

Ends configuration of static list

```
    .end()    ❻
```

❻

Ends Service Call EIP

```
        .log("Response ${body}");
  }
}
```

The route starts from a timer that triggers every other second. Then the Service Call EIP block begins ❶ and ends ❻. The `component` ❷ is used to specify which Camel component to use when calling the service. In this

example, we use HTTP as a protocol and can therefore use any of the many HTTP components from Camel.

TIP The Service Call EIP uses http4 as the default component, so you could leave out this configuration in this example.

The name ❸ of the service must be configured. The configured name is specified as `hello-service/camel/hello`. The name supports URI templating in the form of `name/context-path?parameters`. This means the name of the service is `hello-service` and with an associated `/camel/hello` as context-path.

To keep the example simple, we specify the physical location of the two servers in a static list ❹. Notice that we use `end()`s to end both the static list ❺ and the Service Call EIP ❻.

You can also use Service Call in XML DSL, as shown in the next listing.

Listing 17.8 Using Service Call EIP to call a clustered service (static) in XML DSL

```
<camelContext xmlns="http://camel.apache.org/schema/spring">

  <route>
    <from uri="timer:trigger?period=2000"/>
    <serviceCall name="hello-service/camel/hello" component="http4">  ❶
```

---

❶ Service Call EIP

```
      <staticServiceDiscovery>
        <servers>
          hello-service@localhost:8081,hello-service@localhost:8082  ❷
```

---

❷ Configures static list of servers

```
        </servers>
       </staticServiceDiscovery>
     </serviceCall>
     <log message="Response ${body}"/>
   </route>

 </camelContext>
```

In XML DSL, you specify the service name and component as attributes on the `<serviceCall>` element ❶. You specify the static list of servers ❷ in similar way to Java DSL, where it's embedded inside the Service Call EIP.

Running the example

You can try this example by running the following three Java applications from separate command shells. Running the Foo server:

```
cd chapter17/cluster-servicecall/foo-server
mvn spring-boot:run
```

Running the Bar server:

```
cd chapter17/cluster-servicecall/bar-server
mvn spring-boot:run
```

Running the client:

```
cd chapter17/cluster-servicecall/client-static
mvn camel:run -P embedded
```

You can also try the XML DSL version of the client by running this:

```
cd chapter17/cluster-servicecall/client-static-xml
mvn camel:run -P embedded
```

When running the example, the client will output responses from the two servers in round-robin mode:

```
INFO  Response Hello from Foo server on port 8081
INFO  Response Hello from Bar server on port 8082
INFO  Response Hello from Foo server on port 8081
INFO  Response Hello from Bar server on port 8082
```

This looks good, and it's hardly a surprise that Camel is able to call two local servers. But let's see what happens if one of these servers is down.

17.7.3 Service Call with failover

In chapter 7, we talked about how you should design for failures with distributed systems. You can argue we have a mini distributed system in the example in figure 17.9, where the hello service is distributed among the two servers (Foo and Bar). When the client attempts to call the service using the Service Call EIP, a failure can happen. Let's see what happens if, for example, the Foo server is unavailable.

We assume you have the three Java applications running (Foo server, Bar server, and the client). Now notice what happens if you stop the Foo server:

```
INFO  Response Hello from Bar server on port 8082
ERROR Failed delivery for (MessageId: ID-davsclaus-pro-61686-14929518049
-0-13 on ExchangeId: ID-davsclaus-air-61686-1492951804947-0-14).
Exhausted after delivery attempt: 1 caught:
org.apache.http.conn.HttpHostConnectException: Connect to localhost:8081
[localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed:
Connection refused (Connection refused)
INFO  Response Hello from Bar server on port 8082
```

Every second call to the service will fail with an exception because the client can't connect to `localhost:8081`, which is the physical location of the Foo server. But the service hosted on the Bar server is working, so you can see a successful response when it's called. What can you do to resolve this problem? That's a good question, and there are several solutions. You could, for example, configure Camel's error handler to retry calling the service by configuring the error handler as follows:

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3));
```

Running the client again when the Foo server is unavailable will output responses only from the Bar server:

```
INFO  Response Hello from Bar server on port 8082
INFO  Response Hello from Bar server on port 8082
INFO  Response Hello from Bar server on port 8082
```

And when you start Foo server again, the client will go back to load balance among the two available servers:

```
INFO  Response Hello from Bar server on port 8082
INFO  Response Hello from Foo server on port 8081
INFO  Response Hello from Bar server on port 8082
INFO  Response Hello from Foo server on port 8081
```

Why did Camel's error handler fix this problem? When the Service Call is retried (a redelivery is performed by Camel); the service election process starts all over again, according to the algorithm in figure 17.8. And because the two static servers are chosen in a round-robin fashion, the redelivery attempt will select the Bar server this time, which succeeds. To see this, you can configure Camel's error handler to log at WARN level when an attempt fails:

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3)
  .retryAttemptedLogLevel(LoggingLevel.WARN));
```

Then you can see that each attempt to call the Foo server (`localhost:8081`) fails:

```
INFO  Response Hello from Bar server on port 8082
WARN  Failed delivery for (MessageId: ID-davsclaus-pro-61808-14929525578
-0-1 on ExchangeId: ID-davsclaus-air-61808-1492952557802-0-2). On delive
attempt: 0 caught: org.apache.http.conn.HttpHostConnectException: Connec
to localhost:8081 [localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] faile
Connection refused (Connection refused
INFO  Response Hello from Bar server on port 8082
```

Another approach to resolve this problem is to wrap the Service Call EIP with a Circuit Breaker EIP using Hystrix. Hystrix was covered in chapter

7, section 7.4.3.

HEALTH CHECKS

In a distributed system, you're advised to let your services be observable. For example, the services should provide health status so the platform or service registry can orchestrate this and periodically perform health checks to keep track of the services. The service registry can then filter out unhealthy services, which allows the Service Call EIP to choose only among healthy services. Camel's Service Call EIP allows you to plug in custom providers, such as Consul and Ribbon, that can perform health checks. Chapter 18 covers distributed systems and the importance of health checks.

The Service Call EIP can be configured on different levels. So far, the configuration has been embedded directly within the route. This can become verbose and tedious if you have several service calls in your Camel routes. Instead, you can configure Service Call outside the routes.

17.7.4 Configuring Service Call EIP

When using the Service Call EIP, it's advised to configure this globally. This ensures that the configuration is done once and is reused by every Service Call EIP in use.

When using Java DSL, you create an instance of `ServiceCallConfigurationDefinition` that's used for the configuration. The example in listing 17.7 can be rewritten, as shown in the following listing.

Listing 17.9 Global configuration of Service Call EIP using Java DSL

```
public class MyStaticRouteGlobal extends RouteBuilder {

  public void configure() throws Exception {
  ServiceCallConfigurationDefinition global =    ❶
```

❶ Creates instance of ServiceCallConfigurationDefinition

```
    new ServiceCallConfigurationDefinition();  ❶

  global.component("http4")  ❷
```

❷

Configures Service Call EIP

```
    .staticServiceDiscovery()  ❷
    .servers("hello-service@localhost:8081,"  ❷
            + "hello-service@localhost:8082");  ❷

  getContext().setServiceCallConfiguration(global);  ❸
```

❸

Sets global Service Call EIP configuration

```
      from("timer:trigger?period=2000")
    .serviceCall("hello-service/camel/hello")  ❹
```

❹

Calls the service

```
        .log("Response ${body}");
    }
}
```

At the top of the `configure` method, we set up the global configuration of the Service Call EIP. At first, we create an instance of `ServiceCallConfigurationDefinition` ❶, which is the base for configuring ❷ the EIP. After the configuration is done, it must be registered in `CamelContext`, which is done using the setter method ❸.

The Camel route is now much less verbose, and the Service Call EIP is as simple as it can get with just one line of code ❹:

```
serviceCall("hello-service/camel/hello")
```

The equivalent example using XML DSL is shown in the following listing.

Listing 17.10 Global configuration of Service Call EIP using XML DSL

```
<camelContext xmlns="http://camel.apache.org/schema/spring">

  <defaultServiceCallConfiguration component="http4">     1

    <staticServiceDiscovery>     1
      <servers>     1
        hello-service@localhost:8081,hello-service@localhost:8082     1
      <servers>     1
    </staticServiceDiscovery>     1
  </defaultServiceCallConfiguration>     1

  <route>
    <from uri="timer:trigger?period=2000"/>
    <serviceCall name="hello-service/camel/hello"/>     2

    <log message="Response ${body}"/>
  </route>

</camelContext>
```

1 Configures Service Call EIP

2 Calls the service

In XML DSL, you use `<defaultServiceCallConfiguration>` ❶ to configure the default Service Call EIP configuration. As in Java DSL, the Camel route is much simpler; calling the service is only one line of code ❷:

```
<serviceCall name="hello-service/camel/hello"/>
```

Running the example

The accompanying source code contains these two clients using global configuration, which you can try by running these Java applications from separate command shells. Running the Foo server:

```
cd chapter17/cluster-servicecall/foo-server
mvn spring-boot:run
```

Running the Bar server:

```
cd chapter17/cluster-servicecall/bar-server
mvn spring-boot:run
```

Running the client:

```
cd chapter17/cluster-servicecall/client-static
mvn camel:run -P global
```

Or the XML DSL version of the client:

```
cd chapter17/cluster-servicecall/client-static-xml
mvn camel:run -P global
```

MULTIPLE SERVICE CALL CONFIGURATIONS

Camel allows you to configure multiple Service Calls, with each configuration associated with a unique name. You can then refer to which configuration to use from the Service Call EIP in your Camel routes. You can find information on how to do this from the Camel documentation on GitHub: https://github.com/apache/camel/blob/master/camel-core/src/main/docs/eips/serviceCall-eip.adoc.

Next we'll talk about how the Service Call EIP allows you to specify the Camel endpoint URI to be used when calling the service.

17.7.5 Service Call URI templating

A key feature of Apache Camel is endpoint URIs, which make it easy to use the many Camel components. When the Service Call EIP calls a service, it constructs a Camel endpoint URI to use. These URIs support URI templating.

A picture is worth a thousand words, so let's try with a table instead; see table 17.1.

Table 17.1 Service Call URI template examples

Name	Resolved URI
hello-service	http4:host:port
hello-service/camel/hello	http4:host:port/camel/hello
hello-service/camel/hello?id=123	http4:host:port/camel/hello?id=123

The first row is the most basic example, with only the logical name of the service. The logical name is then replaced with the chosen physical address of the service in the form `host:port`. With the static server list example, this would resolve as either `http4:localhost:8081` or `http4:localhost:8082`. The middle row is the name used in the example. As you can see, this allows you to include context-path in the resolved URI. The last row shows how you can even include query parameters.

Advanced URI templating

If you want full control of the way the URI is resolved, you can specify the URI yourself. You may need to do this when calling services that aren't using HTTP/REST transport. Then you may find yourself using some of the other Camel components, which may require constructing the URIs in a certain way.

But we can still show you how to construct a URI using HTTP transport. For example, you can hardcode the URI to use another Camel component such as undertow:

```
  .serviceCall("hello-service",
               "undertow:http://hello-service/camel/hello?id=123")
```

And in XML DSL:

```
  <serviceCall name="hello-service"
               uri="undertow:http://hello-service/camel/hello?id=123"/>
```

Notice that you separate the service name and the URI:

Name	URI template
hello-service	undertow:http://**hello-service**/camel/hello?id=123

Pay attention to the highlights in the table. This is the name of the service, which must be represented in the URI template. Camel will then replace the service name with the physical address of the chosen server in the format `host:port`—for example:

```
  undertow:http://localhost:8081/camel/hello?id=123
```

You can even specify the exact position of the hostname and port number individually. For instance, the same example can be defined using `name.host` and `name.port` in the syntax as highlighted:

```
  undertow:http://hello-service.host:hello-service.port/camel/hello?id=123
```

The last example we'll cover uses Spring Boot Cloud and Consul as the service registry.

17.7.6 Service Call using Spring Boot Cloud and Consul

HashiCorp Consul can be used as a dynamic service registry in a distributed system. Spring Boot also provides support for Consul, which can be integrated easily with Camel. The example is located in the chapter17/cluster-servicecall/client-consul directory.

In the Maven pom.xml, we've added the Spring Boot starter for Consul:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

And the Camel starters:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-cloud-starter</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-consul-starter</artifactId>
</dependency>
```

Then enable service discovery by adding the `@EnableDiscoveryClient` annotation to the application class (the class with the `@SpringBootApplication` annotation).

Next you configure Consul in the Spring Boot application.properties file:

```
spring.cloud.consul.discovery.enabled=true
spring.cloud.consul.discovery.server-list-query-tags[hello-service] = ca
```

Note the last line, where you map the name of the service to Consul service tags. This is important because this is how Consul knows how services are grouped together.

The Camel route has no need for any Service Call configuration, which makes this easy to use:

```
from("timer:trigger?period=2000")
  .serviceCall("hello-service/camel/hello")
  .log("Response ${body}");
```

That's all you need to develop on the client side. But you may wonder how Consul knows where the services are physically located. When starting Consul, you can specify the services in a JSON file. For example, the Foo and Bar servers are specified as follows:

```
{
  "services": [{
    "id": "hello-foo",
    "name": "hello-service",
    "tags": ["camel"],
    "address": "localhost",
    "port": 8081
  }, {
    "id": "hello-bar",
    "name": "hello-service",
    "tags": ["camel"],
    "address": "localhost",
    "port": 8082
  }]
}
```

You can run this example by following the instructions in the readme file from the chapter17/cluster-servicecall directory. The example uses Docker to run Consul, and you should pay attention on how to do this.

Service Call with failover

If you run this example and, for example, stop one of the servers, you'll see the same problem as in section 17.7.3. The client will fail with an exception. We can't say this enough: when you build distributed applications, *design for failure*. You can resolve this problem the same way as previously suggested. But if you're using Consul in your organization, you may want to use its support for health checks so Consul can observe your applications and keep its service registry up-to-date with the health state of all services.

This requires you to run a Consul agent on each node, which becomes responsible for running the health checks and reporting back to the Consul cluster. In addition, you need to specify how to perform health checks for your services, which you can do using techniques such as an HTTP call or a shell script. This topic is beyond the scope of this book, but you can consult the Consul documentation. What we want to say is that this is all much easier with container-based platforms such as Kubernetes. All of this is baked into the platform, and you have health checks and service discovery out of the box.

That brings us to the end of our coverage of using clustered Camel with traditional Java technologies.

17.8 Summary and best practices

This chapter was a tour of the various ways of creating clustering with Camel using traditional solutions. For example, we talked about using HTTP load balancers, which is a well-established solution. Clustering Camel routes is done using numerous Camel components that integrate with existing proven cluster solutions.

You may be wondering why a chapter on clustering Camel is so far back in this book. One reason is that clustering is a hard topic, and there are many other topics we wanted to cover first. Another reason is related to Camel's background. When Camel was created, it was created as a small integration framework that could be embedded inside existing solutions. For example, combining the following individual Apache projects gives you a powerful solution that supports clustering as well:

- *Apache Camel*—Integration framework with EIPs and components
- *Apache CXF*—Web and RESTful services
- *Apache ActiveMQ*—JMS messaging and clustering
- *Apache Karaf*—OSGi-based application server
- *Apache ServiceMix*—Umbrella project that embeds all the preceding list items into a Camel-based integration platform

In light of this, Camel wasn't intended to be an all-in-one, kitchen-sink solution. Camel was focused on being a small integration framework and left clustering and application management to ActiveMQ, Karaf, and ServiceMix. The story today is different. We're moving toward a microservice architecture approach, so Camel must do clustering well without relying on other Apache projects.

In this chapter, you've seen a variety of ways to cluster Camel. As always, Camel is flexible; it would be possible to build your own Camel component or route policy to add clustering capabilities to Camel that aren't provided out of the box.

The noteworthy takeaways from this chapter are as follows:

- *Clustered components*—Clustering in Camel is primarily provided by Camel components. When it comes to clustering, Camel is just a library, and you should look at what clustering solutions you have that you can use with Camel. For example, you can use HTTP load balancers, JMS message brokers, or clustered caches, to name just a few.
- *Clustered routes*—If you need to cluster Camel routes, techniques are available in either active/passive or active/active mode. You can use several Camel components to integrate with a clustered solution, such as Hazelcast, Infinispan, Consul, or ZooKeeper.
- *Clustered messaging is hard*—Don't underestimate the complexity of distributed messaging. There's no easy solution that works in all situations. We encourage you to conduct comprehensive analysis and read up on relevant material (books, articles, and other resources).
- *Use Service Call*—If your Camel applications must call services that run in a distributed system, consider using the Service Call EIP. This EIP is flexible and allows you to separate configuration from the service call; you can migrate from using, for example, Consul to Kubernetes easily. You can also build custom strategies to filter and choose from among the services. For example, you can plug in logic to choose services using the canary deployment principle, or to prioritize calls according to plans, or SLAs.

The Camel team has begun implementing new clustering and health check APIs and services. This work is an ongoing effort and was first released as a technical preview in Camel 2.20. Because the work is under active development and is changing, it was too early to cover in this book. If you want to follow this work, we recommend reading "A camel running in the clouds" by Luca Burgazolli (creator of this work) at his blog: https://lburgazzoli.github.io.

We'll continue our clustering endeavor when we move on to the world of containers. Chapter 18 will introduce you to container-based technologies such as Docker and Kubernetes. Let's jump on the horse—er, Camel—and ride the wave of new awesomeness of the promised land that containers are shipped from. Okay, get down from your high horse—ugh, darn, not again—we meant Camel. But speaking more seriously, *container* isn't yet another buzzword that will be gone in a few years. Containers are a game changer, so let's see how well Camel rides in containers.