21

Camel and the IoT

BY HENRYK KONSEK

This chapters covers

- Basic introduction to the Internet of Things (IoT)
- Suggestions for purchasing base IoT hardware
- IoT architecture
- Reasons for using Camel for IoT applications
- Camel gateway-to-data-center connectivity
- Integrating Apache Camel and Eclipse Kura

The *Internet of Things* (IoT) is a term used to describe a certain class of distributed IT systems that work with clients located on distributed devices connected to back-end messaging systems. Imagine a centered back-end system with many devices connected to it (like the one in figure 21.1); this is pretty much what IoT is.

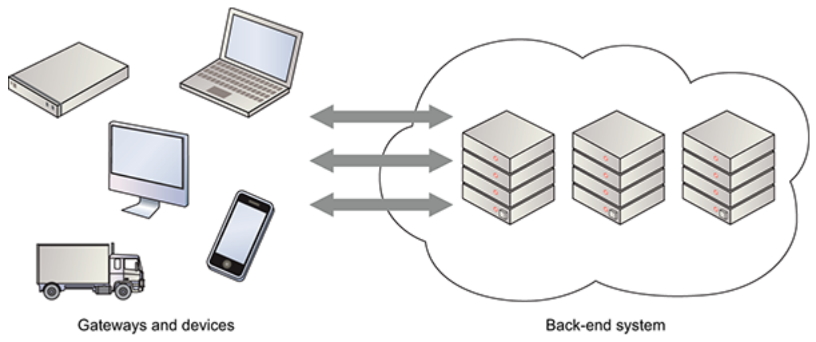


Figure 21.1 General view of the IoT architecture

These connected devices can range from mobile phones, tablets, and Raspberry Pi boards to something as large as a car. The IoT is important for our industry, because the number of connected devices in the world is growing exponentially, and we need to find a way to deal with this kind of scale of distributed clients. In addition, market predictions indicate that sooner or later the majority of developers will be involved in an IoT project of some sort.

Many of us associate the IoT with futuristic devices such as drones and robots, with hipster startups, or with Raspberry Pis. These may be true examples, but the reality is that IoT usually isn't as spectacular as we might expect. The IoT points to a future of boring verticals such as industry, army, intelligence, automotive, home automation, city infrastructure, and medical equipment. And in the majority of cases, end consumers won't even know that they're interacting with an IoT system. For example, all our dishwashers sooner or later are going to send information about their use to their manufacturers. Home furnaces are definitely not as spectacular as drones or robots, but we'll save money on furnace maintenance, as it might alert us whenever our manufacturer detects disturbing patterns in usage.

This chapter provides a gentle introduction to the Internet of Things in general. Then you'll see how Camel can help you build an IoT-class system. Finally, we'll focus on using Apache Camel and Eclipse Kura to greatly simplify providing production-grade IoT gateway solutions. But before proceeding to the details of IoT architecture, let's go shopping. You need some IoT equipment before digging into the IoT world.

21.1 The Internet of Things shopping list

This section provides an opinionated list of hardware you might want to purchase in order to start playing with Camel and the IoT. Although almost every IoT enthusiast could provide another list and argue that it's better than this one, my list is based on the following important factors:

- *Price*—You don't want to spend an excessive amount of money just to start a journey with the Internet of Things. Experiments ought to be cheap so you can play and innovate without special financial concerns.
- *Functionality*—You'll want hardware with many sensors and other features that allow you to experiment with numerous test scenarios. The more sensors (such as temperature or humidity sensors or gyrometers) and actuators that your hardware has, the more exciting experiments you can do with it.
- *Availability in various countries*—A myriad of hardware is on the market, but not all devices are easily available in every country. I've tried to

select those devices that can be easily purchased in and shipped to the most places in the world.

- *Number of tutorials and online resources available*—You want to be sure that hardware you purchase is popular enough that you can easily search online for tutorials and answers to your questions and problems. Some hardware is better than other hardware, but for learning purposes, you should use the most popular ones.

21.1.1 Raspberry Pi

First of all, you need an IoT gateway. I'll explain this term later in this chapter; for now, you just need to know that you need a small computer board that can be used to run a small Camel application. I recommend purchasing a Raspberry Pi for this purpose.

Raspberry Pi is a small computer with 1 GB of RAM and a decent CPU unit (ARM based). The board size is slightly bigger than a pack of cigarettes, which is pretty small for a computer. Raspberry Pi is extremely popular in the *maker* community, so you can find a gazillion tutorials related to this awesome piece of hardware.

To be specific, I recommend buying the Raspberry Pi 3 model B (www.raspberrypi.org/products/raspberry-pi-3-model-b/), which has significant advantages over the previous editions of the board. In particular, it has the following:

- 64-bit CPU
- Embedded Wi-Fi and Bluetooth Low Energy (BLE) units

The price of a Raspberry Pi 3 is about $45. You may also consider buying a case for your Raspberry Pi board (www.raspberrypi.org/products/raspberry-pi-case/), which costs less than $8 and is a fancy plastic cover for your board. It's not necessary to have a case for your Raspberry Pi, but a case can protect your board from physical damage. Last but not least, the Pi looks pretty spiffy in a case, compared to a naked board.

21.1.2 SD card for Raspberry Pi

Raspberry Pi doesn't come with any persistent storage. Instead, it provides a slot for a micro SD card, which can be inserted to serve as Raspberry Pi's disk. An SD card is also used to install an operating system (www.raspberrypi.org/documentation/installation/installing-images/), which the board can bootstrap on startup. Raspberry Pi supports a bunch of Linux distributions, including Raspbian (www.raspberrypi.org/downloads/raspbian/) and Fedora (https://fedoraproject.org/wiki/Raspberry_Pi). I recommend buying at least an 8 GB SD card, especially if you plan to install Raspbian. Prices of decent 8 GB micro SD cards start at $5.

21.1.3 Power bank for Raspberry Pi

Raspberry Pi is powered using a micro USB cable. In practice, you can use your Android phone charger (or any other micro USB cable connected to a laptop, for example) to power your Raspberry Pi unit. Although it's perfectly fine to run your Pi by using one of those cables, I highly recommend purchasing a power bank and powering your board with it. A *power bank* is a rechargeable battery with USB ports.

If you connect your Raspberry Pi to a power bank, you can take it anywhere you want; you aren't limited by a cable plugged into a power socket. You can even take your Pi outdoors—for example, to measure how temperature changes when you go outside your home.

I don't recommend any particular power bank producer or model. Prices of power banks start at a few bucks, but you should invest in a unit with as much power capacity as possible. The more power capacity your power bank has, the longer Raspberry Pi can run connected to it without an additional charging cycle.

21.1.4 Camera for Raspberry Pi

Another optional, but highly recommended, item for IoT developer wannabes is a camera unit. A Raspberry Pi camera, which can be purchased for less than $25, is a useful piece of equipment. It allows you to collect video streams and analyze or store those streams on your board. This is an excellent unit for creating simple applications, demonstrating how Raspberry Pi can be used to detect motion, process images, recognize objects in a stream of video, and so forth.

21.1.5 TI SensorTag

You may already have your Raspberry Pi gateway device ordered, but you won't be able to collect any data with it. You need sensors to do that. Although it's perfectly fine to purchase a bunch of sensors from your favorite electronics store and connect those to your Raspberry Pi via its general-purpose input/output (GPIO) interface, wiring all your sensors to your Pi board requires an electronics background and effort. A great alternative is raw sensors, which can be efficiently used for learning purposes.

Texas Instruments SensorTag is a small device containing a bunch of sensors and wireless connectivity units (www.ti.com/ww/en/wireless_connectivity/sensortag2015/?INTC=SensorTag&HQS=sensortag).

The TI SensorTag unit can be purchased for about $30. The device, powered by a tiny battery, brings an impressive array of sensors for that price, including a temperature sensor, humidity sensor, and gyroscope, among others. The online SensorTag User Guide provides details (see http://processors.wiki.ti.com/index.php/SensorTag_User_Guide#Sensors). You can choose the kind of wireless connectivity you're interested in when buying a unit, but choosing a model with a Bluetooth Low Energy module is a great match for Raspberry Pi 3, which happens to provide BLE connectivity out of the box as well.

In practice, purchasing a BLE-enabled SensorTag unit allows you to send telemetry readings straight into your Raspberry Pi unit and process those readings with Camel. A great match indeed! After you purchase these useful hardware devices, you're ready to dig into the IoT architecture to see how this hardware can be connected to your Camel routes.

21.2 The Internet of Things architecture

Before you dig into Camel in the context of the Internet of Things, let's look at a generic IoT architecture example so you can understand the kinds of components needed to create an operational system for connected devices. You need this knowledge to understand where Camel can help you and where it can't.

Architectures of an IoT system can be less or more complex, but we can extract a main pattern from the majority of applications of this class. Figure 21.2 presents a typical IoT system architecture.

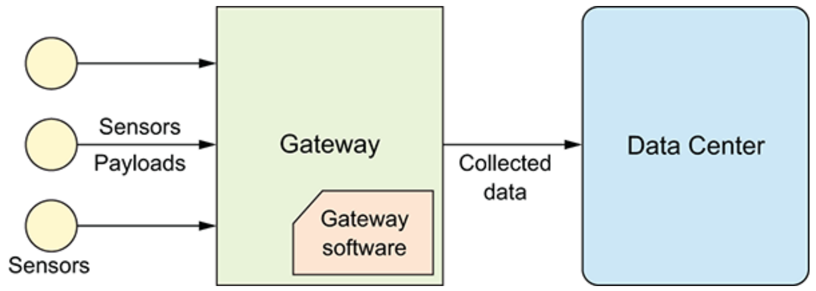


Figure 21.2 Internet of Things application architecture

First of all, the system has sensors (visible on the left of figure 21.2). *Sensors* are small devices that can collect information from an environment. This information may be readings of temperature, air humidity, vibration level, sound, or video from a digital camera. Sensors usually run embedded software and are too small to run Java and Camel. Moreover, sensor devices are usually too small to handle the whole TCP/UDP stack by themselves (they don't provide TCP/UDP connectivity, or *internet* connectivity). Instead, sensors use more lightweight and limited protocols such as BLE, ZigBee, or Z-Wave. Those protocols provide wireless connectivity limited to an effective range of less than 100 meters.

This is where the middle part of figure 21.2 comes in. You need a piece of hardware that can use short-range wireless protocols such as BLE, collect payloads from sensors, and forward these payloads to a data center using TCP or UDP connectivity via Wi-Fi or Ethernet interfaces. This hardware is called a *gateway*. It's a board similar to Raspberry Pi, but usually with a better hardware specification. The gateway is more powerful and can usually run Java Virtual Machine and take advantage of Camel (more on this later in this chapter).

The last piece of the IoT architecture puzzle is data center connectivity (represented on the right side of figure 21.2). There's no point in collecting data on a gateway if you don't send the information you've gathered into a back-end service for further analysis and processing. IoT gateways usually rely on TCP/UDP protocols such as MQTT, AMQP, CoAP, LWM2M, and HTTP/REST for data center connectivity. The gateway's task is to enqueue collected data, preprocess it, filter it if needed, and finally send it to a data center back-end endpoint.

Keep in mind that sensors aren't always connected to a gateway using wireless protocols (such as BLE or Z-Wave). It's common to wire sensors directly into gateway hardware with physical interfaces such as GPIO. Figure 21.3 shows how the architecture might look in such a scenario. As you can see, sensors are not only connected to a gateway via wireless protocols, but also embedded in the gateway itself.

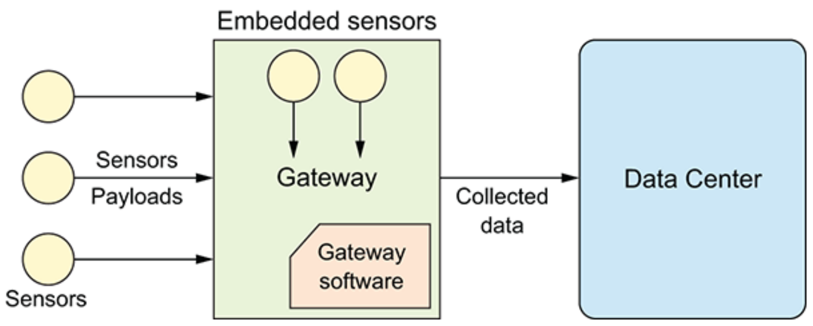


Figure 21.3 IoT application architecture with sensors embedded in a gateway

21.3 Why Camel is the right choice for the IoT

Now you have a sense of the general architecture of IoT applications. But you might still be wondering whether Camel is the right choice for these applications and the real reason to use it for IoT purposes. This section covers the most important areas where Camel functionalities excel in the context of the Internet of Things to bring real value for developers.

Before we dig into the details of using Camel in IoT scenarios, let's take a look at those reasons from 10,000 feet:

- Many Camel components are available out of the box.
- Data formats are supported by Camel.
- Redelivery capabilities.
- Throttling support.
- Possibility of performing content-based routing.
- Support for client-side load balancing.
- Runtime flow control provided by the Camel control-bus component.

Let's discuss those reasons in more detail.

21.3.1 Components

Camel comes with more than 200 components that can be used to connect to the myriad of protocol endpoints. You can use Camel components on your gateway devices to connect to a data center using the protocol that's the best choice for your device connectivity scenario: AMQP, MQTT, REST, CoAP, Kafka, or many others. Having out-of-the-box components that can be used to handle endpoint connectivity is a huge advantage of Camel, as it allows you to reduce the amount of boilerplate code you need to create, test, and maintain in order to connect your devices to the outside world.

21.3.2 Data formats

Connectivity between your gateway device and your data center is one thing, but it's essential to understand the encoding of a message sent by your field device. Camel data formats can be used on the gateway side to encode the message appropriately for your back-end system. On the server side, Camel decoders can be used to deserialize the message via a proper message format: AVRO, JSON, CSV, or others.

21.3.3 Redelivery

As stated in this book, Camel provides support for message redelivery when errors occur in data processing. This feature happens to be one of the most wanted functionalities of IoT systems. Because connected devices usually operate on flaky and highly unreliable network connections, it's common to encounter intermittent issues with endpoint connectivity.

IoT systems should be designed to recover gracefully from interruptions of message flow and should attempt to deliver messages later, when connectivity is back again. Message redelivery is one of the key pieces of the error-handling puzzle, and it's great to have it included in Camel out of the box.

21.3.4 Throttling

Ideally, you want your sensors to produce telemetry data at a fixed pace—for example, reading temperature every second, or ideally, being notified whenever temperature changes. Depending on the sensor type you're reading data from, you might from time to time receive more data payloads than you expected. Because IoT gateway devices aren't as powerful as regular server machines, you may end up with sensor readings overflowing your device. You need a mechanism that protects against such telemetry data peaks. You need a way to tell your gateway to accept at most *N* messages per second. For example, you may want to send at most one temperature reading per second and drop all other payloads (and definitely don't send the other payloads to the data center).

Creating such reliable throttling logic from scratch may be time consuming. Fortunately, Camel comes with Throttler EIP that can be used to limit the data flow. Creating throttling rules in Camel is as simple as the following snippet, which demonstrates code that you might deploy into your gateway device in order to send temperature readings stored on its local filesystem:

```
from("file:/var/temperature?delete=true")
  .throttle(1).timePeriodMillis(1000)
  .to("paho:temperature");
```

The preceding example reads telemetry payloads stored on a local filesystem and sends that data into an MQTT broker by using the Eclipse Paho component for Camel. Even if temperature payloads are generated quickly, Camel forwards only one message per second and drops all the others.

21.3.5 Content-based routing

Some IoT gateways are simple telemetry proxies. They take messages from a sensor and send them to a data center service. In many cases, you want to add some intelligence into your gateway, so you can make it smarter.

For example, you might be interested in filtering out all temperature readings that don't exceed a certain threshold. Imagine, for example, that you want to notify a data center alarm service only when a gateway detects that a temperature in a factory where a gateway is installed is higher than 30 degrees Celsius.

Another scenario that requires the gateway to be smart is a connected car that has to send important telemetry information to a data center using a paid Global System for Mobile Communications (GSM) connection. The same car could send low-priority information, but only when it's within the area of the owner's home, connected to a fast and cheap Wi-Fi network.

This kind of intelligence requires you to analyze the content of each message and define rules to react accordingly. Camel provides excellent support for defining these kinds of rules via content-based routing.

21.3.6 Client-side load balancing

As mentioned, IoT gateways are often connected to highly unreliable networks. Another common scenario is to have an IoT gateway installed in a vehicle that periodically loses connectivity to its network. In such scenarios, it's the gateway's responsibility to attempt to redeliver a message.

As connected vehicles are moving from one destination to another, it's sometimes desirable to attempt to connect to another back-end service when connecting to the first one fails. This kind of behavior can be achieved by using the Camel load balancer EIP. This kind of communication pattern, called a *Circuit Breaker*, can be easily implemented using Camel core features. Also keep in mind that Camel provides a dedicated Hystrix EIP that implements the Circuit Breaker pattern using the popular Hystrix library from Netflix.

21.3.7 Control bus

Another useful piece of Camel is called the *control bus*. It allows you to dynamically enable and disable certain parts of your routing logic. Is it useful for an IoT? Imagine that your connected vehicle is supposed to flush data cached on its local storage only when it's within the range of a trusted Wi-Fi network. The Camel control bus allows you to enable data synchronization logic only when the device gets close to a Wi-Fi network and to disable it when you're out of range.

21.4 Gateway-to-data-center connectivity

You already know what an IoT architecture could look like. You also have a general understanding of the way Camel can be applied to connected device applications. Now let's focus on the connectivity between a gateway device and data center, as this is an area where Camel can be even more useful.

21.4.1 Understanding the architecture

As already discussed, one of the areas where Camel excels when it comes to the IoT is in communication between a data center and a gateway device. This particular feature of Camel requires extra attention, because the way your messages are transferred from the field into your remote servers is one of the most important pieces of your IoT architecture. Figure 21.4 demonstrates a simplified architecture for an IoT application. The communication bits are indicated by the circle.

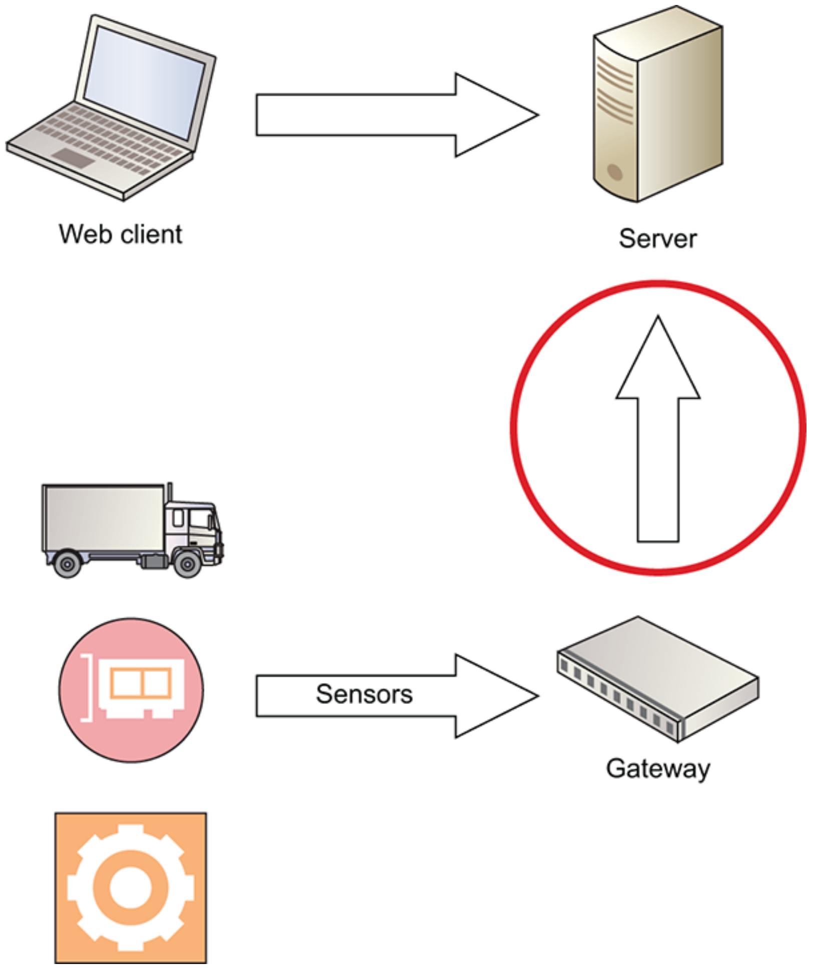


Figure 21.4 Internet of Things connectivity diagram

Protocols working in this layer of an IoT solution are usually TCP-based, but some notable exceptions occur (for example, CoAP and LWM2M are UDP-based). This is good news for back-end developers because they can reuse their existing expertise in working with TCP-based messaging protocols and apply it against a gateway device.

21.4.2 Choosing a protocol

In general, you can use any TCP/UDP protocol to transfer data between a gateway and a data center. The choice of protocol should apply the *use the right tool for the job* principle: you shouldn't assume up front that any single protocol is better than another for gateway connectivity. Keeping that in mind, I'll focus on three protocols supported by Camel: AMQP, MQTT, and REST/HTTP, are the most generic and popular gateway connectivity solutions.

AMQP

One popular choice for connectivity between a gateway and data center is Advanced Message Queuing Protocol (AMQP) 1.0. AMQP message overhead is a bit larger than MQ Telemetry Transport (MQTT) messages, but the protocol is much richer in features and better supported on the back end of the system. Many companies provide scalable AMQP message brokers (for example, Azure Service Bus by Microsoft, or A-MQ by Red Hat), whereas large-scale MQTT back-end offerings are limited.

As for message-size overhead, it's the metadata you need to add to every message that matters. For example, HTTP is considered "chatty" because it adds many verbose plain-text headers and instructions for every request. Message overhead is important, especially for gateway devices using paid GSM plans for data transfer; in this scenario, every extra byte added to each message may count. AMQP messages have a reasonable overhead, but it's not significant. I'm talking about AMQP 1.0, not AMQP 0.9 or earlier. This is an important distinction, because AMQP 1.0 is different from its earlier versions.

The main advantages of using AMQP for gateway-to-data-center connectivity are as follows:

- Good scalability of your messaging infrastructure
- Small message overhead
- Request/reply communication support
- Built-in type system that may additionally reduce message overhead
- Flow-control support (dealing with gateways generating too many messages)
- Low-latency peer-to-peer communication support

To use Camel AMQP on your gateway device, you can use the Camel AMQP component. First, add the Camel AMQP JAR to your application:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-amqp</artifactId>
</dependency>
```

After adding the Camel AMQP JAR into your classpath, register the AMQP component in your Camel context. In particular, you need to specify AMQP broker connection credentials at this stage:

```
import org.apache.camel.component.amqp.AMQPComponent;
...
CamelContex camelContex = ...;
AMQPComponent amqp = AMQPComponent.amqpComponent.amqpComponent("amqp://l
camelContex.addComponent("amqp", amqp);
```

Starting from this point, you can send your telemetry data (for example, temperature values read from a sensor and stored on a gateway filesystem) to the remote AMQP endpoint:

```
from("file:/var/temperature?delete=true")
  .marshal().json(JsonLibrary.Jackson)
  .to("amqp:temperature");
```

This demonstrates how to create a Camel route that consumes files from the local filesystem of the gateway device and sends those converted into a JSON payload to the AMQP destination named temperature.

Before executing the preceding snippet, be sure to add the Camel Jackson JAR file into your classpath. The file is needed to serialize the payload into JSON format. The following snippet demonstrates how to do this:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jackson</artifactId>
</dependency>
```

You may be wondering how temperature readings end up in the /var/temperature directory. Unfortunately, this kind of code is highly hardware dependent. You can use the GPIO interface to read from temperature sensors—for example, DS18B20 (www.sparkfun.com/products/245) connected directly to your gateway board. You can also use BLE to read temperature values from sensors supporting BLE, such as TI SensorTag. Because this chapter is too short to focus on any particular sensor details, I'll assume your temperature readings are persisted to a kind of storage on your gateway disk. Persisting sensor readings on your local disk before forwarding those to a data center is common practice.

MQTT

The other alternative for gateway-to-data-center connectivity is the MQTT protocol. Let's see how it compares to AMQP.

MQTT stands for *MQ Telemetry Transport* and is one of the most popular TCP protocols for IoT data center connectivity. The main advantage of the MQTT protocol in the context of IoT applications is that metadata overhead per message is small (just a few bytes). As mentioned in the previous AMQP section, the size of a message is an important factor for connected devices, as the latter often rely on mobile GSM connectivity. Telecommunication providers charge GSM plan users based on their data consumption, so you want to be sure that your messages don't send unnecessarily large payloads over the wire.

The main advantages of the MQTT protocol are as follows:

- Small message overhead.
- Some devices already support embedded MQTT. These devices can send and receive MQTT messages even though the devices can't run Java applications. You can expect more devices like this to be available in the future.
- Many resources and tutorials related to MQTT are available online.

If you're interested in open source MQTT brokers, you should look at the Eclipse Mosquitto and Apache Artemis projects. Also, Vert.x MQTT is a great brokerless alternative for the MQTT back end.

Camel comes with a component supporting an excellent MQTT client library: Eclipse Paho. To use the Camel Paho component on your gateway device, add the following JAR to your application:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-paho</artifactId>
</dependency>
```

Starting from this point, you can send your telemetry data (for example, temperature values read from a sensor and stored on a gateway filesystem) to a remote MQTT endpoint. The only things you need to provide to

Paho are network coordinates of your target MQTT broker (network address and port number). The easiest way to start using a Paho component is to configure the broker directly in the endpoint URI:

```
from("file:/var/temperature?delete=true")
  .marshal().json(JsonLibrary.Jackson)
  .to("paho:temperature?brokerUrl=tcp://iot.eclipse.org:1883");
```

An alternative to URI-based configuration is configuring a component instance and registering it directly into the Camel context:

```
import org.apache.camel.component.paho.PahoComponent;
...
CamelContex camelContex = ...
PahoComponent paho = new PahoComponent();
paho.setBrokerUrl("tcp://iot.eclipse.org:1883");
camelContex.addComponent("paho", paho);
```

In this case, you don't have to specify connection coordinates in your route, but you can send messages to the `"paho:topicName"` endpoint, as shown here:

```
from("file:/var/temperature?delete=true")
  .marshal().json(JsonLibrary.Jackson)
  .to("paho:temperature");
```

This demonstrates how to create a Camel route that consumes files from the local filesystem of the gateway device and sends those converted into a JSON payload to the MQTT destination named temperature. Before executing the preceding example, be sure to add the Camel Jackson JAR to your classpath.

Please refer to the preceding AMQP section in order to understand how temperature payloads can end up in the /var/temperature directory.

REST/HTTP

You already know how to use the AMQP and MQTT protocols to send messages straight into your messaging broker. That's great, because messaging-oriented solutions are usually the best choice for IoT applications. The primary reason why is that messaging protocols usually are better suited for traffic consisting of many small messages. Also, messaging protocols support not only outbound traffic from a gateway to a data center, but also inbound traffic (receiving messages from a data center without the need to poll any endpoint). This is particularly important when your gateway needs to receive a command from a data center (for example, to be restarted or to upgrade its software to a more recent version).

Keeping all these points in mind, we can safely say that REST/HTTP usually isn’t the best fit for IoT data center connectivity. The message-size overhead of REST/HTTP is rather significant, as HTTP is a chatty, text-based protocol. This is a real disadvantage for GSM-based connectivity. Because of the wide adoption of REST in server-side programming, using HTTP is a popular choice for gateway-to-data-center connectivity. This is especially the case when a gateway device doesn’t generate too many network calls to the data center or when a gateway doesn’t rely on GSM connectivity (for example, when the gateway is located near the factory floor to help with automation of the industrial process).

Camel comes with several components for HTTP clients; this example will focus on one of them. We’ll use a Netty-based HTTP client. To make this client available for your gateway application, add the following line to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4-http</artifactId>
</dependency>
```

From this point, you can send your telemetry data (for example, temperature values read from a sensor and stored on a gateway filesystem) to the remote REST/HTTP endpoint:

```
from("file:/var/temperature?delete=true")
  .marshal().json(JsonLibrary.Jackson)
  .to("netty4-http:http://my.app.com/temperature");
```

This demonstrates how to create a Camel route that consumes files from the local filesystem of the gateway device and sends those converted into a JSON payload to the HTTP URL http://my.app.com/temperature using the `POST` HTTP method. Before executing the preceding example, be sure to add the Camel Jackson JAR to your classpath.

Please refer to the previous AMQP section in order understand how temperature payloads can end up in the /var/temperature directory.

21.5 Camel and Eclipse Kura

Although deploying a standalone fat JAR application into your gateway device is a perfectly valid solution, it may not be enough for more sophisticated scenarios.

A problem with the fat JAR approach is that it's easy to deploy it into a data center using SCP/SSH, Chef, or Ansible, because server-side resources are supposed to be available all the time. You take your JAR and deploy it into a given server. If you need to upgrade your application, you deploy another version of a JAR. You don't expect your data center server to be out of the network for a certain period of time. Unfortunately, this kind of scenario is common for IoT devices. It's your device's responsibility to attempt to connect to a back-end service and ask for potential software updates. The process of installing updates into a remote device is an over-the-air (OTA) update.

Another issue with IoT software upgrades is that usually you'll want to avoid restarting the whole device during the software upgrade process in order to keep your operations continuous. The fat JAR approach, as convenient as it is, doesn't allow you to restart only certain message flows of your application.

As you can see, updating your gateway software without any additional tool dedicated for this purpose can be challenging. An interesting project that tries to solve such issues is Eclipse Kura. Kura is a low-footprint OSGi server dedicated for gateway devices. Camel provides official support for Eclipse Kura, which provides an opinionated way of deploying Camel routes into a Kura server.

The usual reason to deploy Camel routes into Eclipse Kura is to provide enterprise integration pattern support for the gateway. The other reason is to provide a myriad of Camel components for Kura. An example of integrating Camel and Kura is installing Kura on the Raspberry Pi board, reading temperature from a sensor installed into that Raspberry Pi using Kura services, and finally, forwarding a current temperature value to your data center service using Camel routes.

Figure 21.5 shows the general architecture of Camel and Kura integration.

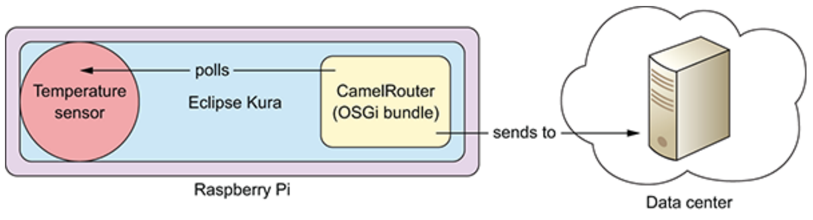


Figure 21.5 Apache Camel and Eclipse Kura integration architecture

Eclipse Kura is installed on the board—for example, on Raspberry Pi, Camel routes are installed as OSGi bundles into the Kura server. Those routes are responsible for using Kura sensor APIs and forwarding messages to a data center.

21.5.1 Starting Kura in emulator mode

It's outside the scope of this book to guide you in installing Kura on a real device board, such as Raspberry Pi. Instead, I'll describe how to run Kura in a board emulator mode as a Docker container. The latter approach is the easiest way to become familiar with Eclipse Kura. If you're interested in installing Kura on a real IoT board, see the official Kura installation guide (http://eclipse.github.io/kura/intro/raspberry-pi-quick-start.html). The Kura emulator "pretends" that it's installed on the Raspberry Pi board and allows you to start playing with the Kura web UI and Camel.

Docker is a virtual container management system that allows you to download images containing runnable software and start those on your local machine. I won't provide detailed installation instructions, and I assume you have Docker installed on your computer.

Let's start your local instance of Kura server. To run the Kura emulator, execute the following Docker command:

```
docker run --name=kura -d -p 8080:8080 ctron/kura-emulator
```

21.5.2 Defining Camel routes using the Kura web UI

When you have the Kura emulator running as a background Docker process, open your favorite web browser and navigate to http://localhost:8080. You'll be prompted to provide a username and password. Type the username `admin` and password `admin`. After providing valid credentials, you'll see the Kura web UI, which should look similar to figure 21.6.

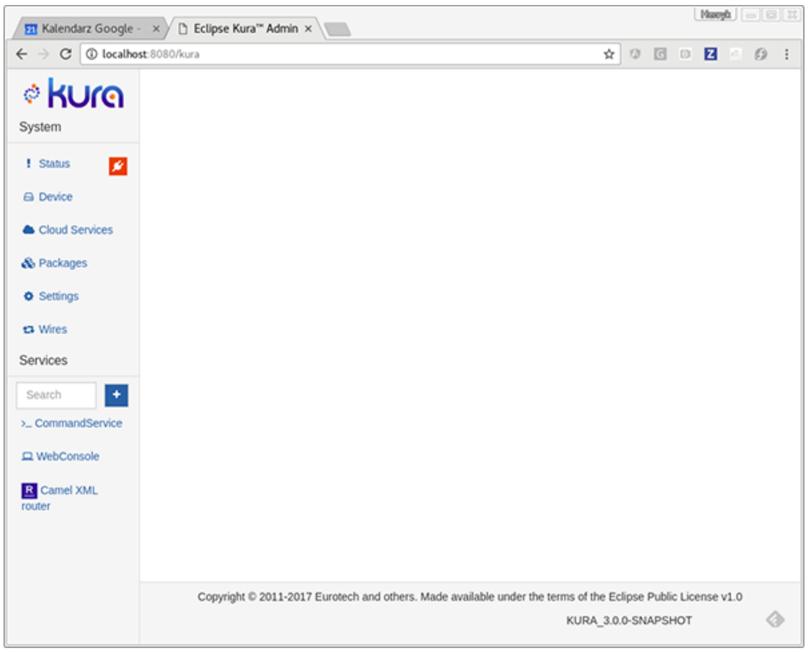


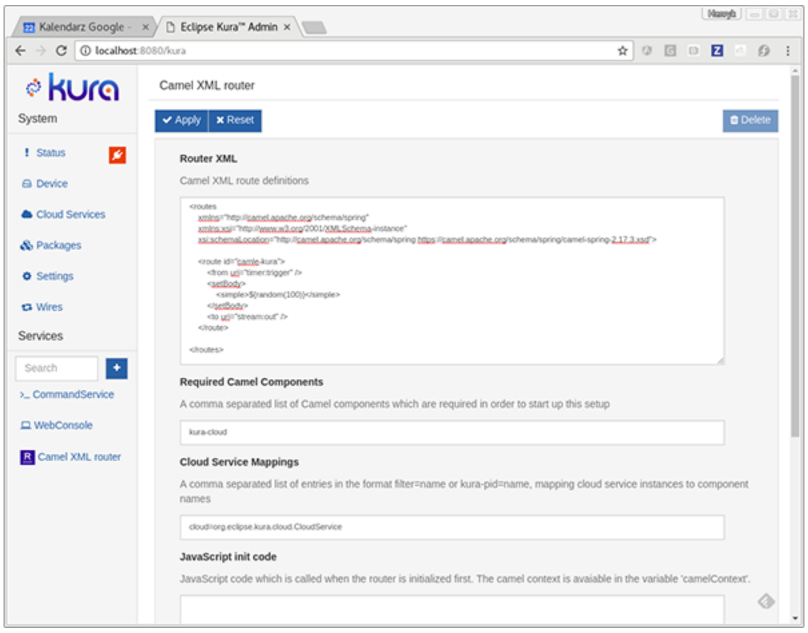
Figure 21.6 Kura web UI

As you can see, Kura provides a bunch of useful features:

- Reviewing information about a given device
- Managing OSGi bundles installed on devices
- Connecting to a back-end services platform (for example, to Eclipse Kapua)

The feature that's the most interesting is hidden under the Camel XML Router tab. Eclipse Kura happens to come with Apache Camel installed by default. Among other things, Camel support allows you to define XML routes using the web UI.

Let's try to create a new Camel XML in a Kura server. To do that, navigate to the Camel XML Router tab. After you click it, you should see a large Router XML tab (shown in figure 21.7).



Camel XML router

Router XML

Camel XML route definitions

```
<routes
  xmlns="http://camel.apache.org/schema/spring"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://camel.apache.org/schema/spring https://camel.apache.org/schema/spring/camel-spring-2.17.3.xsd">

  <route id="camle-kura">
    <from uri="timer:trigger" />
    <setBody>
      <simple>${random(100)}</simple>
    </setBody>
    <to uri="stream:out" />
  </route>

</routes>
```

Required Camel Components

A comma separated list of Camel components which are required in order to start up this setup

kura-cloud

Cloud Service Mappings

A comma separated list of entries in the format filter=name or kura-pid=name, mapping cloud service instances to component names

cloud=org.eclipse.kura.cloud.CloudService

JavaScript init code

JavaScript code which is called when the router is initialized first. The camel context is avaiable in the variable 'camelContext'.

Figure 21.7 Camel routes in the Kura web UI

Now copy the code in the following listing and paste it into the Camel XML input. After that, click the Apply button.

Listing 21.1 Camel XML route to copy into Kura web UI

```
<routes
    xmlns="http://camel.apache.org/schema/spring"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://camel.apache.org/schema/spring https://ca

    <route id="camle-kura">
        <from uri="timer:trigger"/>
        <setBody>
            <simple>${random(100)}</simple>
        </setBody>
        <to uri="stream:out"/>
    </route>

</routes>
```

This code generates a random number from 0 to 99 every second and prints that number into standard output. To see those numbers generated by Camel, navigate to your shell again and execute the following command:

```
docker logs kura
```

As a result, you should see a stream of random numbers similar to the following:

```
3
12
79
28
71
4
38
```

21.5.3 Next steps with Camel and Kura

In the previous section, you managed to start the Eclipse Kura emulator as a Docker container and deploy a Camel route into it. If you're interested in using Camel and Kura together for more serious use cases than a simple Hello World example, you should think about getting your Kura server connected to a back-end IoT platform.

Although Kura can connect to various cloud providers, the primary back-end platform for Kura is the Eclipse Kapua project. Kapua can be used to manage Kura servers connected to it and to provision resources on individual devices. In particular, it's possible to manage OSGi bundles deployed into a particular Kura server. In practice, this means you can use Kapua to manage Camel routes installed into your Kura-enabled device.

21.6 Next steps with Camel and the IoT

In this chapter, you've learned about the Internet of Things and how to take advantage of Apache Camel to create applications for this domain. You've also learned how Eclipse Kura can be used as a device deployment platform for your Camel routes. If you want to continue your IoT journey from this point, I recommend becoming familiar with the following topics.

Eclipse Kapua

When creating an IoT system, sooner or later you'll discover a need for a back-end platform for your connected devices. Eclipse Kapua is an open source back-end platform supporting Camel as a first-class citizen.

LWM2M and the Eclipse Leshan project

LWM2M is an advanced device management protocol that has a real chance to become a de facto standard for orchestrating a fleet of your connected things. An open source implementation of the LWM2M protocol is called Eclipse Leshan (www.eclipse.org/leshan/).

Eclipse Hono

Another IoT project from the Eclipse foundation is called Hono (www.eclipse.org/hono/). Hono, an extension to the AMQP messaging layer, can be used to create a scalable multitenant layer between your devices and your back-end platform. This is a project that's particularly interesting in the context of Camel, as Camel comes with good AMQP connectivity support.

21.7 Summary

This chapter covered the basic concepts of the Internet of Things (IoT) in the context of Camel-based applications. It started with a short shopping list to equip you with basic and cheap, yet powerful, IoT hardware. It also briefly discussed IoT systems architecture and challenges. And this chapter presented scenarios in which Camel brings the most benefits to the IoT messaging infrastructure.

The IoT is an extremely wide topic covered in many books. The aim of this chapter was to give you a taste of how interesting IoT development can be, especially when you're equipped with such a powerful tool as Apache Camel.

I hope you're now convinced that Camel and IoT are a good match and that you'd like to investigate this topic further. I strongly encourage you to follow Eclipse IoT communities such as Eclipse Kura or Eclipse Kapua, which are incubators of innovations related to Camel and the IoT.

Henryk Konsek is a senior software engineer at Red Hat. His areas of expertise include data-intensive back-end systems, data streaming, and messaging solutions. Henryk's primary duty at Red Hat is working on data-streaming back-end services for large-scale IoT deployments. He is a committer to *a wide range of open source projects related to messaging and the IoT, including Apache Camel, Eclipse Hono, Eclipse Kapua, and Eclipse Kura.*