

Erste Schritte

Agile Enterprise-Integration



Enterprise-Integrationsprojekte gehören zu den komplexesten in der IT-Welt. Damit unterschiedliche Applikationen auf eine zuverlässige und effiziente Weise zusammenarbeiten können, braucht es eine Integrations-Middleware, die nicht nur die technischen Anforderungen erfüllt, sondern auch die Agilität unterstützt und fördert. Apache Camel ist eines dieser Open-Source-Projekte, das auf diesem Gebiet nicht nur wegen der großen Anzahl an technischen Features brilliert, sondern sich auch wegen des einfachen Lösungsansatzes für die Enterprise-Integration bestens eignet.

von Alberto Mijares

Eines der Hauptthemen in den heutigen Enterprise-Informatikprojekten ist ihre inhärente Komplexität. Diese entsteht aus dem Bedürfnis, geschäftskritische Prozesse weltweit und zuverlässig umsetzen zu können. Während Skalierbarkeit und Betriebssicherheit durch programmatische Faktoren in den Applikationen modelliert werden können, ist die Modellierung oder gar die Beschreibung von effizienten Geschäftsprozessen nicht ganz so einfach. Meistens sind die Anforderungsanalyse und die effiziente Entwicklung von neuen Funktionen unproblematisch. Schwieriger ist es jedoch, bereits existierende Applikationen als einen eigenständigen Teil in ein Zielsystem zu integrieren. Mögliche Gründe für die Integration bestehender Systeme sind einerseits oft reiner Investitionsschutz oder aber fehlendes Know-how über den Nachbau eines funktionierenden Systems. Das Zusammenführen von komplexen und heterogenen Systemen zu noch komplexeren Systemen wird allgemein als Integration bezeichnet. Damit mehrere Applikationen zusammenarbeiten können, müssen sie die „gleiche Sprache“ sprechen oder aber über eine Middleware kommunizieren, die „sämtliche Sprachen“ aller Applikationen kennt. Das Erste ist wirklich selten der Fall und erfordert Applikationen, die für eine gewisse zukünftige Integration schon so ausgelegt worden sind. Das Finden einer geeigneten Middleware ist eine zeitaufwändige Aufgabe, die üblicherweise einen detaillierten Evaluationsprozess von unterschiedlichen Produkten und Lösungsvorschlägen impliziert. Eine der Schwierigkeiten ist dabei, die richtigen Bewertungskriterien (meist technischer Natur) auszuwählen. Oft zeigt es sich, dass einige Faktoren in der Evaluationsphase als ausschlaggebend angesehen werden, die in der folgenden Implementationsphase nur

mit der Effizienz der Applikation zur Middleware zu tun haben und nicht mit den eigentlichen technischen Features. Zu diesen Faktoren zählen Agilität, Leichtigkeit, adäquate Lernkurve und gute Wiederverwendbarkeit. Um zu veranschaulichen, wie diese abstrakten Konzepte wirklich in eine Middleware-Integration passen, wird gezeigt, was Apache Camel alles zu bieten hat.

Erste Schritte

Wegen des deklarativen Charakters von nachrichtengesteuerten Systemen besteht ein großer Teil einer Applikation mit Apache Camel im Konfigurieren von Nachrichtenflüssen (Message Flows). Camel bietet unterschiedliche Möglichkeiten an, dieses Problem zu lösen. Eine favorisierte und empfohlene Art ist die Anwendung von Camel Java DSL. Das erste Element, das instanziiert und konfiguriert werden muss, ist ein so genannter Camel Context. Sobald dieser gestartet ist, ist das System bereit, Meldungen entsprechend einem konfigurierten Nachrichtenfluss zu versenden oder zu empfangen. Listing 1 zeigt, wie eine minimale Apache-Camel-Applikation aussieht. Dieses erste Beispiel erklärt, wie die Abhängigkeiten der Camel-Applikation unter Verwendung von Maven (Listing 2) konfiguriert werden und wie der Camel Context in Java instanziiert wird.

Um dieses Beispiel laufen lassen zu können, muss sichergestellt sein, dass Apache Maven im Ausführungspfad spezifiziert ist. In einer Konsole, die auf dem Verzeichnis mit den abgespeicherten Dateien geöffnet wurde, wird dann der folgende Befehl eingegeben: `mvn compile exec:java -Dexec.mainClass="com.example.camel.Example1"`. Wie man sieht, führt diese Applikation überhaupt nichts aus, außer dass sie einen leeren Camel Context startet, einige Sekunden wartet und dann den Context stoppt. Wie sieht nun aber ein typisches Beispiel

in Camel aus? Das folgende Schulbeispiel illustriert, wie ein Nachrichtenfluss (in Camel eine Route genannt) spezifiziert und zu diesem Nachrichtenfluss eine Meldung gesendet wird. Um Routen in einem Context zu erstellen, wird ein Route Builder (Listing 3) verwendet. Dazu muss die Applikation leicht geändert (Listing 4) und ein neues Modul von Camel (Listing 5) hinzugefügt werden. In diesem Beispiel wurde ein erstes Camel Route eingeführt, das von einem einzigen Endpunkt (*direct:say_hello*) ausging. Es zeigt, wie der Inhalt der Nachricht mit Java DSL umgewandelt und an den Endpunkt gesendet wird und wie die Nachricht durch die Verwendung eines Producer Templates an einen Context weitergeleitet wird. Dieses Beispiel berücksichtigt folgende wichtige Konzepte:

- Routen können innerhalb von Route Builders gepackt werden und es können sich beliebig viele auf einen Context beziehen, die die verlangte Funktion auf eine modulare Weise einrichten.
- Dazu kann eine Applikation mehrere Contexts haben, die parallel mit einem passenden Isolationslevel arbeiten.
- Die Funktion eines Endpunkts wird durch ein URI (*stream:out*) definiert, das aus einem Präfix (*stream*) und einem Suffix (*out*) besteht. Der Präfix identifiziert eine von mehreren Camel-Komponenten [2] und somit die Funktionalität. Das Suffix konfiguriert die Funktion mittels einer komponentenspezifischen Parametersyntax.
- Einige Endpunkte funktionieren in einer passiven Weise (diese warten auf eingehende Meldungen), während andere aktiv unterschiedliche Arten von Ressourcen überwachen (z. B. durch festgelegte, zyklische Abfragen).

Eine realistischere Anwendung

Um die Fülle an Features in Camel besser einzuschätzen, und um zu zeigen, wie elegant und präzise die Camel Java DSL sein kann, wird im nächsten Beispiel ein Legacy-System, das die Bestellungen für Kunden generiert, mit einem System für die Preisbestimmung ermittelt. Im System für die Preisbestimmung werden die Rabatte mit komplexen Geschäftsregeln berechnet. Das Legacy-System, auf dem die Bestellungen generiert werden, ist ein wirklich altes und schlecht wartbares System, das als Integrationsmöglichkeit nur die Generierung von CSV-(Comma-Separated-Values-)Dateien anbietet. Andererseits ist das System für die Preisbestimmung eine moderne Webapplikation, die einen einfachen, aber effektiven REST-Service für die Berechnung der Preise der Bestellungen offeriert. Da die verwendeten Regeln für die Berechnung des Preises wirklich komplex und somit auf dem System sehr rechenintensiv sind, muss eine Belastung des Systems mit mehr als einer Preisberechnung pro zwei Sekunden vermieden werden. Das Einlesen einer großen CVS-Datei und das Senden der Preisberechnungen auf einmal würden zu einem Zusammenbruch des Systems führen. Das System soll deshalb die Bestellungen prinzipiell von der CSV-Datei *orders.csv*

Listing 1

```
package com.example.camel;
import org.apache.camel.impl.DefaultCamelContext;

public class Example1 {
    public static void main(String[] args) throws Exception {
        DefaultCamelContext camelContext = new DefaultCamelContext();
        camelContext.start();
        Thread.sleep(10000); // Wait some time before stopping the context
        camelContext.stop();
    }
}
```

Listing 2

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>Camel</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-core</artifactId>
      <version>${camel.version}</version>
    </dependency>
  </dependencies>
  <properties>
    <camel.version>2.8.0</camel.version>
  </properties>
</project>
```

Grundlagen

Die Grundzüge von Apache Camel können durch folgende Konzepte aufgeführt werden: gute Unterstützung der Enterprise Integration Patterns, Java-basierend, ausgezeichnete Spring-Integration und für Prototyping und Modularisierung konzipiert. Diese Eigenschaften plus der Tatsache, dass es auf Maven basiert, machen dieses Projekt sehr populär im Umfeld der Enterprise-Integration. Wie schon erwähnt, bietet Camel eine Java-Implementierung der Enterprise Integration Patterns an. Diese Muster sind im Buch von Gregor Hohpe und Bobby Woolf [1] zusammengestellt, gut beschrieben und im Grunde genommen Rezepte für einen Nachrichtenaustausch auf einem abstrakten Level. Sie zeigen, wie die Probleme des Datenaustauschs auf Applikationsebene effizient und zuverlässig gelöst werden können. Wie im Buch erklärt, ist ein Datenaustausch, der in unterschiedlichen Ausprägungen (in, out, in/out, asynchron und synchron) konfiguriert wird und die Kommunikationsanforderungen von Enterprise-Applikationen auf eine generische Weise implementiert, ein mächtiger Mechanismus.

Listing 3

```
package com.example.camel;
import org.apache.camel.builder.RouteBuilder;

public class Example2Routes extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:say_hello")
            .setBody(constant("Hello ").append(body()))
            .to("stream:out");
    }
}
```

Listing 4

```
public static void main(String[] args) throws Exception {
    DefaultCamelContext camelContext = new DefaultCamelContext();
    camelContext.addRoutes(new Example2Routes());
    camelContext.start();

    ProducerTemplate template = camelContext.createProducerTemplate();
    template.sendBody("direct:say_hello", "Alberto");

    Thread.sleep(1000);
    camelContext.stop();
}
```

Listing 5

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stream</artifactId>
  <version>${camel.version}</version>
</dependency>
```

Listing 6

```
customer_1, 10, 100
customer_2, 20, 200
customer_2, 30, 300
customer_3, 40, 400
customer_1, 50, 500
```

Listing 7

```
public static void main(String[] args) throws Exception {
    DefaultCamelContext camelContext = new DefaultCamelContext();
    camelContext.addRoutes(new Example3Routes());
    camelContext.start();
    Thread.sleep(10000);
    camelContext.stop();
}
```

(Listing 6) lesen, für jede Bestellung den Preisberechnungsservice aufrufen, das berechnete Total in eine entsprechende Reihenfolge setzen und in einem späteren Schritt die Ergebnisse in eine andere CSV-Datei für die Auslieferung speichern. Weil Camel Ressourcen (z. B. Dateien) zyklisch abfragen kann, ist es in diesem Fall gar nicht nötig, einen Datenproduzenten (Producer) für das Senden der Meldungen an einen Context (Listing 7) zu verwenden. Die ganze Konfiguration sollte nun wie in Listing 8 aussehen.

Während die Konfiguration der Routen für das Verständnis nicht wirklich ausführlich beschrieben werden muss, verlangen einige wirklich neue Punkte an dieser Stelle ergänzende Erklärungen. Um den Inhalt der CSV-Datei, die die Bestellungen enthält, zu lesen, bietet Camel einen so genannten Endpunkttyp. Dieser agiert als Verbraucher (Consumer, File Reader) oder als Produzent (File Writer). Sobald der Inhalt der CSV-Datei eingelesen worden ist, wird jede Zeile in eine Instanz der *ProduktOrder*-Klasse (Listing 9) konvertiert [3]. Das ist ein komfortables POJO, annotiert mit Apache-Bindy-Annotationen [4], das beim Unmarshalling-Prozess zum Erzeugen des entsprechenden Beans verwendet wird. Sobald die Zeilen der Bestellungsdatei eingelesen und in POJOs entpackt (unmarshalled) sind, kann das Splitter Enterprise Integration Pattern [5] angewendet werden. Mittels einer einfachen Java-Klasse (Listing 10) wird die Nachricht, die alle Produktbestellungen enthält, in individuelle Meldungen aufgesplittet, sodass eine Nachricht

Listing 8

```
public class Example3Routes extends RouteBuilder {
    private static final String REQUEST_TEMPLATE = "<?xml
        version='1.0'?>" +
        "<priceRequest customer='\"${body.customer}\"' +
        \" quantity='\"${body.quantity}\"' price='\"${body.price}\"'/>";
    private static final String BINDY_PACKAGE = "com.example.camel";

    @Override
    public void configure() throws Exception {
        from("file:src/resources/?fileName=orders.csv&noop=true")
            .unmarshal().bindy(BindyType.Csv, BINDY_PACKAGE)
            .split().method(ProductOrderSplitter.class)
            .log("Read: '${body}'")
            .enrich("direct:calculate_total",
                new PriceAggregationStrategy())
            .log("Written: '${body}'")
            .marshal().bindy(BindyType.Csv, BINDY_PACKAGE)
            .to("file:src/resources/?fileName=totals.
                csv&fileExist=Append");

        from("direct:calculate_total")
            .setBody().simple(PRICE_REQUEST_TEMPLATE)
            .throttle(1).timePeriodMillis(2000)
            .to("restlet:http://localhost:9999/priceCalculator?
                restletMethod=post")
            .transform().xpath("/priceResponse/price/text()",
                Integer.class);
    }
}
```

nur eine Bestellung enthält. Diesen Bestellungen fehlen noch die Informationen für den Totalpreis, der nicht in der originalen CSV-Datei steht und vom Preisbestimmungssystem neu berechnet werden muss. Für die Bewältigung dieser Aufgabe wird das Enricher Enterprise Integration Pattern [6] konfiguriert, das die Meldungen für einen anderen Endpunkt weiterleitet und den Totalpreis in die Bestellung setzt. Das geschieht am Schluss mittels der Aggregation-Strategy-Implementation (Listing 11). Sobald die Bestellinformation vollständig ist, werden die Bestellungen zurück in eine CSV-Datei umgewandelt (marshalling) und mittels eines Dateiendpunkts werden sie in eine Totaldatei geschrieben.

Um den REST-Service des Preisbestimmungssystems aufzurufen (was innerhalb des *direct:calculate_total*-Endpunkts geschieht), muss zuerst ein XML-Fragment vom Typ *PriceRequest* generiert werden. Dafür könnte wieder der JAXB Marshalling Support in Camel (wie schon beim Apache Bindy Support gesehen) verwendet werden. Aber für diesen Fall ist es besser, einen rudimentäreren Mechanismus, wie das Simple-Template mit der XML-Struktur zu verwenden. Das wird durch den Camel Feature Expression Languages Support [7] illustriert. Vor dem Aufruf und um zu vermeiden, dass der Preisbestimmungsservice mit einer großen Anzahl an Anfragen geflutet wird, wird ein weiteres der mächtigen Enterprise Integration Patterns von Camel, das Throttler Pattern, konfiguriert [8]. Mit diesem einfachen Konstrukt kann die Anzahl der Meldungen pro Sekunde zwischen zwei Endpunkten kontrolliert werden. Zuletzt wird in diesem Beispiel der REST-Preisbestimmungsservice aufgerufen. Dazu wird der *Restlet*-Endpunkt verwendet, der den Preis, der als Zahlenwert in der XML-Response-Nachricht steht, mittels einer XPath Expression entpackt. Da in diesem Beispiel neue Typen von Endpunkten (*Restlet*, *Bindy*) verwendet werden, müssen die POM-Abhängigkeiten dementsprechend erweitert werden (Listing 12).

Camel Features

Nach diesem nicht unbedingt trivialen Beispiel soll noch einmal auf die am Anfang des Artikels vorgestellten Argumente eingegangen und aufgezeigt werden, wie sie auf das gezeigte Beispiel zugeordnet sind:

- **Agilität:** Die meisten der bevorzugten Ansätze in Softwareentwicklungsprozessen basieren heutzutage auf dem Konzept „Agile Development“. Grundsätzlich fördert diese Idee Dinge wie die pragmatische Anforderungsanalyse, ständige Verbesserung durch Benutzerfeedback, automatisches Testen und dynamische Priorisierung von Featureimplementierungen. Im Hinblick darauf ermöglicht Camel, mit einer kleinen Applikation zu beginnen und sie stetig bis auf Produktionsstufe auszubauen. Ein weiteres Designprinzip von Camel ist die Testbarkeit. Camel bietet alle notwendigen Werkzeuge an, um den Testprozess zu automatisieren, und stellt viele Unit Tests und Mock-Objekte zur Verfügung.

Listing 9

```
@CsvRecord(separator = ", ")
public class ProductOrder {
    @DataField(pos = 1, trim = true)
    private String customer;

    @DataField(pos = 2, trim = true, pattern = "0")
    private int quantity;

    @DataField(pos = 3, trim = true, pattern = "0")
    private int price;

    @DataField(pos = 4, trim = true, pattern = "0")
    private int total;

    // Getters and setters omitted
}
```

Listing 10

```
public class ProductOrderSplitter {
    public List<ProductOrder> split(List<Map<Class, ProductOrder>> body) {
        List<ProductOrder> result = new ArrayList<ProductOrder>();
        for (Map<Class, ProductOrder> map : body) {
            for (ProductOrder productOrder : map.values()) {
                result.add(productOrder);
            }
        }
        return result;
    }
}
```

Listing 11

```
public class PriceAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange exchange, Exchange exchange1) {
        ProductOrder order = exchange.getIn().getBody(ProductOrder.class);
        Integer total = exchange1.getIn().getBody(Integer.class);
        order.setTotal(total);
        return exchange;
    }
}
```

Listing 12

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-bindy</artifactId>
  <version>${camel.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-restlet</artifactId>
  <version>${camel.version}</version>
</dependency>
```

- **Leichtigkeit:** Unternehmenssoftware hat den Ruf, groß dimensionierte Runtime-Plattformen zu benötigen, die auch die Ressourcen großzügig dimensionierter Entwicklerplattformen stark beanspruchen. Das führt schnell zu Frustrationen unter den Entwicklerteams, da dieser Ressourcen hunger die Produktivität der Teams drastisch reduzieren kann. Camel ist auch bezüglich der Softwareverteilung eine vielseitige Middleware. Die vorgestellten Beispiele zeigen die einfachste Option (Java-Standalone-Applikation). Aber eine Camel-Applikation kann auch in einem Framework, z. B. Spring, in einem Application-Container oder sogar als OSGi-Modul eingesetzt werden. Die Änderungen zum Starten der Applikation in unterschiedlichen Umgebungen sind minimal, sei das in einer Test- oder aber in einer Produktionsumgebung.
- **Adäquate Lernkurve:** Unter Einbeziehung der Tatsache, dass Nachrichtensysteme sehr komplex sind, und obwohl die Grundlagen meist gut dokumentiert, aber nicht allen Entwicklern gut vertraut sind, sieht man, dass die Steigung der Lernkurve, bei Themen, z. B. asynchrone Aufrufe oder Patterns zum Austausch von Nachrichten, stark abnimmt und somit die Produktivität der Entwicklerteams senkt. Eine adäquate Ausbildung und Referenzdokumentation sind zwar ein Muss, aber sehr oft ziehen es Entwickler vor, anhand eines Beispiels ein Feature umzusetzen oder mit einem High Level API die Möglichkeiten einer Software auszuloten, um sie dann für einen Anwendungsfall der Applikation zu adaptieren. Camel bietet diese technischen Ressourcen an, und mittels vieler verfügbarer DSLs wird hinsichtlich des Enterprise Integration Patterns und anderer domänenspezifischer Abstraktionen ein High Level API zur Verfügung gestellt.
- **Wiederverwendbarkeit:** Diese kommt auf unterschiedliche Art und Weise in Apache Camel vor. Nachfolgend sind die Relevantesten davon aufgeführt:
 - **Wiederverwendung von Code:** Da Apache Camel ein Open-Source-Projekt ist, wird nicht nur der produktive Quellcode zur Verfügung gestellt, sondern auch umfangreiche und einheitliche Tests, die die Funktion zusätzlich dokumentieren. Der Quellcode dient auch als Fundgrube von Codebeispielen für die Entwickler der Applikationen. Wie Nachrichtensysteme funktionieren, ist nicht trivial, und so unterstützen diese vielfältigen Codebeispiele den Lernprozess auf einfache Weise. Camel erlaubt die Integration zahlreicher Open-Source-Projekte mit Unterstützung von Datenformaten, Sprachensupport oder mittels spezieller Komponenten.
 - **Wiederverwendung von Modulen:** Eine direkte Konsequenz des modularen Designs. Es erlaubt, die Camel-Funktionen in Module zu packen und in anderen, auf Camel basierenden Projekten wiederzuverwenden. Camel ist modular. Das erlaubt Entwicklern, Packages selektiv für die Applikation auszuwählen (durch die Dependencies-Definitionen in der POM-Datei).

- **Wiederverwendung von Fachkenntnissen:** Camel ist ein Java-Projekt, das von Beginn an sehr gut mit dem Spring Framework integriert war. Java und Spring gehören zu den Schlüsseltechnologien innerhalb der Enterprise-IT-Welt und helfen enorm, ein dazu passendes Entwicklerteam zu finden.
- **Wiederverwendung von Know-how:** Ein Hauptmerkmal von Apache Camel ist die Implementierung der Enterprise Integrations Patterns, die im gleichnamigen Buch von Gregor Hohpe und Bobby Woolf beschrieben werden. Durch die Anwendung dieser Architektur-Patterns profitiert ein Entwickler automatisch vom Know-how und der Erfahrung dieser Softwareingenieure. Dadurch wird ein kostspieliger Prozess vermieden, eine eigene Lösung durch Ausprobieren zu finden. Weiter wird die breite Anwendung dieses wertvollen Know-hows zusätzlich gefördert.

Fazit

Camel ist ein ausgewachsenes Projekt, mit dem einerseits leicht zu beginnen ist, andererseits auch sämtliche erforderlichen Features einer typischen Enterprise-Integration-Applikation, das heißt von der Transaktionsabwicklung bis zur Fehlerbehandlung, anbietet. Aber Camel ist entschieden kein Enterprise Service Bus (ESB). Das ist aus der Sicht vieler ein Vorteil, da dadurch die Projektgröße nicht zu umfangreich ist und sich Camel auf die Funktionen konzentriert, für die es entworfen worden ist. Um diesen Mangel an Funktionen auszugleichen, bietet Camel eine ausgezeichnete Integration zu Apache ServiceMix und anderen Produkten mit ESB-Möglichkeiten an.

Ich hoffe, dass der Artikel Ihr Interesse geweckt und Sie davon überzeugt hat, das nächste Mal bei der Integration von Services Camel ausprobieren oder wie Entwickler sagen: „to give Camel a ride“.



Alberto Mijares ist Softwareingenieur mit mehr als zehn Jahren Erfahrung in der IT-Welt. Er ist Scrum Master und Agile-Anwender und bietet großes Know-how zu Webtechnologien und Java Enterprise Applications. Auch nahm er in der Vergangenheit an W3C-Projekten in Verbindung mit der Semantic-Web-Initiative teil.

Außerdem gibt Alberto Mijares regelmäßig Tipps und Tricks an Entwicklerteams weiter und ist Spezialist für Technologien wie das Google Web Toolkit, Apache Camel und Apache Solr. Seit 2008 arbeitet er für die Canoo Engineering AG in der Schweiz.

Links & Literatur

- [1] Hohpe, Gregor; Woolf, Bobby: „Enterprise Integration Patterns“, Addison-Wesley, 2003
- [2] <http://camel.apache.org/components.html>
- [3] <http://camel.apache.org/data-format.html>
- [4] <http://camel.apache.org/bindy.html>
- [5] <http://camel.apache.org/splitter.html>
- [6] <http://camel.apache.org/content-enricher.html>
- [7] <http://camel.apache.org/languages.html>
- [8] <http://camel.apache.org/throttler.html>