

Camel IN ACTION



Claus Ibsen
Jonathan Anstey

Forewords by
Gregor Hohpe and James Strachan

SAMPLE CHAPTER

 MANNING



Camel in Action

Claus Ibsen
Jonathan Anstey

Appendix E

Copyright 2011 Manning Publications

brief contents

PART 1	FIRST STEPS	1
	1 ■ Meeting Camel	3
	2 ■ Routing with Camel	22
PART 2	CORE CAMEL	59
	3 ■ Transforming data with Camel	61
	4 ■ Using beans with Camel	93
	5 ■ Error handling	120
	6 ■ Testing with Camel	154
	7 ■ Understanding components	188
	8 ■ Enterprise integration patterns	237
PART 3	OUT IN THE WILD	281
	9 ■ Using transactions	283
	10 ■ Concurrency and scalability	315
	11 ■ Developing Camel projects	359
	12 ■ Management and monitoring	385
	13 ■ Running and deploying Camel	410
	14 ■ Bean routing and remoting	443

appendix E

Akka and Camel

by Martin Krasser

Akka aims to be the platform for the next-generation, event-driven, scalable, and fault-tolerant architectures on the JVM. One of the core features of Akka is an implementation of the Actor model. It alleviates the developer from having to deal with explicit locking and thread management. Using the Actor model raises the abstraction level and provides a better platform for building correct concurrent and scalable applications.

Akka comes with a Camel integration module that allows Akka actors to interact with communication partners over a great variety of protocols and APIs. This appendix presents selected Akka-Camel integration features by example. In particular, it covers the following:

- An introduction to Akka's Actor API
- Implementing consumer actors for receiving messages from Camel endpoints
- Implementing producer actors for sending messages to Camel endpoints
- Using and customizing Akka's `CamelService`
- Camel's `ActorComponent` for exchanging messages with actors

We'll also look at a complete routing example that combines many of the features presented in this appendix.

The examples only scratch the surface of what can be done with Akka. Interested readers may want to refer to the Akka online documentation for details (<http://akka.io>). The Actor model is also discussed on Wikipedia: http://en.wikipedia.org/wiki/Actor_model. The code examples in this appendix are available in the source code for the book, and they include a README file that explains how to build and run them.

Akka offers both a Scala API and a Java API for actors. Here, only the Scala API will be covered. We'll assume that you already have a basic knowledge of the Scala programming language.

E.1 Introducing the Akka-Camel integration

In the Actor model, each object is an actor; an actor is an entity that has a mailbox and a behavior. Messages can be exchanged between actors, and they'll be buffered in the mailbox. Upon receiving a message, the behavior of the actor is executed. An actor's behavior can be any piece of code, such as code that changes internal state, sends a number of messages to other actors, creates a number of actors, or assumes new behavior for the next message to be received.

An important property of the Actor model is that there's no shared state between actors; all communications happen by means of messages. Messages are exchanged asynchronously, but Akka supports waiting for responses as well. Also, messages are always processed sequentially by an actor. There's no concurrent execution of a single actor instance, but different actor instances can process their messages concurrently.

Akka, itself, is written in Scala, so applications often use Akka's Scala API for exchanging messages with actors. But this is not always an option, especially in the domain of application integration. Existing applications often can't be modified to use the Scala API directly, but can use file transfer on FTP servers or low-level TCP to exchange messages with other applications. For existing applications to communicate with actors, a separate integration layer is needed, and this is where Camel fits in: Camel is designed around the messaging paradigm, and it supports asynchronous message exchanges as well.

For implementing an integration layer between Akka actors and third-party applications or components, Akka provides the akka-camel module (<http://doc.akkasource.org/camel>). With the akka-camel module, it's almost trivial to implement message exchanges with actors over protocols and APIs such as HTTP, SOAP, TCP, FTP, SMTP, JMS, and others. Actors can both consume messages from and produce messages for Camel endpoints.

Another important feature of the akka-camel module is that it fully supports Camel's asynchronous, nonblocking routing engine: asynchronous message exchanges with actors can be extended to a number of additional protocols and APIs. Furthermore, all Camel components are supported in a generic way: whenever a new Camel component is released by the Camel community, it can readily be used to exchange messages with Akka actors.

The following section gives a brief introduction to Akka's Actor API and shows how to create actors and exchange messages with them.

E.2 Getting started with Akka actors

Let's start with a simple example: an actor that prints any message it receives to `stdout`. When it receives a special stop message, the actor stops itself.

An actor implementation class must extend the `Actor` trait and implement the `receive` partial function. Incoming messages are matched against the case patterns defined in `receive`.

```
import akka.actor.Actor

class SimpleActor extends Actor {
  protected def receive = {
    case "stop" => self.stop
    case msg    => println("message = %s" format msg)
  }
}
```

If one of the patterns matches, the statement after `=>` is executed. Akka's Actor API doesn't impose any constraints on the message type and format—any Scala object can be sent to an actor.

Before sending a message to the preceding actor, clients need to create and start an instance of `SimpleActor`. This is done with the `actorOf` factory method, which returns an actor reference, and by calling the `start` method on that reference. This can be done as follows, where the client creates the actor and sends two messages to it:

```
import akka.actor.Actor._

val simpleActor = actorOf[SimpleActor].start

simpleActor ! "hello akka"
simpleActor ! "stop"
```

The actor reference, once it's created, is also used for sending messages to the actor. With the `!` (bang) operator, clients send messages with fire-and-forget semantics. The `!` operator adds the message to the actor's mailbox, and the actor processes the message asynchronously. The preceding example first sends a "hello akka" string that matches the second pattern in the `receive` method. The message is therefore written to `stdout`. The "stop" message sent afterwards is matched by the first pattern, which stops the actor. Note that sending a message to a stopped actor throws an exception.

To run the example from the `appendixE` directory, enter `sbt run` on the command line and select `camelInaction.SectionE2` from the list of main classes.

NOTE The source code for this book contains a `README` file in the `appendixE` directory that explains how to install and set up the Simple Build Tool (`sbt`). All the examples in this appendix can be run by executing `sbt run` from the command line. This command displays a menu, from which you can choose the example to run by its number.

The preceding example only uses a small part of Akka's Actor API. It demonstrates how clients can send messages to actors and how actors can match and process these messages. The next step is to add an additional interface to the actor so that it can receive messages via a Camel endpoint.

E.3 Consuming messages from Camel endpoints

If you want to make actors accessible via Camel endpoints, actor classes need to mixin the `Consumer` trait and implement the `endpointUri` method. Consumer actors can be used for both one-way and request-response messaging.

E.3.1 One-way messaging

The following listing extends the example from the previous section and enables the actor to receive messages from a SEDA endpoint.

Listing E.1 Actor as consumer receiving messages from a SEDA endpoint

```
import akka.actor.Actor
import akka.camel.{Consumer, Message}

class SedaConsumer extends Actor with Consumer {
  def endpointUri = "seda:example"

  protected def receive = {
    case Message("stop", headers) => self.stop
    case Message(body, headers)   => println("message = %s" format body)
  }
}
```

The `endpointUri` method is implemented to return a SEDA endpoint URI. This causes the actor to consume messages from the `seda:example` queue once it is started. One important difference, compared to `SimpleActor`, is that the received messages are of type `Message`, which are immutable representations of Camel messages. A `Message` object can be used for pattern matching, and the message body and headers can be bound to variables, as shown in listing E.1.

For any consumer actor to receive messages, an application needs to start a `CamelService` before starting a consumer actor:

```
import akka.actor.Actor._
import akka.camel._

val service = CamelServiceManager.startCamelService

service.awaitEndpointActivation(1) {
  actorOf[SedaConsumer].start
}

for (template <- CamelContextManager.template) {
  template.sendBody("seda:example", "hello akka-camel")
  template.sendBody("seda:example", "stop")
}

service.stop
```

A `CamelService` can be started with `CamelServiceManager.startCamelService`. The started `CamelService` instance is returned from the `startCamelService` method call.

When a consumer actor is started, the `CamelService` is notified and it will create and start (activate) a route from the specified endpoint to the actor. This is done

asynchronously, so if an application wants to wait for a certain number of endpoints to be activated, it can do so with the `awaitEndpointActivation` method. This method blocks until the expected number of endpoints have been activated in the block that follows that method.

The application is then ready to produce messages to the `seda:example` queue. A Camel `ProducerTemplate` for sending messages can be obtained via `CamelContextManager.template`.

This returns an instance of `Option[ProducerTemplate]` that can be used with a `for comprehension`. If the `CamelService` has been started, the body of the `for comprehension` will be executed once with the current `ProducerTemplate` bound to the `template` variable. If the `CamelService` hasn't been started, the `for` body won't be executed at all.

Alternatively, applications may also use `CamelContextManager.mandatoryTemplate` which returns the `ProducerTemplate` directly or throws an `IllegalStateException` if the `CamelService` hasn't been started.

The application first sends a message that will be printed to `stdout`, and then it sends a special stop message that stops the actor. Alternatively, clients can also send `Message` objects directly via the native Actor API:

```
actor ! Message("hello akka-camel")
```

Finally, the application gracefully shuts down the `CamelService`.

`SedaConsumer` is an actor that doesn't reply to the initial sender. Request-response message exchanges require a minor addition, as shown in the next section for a consumer actor with a Jetty endpoint.

E.3.2 Request-response messaging

Listing E.2 shows how an actor can reply to the initial sender using the `self.reply` method. In this example, the initial sender is an HTTP client that communicates with the actor over a Jetty endpoint.

Listing E.2 Actor acting as consumer which sends back replies to sender

```
class HttpConsumer1 extends Actor with Consumer {
  def endpointUri = "jetty:http://0.0.0.0:8811/consumer1"

  protected def receive = {
    case msg: Message => self.reply("received %s"
                                   format msg.bodyAs[String])
  }
}
```

NOTE Valid initial senders can either be other actors or Camel routes to that actor. This is an implementation detail: for the receiving actor, both initial sender types appear to be actor references.

When POSTing a message to `http://localhost:8811/consumer1`, the actor converts the received message body to a `String` and prepends "received " to it. The result is

returned to the HTTP client with the `self.reply` method. The `self` object is the self-reference to the current actor.

The reply message is a plain `String` that's internally converted to a Camel `Message` before returning it to the Jetty endpoint. If applications additionally want to add or modify response headers, they can do so by returning a `Message` object containing the response body and headers. The next example creates an XML response and sets the `Content-Type` header to `application/xml`.

Listing E.3 An actor sending back XML messages as reply to sender

```
class HttpConsumer2 extends Actor with Consumer {
  def endpointUri = "jetty:http://0.0.0.0:8811/consumer2"

  protected def receive = {
    case msg: Message => {
      val body = "<received>%s</received>" format msg.bodyAs[String]
      val headers = Map("Content-Type" -> "application/xml")
      self.reply(Message(body, headers))
    }
  }
}
```

Consumer actors wait for their clients to initiate message exchanges. If actors themselves want to initiate a message exchange with a Camel endpoint, a different approach must be taken.

E.4 Producing messages to Camel endpoints

For producing messages to Camel endpoints, actors have two options. They can either use a Camel `ProducerTemplate` directly or mixin the `Producer` trait in the actor implementation class. This section will cover the use of the `Producer` trait. The use of the Camel `ProducerTemplate` is explained in appendix C.

The advantage of using the `Producer` trait is that actors fully leverage Camel's asynchronous routing engine. To produce messages to a Camel endpoint, an actor must implement the `endpointUri` method from the `Producer` trait, as follows.

Listing E.4 An actor as a producer sending messages to the defined HTTP endpoint

```
import akka.actor.Actor
import akka.camel.Producer

class HttpProducer1 extends Actor with Producer {
  def endpointUri = "http://localhost:8811/consumer2"
}
```

In this example, any message sent to an instance of `HttpProducer1` will be POSTed to `http://localhost:8811/consumer2`, which is the endpoint of the consumer actor in listing E.3 (these two actors are communicating over HTTP).

In the following code example, an application sends a message to the producer actor with the `!!` (bangbang) operator, which means it sends and receives eventually:

the message is sent to the actor asynchronously, but the caller also waits for a response.

```
import akka.actor.Actor._
import akka.camel.{Failure, Message}

val httpProducer1 = actorOf[HttpProducer1].start

httpProducer1 !! "Camel rocks" match {
  case Some(m: Message) => println("response = %s" format m.bodyAs[String])
  case Some(f: Failure) => println("failure = %s" format f.cause.getMessage)
  case None              => println("timeout")
}
```

The return type of `!!` is `Option[Any]`. For a producer actor at runtime, the type can be one of the following:

- `Some(Message)` for a normal response
- `Some(Failure)` if the message exchange with the endpoint failed
- `None` if waiting for a response timed out

The timeout for a response is defined by the `timeout` attribute on the actor reference (`ActorRef.timeout`). The default value is 5000 (ms) and it can be changed by applications.

As you've probably realized, `HttpProducer1` doesn't implement a `receive` method. This is because the `Producer` trait provides a default `receive` implementation that's inherited by `HttpProducer1`. The default behavior of `Producer.receive` is to send messages to the specified endpoint and to return the result to the initial sender. In the preceding example, the initial sender obtains the result from the `!!` method call.

Actor classes can override the `Producer.receiveAfterProduce` method to, for example, forward the result to another actor, instead of returning it to the original sender. In this case, the original sender should use the `!` operator for sending the message; otherwise it will wait for a response until timeout. In the following example, the producer actor simply writes the result to `stdout` instead of returning it to the sender.

```
class HttpProducer2 extends Actor with Producer {
  def endpointUri = "http://localhost:8811/consumer3"

  override protected def receiveAfterProduce = {
    case m: Message => println("response = %s" format m.bodyAs[String])
    case f: Failure => println("failure = %s" format f.cause.getMessage)
  }
}
```

A producer actor by default initiates in-out message exchanges with the specified endpoint. For initiating in-only message exchanges, producer implementations must either override the `Producer.oneway` method to return `true` or mixin the `Oneway` trait. The following code shows the latter approach by mixin the `Oneway` trait:

```
import akka.actor.Actor
import akka.camel.{Oneway, Producer}

class JmsProducer extends Actor with Producer with Oneway {
  def endpointUri = "jms:queue:test"
}
```

For producer and consumer actors to work with Camel, applications need to start the `CamelService`, which sets up the `CamelContext` for an application. The next section shows some examples of how applications can customize the process of setting up a `CamelContext`.

E.5 Customizing CamelService

When started, a `CamelService` creates a default `CamelContext` and makes it accessible via the `CamelContextManager` singleton. Applications can access the current `CamelContext` within a `for (context <- CamelContextManager.context) { ... }` comprehension and make any modifications they want. Alternatively, a `CamelContext` can also be obtained directly via `CamelContextManager.mandatoryContext`. This will throw an `IllegalStateException` if the `CamelService` hasn't been started.

But modifying a `CamelContext` after it's been started isn't always an option. For example, applications may want to use their own `CamelContext` implementations or to make some modifications before the `CamelContext` is started. This can be achieved either programmatically or declaratively, as explained in the following subsections.

E.5.1 Programmatic customization

If an application wants to disable JMX, for example, it should do so before the `CamelContext` is started. This can be achieved by manually initializing the `CamelContextManager` and calling `disableJMX` on the created `CamelContext`:

```
import akka.camel._
CamelContextManager.init
CamelContextManager.context.disableJMX
CamelServiceManager.startCamelService
```

By default, the `CamelContextManager.init` method creates a `DefaultCamelContext` instance, but applications may also pass any other `CamelContext` instance as an argument to the `init` method:

```
import akka.camel._
val camelContext: CamelContext = ...
CamelContextManager.init(camelContext)
CamelContextManager.context.disableJMX
CamelServiceManager.startCamelService
```

When the `CamelService` is started, it will also start the user-defined `CamelContext`.

E.5.2 Declarative customization

Alternatively, a CamelService can be created and configured within a Spring application context. The following Spring XML configuration uses the Akka and Camel XML namespaces to set up a CamelService and a CamelContext respectively. The custom CamelContext is injected into the CamelService.

Listing E.5 Spring XML file setting up Akka and Camel


```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://www.akkasource.org/schema/akka"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.akkasource.org/schema/akka
http://scalablesolutions.se/akka/akka-1.0.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

  <camel:camelContext id="camelContext">

</camel:camelContext>

  <akka:camel-service id="camelService">
    <akka:camel-context ref="camelContext" />
  </akka:camel-service>

</beans>
```




After creating an application context from the XML configuration, a CamelService **1** runs and listens for consumer actors to be started. If an application wants to interact with the CamelService directly, it can obtain the running CamelService instance either via CamelServiceManager.service, CamelServiceManager.mandatoryService, or directly from the Spring application context.

The following code shows how you can use the former approach for obtaining the CamelService from the CamelServiceManager:

```
import org.springframework.context.support.ClassPathXmlApplicationContext
import akka.actor.Actor._
import akka.camel._

val appctx = new ClassPathXmlApplicationContext("/sample.xml")
val camelService = CamelServiceManager.mandatoryService

appctx.destroy
```



When the application context (appctx) is destroyed, the CamelService and the CamelContext are shut down as well.

In all the examples so far, routes to actors have been automatically created by the CamelService. Whenever a consumer actor has been started, this was detected by the CamelService and a route from the actor's endpoint to the actor itself was added to

the current CamelContext. Alternatively, applications can also define custom routes to actors by using Akka's ActorComponent.

E.6 The Actor component

Accessing an actor from a Camel route is done with the Actor component, a Camel component for producing messages to actors. For example, when starting SedaConsumer from listing E.1, the CamelService adds the following (simplified) route to the CamelContext:

```
from("seda:example").to("actor:uuid:<actoruuid>")
```

The route starts from `seda:example` and goes to the started `SedaConsumer` instance, where `<actoruuid>` is the consumer actor's UUID. An actor's UUID can be obtained from its reference. Endpoint URIs starting with the actor scheme are used to produce messages to actors.

The Actor component isn't only intended for internal use but can also be used by user-defined Camel routes to access any actor; in this case, the target actor doesn't need to implement the Consumer trait. The Actor component also supports Camel's asynchronous routing engine and allows asynchronous in-only and in-out message exchanges with actors.

Listing E.6 shows an example: a user-defined Camel route that sends a message to an instance of `HttpProducer1` (the producer actor from listing E.4). This producer actor sends a message to `http://localhost:8811/consumer2`. If the communication with the HTTP service succeeds, the producer actor returns a `Message` object containing the service response or a `Failure` object with the cause of the failure. If the producer can't connect to the service, for example, the failure cause will be a `ConnectException`. This exception can be handled in the route. Other exceptions are possible, but they aren't included here, to keep the example simple.

Listing E.6 Camel route sending message to Akka actor

```
import java.net.ConnectException
import org.apache.camel.builder.RouteBuilder
import akka.actor.Actor._
import akka.actor.Uid
import akka.camel._

class CustomRoute(uuid: Uid) extends RouteBuilder {
  def configure = {
    from("direct:test")
      .onException(classOf[ConnectException])
        .handled(true).transform.constant("feel bad").end
      .to("actor:uuid:%s" format uuid)
  }
}

val producer = actorOf[HttpProducer1].start
CamelServiceManager.startCamelService
```

Camel route
sending to actor

```

for (context <- CamelContextManager.context;
     template <- CamelContextManager.template) {
  context.addRoutes(new CustomRoute(producer.uuid))
  template.requestBody("direct:test", "feel good", classOf[String]) match {
    case "<received>feel good</received>" => println("communication ok")
    case "feel bad" => println("communication failed")
    case _ => println("unexpected response")
  }
}

```

After starting the target actor and a CamelService, the application adds the user-defined route to the current CamelContext. It then uses a `ProducerTemplate` to initiate an in-out exchange with the route and tries to match the response, where the response either comes from the HTTP service or from the error handler.

We'll now move on to a more advanced example that applies many of the features described so far. It combines different actor types to a simple integration solution for transforming the content of a web page.

E.7 A routing example

Camel applications usually define message-processing routes with the Camel DSL. Akka applications can alternatively define networks of interconnected actors, in combination with consumer and producer actors, to set up message-processing routes.

This section shows a simple example of how to set up a message-processing route with actors. The goal of this example is to display the Akka homepage (<http://akka-source.org>) in a browser, with occurrences of *Akka* in the page content replaced with an uppercase *AKKA*. The example combines a consumer and a producer actor with another actor that transforms the content of the homepage.

The setup of the example application is sketched in figure E.1; the corresponding code is shown in listing E.7.

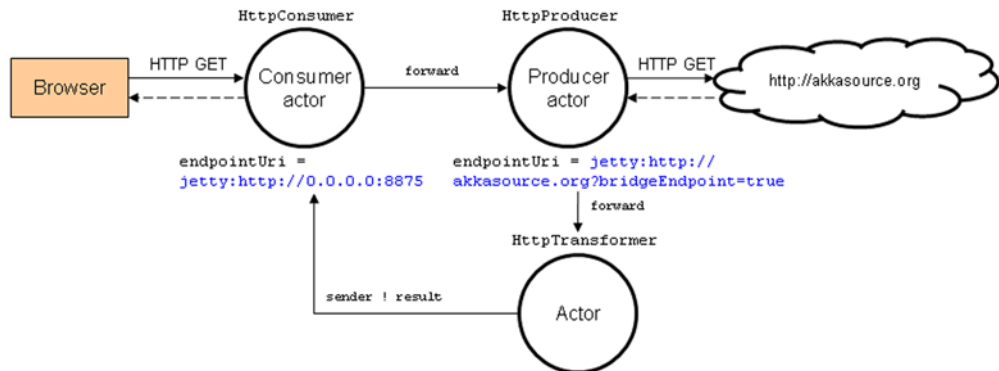


Figure E.1 The setup of the example application. A consumer and a producer actor provide connectivity to external systems. The consumer actor receives requests from a browser and forwards them to a producer actor, which fetches the HTML page. The HTML page is then forwarded to an actor that transforms the content of the page and returns the transformation result to the initial sender, so that it can be displayed in the browser.

Listing E.7 Akka consumer and producer example

```

import org.apache.camel.Exchange
import akka.actor.Actor._
import akka.actor.{Actor, ActorRef}
import akka.camel._

class HttpConsumer(producer: ActorRef) extends Actor with Consumer {
  def endpointUri = "jetty:http://0.0.0.0:8875/"
  protected def receive = {
    case msg => producer forward msg
  }
}

class HttpProducer(transformer: ActorRef) extends Actor with Producer {
  def endpointUri = "jetty:http://akkasource.org/?bridgeEndpoint=true"

  override protected def receiveBeforeProduce = {
    case msg: Message =>
      msg.setHeaders(msg.headers(Set(Exchange.HTTP_PATH)))
  }

  override protected def receiveAfterProduce = {
    case msg => transformer forward msg
  }
}

class HttpTransformer extends Actor {
  protected def receive = {
    case msg: Failure => self.reply(msg)
    case msg: Message => self.reply(msg.transformBody[String] {
      _ replaceAll ("Akka ", "AKKA ")
    })
  }
}

CamelServiceManager.startCamelService

val httpTransformer = actorOf(new HttpTransformer).start
val httpProducer = actorOf(new HttpProducer(httpTransformer)).start
val httpConsumer = actorOf(new HttpConsumer(httpProducer)).start

```

HttpConsumer is an actor that accepts HTTP GET requests on port 8875 and is configured to forward requests to an instance of HttpProducer. When an actor forwards a message to another actor, it forwards the initial sender reference as well. This reference is needed later for returning the result to the initial sender.

A forwarded message causes the HttpProducer to send a GET request to <http://akkasource.org>. Before doing so, it drops all headers **1** from the request message, except for the HTTP_PATH header, which is needed by the bridge endpoint. This preprocessing is done in the producer's receiveBeforeProduce method. The received HTML content from <http://akkasource.org> is then forwarded **2** to an instance of HttpTransformer.

The HttpTransformer **3** is an actor that replaces all occurrences of Akka in the message body with an uppercase AKKA and returns the result to the initial sender. The reference to the initial sender has been forwarded by the producer actor **2**.

After starting the `CamelService`, the actors are wired and started.

To run the example from the `appendixE` directory, enter `sbt run` on the command line, and select `camelinaction.SectionE7` from the list of main classes. Access <http://localhost:8875> from a browser, and a transformed version of the Akka homepage should be displayed.

Finally, it should be noted that the actors and the Jetty endpoints in this example exchange messages asynchronously; no single thread is allocated or blocked for the full duration of an in-out message exchange. Although it's not critical for this example, exchanging messages asynchronously can help to save server resources, especially in applications with long-running request-response cycles and frequent client requests.

E.8 Summary

This appendix shows you how to exchange messages with Akka actors over protocols and APIs supported by the great variety of Camel components. You saw how consumer actors can receive messages from Camel endpoints and producer actors can send messages to Camel endpoints. Setting up a Camel endpoint for an actor is as easy as defining an endpoint URI for that actor.

The prerequisite for running consumer and producer actors is a started `CamelService` that manages an application's `CamelContext`. Applications can configure the `CamelService` either programmatically or declaratively based on custom Spring XML schemas provided by Akka and Camel.

You also saw how to use Akka's Actor component to access any actor from a user-defined Camel route. Actor endpoints are implemented by defining an actor endpoint URI in the route. A routing example finally demonstrated how to combine consumer and producer actors to develop a simple integration solution for transforming the content of a web page.

The features described in this appendix are those of Akka version 1.0. If you want to keep track of the latest development activities, get in touch with the Akka community via the Akka User List (<http://groups.google.com/group/akka-user>) and the Akka Developer List (<http://groups.google.com/group/akka-dev>). Your feedback is highly welcome.

Camel IN ACTION

Claus Ibsen and Jonathan Anstey

Forewords by Gregor Hohpe and James Strachan

Apache Camel is a Java framework that lets you implement the standard enterprise integration patterns in a few lines of code. With a concise but sophisticated DSL you snap integration logic into your app, Lego-style, using Java, XML, or Scala. Camel supports over 80 common transports such as HTTP, REST, JMS, and Web Services.

Camel in Action is a Camel tutorial full of small examples showing how to work with the integration patterns. It starts with core concepts like sending, receiving, routing, and transforming data. It then shows you the entire lifecycle and goes in depth on how to test, deal with errors, scale, deploy, and even monitor your app—details you can find only in the Camel code itself. Written by the developers of Camel, this book distills their experience and practical insights so that you can tackle integration tasks like a pro.

What's Inside

- Valuable examples in Java and XML
- Explanations of complex patterns
- Error handling, testing, deploying, managing, and running Camel
- Accessible to beginners, useful to experts

About the Authors

Claus Ibsen is project lead on Camel and an integration specialist. **Jonathan Anstey** is Camel committer and engineer specializing in enterprise integration. Both work for FuseSource Corporation.

For online access to the authors and a free ebook for owners of this book, go to manning.com/CamelinAction



"I highly recommend this book. It kicks ass!"

—James Strachan, Cofounder of Apache Camel

"Strikes the right balance between core concepts and running code."

—Gregor Hohpe, Coauthor of *Enterprise Integration Patterns*

"Comprehensive guide to enterprise integration with Camel."

—Gordon Dickens
Chariot Solutions

"A deep book ... with great examples"

—Jeroen Benckhuijsen
Atos Origin

"Great content from the source developers."

—Domingo Suarez Torres
SynergyJ

"A must-have."

—Tjits Rademakers, Atos Origin

ISBN-10: 1-135182-36-6

ISBN-13: 978-1-135182-36-8



9 781935 182368