16

Management and monitoring

This chapter covers

- Monitoring Camel instances
- Tracking application activities
- Using notifications
- Managing Camel applications with JMX and REST
- Understanding and using the Camel management API
- Gathering runtime performance statistics
- Using Dropwizard metrics with Camel
- Developing custom components for management

Applications in production are often critical for businesses. That's especially true for applications that sit at an intermediate tier and integrate all the business applications and partners. Camel is often in this role.

To help ensure high availability, your organization must monitor its production applications. By doing so, you can gain important insight into the applications and foresee trends that otherwise could cause business processes to suffer. In addition, monitoring helps with related issues such as operations reporting, service-level agreement (SLA) enforcement, and audit trails.

It's also vital for the operations staff to be able to fully manage the applications. For example, if an incident occurs, staff may need to stop parts of the application from running while the incident investigations occur. You'll also need management capabilities to carry out scheduled maintenance or upgrades of your applications.

Management and monitoring are often two sides of the same coin. For example, management tooling includes monitoring capabilities in a single coherent dashboard, allowing a full overview for the operations staff.

This chapter reviews various strategies for monitoring your Camel applications. We'll first cover the most common approach, which is to check on the health of those applications. Then we'll look at the options for tracking activity and managing those Camel applications.

16.1 Monitoring Camel

It's standard practice to monitor systems with periodic health checks. For people, checking one's health involves measuring parameters at regular intervals, such as pulse, temperature, and blood pressure. By checking over a period of time, you know not only the current values but can also spot trends, such as whether your temperature is rising. All together, these data give insight into the health of the person.

For a software system, you can gather system-level data such as CPU load, memory usage, and disk usage. You can also collect application-level data, such as message load, response time, and many other parameters. This data tells you about the health of the system. Checks on the health of Camel applications can occur at three levels:

- *Network level*—This is the most basic level, where you check that the network connectivity is working.
- *JVM level*—At this level, you check the JVM that hosts the Camel application. The JVM exposes a standard set of data using the JMX technology.
- *Application level*—Here you check the Camel application using JMX or other techniques.

To perform these checks, you need various tools and technologies. The Simple Network Management Protocol (SNMP) enables both JVM and system-level checks. Java Management Extensions (JMX) is another technology that offers similar capabilities to SNMP. You might use a mix of both: SNMP is older, more mature, and often used in large system-management tools including established technologies such as IBM Tivoli and HP OpenView. JMX, on the other hand, is a pure Java standard supported by open source tools such as Jolokia, Hawkular, Prometheus, and Nagios.

The following sections go over the three levels and offer approaches you can use for performing automatic and periodic health checks on your Camel applications.

16.1.1 Checking health at the network level

The most basic health check you can do is to check whether a system is alive. You may be familiar with the ping command, which you use to send a ping request to a remote host. Camel doesn't provide a ping service out of the box, but creating such a service is easy. The ping service reveals only whether Camel is running, but that will do for a basic check.

Suppose you've been asked to create such a ping service for Rider Auto Parts. The service is to be integrated with the existing management tools. You choose to expose the ping service over HTTP, which is a universal protocol that the management tool can easily use, a scenario illustrated in figure 16.1.

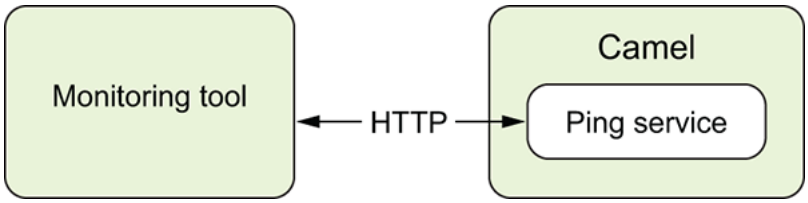

Figure 16.1 A monitoring tool monitors Camel with a ping service by sending periodic HTTP `GET` requests.

Implementing the service in Camel is easy when using the Jetty component. All you have to do is expose a route that returns the response, as follows:

```
from("jetty:http//0.0.0.0:8080/ping").transform(constant("PONG\n"));
```

When the service is running, you can invoke an HTTP `GET`, which should return the `PONG` response.

You can try this on your own with the book's source code. In the chapter16/health directory, invoke this Maven goal:

```
mvn compile exec:java
```

Then invoke the HTTP `GET` using either a web browser or the `curl` command:

```
$ curl http://0.0.0.0:8080/ping
PONG
```

The ping service can be enhanced to use the JVM and Camel APIs to gather additional data about the state of the internals of your application.

Using Jolokia as ping service

Instead of building your own Camel route as a ping service, you can use Jolokia. The Jolokia Java JVM agent, upon starting the Camel application, will bootstrap Jolokia automatically.

JOLOKIA

Jolokia is an excellent library that makes JMX management fun again. At its core, it's an HTTP/JSON bridge for remote JMX access. It provides a Java agent that makes it easy to integrate into a JVM without any code changes. You can find more details about Jolokia at its website: https://jolokia.org.

You can try this example on your own with the book's source code. In the chapter16/jolokia directory, invoke this Maven goal:

```
mvn compile exec:exec
```

Then invoke the HTTP `GET` using either a web browser or the `curl` command (make sure to include the trailing slash):

```
curl http://0.0.0.0:8778/jolokia/
```

Jolokia returns a JSON response with information about the Jolokia agent running in the JVM, as shown in figure 16.2.

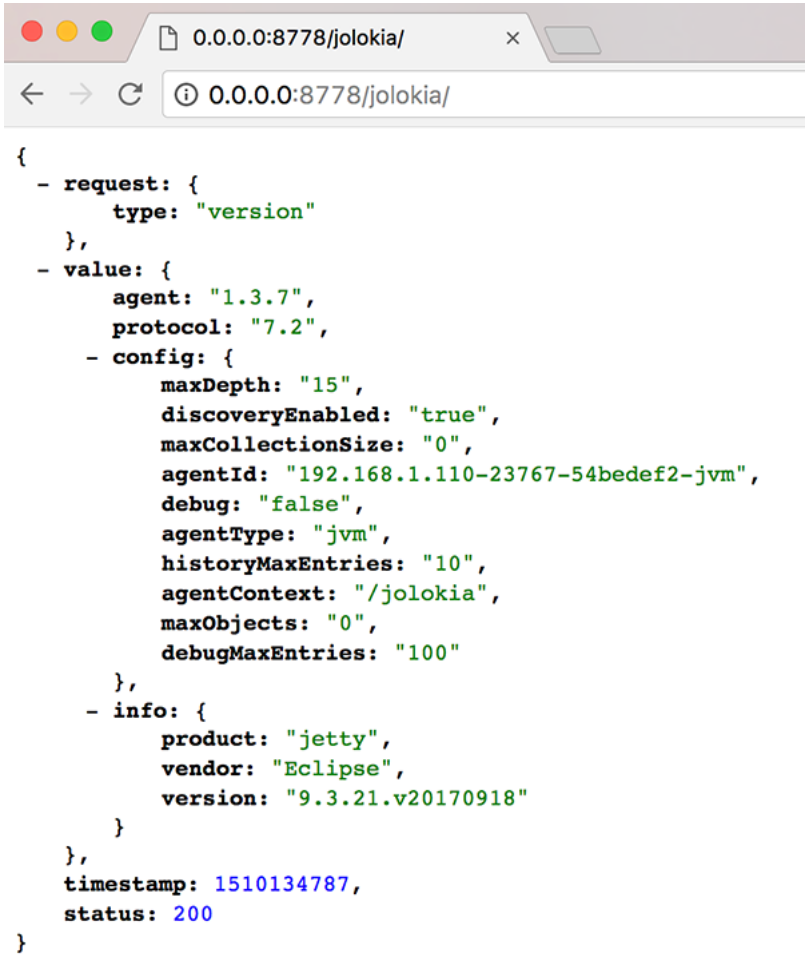

Figure 16.2 Web browser showing the returned information from calling the Jolokia agent on the given URL

Jolokia allows you to access JMX attributes that you can use to read the Uptime attribute from the Camel application. The following URL retrieves this information:

```
http://localhost:8778/jolokia/read/org.apache.camel:context=camel-1
,name=%22camel-1%22,type=context/Uptime
```

And Jolokia will respond with JSON, where you can see that the Uptime value is 8 minutes:

```
{"timestamp":1510141224,"status":200,"request":
{"mbean":"org.apache.camel:context=camel-1,name=\"camel-1\",
type=context","attribute":"Uptime","type":"read"},"value":"8 minutes"}
```

TIP Instead of reading the Uptime attribute, you can try some of the many attributes that Camel exposes in JMX, such as CamelVersion and ExchangesTotal.

We'll now leave Jolokia for a bit and continue. Another use for the ping service is when using a load balancer in front of multiple instances of Camel applications. This is often done to address high availability, as shown in figure 16.3. The load balancer will call the ping service to assess whether the particular Camel instance is ready for regular service calls.

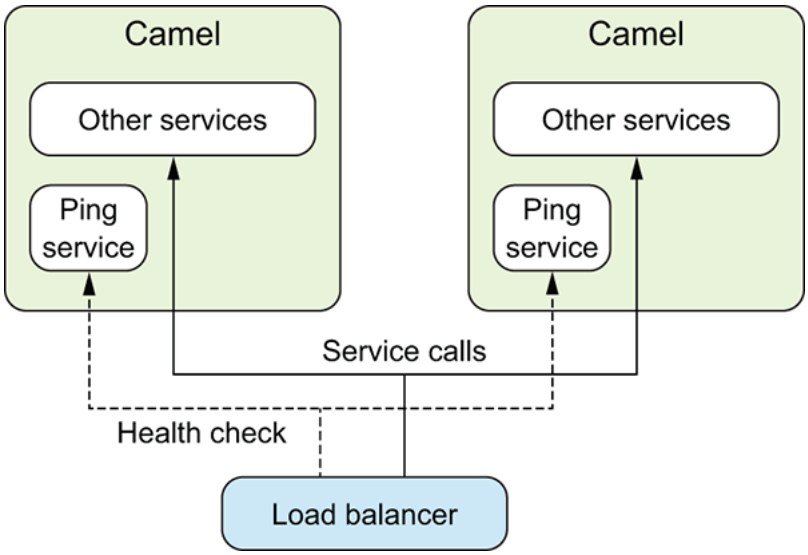

Figure 16.3 The load balancer uses health checks to ensure connectivity before it lets the service calls pass through.

Network-level checks offer a quick and coarse assessment of the system's state of health. Let's move on to the JVM level, where you monitor Camel using JMX.

16.1.2 Checking health level at the JVM level

SNMP is a standard for monitoring network-attached devices. It's traditionally used to monitor the health of servers at the OS level by checking parameters such as CPU load, disk space, memory usage, and network traffic, but it can also be used to check parameters at the application level, such as the JVM.

Java has a built-in SNMP agent that exposes general information, such as memory and thread usage, and issues notifications on low-memory conditions. This allows you to use existing SNMP-aware tooling to monitor the JVM where Camel is running. A wide range of commercial and open source monitoring tools also use SNMP. Some are simple and have a shell interface, and others have a powerful GUI. You may work in an organization that already uses a few monitoring tools, so make sure these tools can be used to monitor your Camel applications as well.

The SNMP agent in the JVM is limited to exposing data at only the JVM level; it can't be used to gather information about the Java applications that are running. JMX, in contrast, is capable of monitoring and managing both the JVM and the applications running on it.

In the next section, we'll look at how to use JMX to monitor Camel at the JVM and application levels.

16.1.3 Checking health at the application level

Camel provides JMX monitoring and management out of the box in the form of an agent that uses the JMX technology. This is illustrated in figure 16.4.

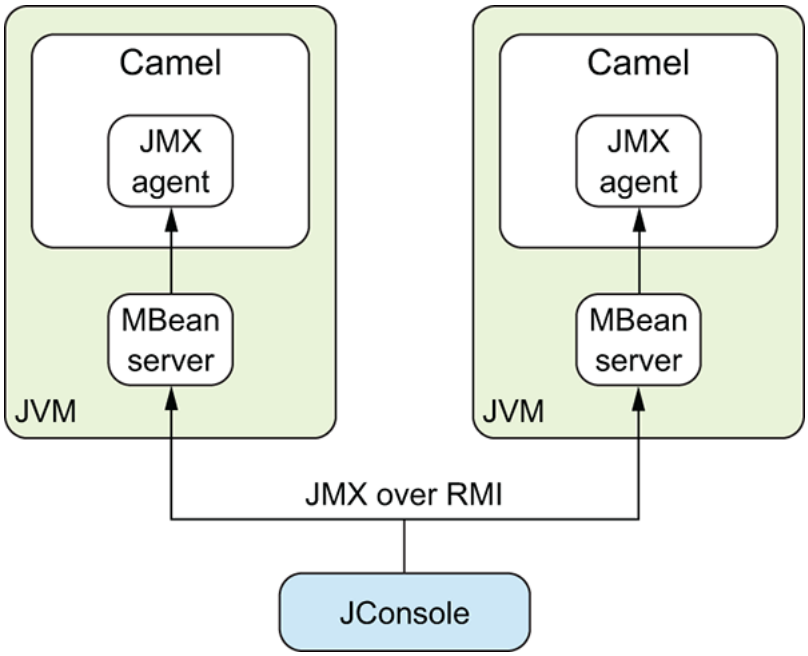

Figure 16.4 JConsole connects remotely to an MBean server inside the JVM, which opens up a world of in-depth information and management possibilities for Camel instances.

The JMX agent remotely exposes (over Remote Method Invocation) a wealth of standard details about the JVM as well as some Camel information. The former comes standard from the JDK, and the latter is provided by Camel. The most prominent feature the Camel JMX agent offers is the ability to remotely control the lifecycle of any service in Camel. For example, you can stop routes and later bring those routes into action again. You can even shut down Camel itself.

How do you use JMX with Camel? Camel comes preconfigured with JMX enabled at the developer level, by which we mean that Camel allows you to connect to the JVM from the same localhost where the JVM is running.

If you need to manage Camel from a remote host, you need to explicitly enable this in Camel.

We think this is important to cover thoroughly, so we devote the next section to this topic.

16.2 Using JMX with Camel

Camel comes with JMX enabled out of the box. When Camel starts, it logs at the `INFO` level whether JMX is enabled or not:

```
2017-08-06 15:00:51,361 [viceMain.main()] INFO  ManagedManagementStrateg
```

And when JMX is disabled:

```
2017-08-06 15:02:53,885 [viceMain.main()] INFO  ManagedManagementStrateg
```

In this section, we'll look at two management tools that can manage Java applications using JMX. The first is JConsole, which is shipped out of the box in Java. The other tool is Jolokia, with hawtio used as the web console.

16.2.1 Using JConsole to manage Camel

Java provides a JMX tool named JConsole. You'll use it to connect to a Camel instance and see what information is available.

TIP Java provides another management tool named JVisualVM that you can use as well. But you'd need to install the VisualVM-MBeans plugin to allow this tool to work with JMX Management Beans.

First, you need to start a Camel instance. You can do this from the chapter16/health directory using this Maven command:

```
mvn compile exec:java
```

Then, from another shell, you can start JConsole by invoking `jconsole`.

When JConsole starts, it displays a window with two radio buttons. The Local radio button is used to connect to existing JVMs running on the same host. The Remote radio button is used for remote management, which we'll cover shortly. The Local option should already list the local running processes. You can find the correct process by hovering the mouse to display a tooltip with the full name. The process containing the word `Launcher` corresponds to the running application that was launched from Maven. Select this process, and you can click the Connect button to connect JConsole to the Camel instance.

Figure 16.5 shows the Camel management beans (MBeans) that are visible from JConsole.

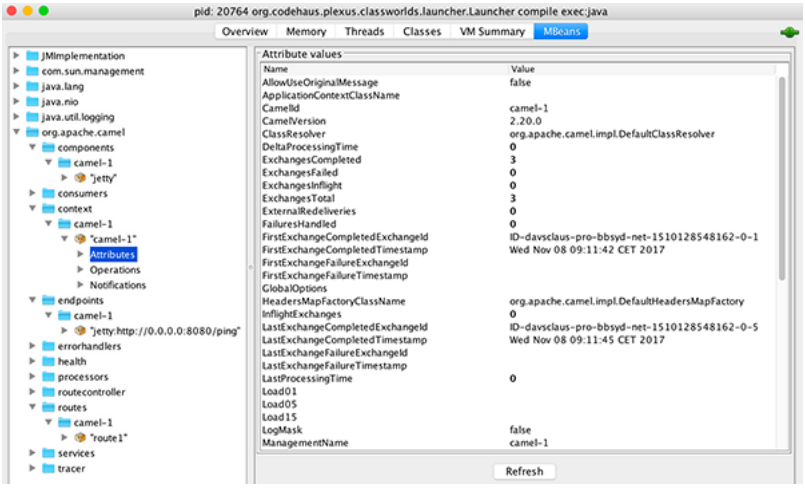

Figure 16.5 Camel registers numerous MBeans that expose internal details, such as usage statistics and management operations.

Camel registers many MBeans that expose statistics and operations for management. Those MBeans are divided into 12 categories, which are listed in table 16.1. Most MBeans expose a set of standard information and operations concerning things such as lifecycle. We encourage you to spend a moment browsing the MBeans in JConsole to see what information they provide.

Table 16.1 Categories of exposed Camel MBeans

Category	Description
Components	Lists the components in use.
Consumers	Lists all the consumers for the Camel routes. Some consumers have additional information and operations, such as the JMS, SEDA, Timer, and File/FTP consumers.
Context	Identifies the `CamelContext` itself. This is the MBean you need if you want to shut down Camel via JMX.
Data formats	Lists the data formats in use.
Endpoints	Lists the endpoints in use.
Error handlers	Lists the error handlers in use. You can manage error handling at runtime, such as by changing the number of redelivery attempts or the delay between redeliveries.
Event notifiers	Lists the event notifiers in use. We'll cover the use of event notifiers in section 16.3.5.
Processors	Lists all the processors (enterprise integration patterns) in use. The EIPs provide various runtime information and operations you can access. For example, the Content-Based Router EIP contains statistics indicating the number of times each predicate has matched.
Producers	Lists all the producers for the Camel routes. Some producers have additional information and

Category	Description
	operations such as JMS, SEDA, and File/FTP producers.
Routes	Lists all the routes in use. Here you can obtain route statistics, such as the number of messages completed, failed, and so on.
Services	Lists miscellaneous services in use.
Thread pools	Lists all the thread pools in use. Here you can obtain statistics about the number of threads currently active and the maximum number of threads that have been active. You can also adjust the core and maximum pool size of the thread pool.
Tracer	Allows you to manage the Tracer service. The Tracer is a Camel-specific service that's used for tracing how messages are routed at runtime. We cover the use of the Tracer in detail in section 16.3.4.

When you need to monitor and manage a Camel instance from a remote computer, you must enable remote management in Camel.

16.2.2 Using JConsole to remotely manage Camel

To be able to remotely manage Camel, you need to instruct Camel to register a JMX connector. That can be done in the following three ways:

- Using JVM properties
- Configuring `ManagementAgent` from Java
- Configuring the JMX agent from XML DSL

We'll go over each of these three methods in the following sections.

Using JVM properties

By specifying the following JVM property on JVM startup, you can tell Camel to create a JMX connector for remote management:

```
-Dorg.apache.camel.jmx.createRmiConnector=true
```

If you do this, Camel will log, at `INFO` level on startup, the JMX service URL that's needed to connect. It looks something like this:

```
2017-08-06 15:58:37,264 [viceMain.main()] INFO  ManagedManagementStrateg
- JMX is enabled
2017-08-06 15:58:37,267 [viceMain.main()] INFO  DefaultManagementAgent
- ManagementAgent detected JVM system properties:{org.apache.camel.jmx.
createRmiConnector=true}
2017-08-06 15:58:37,492 [99/jmxrmi/camel] INFO  DefaultManagementAgent
- JMX Connector thread started and listening at:
service:jmx:rmi:///jndi/rmi://davsclaus-pro:1099/jmxrmi/camel
```

To connect to a remote JMX agent, you can use the Remote radio button from JConsole and enter the service URL listed in the log. By default, port 1099 is used, but this can be configured using the `org.apache.camel.jmx.rmiConnector.registryPort` JVM property.

Configuring the ManagementAgent from Java

The `org.apache.camel.management.DefaultManagementAgent` class is provided by Camel as the default JMX agent. All you need to do is to configure the settings directly using the `ManagementAgent` interface, as highlighted here:

```
public class PingServiceMain {
    public static void main(String[] args) throws Exception {
        CamelContext context = new DefaultCamelContext();

        context.getManagementStrategy().getManagementAgent()
            .setCreateConnector(true);

        context.addRoutes(new PingService());
        context.start();
```

```
        }
    }
```

Configuring a JMX agent from XML DSL

If you use XML DSL with Camel, configuring a JMX connector is even easier. All you have to do is add `<jmxAgent>` in the `<camelContext>`, as shown here:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <jmxAgent id="agent" createConnector="true"/>
    ...
</camelContext>
```

`<jmxAgent>` also offers a `registryPort` attribute that you can use to set a specific port number if the default port 1099 isn't suitable.

JMX: THE GOOD, BAD, AND UGLY

JMX is a good technology for Java libraries and applications to expose a management API, and provide runtime metrics about the state of the JVM and application. But it has its serious drawbacks, as it's a Java-only technology that requires clients to use Java as well (or CORBA). Also, as you've witnessed, remote management requires using a connector port (usually 1099 as the default), and then the JVM assigns a random port as well that's in use. This is seriously not firewall friendly, as organizations have to open ports in their network for JMX remote management. For more details, see, for example, the Jolokia site: https://jolokia.org/features/firewall.html.

How can clients using another technology access and manage Java applications, which are also firewall friendly? A great answer is to use Jolokia.

16.2.3 Using Jolokia to manage Camel

Jolokia allows you to access JMX using HTTP. This ubiquitous technology can be accessed from a web browser, an HTTP client from the command line, or a web console such as hawtio.

TIP We recommend that you read about the features of Jolokia from its website: https://jolokia.org/features-nb.html.

Jolokia offers numerous agents providing Jolokia services in various environments. This chapter covers these three agents:

- WAR agent for deployment as a web application in an application server such as Apache Tomcat or WildFly
- OSGi agent for deployment in an OSGi container such as Apache Karaf or JBoss Fuse
- JVM agent as a generic agent using Java agent style

The WAR agent is the easiest to use, so let's start there.

Using the Jolokia WAR agent

The WAR agent is used by deploying the agent in a web container such as Apache Tomcat or WildFly. You can either deploy the preexisting Jolokia WAR or embed Jolokia into your existing WAR application. In this section, we'll walk you through both approaches, starting by deploying the out-of-the box WAR in the following steps:

1. Start Apache Tomcat.
2. Download Jolokia WAR from https://jolokia.org/download.html.
3. Copy the downloaded WAR into the webapps directory of the running Apache Tomcat.
4. Check whether Jolokia is running by opening http://localhost:8080/jolokia-war-1.3.7 (assuming you're using version 1.3.7 of Jolokia).
5. If Jolokia is running, the web page should output a Jolokia status page (similar to figure 16.2).

By installing the Jolokia WAR once into Apache Tomcat, you can use Jolokia to manage any of the applications deployed in Tomcat. For example, if you deploy five Camel applications and three other applications, Jolokia can access all of those deployments.

Another approach is to embed Jolokia into your existing WAR application, which you may want to do to make a single deployment have *batteries in-*

cluded.

Embedding Jolokia WAR agent into existing WAR application

You can embed Jolokia into your existing WAR application by adding a dependency to Jolokia in the Maven pom.xml file:

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
    <version>1.3.7</version>
</dependency>
```

Then add Jolokia to the web.xml file to expose the Jolokia as an HTTP service as a servlet:

```
<servlet>
    <servlet-name>jolokia-agent</servlet-name>
    <servlet-class>org.jolokia.http.AgentServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>jolokia-agent</servlet-name>
    <url-pattern>/jolokia/*</url-pattern>
</servlet-mapping>
```

The book's source code contains an example in the chapter16/jolokia-embedded-war that you can try using the following Maven goal:

```
mvn jetty:run
```

You can also build the example and deploy the WAR file to a web container such as Apache Tomcat using the following:

```
mvn clean install
```

Then copy the generated WAR file target/chapter16-jolokia-war.war to the webapps directory of Apache Tomcat.

If you're using Apache Tomcat with the default HTTP port 8080, you can access Jolokia using the following URL from a web browser, which should

report back a status response from Jolokia:

```
http://localhost:8080/chapter16-jolokia-embedded-war/jolokia/
```

The principle of embedding a Jolokia WAR agent together with your Camel application in the same deployment unit is illustrated in figure 16.6.

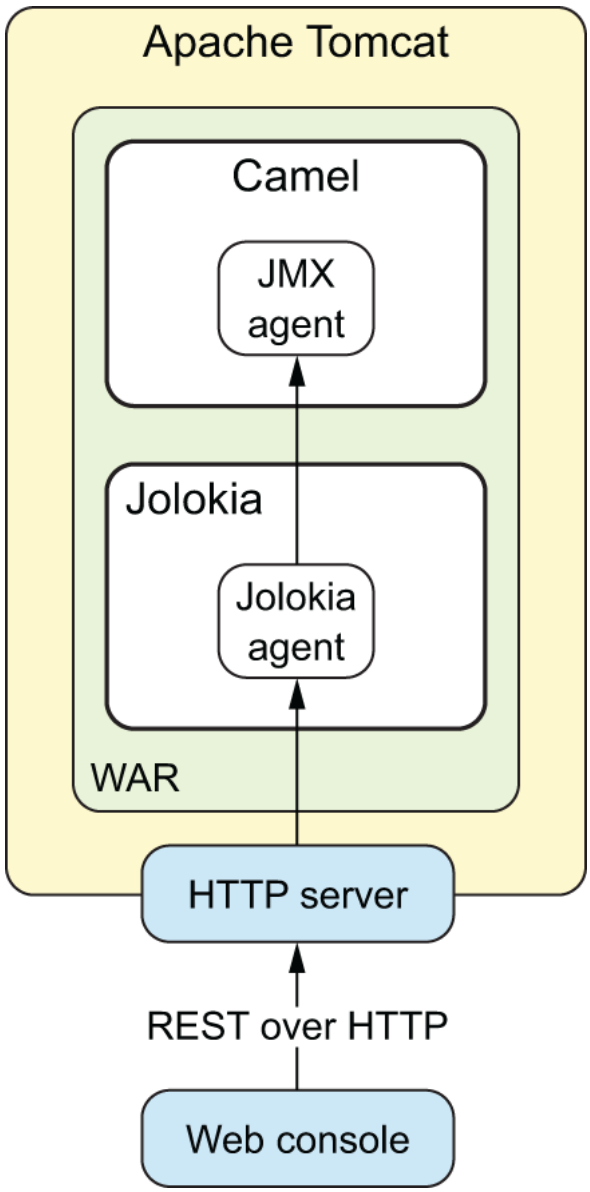

Figure 16.6 Jolokia is embedded together with Camel into the same WAR deployment unit, which is running inside Apache Tomcat. The Jolokia service is exposed using the HTTP server of Apache Tomcat, which the web console can access to obtain information about the running Camel application.

HAWTIO, THE HOT WEB CONSOLE

As a cool web console, you can use hawtio. Try the example from chapter16/jolokia-embedded-war and deploy hawtio into Apache Tomcat. The hawtio console will autodetect the Camel applications running in the JVM, which allows you to use the plugin to visualize the running Camel routes as well as to manage them, and much more. Chapter 19 covers hawtio in more detail.

The next kind of agent is the OSGi agent, which works in OSGi containers such as Apache Karaf and JBoss Fuse. Because Jolokia is preinstalled in JBoss Fuse, we use Apache Karaf as a demonstration for installing and using Jolokia.

Using Jolokia with Apache Karaf

Using Jolokia on Apache Karaf is easy because you can install Jolokia using the Karaf shell. After Karaf has been started, type the following in the shell (Karaf 4 onward):

```
feature:install war
install -s mvn:org.jolokia/jolokia-osgi/1.3.7
```

Jolokia is then available on the default Karaf HTTP port (8181) using the context path jolokia. You should be able to access Jolokia using the following URL:

```
http://localhost:8181/jolokia/
```

Okay, let's move on and talk about the last kind of Jolokia agent: the Java JVM agent.

Using Jolokia Java Agent

Jolokia provides a JVM agent (a.k.a. *Java agent*) that doesn't need any special runtime environment because it uses the embedded HTTP server from Java itself (requires Java 1.6 onward). This agent is the most generic and can be used for any Java application. It's particularly useful when the other specialized agents don't fit. Figure 16.7 illustrates how this agent works.

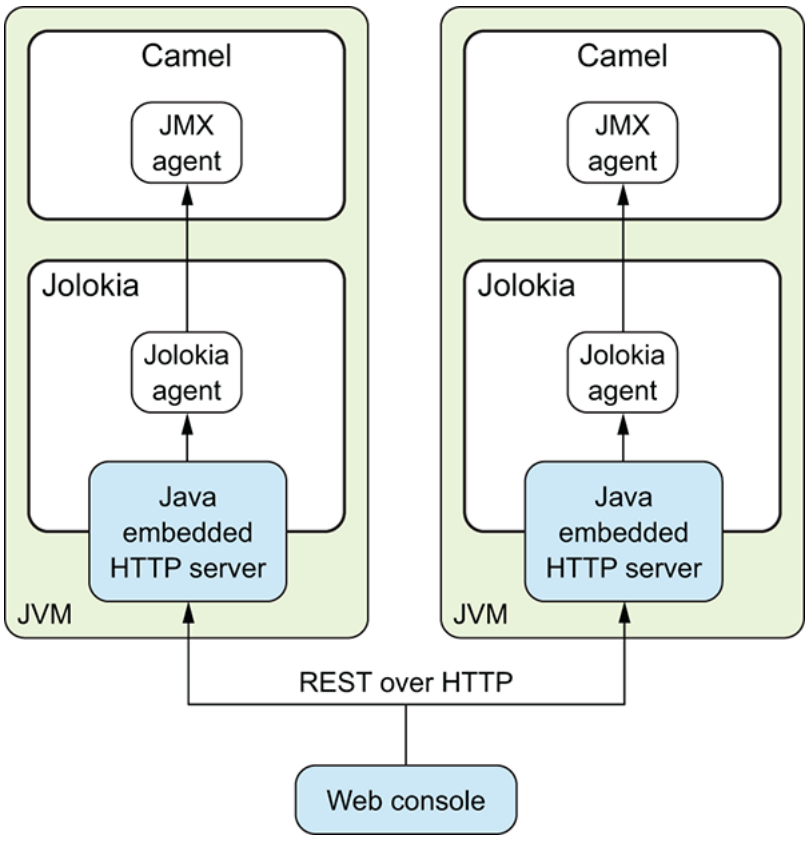

Figure 16.7 A web console (or HTTP client) connects remotely to a JVM of choice to manage using HTTP. The JVM embeds the HTTP server, and the Jolokia agent bridges HTTP to JMX so the web console can gather all JMX information from the Camel application.

The Jolokia Java agent is installed and running as a Java JVM agent, which is configured as part of starting the JVM. The Camel application is running in the same JVM but not embedding Jolokia, as you did in the previous example. The Jolokia agent uses the HTTP server from Java to expose its services, which the web console accesses. The agent queries the JVM using JMX to discover the available JMX MBeans, which the agent can use to obtain the requested information from the web console.

The book's source code contains an example of using the Jolokia Java agent in the chapter16/jolokia directory. You can try the example with the following Maven goal:

```
mvn compile exec:exec
```

If you're not familiar with JVM agents, we suggest you take a look at the pom.xml file from this example, as it shows how to use Maven to execute a Java application with a JVM agent.

The example can also be run without Maven. First you need to use Maven to download the needed dependencies, which can be done once:

```
mvn dependency:copy-dependencies
```

Then you need to build the example:

```
mvn clean install
```

And finally, you can run the example using this rather long command:

```
java -javaagent:lib/jolokia-jvm-1.3.7-agent.jar -cp
target/dependency/*:target/chapter16-jolokia-2.0.0.jar
camelinaction.JolokiaMain
```

Notice how to specify the JVM agent on the command line, using the `-javaagent` argument.

TIP You can find more information about the Jolokia JVM agent from its website: https://jolokia.org/agent/jvm.html.

JOLOKIA IS HOT AND AWESOME

We highly recommend Jolokia. It's an awesome library that makes JMX fun and easy to use again. A little-known fact is that the author of Jolokia, Roland Huß, is an avid chili fan, and therefore named the project after the strongest chili, Jolokia, also known as the ghost chili (https://en.wikipedia.org/wiki/Bhut_jolokia).

The fabric8 and Fuse team from Red Hat have been using Jolokia for a long time, and we provide Jolokia out of the box in many of our software projects. For example, JBoss Fuse and all Java-based Docker images come with Jolokia JVM agent installed.

Now that you've seen how to check the health of your applications, as well as dipped your toes into the waters of managing Camel applications, it's time to learn how to keep an eye on what your applications are doing.

16.3 Tracking application activity

Beyond monitoring an application's health, you need to ensure that it operates as expected. For example, if an application starts malfunctioning or has stopped entirely, it may harm business. There may also be business or security requirements to track particular services for compliance and auditing.

A Camel application used to integrate numerous systems may often be difficult to track because of its complexity. It may have inputs using a wide range of transports, and schedules that trigger more inputs as well. Routes may be dynamic if you're using content-based routing to direct messages to different destinations. And errors are occurring at all levels related to validation, security, and transport. Confronted with such complexity, how can you keep track of the behavior of your Camel applications?

You do that by tracking the traces that various activities leave behind. By configuring Camel to leave traces, you can get fairly good insight into what's going on, both in real time and after the fact. Activity can be tracked using logs, whose verbosity can be configured to your needs. Camel also offers a notification system that you can use.

Let's look at how to use log files and notifications to track activities.

16.3.1 Using log files

Monitoring tools can be tailored to look for patterns, such as error messages in logs, and they can use pattern matching to react appropriately, such as by raising an alert. Log files have been around for decades, so any monitoring tool should have good support for efficient log file scanning. Even if this solution sounds basic, it's one used extensively in today's IT world.

Log files are read not only by monitoring tools but also by people, such as operations, support, or engineering staff. That puts a burden on both Camel and your applications to produce enough evidence so that both humans and machines can diagnose the issues reported.

Camel offers four options for producing logs to track activities:

- *Using core logs*—Camel logs various types of information in its core logs. Major events and errors are reported by default.
- *Using custom logging*—You can use Camel's logging infrastructure to output your own log entries. You can do this from different places, such as from the route using the log EIP or log component. You can also use regular logging from Java code to output logs from your custom beans.
- *Using Tracer*—Tracer is used for tracing how and when a message is routed in Camel. Camel logs, at the INFO level, each and every step a message takes. Tracer offers a wealth of configuration options and features.
- *Using notifications*—Camel emits notifications that you can use to track activities in real time.

Let's look at these options in more detail.

16.3.2 Using core logs

Camel emits a lot of information at the `DEBUG` logging level and an incredible amount at the `TRACE` logging level. These levels are appropriate only for development, where the core logs provide great details for the developers.

In production, you'll want to use the `INFO` logging level, which generates a limited amount of data. At this level, you won't find information about activity for individual messages—for that, you need to use notifications or the Tracer, which we cover in section 16.3.4.

The core logs in production usage usually provide only limited details for tracking activity. Important lifecycle events such as the application being started or stopped are logged, as are any errors that occur during routing.

16.3.3 Using custom logging

Custom logging is useful if you're required to keep an audit log. With custom logging, you're in full control of what gets logged.

In EIP terms, it's the Wire Tap pattern that tackles this problem. By tapping into an existing route, you can tap messages to an audit channel. This audit channel, which is often an internal queue (SEDA or VM trans-

port), is then consumed by a dedicated audit service, which takes care of logging the messages.

Using Wire Tap for custom logging

Let's look at an example. At Rider Auto Parts, you're required to log any incoming orders. Figure 16.8 shows orders flowing in from CSV files that are then wiretapped to an audit service before moving on for further processing.

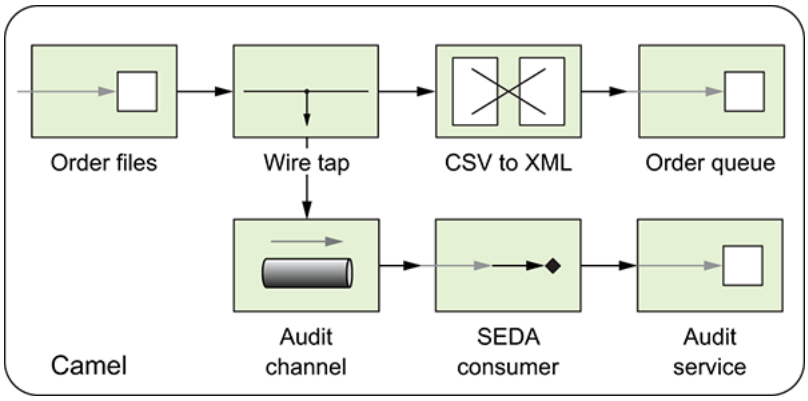

Figure 16.8 Using a wire tap to tap incoming files to an audit service before the file is translated to XML and sent to an order queue for further processing

Implementing the routes outlined in figure 16.8 in Camel is fairly straightforward, as shown in the following listing.

Listing 16.1 Using Wire Tap to tap messages for audit logging

```
public void configure() throws Exception {
    from("file://rider/orders")
    .wireTap("seda:audit")    (1)
```

(1) wireTap message to a seda queue

```
        .bean(OrderCsvToXmlBean.class)
        .to("jms:queue:orders");

    from("seda:audit")
    .bean(AuditService.class, "auditFile");    (2)
```

❷

Uses a bean to implement logic to write to audit log

```
}
```

The first route routes incoming order files. These are wiretapped to an internal SEDA queue (`"seda:audit"`) ❶ for further processing. The messages are then transformed from CSV to XML using the `OrderCsvToXmlBean` bean before being sent to a JMS queue.

The second route is used for auditing. It consumes the tapped messages and processes them with an `AuditService` bean ❷, as shown in the following listing.

Listing 16.2 Implementation of a simple audit logging service using Java bean

```
public class AuditService {

  private Log LOG = LogFactory.getLog(AuditService.class);  ❶
```

❶

Logger to use

```
    public void auditFile(String body) {
        String[] parts = body.split(",");
        String id = parts[0];
        String customerId = parts[1];
        String msg = "Customer " + customerId + " send order id " + id;
  LOG.info(msg);  ❷
```

❷

Logging an audit trail of the message

```
    }
}
```

This implementation of the `AuditService` bean has been kept simple by logging the audit messages via a Java logger library ❶. The Java bean constructs a logging message using various parts of the message body, which then gets logged ❷.

NOTE The Wire Tap EIP uses a thread pool to process the tapped message concurrently. See more details in chapter 13, section 13.3.3.

The book's source code contains this example in the chapter16/logging directory. You can try the example using the following Maven goal:

```
mvn test -Dtest=AuditTest
```

Using the Camel Log component

Camel provides a Log component that's capable of logging the Camel `Message` using a standard format at certain interesting points. To use the Log component, you route a message to it as follows:

```
public void configure() throws Exception {
    from("file://rider/orders")
        .to("log:input")
        .bean(OrderCsvToXmlBean.class)
        .to("log:asXml")
        .to("jms:queue:orders");
}
```

In this route, you use the Log component in two places. The first is to log the incoming file, and the second is after the transformation.

You can try this example using the following Maven goal from the chapter16/logging directory:

```
mvn test -Dtest=LogComponentTest
```

If you run the example, it will log the following:

```
2017-08-10 14:47:42,349 [et/rider/orders] INFO  incoming - Exchange[Exch
2017-08-10 14:47:42,351 [et/rider/orders] INFO  asXml - Exchange[Exchang
      <item><id>222</id><amount>1</amount></item></order>]
```

By default, the Log component will log the message body and its type at the `INFO` logging level. Notice that in the first log line, the type is `GenericFile`, which represents a `java.io.File` in Camel. In the second log line, the type has been changed to `String`, because the message was transformed to a String using the `OrderCsvToXmlBean` bean.

You can customize what the Log component should log using the many options it supports. Consult the Camel Log documentation for the options (http://camel.apache.org/log.html). For example, to make the messages less verbose, you can disable showing the body type and limit the length of the message body being logged using the following configuration:

```
log:incoming?showExchangePattern=false&showBodyType=false&maxChars=100
```

That results in the following output:

```
2017-08-10 14:50:36,575 [et/rider/orders] INFO asXml -
Exchange[Body: <order><id>123/id><customerId>4444</customerId
>
<date>20170810</date><item><id>222<id><amount...]
```

TIP The Log component has a `showAll` option to log everything from `Exchange`.

The Log component is used to log information from `Exchange`, but what if you want to log a message in a custom way? What you need is something like `System.out.println`, so you can input whatever `String` message you like into the log. That's where the Log EIP comes in.

Using the Log EIP

The Log EIP is built into the Camel DSL. It allows you to log a human-readable message from anywhere in a route, as if you were using `System.out.println`. It's primarily meant for developers, so they can quickly output a message to the log console. But that doesn't mean you can't use it for other purposes as well.

Suppose you want to log the filename you received as input. That's easy with the Log EIP—all you have to do is pass in the message as a String:

```
public void configure() throws Exception {
    from("file://riders/orders")
        .log("We got incoming file ${file:name} containing: ${body}")
        .bean(OrderCsvToXmlBean.class)
        .to("jms:queue:orders");
}
```

The String is based on Camel's Simple expression language, which supports the use of placeholders that are evaluated at runtime. In this example, the filename is represented by `${file:name}`, and the message body by `${body}`. If you want to know more about the Simple expression language, refer to appendix A.

You can run this example using the following Maven goal from the chapter16/logging directory:

```
mvn test -Dtest=LogEIPTest
```

If you run this example, it will log the following:

```
2017-08-10 14:52:32,576 [et/rider/orders] INFO  route1 - We got incoming
```

The Log EIP will, by default, log at the `INFO` level using the route ID as the logger name. In this example, the route isn't explicitly named, so Camel assigns it the name `route1`.

Using the Log EIP from XML DSL is also easy, as shown here:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/rider/orders"/>
    <log message="Incoming file ${file:name} containing: ${body}"/>
    <bean beanType="camelinaction.OrderCsvToXmlBean"/>
    <to uri="jms:queue:orders"/>
  </route>
</camelContext>
```

The XML example is also provided in the book's source code. You can try the example using the following Maven goal:

```
mvn test -Dtest=LogEIPSpringTest
```

The Log EIP also offers options to configure the logging level and log name, in case you want to customize those as well, as shown here in XML DSL:

```
<log message="Incoming file ${file:name} containing: ${body}"
      logName="Incoming" loggingLevel="DEBUG"/>
```

In the Java DSL, the logging level and log name are the first two parameters. The third parameter is the log message:

```
.log(LoggingLevel.DEBUG, "Incoming",
     "Incoming file ${file:name} containing: ${body}")
```

Anyone who has had to browse millions of log lines to investigate an incident knows it can be hard to correlate messages.

Using correlation IDs

When logging messages in a system, the messages being processed can easily get interleaved, which means the log lines will be interleaved as well. What you need is a way to correlate those log messages so you can tell which log lines are from which messages.

You do that by assigning a unique ID to each created message. In Camel, this ID is `ExchangeId`, which you can grab from `Exchange` using the `exchange.getExchangeId()` method.

TIP You can tell the Log component to log the `ExchangeId` using the following option: `showExchangeId=true`. When using the Log EIP, you can use `${id}` from the Simple expression language to grab the ID.

To help understand how and when messages are being routed, Camel offers Tracer, which logs message activity as it occurs.

16.3.4 Using Tracer

Tracer's role is to trace how and when messages are routed in Camel. It does this by intercepting each message being passed from one node to another during routing. Figure 16.9 illustrates this principle.

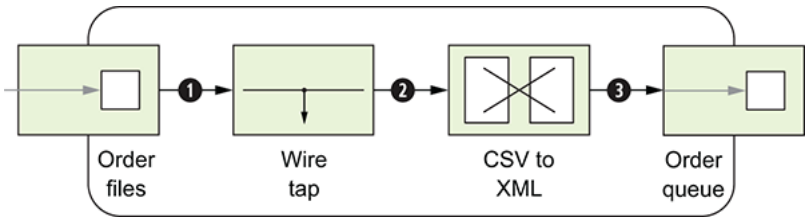

Figure 16.9 Tracer sits between each node in the route (at ❶, ❷, and ❸) and traces the message flow.

You may remember being told that Camel has a channel sitting between each node in a route—at points ❶, ❷, and ❸ in figure 16.9. `Channel` has multiple purposes, such as error handling, security, and interception. Because the Tracer is implemented as an interceptor, it falls under the control of `Channel`, which at runtime will invoke it.

To use Tracer, you need to enable it, which is easily done in either the Java DSL or XML DSL. In the Java DSL, you can enable it by calling `context.setTracing(true)` from within the `RouteBuilder` class:

```
public void configure() throws Exception {
    context.setTracing(true);
    ...
 }
```

In XML DSL, you enable Tracer from `<camelContext>` as follows:

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

When running with Tracer enabled, Camel records trace logs at the `INFO` level, which at first may seem a bit verbose. To reduce the verbosity, we've configured Tracer to not show properties and headers. Here's an example of Tracer output:

```
2017-08-13 15:37:04,072 [et/rider/orders] INFO  Tracer - ID-davsclaus-pr

2017-08-13 15:37:06,076 [et/rider/orders] INFO  Tracer - ID-davsclaus-pr

2017-08-13 15:37:06,076 [ - seda://audit] INFO  Tracer - ID-davsclaus-pr
```

The interesting thing to note from the trace logs is that the log starts with the exchange ID, which you can use to correlate messages. In this example, two IDs (highlighted in bold) are in play: `ID-davsclaus-pro-56412-1439473020118-0-202` and `ID-davsclaus-pro-56412-1439473020118-0-205`. You may wonder why you have two IDs when there's only one incoming message. That's because the wire tap creates a copy of the incoming message, and the copied message will use a new exchange ID because it's being routed as a separate process.

Next, Tracer outputs the message's current route, followed by the `from -> to` nodes. This is probably the key information when using Tracer, because you can see each individual step the message takes in Camel.

Then Tracer logs the message exchange pattern, which is either `InOnly` or `InOut`. Finally, it logs the information from the `Message`, just as the Log component would do.

DISTRIBUTED TRACING

In this chapter, we're covering tracing Camel applications in a non-clustered environment. In a distributed system, you need something more powerful to help orchestrate and gather data from all the running services in the cluster. We can recommend looking at the following open source projects for distributed tracing: Zipkin, OpenTracing, Jaeger (Uber), and Hawkular. Apache Camel has components that support Zipkin and OpenTracing along with examples you can find at https://github.com/apache/camel/tree/master/examples.

Monitoring applications via the core logs, custom logging, and Tracer is like looking into Camel's internal journal after the fact. If the log files get big, it may feel like you're looking for a needle in a haystack. Sometimes you may prefer to have Camel call you when particular events occur. That's where the notification mechanism comes into play.

16.3.5 Using notifications

For fine-grained tracking of activity, Camel's management module offers notifiers for handling internal notifications. These notifications are generated when specific events occur inside Camel, such as when an instance starts or stops, when an exception has been caught, or when a message is created or completed. The notifiers subscribe to these events as listeners, and they react when an event is received.

Camel uses a pluggable architecture, allowing you to plug in and use your own notifier, which we cover later in this section. Camel provides the following notifiers out of the box:

- `LoggingEventNotifier` —A notifier for logging a text representation of the event using the SLF4J logging framework. This means you can use loggers, such as log4j, which has a broad range of appenders that can dispatch log messages to remote servers using UDP, TCP, JMS, SNMP, email, and so on.
- `PublishEventNotifier` —A notifier for dispatching the event to any kind of Camel endpoint. This allows you to use Camel transports to broadcast the message any way you want.
- `JmxNotificationEventNotifier` —A notifier for broadcasting the events as JMX notifications. For example, management and monitoring tooling can be used to subscribe to the notifications.

You'll learn in the following sections how to set up and use an event notifier and how to build and use a custom notifier.

WARNING Because routing each exchange produces at least two (created and completed) notifications, you can be overloaded with thousands of notifications. That's why you should always filter out unwanted notifications. The `PublishEventNotifier` uses Camel to route the event message, which will potentially induce a second load on your system. That's why the notifier is configured by default to not generate new events during processing of events.

Configuring an event notifier

Camel doesn't use event notifiers by default, so to use a notifier, you must configure it. This is done by setting the notifier instance you want to use on the `ManagementStrategy`. When using the Java DSL, this is done as shown here:

```
LoggingEventNotifier notifier = new LoggingEventNotifier();
notifier.setLogName("rider.EventLog");
notifier.setIgnoreCamelContextEvents(true);
notifier.setIgnoreRouteEvents(true);
notifier.setIgnoreServiceEvents(true);
context.getManagementStrategy().addEventNotifier(notifier);
```

First you create an instance of `LoggingEventNotifier`, because you're going to log the events using log4j. Then you set the log name you want to use. In this case, you're interested in only some of the events, so you ignore the ones you aren't interested in.

The configuration when using XML DSL is a bit different, because Camel will pick up the notifier automatically when it scans the registry for beans of type `EventNotifier` on startup. This means you just have to declare a bean, like this:

```
<bean id="eventLogger"
       class="org.apache.camel.management.LoggingEventNotifier">
    <property name="logName" value="rider.EventLog"/>
    <property name="ignoreCamelContextEvents" value="true"/>
    <property name="ignoreRouteEvents" value="true"/>
```

```
    <property name="ignoreServiceEvents" value="true"/>
</bean>
```

You can also write your custom `EventNotifier` instead of using the built-in notifiers.

Using a custom event notifier

Rider Auto Parts wants to integrate an existing Camel application with the company's centralized error log database. They already have a Java library that's capable of publishing to the database, and this makes the task much easier. Figure 16.10 illustrates the situation.

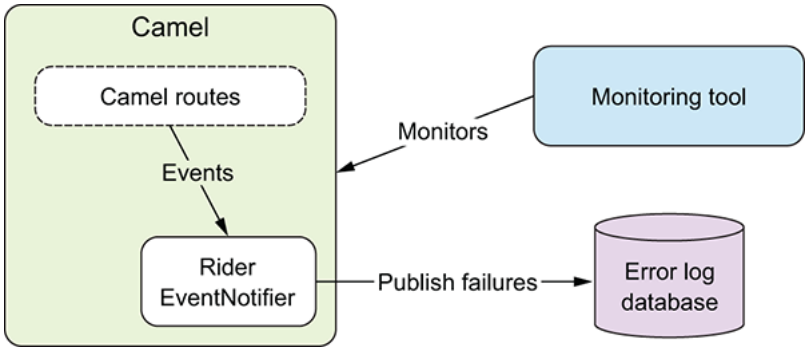

Figure 16.10 Failure events must be published into the centralized error log database using the custom `RiderEventNotifier`.

They decide to implement a custom event notifier named `RiderEventNotifier`, which uses its own Java code, allowing ultimate flexibility. Listing 16.3 shows the important snippets for implementing this. Here, you extend the `EventNotifierSupport` class, an abstract class meant to be extended by custom notifiers. If you don't want to extend this class, you can implement the `EventNotifier` interface instead. The `RiderFailurePublisher` class is the existing Java library for publishing failure events to the database.

Listing 16.3 A custom event notifier publishes failure events to a central log database

```
public class RiderEventNotifier extends EventNotifierSupport {
    private RiderFailurePublisher publisher;

    public boolean isEnabled(EventObject eventObject) {
        return eventObject instanceof ExchangeFailedEvent;  (1)
```

❶

Filters which events to trigger upon

```
    }

    public void notify(EventObject eventObject) throws Exception {    ❷
```

❷

Accepted events

```
        if (eventObject instanceof ExchangeFailedEvent) {
            ExchangeFailedEvent event = (ExchangeFailedEvent) eventObjec
            String id = event.getExchange().getExchangeId();
            Exception cause = event.getExchange().getException();
            Date now = new Date();

            publisher.publish(appId, id, now, cause.getMessage());    ❸
```

❸

Publishes failure events

```
        }
    }

    protected void doStart() throws Exception {}

    protected void doStop() throws Exception {}
}
```

The `isEnabled` method is invoked by Camel with the event being passed in as a `java.util.EventObject` instance. You use an `instanceof` test to filter for the events you're interested in, which are failure events ❶ in this example. If the `isEnabled` method returns `true`, the event is propagated to the `notify` method ❷. Then information is extracted from the event, such as the unique exchange ID and the exception message to be pub-

lished. This information is then published using the existing Java library ❸.

TIP If you have any resources that must be initialized, Camel offers `doStart` and `doStop` methods for this kind of work, as shown in listing 16.3.

The book's source code contains this example in the chapter16/notifier directory, which you can try using the following Maven goal:

```
mvn test -Dtest=RiderEventNotifierTest
```

We've now reviewed four ways to monitor Camel applications. You learned to use Camel's standard logging capabilities and to roll a custom solution when needed. In the next section, you take a further look at how to manage both Camel and your custom Camel components.

16.4 Managing Camel applications

Section 16.2 already touched on how to manage Camel, as you learned how to use JMX with Camel. This section takes deep dives into various management-related topics:

- *Camel application lifecycle*—Control your Camel application, such as stopping and starting routes, and much more using a broad range of ways with JMX, Jolokia, hawtio, and the ControlBus component
- *Camel management API*—Learn about the programming API from Camel that defines the management API.
- *Performance statistics*—Discover which key metrics Camel captures about your Camel application performance, and how to access these metrics for custom reporting and hook into monitoring and alert tools.
- *Management enabling custom components*—Learn to program your custom Camel components and Java beans so they're management enabled out of the box, as if they were first-class from the Camel release.

We'll start by looking at how to manage the lifecycles of your Camel applications.

16.4.1 Managing Camel application lifecycles

Being able to manage the lifecycles of your Camel applications is essential. You should be able to stop and start Camel in a reliable manner, and you should be able to pause or stop a Camel route temporarily, to avoid taking in new messages while an incident is being mitigated. Camel offers you full lifecycle management on all levels.

Suppose you want to stop an existing Camel route. To do that, you connect to the application with JMX as you learned in section 16.2. Figure 16.11 shows JConsole with the route in question selected.

As you can see, `route1` has been selected from the MBeans tree. You can view its attributes, which reveal various stats such as the number of exchanges completed and failed, its performance, and so on. The `State` attribute displays information about the lifecycle—whether it's started, stopped, suspended, or resumed.

To stop the route, select the Operations entry and click the `stop` operation. Then return to the Attributes entry. You should see the `State` attribute's value change to `Stopped`.

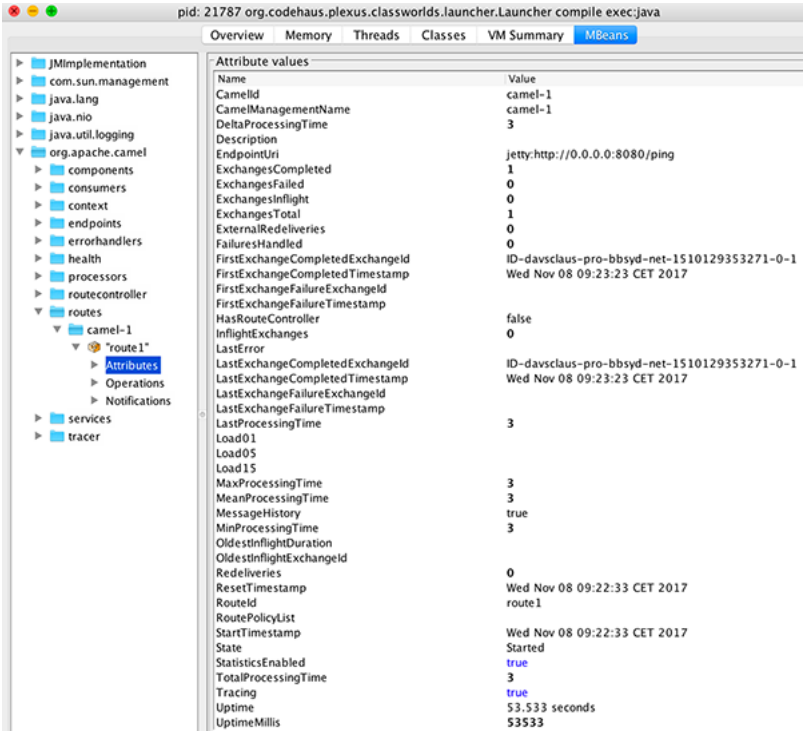

Figure 16.11 Selecting the route to manage in JConsole

TIP In JConsole, you can hover the mouse over an attribute or operation to show a tooltip with a short description.

You can also manage the lifecycle using Jolokia and hawtio.

16.4.2 Using Jolokia and hawtio to manage Camel lifecycles

Jolokia allows you to invoke JMX operations that you can use to start and stop Camel routes. We use an example from the source code to demonstrate this. In the chapter16/jolokia-embedded-war file, run the following Maven goal:

```
mvn clean install
```

Then copy the WAR file to a running Apache Tomcat in the webapps directory:

```
cp target/chapter16-jolokia-embedded-war.war /opt/apache-tomcat-8.5.23/w
```

You can then stop the Camel route using the following REST call using `curl` or from a web browser:

```
curl 'http://localhost:8080/chapter16-jolokia-embedded
-war/jolokia/exec/org.apache.camel:context=camel-1,type=routes,
name=%22route1%22/stop()'
```

From the Apache Tomcat log, you should see that Camel reports that it's stopping the route:

```
2017-08-19 11:05:20,700 [#0 - timer://foo] INFO  route 1 - I am running.
2017-08-19 11:05:22,445 [nio-8080-exec-1]  INFO  DefaultShutdownStrategy
```

And likewise, you can start the route again using the `start` operation:

```
curl 'http://localhost:8080/chapter16-jolokia-embedded-
war/jolokia/exec/org.apache.camel:context=camel-1,type=routes,
name=%22route1%22/start()'
```

Now, you may have noticed that the REST call to Jolokia is a rather long command. How do you know this command? You use hawtio, which can display the Jolokia REST call for each JMX attribute or operation. Download hawtio and install it into the running Apache Tomcat, such as by copying the WAR file:

```
cp ~/Downloads/hawtio-default-1.5.6.war /opt/apache-tomcat
-8.5.23/webapps/hawtio.war
```

Then access hawtio from a web browser: http://localhost:8080/hawtio.

The JMX tab in hawtio is similar to JConsole, with a JMX tree on the left side. In the tree, you can find the Camel application and select the route MBean. And then select the Operations sub tab, to list the operations on this MBean. Find the `stop` operation and click it. This should lead to the screen shown in figure 16.12.

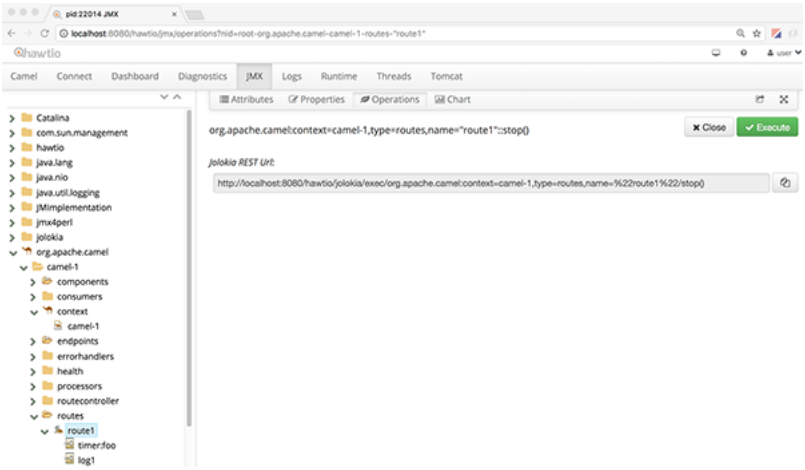

Figure 16.12 Using hawtio to manage a Camel application. In the JMX tab, the Camel route MBean is selected in the tree, and the `stop` operation is selected that allows you to invoke the operation using the Execute button. The corresponding Jolokia REST URL is also shown.

As figure 16.12 shows, you can find every Jolokia REST URL using the hawtio web console.

TIP To manage Camel routes with hawtio, the Camel tab is more useful than the JMX tab. The Camel tab displays a list of all the routes in a table. You can easily select one or more routes to start or stop with a click of a button.

The Control Bus EIP pattern from the EIP book is the next topic.

16.4.3 Using Control Bus to manage Camel

So far, the various ways for managing and controlling your Camel applications have been from the *outside*, but what if you want to do this from the *inside*? That's what the Control Bus EIP does, to monitor and manage the Camel application from within the framework.

In Camel, the Control Bus EIP is implemented as a Camel component that accepts commands to control the lifecycle of the Camel application. For example, you can send a message to a `controlbus` endpoint to stop a route or gather performance statistics.

Let's illustrate how this works with the example in the following listing.

Listing 16.4 The ping service implemented with Rest DSL and using Control Bus to control the route lifecycle

```
public class PingService extends RouteBuilder {

  public void configure() throws Exception {

    restConfiguration().component("restlet").port(8080);    ❶
```

❶

Uses restlet component on port 8080 as rest server

```
    rest("/rest").consumes("application/text").produces("application/tex

    .get("ping")    ❷
```

❷

Specifies the ping service

```
        .route().routeId("ping")
          .transform(constant("PONG\n"))
      .endRest()

      .get("route/{action}")
        .toD("controlbus:route?routeId=ping&action=${header.action}");    ❸
```

❸

Control bus to control the route

```
    }
}
```

When using the Camel Rest DSL (covered in chapter 10), you need to configure which Camel component to use as the HTTP REST-Server; in this example, we use the restlet component ❶. The ping service is an HTTP `GET` operation that returns the pong response ❷. The following rest service uses the context-path `route/{action}`, where action is a dynamic

value that's bound as a Camel header, and therefore you're using dynamic-to as the `controlbus` endpoint. This allows you to specify the action parameter using the dynamic value of the header ❸.

The book's source code contains this example in the chapter16/controlbus directory. You can try this example by running the following Maven command:

```
mvn compile exec:java
```

Then from a web browser or using a tool like `curl`, you can control the route as shown here:

```
$ curl http://localhost:8080/rest/ping
PONG
$ curl http://localhost:8080/rest/route/status
Started
$ curl http://localhost:8080/rest/route/stop
$ curl http://localhost:8080/rest/route/status
Stopped
```

This example uses three of the possible action options the control bus accepts. All the possible values supported by the action option are listed in table 16.2.

Table 16.2 Supported values of the action option from the ControlBus component

Value	Description
`start`	Starts the route. There's no return value.
`stop`	Stops the route. There's no return value.
`suspend`	Suspends the route. There's no return value.
`resume`	Resumes the route. There's no return value.
`status`	Gets the route status returned as a plain-text value such as `Started`, and/or `Stopped`.
`stats`	Gets the route performance status returned in XML format.

The control bus executes the action synchronously by default. For example, if a route takes 15 seconds to stop gracefully, that action has to complete before the Camel routing engine can continue routing the message. If you want to execute the task asynchronously, you can do that by setting the option `async` to `true`:

```
.toD("controlbus:route?routeId=ping&action=${header.action}&async=true")
```

So far, you've seen how to manage Camel applications using various tools such as JMX and the control-bus component. They all operate on top of the Camel management API, which we'll look at in the next section.

16.5 The Camel management API

The Camel management API is defined as a set of JMX MBean interfaces in the `org.apache.camel.api.management.mbean` package. These interfaces declare the JMX operations and JMX attributes that each and every Camel MBean provides. This is done using annotations to declare whether it's a read-only or read-write attribute, or an operation. For example, `ManagedCamelContextMBean` is the management API of

`CamelContext`. The following listing shows snippets of the source code for this interface.

Listing 16.5 Source code of `ManagedCamelContextMBean`

```
package org.apache.camel.api.management.mbean;

import org.apache.camel.api.management.ManagedAttribute;
import org.apache.camel.api.management.ManagedOperation;

public interface ManagedCamelContextMBean extends
        ManagedPerformanceCounterMBean {

    @ManagedAttribute(description = "Camel ID")     ❶
```

❶ Exposes attribute for management (read-only on getter)

```
    String getCamelId();

    @ManagedAttribute(description = "Camel ManagementName")     ❶
    String getManagementName();

    @ManagedAttribute(description = "Camel Version")     ❶
    String getCamelVersion();

    @ManagedOperation(description = "Starts all the routes")     ❷
```

❷ Exposes operation for management

```
    void startAllRoutes() throws Exception;
```

In the `ManagedCamelContextMBean` interface, each getter/setter becomes a JMX attribute by annotating the getter and setter methods with the `@ManagedAttribute` annotation ❶. The attribute can be read-only if there's only an annotation on the getter method. The attribute is read-write when both the getter and setter have been annotated. Operations

are regular Java methods that may return a response or not (`void`). The operation ❷ in listing 16.5 doesn't return a response and is therefore declared as `void`.

The Camel management API is vast, as we have MBeans for almost all the moving pieces that Camel uses at runtime. The MBeans are divided into categories, as previously listed in table 16.1. For example, there are MBeans for all the Camel components, data formats, endpoints, routes, processors (EIPs), thread pools, and miscellaneous services. These MBeans are defined as interfaces in the `org.apache.camel.api.management.mbean` package.

In section 16.5.2, you'll see an example of using the Camel MBean API to get runtime information from a throttler EIP. The management API can be accessed using regular JMX Java code, and also from within Camel. We cover all these aspects in the following two sections.

16.5.1 Accessing the Camel management API using Java

The Camel management API is defined as a set of JMX MBeans, which any Java client can access. This allows you to write Java code that can manage Camel applications (or any Java application that offers a JMX management API). You may want to do this as part of integrating Camel management from an existing Java management library or tool.

NOTE Using JMX to manage Java applications has some drawbacks. For example, the client must use Java, and the network transport uses Java RMI, which isn't firewall friendly, as multiple ports must be open. See the sidebar "JMX: The good, bad, and ugly" in section 16.2.2 for more details.

In this section, you'll use a simple example that runs a Camel application in a JVM. Then from another JVM, the JMX client connects to the Camel JVM and, using the JMX API, is able to obtain information about the Camel application. This scenario is illustrated in figure 16.13.

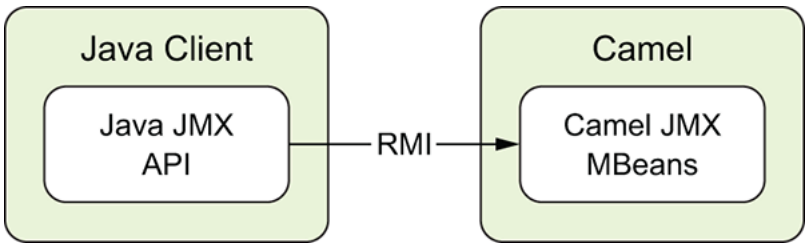

Figure 16.13 The Java client uses JMX API to connect and manage a remote Camel application, using the Camel JMX management API.

The example includes two clients. The first client uses solely the standard JMX API from the Java runtime. The second client uses a technique called a JMX proxy.

Using standard Java JMX API

For a Java JMX client to be able to remotely manage another Java application, the remote application must expose a JMX connector that the client can connect and use. The Camel application does this by setting the `createConnector` option to `true`:

```
context.getManagementStrategy()
    .getManagementAgent().setCreateConnector(true);
```

When Camel starts, a JMX connector is created and listens on port 1099 (the default port number). The URL that clients need to use for connecting to Camel is logged:

```
JMX Connector thread started and listening at:service:jmx:rmi:///jndi/rm
```

This can be handy, as the URL isn't easy to remember.

The following listing shows how to write a Java application that connects to the remote Camel application and outputs the Camel version and the uptime.

Listing 16.6 Java client connecting to a remote Camel application

```
public class JmxClientMain {
    private String serviceUrl = "service:jmx:rmi:///jndi/rmi://"
                + "localhost:1099/jmxrmi/camel";  (1)
```

JMX URL to connect to remote Camel application

```
    public static void main(String[] args) throws Exception {
        JmxClientMain main = new JmxClientMain();
        main.run();
    }

    public void run() throws Exception {
        JmxCamelClient client = new JmxCamelClient();
    client.connect(serviceUrl);    ❷
```

Connecting to the Camel application

```
    System.out.println("Version: " + client.getCamelVersion());    ❸
```

Getting information about the Camel application

```
    System.out.println("Uptime: " + client.getCamelUptime());    ❸

    client.disconnect();    ❹
```

Disconnecting from the remote Camel application

```
    }
}
```

There's more to this client, as we've implemented the lower-level JMX code in another class called `JmxCamelClient`, shown in the following listing.

Listing 16.7 Lower-level JMX API to remotely manage a Camel application

```
import javax.management.MBeanServerConnection;    ❶
```

❶

Imports only standard Java JMX API

```
import javax.management.ObjectName;    ❶
import javax.management.remote.JMXConnector;    ❶
import javax.management.remote.JMXConnectorFactory;    ❶
import javax.management.remote.JMXServiceURL;    ❶

public class JmxCamelClient {
    private JMXConnector connector;
    private MBeanServerConnection connection;

    public void connect(String serviceUrl) throws Exception {
        JMXServiceURL url = new JMXServiceURL(serviceUrl);    ❷
```

❷

Sets up URL to connect to the remote JVM

```
        connector = JMXConnectorFactory.connect(url, null);    ❸
```

❸

Connects to the remote JVM using the URL

```
        connection = connector.getMBeanServerConnection();    ❹
```

❹

Opens a connection the client uses

```
    }

    public String getCamelVersion() throws Exception {
```

```
        ObjectName on = new ObjectName("org.apache.camel:"
    + "context=camel-1,type=context,name=\"camel-1\"");  ❺
```

❺

JMX ObjectName for the CamelContext

```
    return (String) connection.getAttribute(on, "CamelVersion");  ❻
```

❻

Reads the JMX attribute using the remote connection

```
    }

    public String getCamelUptime() throws Exception {
        ObjectName on = new ObjectName("org.apache.camel:"
    + "context=camel-1,type=context,name=\"camel-1\"");  ❺
```

❺

JMX ObjectName for the CamelContext

```
    return (String) connection.getAttribute(on, "Uptime");  ❻
```

❻

Reads the JMX attribute using the remote connection

```
    }

    public void disconnect() throws Exception {
    connector.close();  ❼
```

❼

Disconnects from the remote JVM

```
        }

    }
```

You can see from the top of listing 16.7 that this class imports only the standard Java JMX API ❶. To connect to a remote JVM, three lines of code are needed (❷ ❸ ❹) to obtain an `MBeanServiceConnection` instance, which is used to read JMX attributes (❺ ❻). When the client is done, it must close ❼ so the connection is properly shut down.

The lower-level JMX code in listing 16.7 can also use the Camel management API.

Using a JMX Proxy

When using a JMX proxy, the lower-level JMX code is hidden behind a Java interface, which acts as a facade, allowing the client to use the API from the interface as if it was a regular Java API. The Camel management API is defined in Java interfaces, and therefore JMX clients can use the JMX proxy technique with Camel.

The following listing shows how to use these interfaces to simplify the code.

Listing 16.8 Lower-level JMX API using JMX proxy as a facade for Camel management

```
import javax.management.JMX;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

import org.apache.camel.api.management.mbean.ManagedCamelContextMBean;
```

❶

Imports the Camel management API

```
public class JmxCamelClient2 {

    private JMXConnector connector;
    private MBeanServerConnection connection;
    private ManagedCamelContextMBean proxy;

    public void connect(String serviceUrl) throws Exception {
        JMXServiceURL url = new JMXServiceURL(serviceUrl);
```

Connects to the remote JVM

```
        connector = JMXConnectorFactory.connect(url, null);
        connection = connector.getMBeanServerConnection();

        // create an mbean proxy so we can use the type-safe api
        ObjectName on = new ObjectName("org.apache.camel:"
            + "context=camel-1,type=context,name=\"camel-1\"");
```

JMX ObjectName for the CamelContext

```
        proxy = JMX.newMBeanProxy(connection, on,
```

Creates a JMX Proxy using ManagedCamelContextMBean as facade

```
                ManagedCamelContextMBean.class);
    }

    public void disconnect() throws Exception {
        connector.close();
    }

    public String getCamelVersion() throws Exception {
        return proxy.getCamelVersion();
```

Reads the JMX attributes using the facade

```
    }

    public String getCamelUptime() throws Exception {
        return proxy.getUptime();
```

Reads the JMX attributes using the facade

```
    }
}
```

A difference between listing 16.7 and 16.8 is that you're now using the Camel management API as the facade ❶. When doing this, you must add camel-core on the classpath. The benefit of using the facade is that the entire Camel management API is accessible, as if you were coding with Camel. Another benefit is type safety, so you don't have to worry about whether your JMX attributes and operations are defined correctly to avoid runtime errors when starting your application.

The difference becomes more apparent when comparing how listings 16.7 and 16.8 have implemented the two methods `getCamelVersion` and `getCamelUptime`. Listing 16.7 uses the clumsy JMX API, and listing 16.8 uses type-safe Java method calls.

The book's source code contains this example in the chapter16/jmx-client directory. To run the example, you need to start the Camel application first using the following:

```
mvn compile exec:java -Pserver
```

Then you can run the clients:

```
mvn compile exec:java -Pclient
mvn compile exec:java -Pclient2
```

The Camel management API can also be used from within Camel applications, where you can use Java code to obtain any information from Camel—for example, to integrate with custom monitoring systems, or gather information for reporting.

16.5.2 Using Camel management API from within Camel

Rider Auto Parts has a legacy order system that can handle only at most five orders per minute. To accommodate this, a Camel route has been put in front of the legacy system that throttles the messages:

```
from("seda:orders")
  .throttle(5).timePeriodMillis(60000).asyncDelayed().id("orderThrottler
    .to("seda:legacy");
```

You've then been told to build a solution that can report to the central monitoring system the number of orders currently being throttled by Camel. How can you build such a solution?

With all the knowledge you've gained from this chapter, you're in safe hands. You've learned that Camel exposes a wealth of information at runtime. Table 16.1 listed all the categories of information available. What you need to get is the runtime information from the Throttler EIP. As you know, the Camel management API is defined in the `org.apache.camel.api.management.mbean` package, so you take a look there, and discover the `ManagedThrottlerMBean` interface, which represents the Throttler EIP. This MBean has the information you need:

```
@ManagedAttribute(description = "Number of exchanges currently throttled
int getThrottledCount();
```

To access the information from the throttler, you need to use the Camel management API as a client from within Camel. You can do this, as you learned previously, using the JMX and JMX proxy technique. But, hey, you're using Camel, so there's usually an easier way.

The easy way with Camel is using `CamelContext` to quickly get an MBean that represents a given EIP from any of the Camel routes. This can be done using the `getManagedProcessor` method from `CamelContext`.

The following listing shows how to develop a Java class that can get the number of throttled messages.

Listing 16.9 Class that reports the number of current throttled messages

```
import org.apache.camel.CamelContext;
import org.apache.camel.CamelContextAware;
import org.apache.camel.api.management.mbean.ManagedThrottlerMBean;

public class RiderThrottlerReporter implements CamelContextAware {    (1)
```

(1) Class is CamelContextAware to have CamelContext injected

```
    private CamelContext context;

    public CamelContext getCamelContext() {
        return context;
    }

    public void setCamelContext(CamelContext camelContext) {
        this. context = camelContext;
    }

    public long reportThrottler() {
        ManagedThrottlerMBean throttler =
            context.getManagedProcessor("orderThrottler",
                    ManagedThrottlerMBean.class);    (2)
```

(2) Gets the throttler MBean

```
    return throttler.getThrottledCount();    (3)
```

(3) Returns the number of current throttled messages

```
    }
}
```

This source code is a simple Java class that gets the `CamelContext` injected because it implements `CamelContextAware` ❶. To get the current number of throttled messages, you use `CamelContext` to look up the throttler MBean ❷, which has the ID `orderThrottler` in the Camel route. With the MBean, you can easily get the number of throttled messages ❸.

The accompanying source code contains this example in the chapter16/jmx-camel directory. You can try the example using the following Maven goal:

```
mvn test -Dtest=RiderThrottlerTest
```

The throttler MBean exposes information about the Throttler EIP. Camel provides similar MBeans for all the other kinds of EIPs, such as Content-Based Router, Splitter, Aggregator, Recipient List, and so on. In addition, those MBeans have common information such as performance statistics.

16.5.3 Performance statistics

The Camel management API also exposes a lot of performance statistics, such as the average, minimum, and maximum processing time, and much more. This information is available on three levels:

- *CamelContext*—Aggregated statistics for all the Camel routes and processors
- *Route*—Aggregated statistics for the current route and its processors
- *Processor*—Statistics for each individual processor

The information that's exposed is defined in the interface `org.apache.camel.api.management.mbean.ManagedPerformanceCounterMBean` as JMX attributes. The following lists the most usable information:

```
@ManagedAttribute(description = "Number of completed exchanges")
long getExchangesCompleted() throws Exception;

@ManagedAttribute(description = "Number of failed exchanges")
long getExchangesFailed() throws Exception;
```

```
@ManagedAttribute(description = "Number of inflight exchanges")
long getExchangesInflight() throws Exception;

@ManagedAttribute(description = "Min Processing Time [milliseconds]")
long getMinProcessingTime() throws Exception;

@ManagedAttribute(description = "Mean Processing Time [milliseconds]")
long getMeanProcessingTime() throws Exception;

@ManagedAttribute(description = "Max Processing Time [milliseconds]")
long getMaxProcessingTime() throws Exception;

@ManagedAttribute(description = "Total Processing Time [milliseconds]")
long getTotalProcessingTime() throws Exception;

@ManagedAttribute(description = "Last Processing Time [milliseconds]")
long getLastProcessingTime() throws Exception;
```

All this information is available out of the box. For example, figure 16.11 is a screenshot from JConsole with the `CamelRouteMBean`. On this screenshot, you can see some of the performance statistics such as the min/mean/max values.

New information that was recently added to Camel has the oldest duration of all the current inflight messages. This indicates which of all the inflight messages is currently taking the most time:

```
@ManagedAttribute(description = "Oldest inflight exchange duration")
Long getOldestInflightDuration();
```

The book's source code contains a little example of obtaining this information every second and logging to the console. You can run this example, which is in the chapter 16/jmx-camel directory, using the following Maven goal:

```
mvn test -Dtest=OldestTest
```

The implementation is similar to what you've done in listing 16.9. `ManagedRouteMBean` is retrieved from `CamelContext` using the following code, in which `myRoute` is the route ID:

```
ManagedRouteMBean route = context.getManagedRoute("myRoute",
                                                  ManagedRouteMBean.clas
```

TIP The performance statistics can be dumped as XML using the `dumpStatsAsXml(boolean)` method. If the boolean parameter is `true`, the statistics include extended information.

The performance statistics out of the box from Apache Camel are specific to Camel. A popular metrics library is Dropwizard metrics (formerly known as codehale metrics). You can use this library with Camel using the camel-metrics component.

Using camel-metrics for route performance statistics

The camel-metrics component allows you to easily capture route performance statistics for all your Camel routes. The statistics can then be reported to various monitoring systems, or you can expose the information in JSON format, from a service over HTTP or JMX transport.

Enabling camel-metrics on all your Camel routes is done by `org.apache.camel.spi.RoutePolicyFactory`, which creates a new `RoutePolicy` for all your routes. The camel-metrics component offers `MetricsRoutePolicyFactory`, which can be used in Java or XML DSL:

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

And in XML DSL, you declare the factory as a `<bean>`, and it's automatically enabled:

```
<bean id="metricsFactory" class="org.apache.camel.component
    .metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

This uses Dropwizard metrics to capture the amount of time the route takes to process each message. This is done by a Dropwizard timer that

measures both the rate that a particular piece of code is called and the distribution of its duration.

You can access the statistics from Java or JMX, as shown in the following listing.

Listing 16.10 Accessing Dropwizard performance statics from Java code

```
MetricsRegistryService registryService =
    context.hasService(MetricsRegistryService.class);  (1)
```

(1) Obtains the MetricsRegistryService from CamelContext

```
if (registryService != null) {
  MetricRegistry registry = registryService.getMetricsRegistry();  (2)
```

(2) Gets hold of Dropwizard MetricRegistry that holds the statistics

```
  long count = registry.timer("camel-1:foo.responses").getCount();  (3)
```

(3) Gets the response timer for the foo route as count, mean, rate1, rate5, and rate15

```
    double mean = registry.timer("camel-1:foo.responses").getMeanRate();
    double rate1 = registry.timer("camel-1:foo.responses")
                    .getOneMinuteRate();
    double rate5 = registry.timer("camel-1:foo.responses")
                    .getFiveMinuteRate();
    double rate15 = registry.timer("camel-1:foo.responses")
                     .getFifteenMinuteRate();
    log.info("count={}, mean={}, rate1={}, rate5={}, rate15={}",
```

```
                count, mean, rate1, rate5, rate15);
    }
```

This code is from an example from the book's source code, which is located in the chapter16/metrics directory.

To get hold of the Dropwizard metrics, you need first to get hold of `MetricsRegistryService` from the `CamelContext` ❶. The `MetricRegistry` ❷ then gives access to all the performance statistics that Dropwizard has gathered. The camel-metrics component uses a timer for each route, using the naming pattern `camelId:routeId.responses`. The example contains two routes named foo and bar, hence the code snippet in listing 16.10 gathers the statistics for the foo route.

You can also get all the statistics at once in JSON format. You can do that using JMX, as shown in the following listing.

Listing 16.11 Using JMX to get Dropwizard performance statistics in JSON format

```
ObjectName on = new ObjectName("org.apache.camel:context=camel-1,"
    + "type=services,name=MetricsRegistryService");  ❶
```

❶ JMX ObjectName to the MetricsRegistryService

```
MBeanServer server = context.getManagementStrategy()
    .getManagementAgent().getMBeanServer();  ❷
```

❷ Gets hold of the MBeanServer

```
String json = (String) server.invoke(on, "dumpStatisticsAsJson",
                    null, null);
```

```
Invokes the dumpStatisticsAsJson operation
```

This code is from the same example shown in listing 16.10. To get the performance statistics in JSON from JMX, you need to get hold of the JMX MBean `ObjectName` ❶ that has the `dumpStatisticsAsJson` operation ❷. JMX is what hawtio uses in the Camel plugin to show the Dropwizard performance statistics in a graphical chart, as shown in figure 16.14.

You can try this example by running the following goal from Maven from the chapter16/metrics directory:

```
mvn compile hawtio:run
```

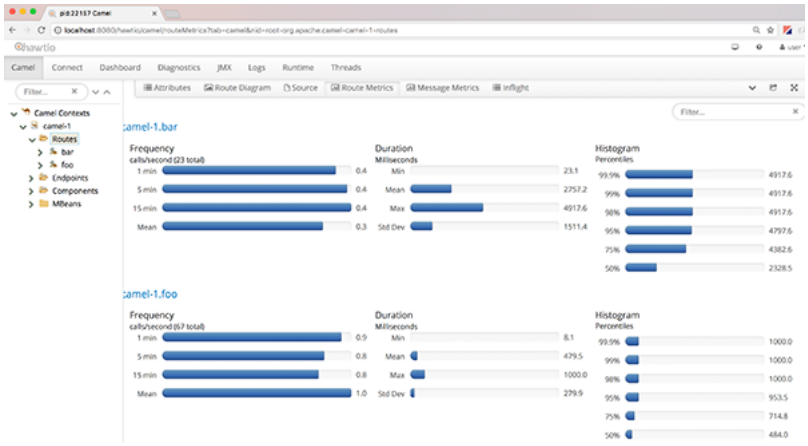

Figure 16.14 Hawtio web console showing Dropwizard performance statistics of each Camel route running in the application

And then you can find the Route Metrics tab in the Camel plugin and see the graphical chart in real time.

You may have built some Camel components of your own that you'd like to manage.

16.5.4 Management-enable custom Camel components

Suppose Rider Auto Parts has developed a Camel ERP component to integrate with its ERP system, and the operations staff has requested that the component be managed. The component has a verbosity switch that should be exposed for management. Running with verbosity enabled allows the operations staff to retrieve additional information from the logs, which is needed when some sort of issue has occurred.

Listing 16.12 shows how to implement this on the `ERPEndpoint` class, which is part of the ERP component. This code listing has been abbreviated to show only the relevant parts of the listing; the full example is in the book's source code in the chapter16/custom directory.

Listing 16.12 Management enabling a custom endpoint

```
@ManagedResource(description = "Managed ERPEndpoint")    ❶
```

❶ Exposes class as MBean

```
public class ERPEndpoint extends DefaultEndpoint {

    private String name;
    private boolean verbose;

    public ERPEndpoint(String endpointUri, Component component) {
        super(endpointUri, component);
    }


    @ManagedAttribute(description = "Verbose logging enabled")
  public boolean isVerbose() {    ❷
```

❷ Exposes attribute for management (read/write on getter and setter)

```
        return verbose;
    }

    @ManagedAttribute(description = "Verbose logging enabled")
  public void setVerbose(boolean verbose) {    ❷
```

❷ Exposes attribute for management (read/write on getter and setter)

```
        this.verbose = verbose;
    }

    @ManagedAttribute(description = "Logical name of endpoint")
  public String getName() {   ❸
```

❸

Exposes attribute for management (read-only on getter)

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManagedOperation(description = "Ping test of the ERP system")
  public String ping() {   ❹
```

❹

Exposes operation for management

```
        return "PONG";
    }

}
```

If you've ever tried using the JMX API to expose the management capabilities of your custom beans, you know it's a painful API to use. It's better to go for the easy solution and use Camel JMX. You'll notice that it uses the Camel `@ManagedResource` annotation ❶ to expose this class as an MBean. In the same way, you can expose the `verbose` property as a managed read-write attribute using the `@ManagedAttribute` ❷ annotation on the setter method. Likewise, you can expose the `name` property as a managed read-only attribute when you annotate the getter method ❸. JMX operations can also easily be exposed by annotation methods with `@ManagedOperation` ❹.

You can run the following Maven goal from the chapter16/custom directory to try this example:

```
mvn compile exec:java
```

When you do, the console will output a log line every 5 seconds, as the following route illustrates:

```
from("timer:foo?period=5000")
    .setBody().simple("Hello ERP calling at ${date:now:HH:mm:ss}")
    .to("erp:foo")
    .to("log:reply");
```

What you want to do now is turn on the verbose switch from your custom ERP component. Figure 16.15 shows how this is done from JConsole.

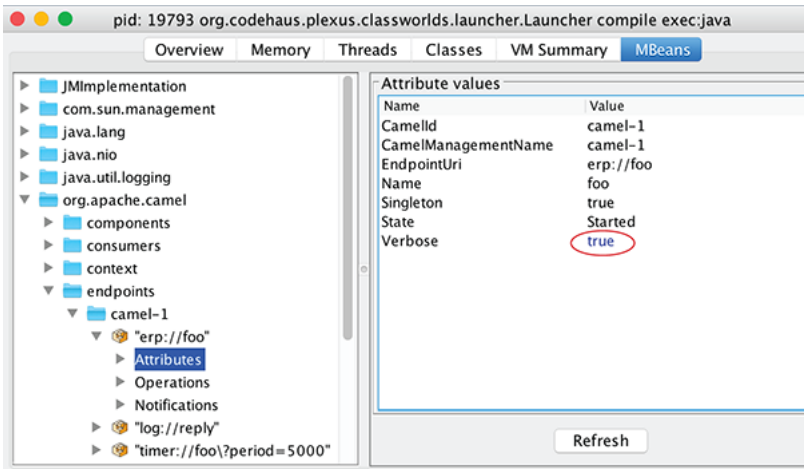

Figure 16.15 Enabling the `Verbose` attribute at runtime using JConsole. Click the Value column to edit and change the text from `false` to `true`.

As you can see, your custom component is listed under endpoints as `erp://foo`, which was the URI used in the route. The figure also shows the `Verbose` attribute. If you change this value to `true`, the console should immediately reflect this change. The first two of the following lines are from before the verbose switch was enabled. When the switch is enabled, it starts to output `Calling ERP...`:

```
2017-08-19 12:53:12,145 [0 - timer://foo] INFO  reply - Exchange[Exchang

2017-08-19 12:53:17,145 [0 - timer://foo] INFO  reply - Exchange[Exchang

Calling ERP with: Hello ERP calling at 12:53:22
```

```
2017-08-19 12:53:22,145 [0 - timer://foo] INFO  reply - Exchange[Exchang
```

What you've just learned about management-enabling a custom component is the same principle Camel uses for its components. A Camel component consists of several classes, such as `Component`, `Endpoint`, `Producer`, and `Consumer`, and you can management-enable any of those. For example, the schedule-based components, such as the Timer component, allow you to manage the consumers to adjust how often they should trigger.

It's not only custom Camel components that you can management-enable using the Camel annotations. You can also do this on regular Java beans (POJOs).

16.5.5 Management-enable custom Java beans

To enable custom Java beans for management, you use the same approach as for custom Camel components: you use the Camel management annotations.

The following listing shows how a simple Java bean can be enabled for management.

Listing 16.13 The Java bean has been management-enabled using the Camel annotations

```
@ManagedResource    (1)
```

---

(1) Exposes class as MBean

```
public class HelloBean {

    private String greeting = "Hello";

  @ManagedAttribute(description = "The greeting to use")    (2)
```

❷

Exposes attribute for management (read/write on getter and setter)

```
    public String getGreeting() {
        return greeting;
    }

    @ManagedAttribute(description = "The greeting to use")  ❷
```

❷

Exposes attribute for management (read/write on getter and setter)

```
    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    @ManagedOperation(description = "Say the greeting")  ❸
```

❸

Exposes operation for management

```
    public String say() {
        return greeting;
    }
}
```

By adding the Camel management annotations to the bean, you can expose it as an MBean. This is done by adding `@ManagedResource` on the class level ❶. Attributes on the bean can be exposed using `@ManagedAttribute` ❷, which supports read-only and read-write mode. In this example, you add the `@ManagedAttribute` on both the getter and setter, hence the attribute is read-write. For read-only mode, the annotation should be configured only on the getter. A JMX operation can be exposed using `@ManagedOperation` ❸, which in this example invokes the `say` method.

The bean must be used in a Camel route to let Camel enlist the bean as a Camel processor in JMX. This process happens when Camel is starting and all the routes are built from the model, and part of that process is to enlist MBeans that represent the various parts of the routes such as consumers, producers, endpoints, EIPs, and so on. These MBeans are categorized as shown in table 16.1.

The book's source code contains an example with the hello bean in the chapter16/custom-bean directory. In this example, the bean is used in a simple Camel route:

```
from("timer:foo?period=5s")
    .bean("hello", "say")
    .log("${body}");
```

The application uses a Camel `Main` class to boot Camel. As part of the configuring of this class, a `HelloBean` instance is created and enlisted in the Camel registry using the name `hello` ❶:

```
public static void main(String[] args) throws Exception {
    Main main = new Main();
    main.bind("hello", new HelloBean());    ❶
```

❶ Binds a HelloBean instance in the Camel registry

```
    main.addRouteBuilder(new HelloRoute());
    main.run();
}
```

You can try this example by running the following Maven goal from the chapter16/custom-bean directory:

```
mvn compile exec:java
```

If you run the example and connect to the JVM using JConsole, you can find the custom bean in the JMX tree under the Camel processor tree, as shown in figure 16.16.

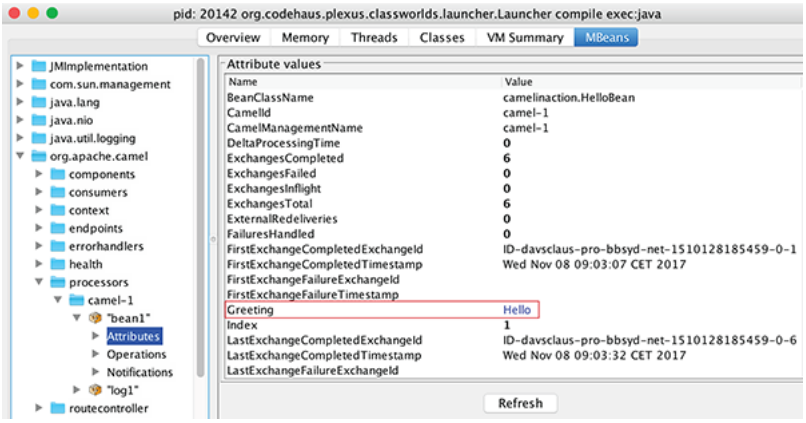

Figure 16.16 `HelloBean` with the custom `Greeting` attribute is enlisted in the Camel processor tree, also inheriting the standard set of Camel JMX attributes such as all the performance details.

Congratulations! You've now learned all there is to managing Camel applications and enlisting your custom components or Java beans for management.

16.6 Summary and best practices

A sound strategy for monitoring your applications is necessary when you take them into production. Your organization may already have strategies that must be followed for running and monitoring applications.

In this chapter, you looked at monitoring your Camel applications using health-level checks. You learned that existing monitoring tools could be used via SNMP, JMX protocols, or REST with help from Jolokia. Using JMX allows you to manage Camel at the application level, which is essential for lifecycle management, and performing functions such as stopping a route. You also looked at what Camel has to offer in terms of logging. You learned about the Camel logs and using custom logging. You also explored the Camel notification system, which is pluggable, allowing you to hook in your own notification adapter and send notifications to a third party. The last section of this chapter covered the Camel management API. We explored how you can access, from this API, details such as performance statistics. We also discussed how you can build custom Camel components or Java beans that can tap into the Camel management API as a first-class citizen.

Here are a few simple guidelines:

- *Involve the operations team*—Monitoring and management aren't afterthoughts. You should involve the operations team early in the project's

lifecycle. Your organization likely already has procedures for managing applications, which must be followed.

- *Use health checks*—For example, develop a *happy* page that does an internal health check and reports back on the status. A happy page can then easily be accessed from a web browser and monitoring tools.
- *Provide informative error messages*—When something goes wrong, you want the operations staff receiving the alert to be able to understand what the error is all about. If you throw exceptions from business logic, include descriptive information about what's wrong.
- *Use the Tracer*—If messages aren't being routed as expected, you can enable Tracer to see how they're being routed. But beware; Tracer can be verbose, and your logs can quickly fill up with lines if your application processes a lot of messages.
- *Read log files from testing*—Have developers read the log files to see which exceptions have been logged. This can help them preemptively fix issues that otherwise could slip into production.
- *Jolokia is awesome*—Jolokia brings fun back to Java management. You can now easily enable your custom Camel components and Java beans for management and let clients easily access and manage them with the help from Jolokia, using REST and HTTP.
- *Distributed tracings*—Users building distributed applications or a microservice-based architecture should consider a strategy for tracing applications. Unfortunately, this topic came too late to be covered in this book. We recommend looking at OpenTracing or Zipkin and what's happening with distributed tracing in cloud platforms, because such a functionality is likely to be a must-have feature in the near future. You can find examples of using both Zipkin and OpenTracing from Apache Camel at https://github.com/apache/camel/tree/master/examples.

The next chapter covers how to cluster your Camel applications. You'll learn how to set up Camel in a highly available cluster, how to cluster your Camel routes, and how to use clustered messaging with Camel. Another way to say it: it's clustering time!