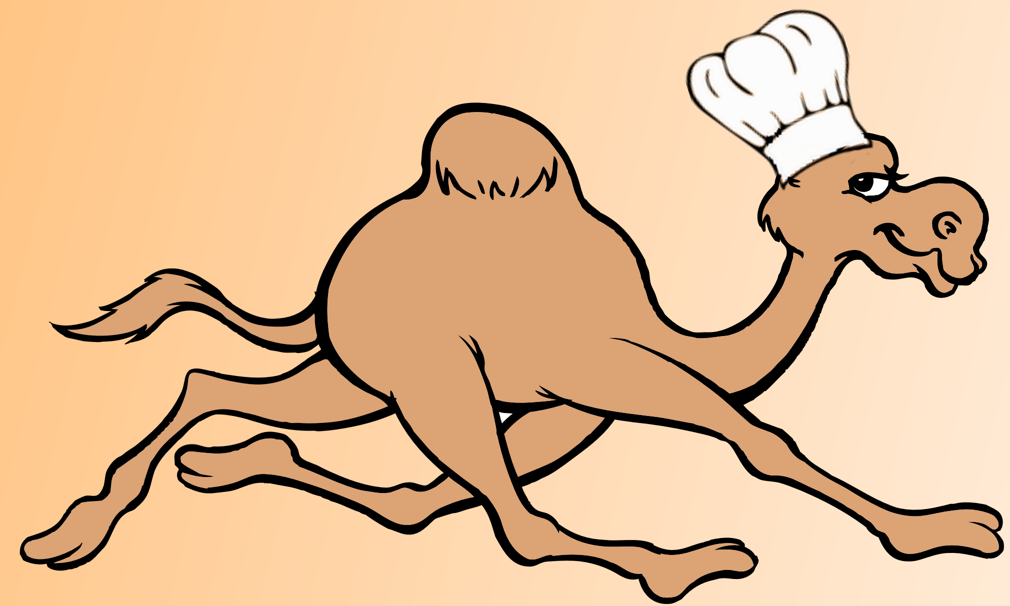
CamelOne 2013

June 10-11 2013

Boston, MA



Get Cooking with Apache Camel

Cookbook of Tips and Tricks

Scott Cranton

Who is this Scott Cranton?

Currently at Red Hat - Middleware (technical) sales

Joined FuseSource Feb, 2009 - 4.5 years with Camel

Previously at:

BEA / Oracle, Nexaweb, Object Design / Progress, MapInfo, OpenMap, TerraLogics / Strategic Mapping

First 10+ years full time coder; last 12+ selling middleware

https://github.com/FuseByExample
https://github.com/CamelCookbook

Why a Camel Cookbook?

Help beginner to intermediate users get productive by example with references for later reading

Break common Camel tasks into Recipes

Each recipe contains:

- Introduction
- How to do it (task oriented)
- How it works
- There's more

CamelOne

How Do I Control Route Startup Order?

Introduction

Sometimes order matters, and routes need to start and stop in a defined order

How to do it

Set route `startupOrder` attribute

XML DSL - <route **startupOrder="20"** routeId="myRoute">...</route>

Java DSL - `from("direct:in")`**`.startupOrder(20)`**`.routeId("myRoute")...;`

How it works

Routes started in ascending order, and stopped in descending order

http://camel.apache.org/configuring-route-startup-ordering-and-autostartup.html

There's more

You can programmatically startup and shutdown routes in response to events

```
exchange.getContext().startRoute("myRoute");
```

Camel Enterprise Integration Cookbook

Topics include:

Structuring Routes, Routing, Split/Join, Transformation, Testing, Error Handling, Monitoring and Debugging, Extending Camel, Parallel processing, Security, Transactions, ...

~110 Recipes

Co-Authors: Scott Cranton and Jakub Korab

https://github.com/CamelCookbook/

Welcome feedback, reviews, contributions, ...

Camel Cookbook Tasting Menu

- Route Design
- Routing messages (EIP)
- Transformation
- Unit Testing
- Extending Camel

CamelOne

Interconnecting Routes

Camel supports breaking routes up into re-usable sub-routes, and synchronously or asynchronously calling those sub-routes.

CamelOne

Interconnecting Routes

	Within Context	**Within JVM**
Synchronous	`Direct`	`Direct-VM`
Asynchronous	`SEDA`	`VM`

CamelOne

Interconnecting Routes

Direct and Direct-VM

CamelContext-1

```
from("activemq:queue:one").to("direct:one");
from("direct:one").to("direct-vm:two");
```

CamelContext-2

```
from("direct-vm:two").log("Direct Excitement!");
```

Run on same thread as caller

`direct-vm` within JVM, including other OSGi Bundles

CamelOne

Interconnecting Routes

SEDA and VM

CamelContext-1

```
from("activemq:queue:one").to("seda:one");
from("seda:one").to("vm:two");
```

CamelContext-2

```
from("vm:two").log("Async Excitement!");
```

Run on different thread from caller

`concurrentConsumers=1` controls thread count

`vm` within JVM, including other OSGi Bundles

Interconnecting Routes

SEDA and VM

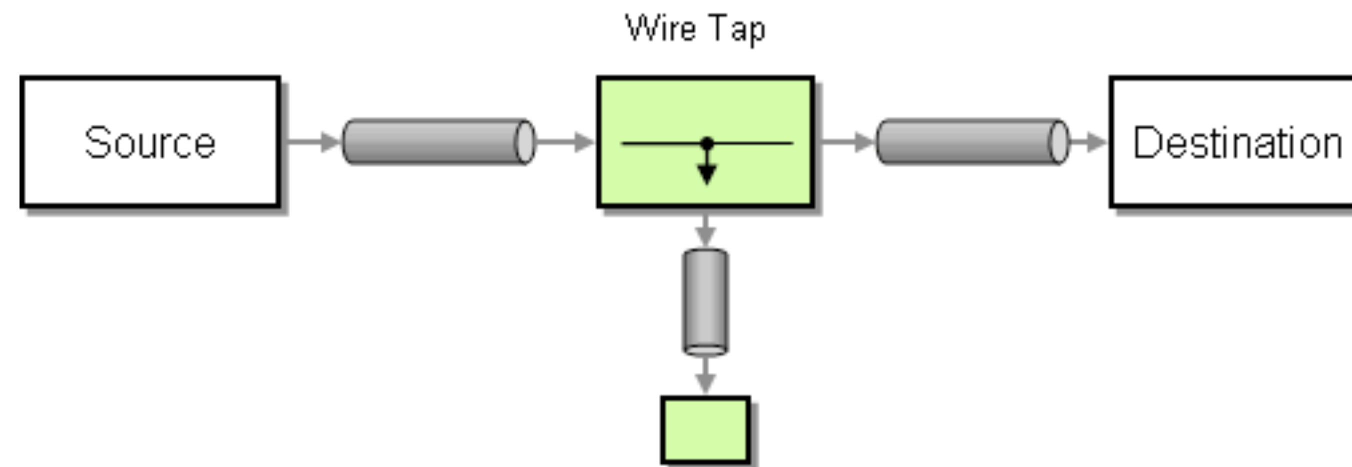
```
from("activemq:queue:one").to("seda:one");

from("seda:one?multipleConsumers=true").log("here");

from("seda:one?multipleConsumers=true").log("and there");
```

Publish / Subscribe like capability

Each route gets its own copy of the Exchange

Multicast EIP better for known set of routes

Wire Tap



To create an asynchronous path from your main route. Useful for audit and logging operations.

Wire Tap

CamelOne

```
from("direct:start")
   .wiretap("direct:tap")
   .to("direct:other");

from("direct:tap")
   .log("something interesting happened");

from("direct:other")
   .wiretap("activemq:queue:tap")
   .to("direct:other");
```

Runs on different thread / thread pool

Default Thread Pool - initial 10; grows to 20

Wire Tap

```
from("direct:start")
   .wiretap("direct:tap-mod")
   .delay(constant(1000))
   .log("Oops! Body changed unexpectedly");

from("direct:tap-mod")
   .bean(BodyModifier.class, // Modifies message body
         "changeIt");
```

Message, Header, and Properties passed by reference

CamelOne

Wire Tap

```
from("direct:start")
   .wiretap("direct:tap-mod")
     .onPrepare(new DeepCloningProcessor())
   .delay(constant(1000))
   .log("Yay! Body not changed.");

from("direct:tap-mod")
   .bean(BodyModifier.class, "changeIt");
```

`onPrepare` calls Processor **before** `wiretap` endpoint

Throttler

For when you need to limit the number of concurrent calls made a sequence of actions. For example, limit calls to a back-end ERP.

CamelOne

Throttler

```
from("direct:start")
   .throttle(constant(5))     // max per period expression
     .to("direct:throttled") // step(s) being throttled
   .end()                     // end of step(s)
   .to("direct:post-throttle");
```

Example limits to 5 messages per 1,000 ms (default)

Should use `end()` to delimit steps being throttled

CamelOne

Throttler

```
from("direct:start")
  .throttle(header("message-limit")) // expression
    .to("direct:throttled")
  .end()
  .to("direct:post-throttle");
```

Example throttles to value in header `"message-limit"`

Will use last value if Expression evaluates to `null`

CamelOne

Throttler

```
from("direct:start")
   .throttle(constant(5))
         .timePeriodMillis(2000) // value in milliseconds
      .to("direct:throttled")
   .end()
   .to("direct:post-throttle");
```

Example limits to 5 messages per 2,000 ms

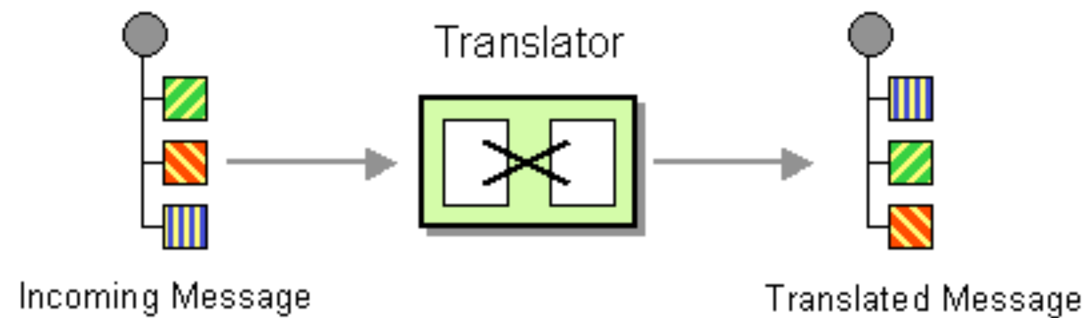
CamelOne

Throttler

```
from("direct:start")
  .throttle(constant(5))
      .asyncDelayed()
    .to("direct:throttled")
  .end()
  .to("direct:post-throttle");
```

`asyncDelayed()` causes throttled requests to be queued for future processing, releasing calling thread (non-blocking)

XSLT



Camel supports many different ways to transform data, for example using XSLT to transform XML.

XSLT

```
from("direct:start")
  .to("xslt:books.xslt");
```

Example uses books.xslt on `classpath:` (default)
Also supports `file:` and `http:`

CamelOne

XSLT

```
from("direct:start")
  .to("xslt:books.xslt?output=DOM");
```

`output` controls output data type

Supports: `string` (default), `bytes`, `DOM`, and `file`

`output=file` writes directly to disk; path must exist

File name controlled by header `"CamelXsltFileName"`

XSLT

```
from(“direct:start”)
   .setHeader(“myParamValue”, constant(“29.99”))
   .to(“xslt:books-with-param.xslt”);
```

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="myParamValue"/>

  <xsl:template match="/">
    <books>
      <xsl:attribute name="value">
        <xsl:value-of select="$myParamValue"/>
      </xsl:attribute>
      <xsl:apply-templates
           select="/bookstore/book/title[../price>$myParamValue]">
        <xsl:sort select="."/>
      </xsl:apply-templates>
    </books>
  </xsl:template>

  ...

</xsl:stylesheet>
```

CamelOne

Mock Endpoints

Camel provides some very powerful Unit Testing capabilities. Its MockEndpoint class supports complex expressions to validate your routes.

CamelOne

Mock Endpoints

```
from("direct:start")
   .filter().simple("${body} contains 'Camel'")
      .to("mock:camel")
   .end();


public class ContentBasedRouterTest
      extends CamelTestSupport {
   @Test
   public void testCamel() throws Exception {
      getMockEndpoint("mock:camel")
         .expectedMessageCount(1);

      template.sendBody("direct:start", "Camel Rocks!");

      assertMockEndpointsSatisfied();
   }

...
}
```

CamelOne

Mock Endpoints

```
MockEndpoint mockCamel = getMockEndpoint("mock:camel");
mockCamel.expectedMessageCount(2);
mockCamel.message(0).body().isEqualTo("Camel Rocks");
mockCamel.message(0).header("verified").isEqualTo(true);
mockCamel.message(0).arrives().noLaterThan(50).millis()
  .beforeNext();
mockCamel.allMessages()
  .simple("${header.verified} == true");

template.sendBody("direct:start", "Camel Rocks");
template.sendBody("direct:start", "Loving the Camel");

mockCamel.assertIsSatisfied();

Exchange exchange0 = mockCamel.assertExchangeReceived(0);
Exchange exchange1 = mockCamel.assertExchangeReceived(1);
assertEquals(exchange0.getIn().getHeader("verified"),
             exchange1.getIn().getHeader("verified"));
```

CamelOne

Mock Endpoints

Mock Responding

```
from("direct:start")
   .inOut("mock:replying")
   .to("mock:out");


getMockEndpoint("mock:replying")
   .returnReplyBody(
       SimpleBuilder.simple("Hello ${body}"));

getMockEndpoint("mock:out")
   .expectedBodiesReceived("Hello Camel");

template.sendBody("direct:start", "Camel");

assertMockEndpointsSatisfied();
```

CamelOne

Mock Endpoints

Auto Mocking

```
from("direct:start")
  .to("activemq:out");


public class ContentBasedRouterTest
    extends CamelTestSupport {
  @Override
  public String isMockEndpoints() {
    return "activemq:out";
  }

  @Test
  public void testCamel() throws Exception {
    getMockEndpoint("mock:activemq:out")
      .expectedMessageCount(1);

    template.sendBody("direct:start", "Camel Rocks!");

    assertMockEndpointsSatisfied();
  }
}
```

POJO Producing

Camel's ProducerTemplate is seen mostly in Unit Tests, and it provides a great way to interact with Camel from your existing code.

POJO Producing

```
public class ProducePojo {
  @Produce
  private ProducerTemplate template;

  public String sayHello(String name) {
    return template.requestBody("activemq:sayhello",
                                name, String.class);
  }
}
```

Send (InOnly) or Request (InOut) to Endpoint or Route

CamelOne

POJO Producing

```
public interface ProxyPojo {
    String sayHello(String name);
}

public class ProxyProduce {
    @Produce(uri = "activemq:queue:sayhello")
    ProxyPojo myProxy;

    public String doSomething(String name) {
        return myProxy.sayHello(name);
    }
}
```

Proxy template Java interface to make use clearer

Parameter Binding

CamelOne

Camel was designed to work well with calling POJOs. Parameter Binding allows you to, within the route, map the message to the method parameters. This makes it even easier to call POJOs from Camel.

Parameter Binding

```
public String myMethod(
     @Header("JMSCorrelationID") String id,
     @Body String message) {
   ...
}

public String myOtherMethod(
     @XPath("/myDate/people/@id") String id,
     @Body String message) {
   ...
}
```

CamelOne

Parameter Binding

```
public class MyBean {
  public String sayHello(String name, boolean fanboy) {
    return (fanboy) ? ("Hello iPhone")
                    : ("Hello " + name);
  }
}
```

```
from("direct:fanboy")
  .bean(MyBean.class, "sayHello(${body}, true)");

from("direct:undecided")
  .bean(MyBean.class, "sayHello(${body},
                               ${header.fanboy})");
```

Send (InOnly) or Request (InOut) to Endpoint or Route

CamelOne

Questions?

Camel Enterprise Integration Cookbook

Topics include:

Structuring Routes, Routing, Split/Join, Transformation, Testing, Error Handling, Monitoring and Debugging, Extending Camel, Parallel processing, Security, Transactions, ...

~110 Recipes

Co-Authors: Scott Cranton and Jakub Korab

https://github.com/CamelCookbook/

Welcome feedback, reviews, contributions, ...