Integration ganz einfach mit Apache Camel

Christian Schneider

Talend

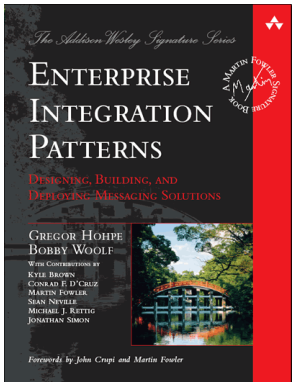
Integration ganz einfach mit Apache Camel

- Überblick
- Erste Integration (erstellen, testen, deployen)
- Camel in OSGi
- Live Beispiele
- Die Architektur von Apache Camel

Das Apache Camel Projekt

Leistungsfähiges Integrationsframework, basiert auf den Enterprise Integration Patterns

- Open Source Apache License v2
- Homepage des Projekts: http://camel.apache.org
- Aktive und wachsende Community



Highlights

- Projekt wurde 2007 gestartet
- Ausgereift, für den produktiven Einsatz geeignet
- über 100 Komponenten
- Umfangreiche Dokumentation
- Mehrere Hersteller unterstützen das Projekt

Integration ist nicht einfach

- Das Thema Integration ist riesig
- Es gibt keinen goldenen Hammer, keine “one size fits all” Lösung

Zu verbindende Applikationen sind inkompatibel in:

- Protokoll
- Datenformat
- Semantik

Camel macht Integration einfacher

- Ausrichtung auf Standards (mehr als 100 Technologien, Protokolle und Dataformate werden unterstützt)
- Direkte Nutzung der Enterprise Integration Patterns
- Konvention vor Konfiguration
- Nachrichteninhalt ist egal
- Wahl des Containers ist egal
- Leichtgewichtig, modular und einfach erweiterbar
- Kein schwergewichtiger ESB => nur Bibliothek

Quelle: Universal Business Adapter (IBM)

Wo lade ich Camel runter?

Die Distribution von Apache

- Die Apache Distribution enthält alle Camel JARs

  http://camel.apache.org/download.html

- … oder Maven benutzen

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-core</artifactId>
  <version>2.8.1</version>
</dependency>
```

Distribution z.B. TESB SE von Talend

- OSGi Container (Apache Karaf und Eclipse Equinox)
- Ein Installationsarchiv (Auspacken, loslegen)
- Für den produktiven Einsatz geeignet, Support von Talend

http://www.talend.com/products-application-integration/talend-integration-factory-community-edition.php

Hello World

Wie implementiere ich meine erste Camel Anwendung?

- Camel nutzt Apache Maven
- Wir nehmen einen der vorhandenen Maven Archetypes

http://camel.apache.org/camel-maven-archetypes.html

- …generieren das Projekt

```
mvn archetype:generate \
 -DarchetypeGroupId=org.apache.camel.archetypes \
 -DarchetypeArtifactId=camel-archetype-java \
 -DarchetypeVersion=2.9.1 \
 -DgroupId=org.talend.example \
 -DartifactId=example-cbr \
 -Dversion=1.0-SNAPSHOT
```

- …und lassen es laufen

```
mvn camel:run
```

Genug der Theorie...

Deployen einer Camel Anwendung

Camel ist leichtgewichtig und braucht keinen bestimmten Container, mögliche Deploymentoptionen sind:

- Als standalone Java Applikation
- Als Bestandteil (eingebettet in) einer anderen Java Applikation
- In einem Servlet Container (Tomcat, Jetty)
- In einem OSGi Container (z.B. Karaf)

- In die Cloud (Amazon EC2, Google AppEngine)

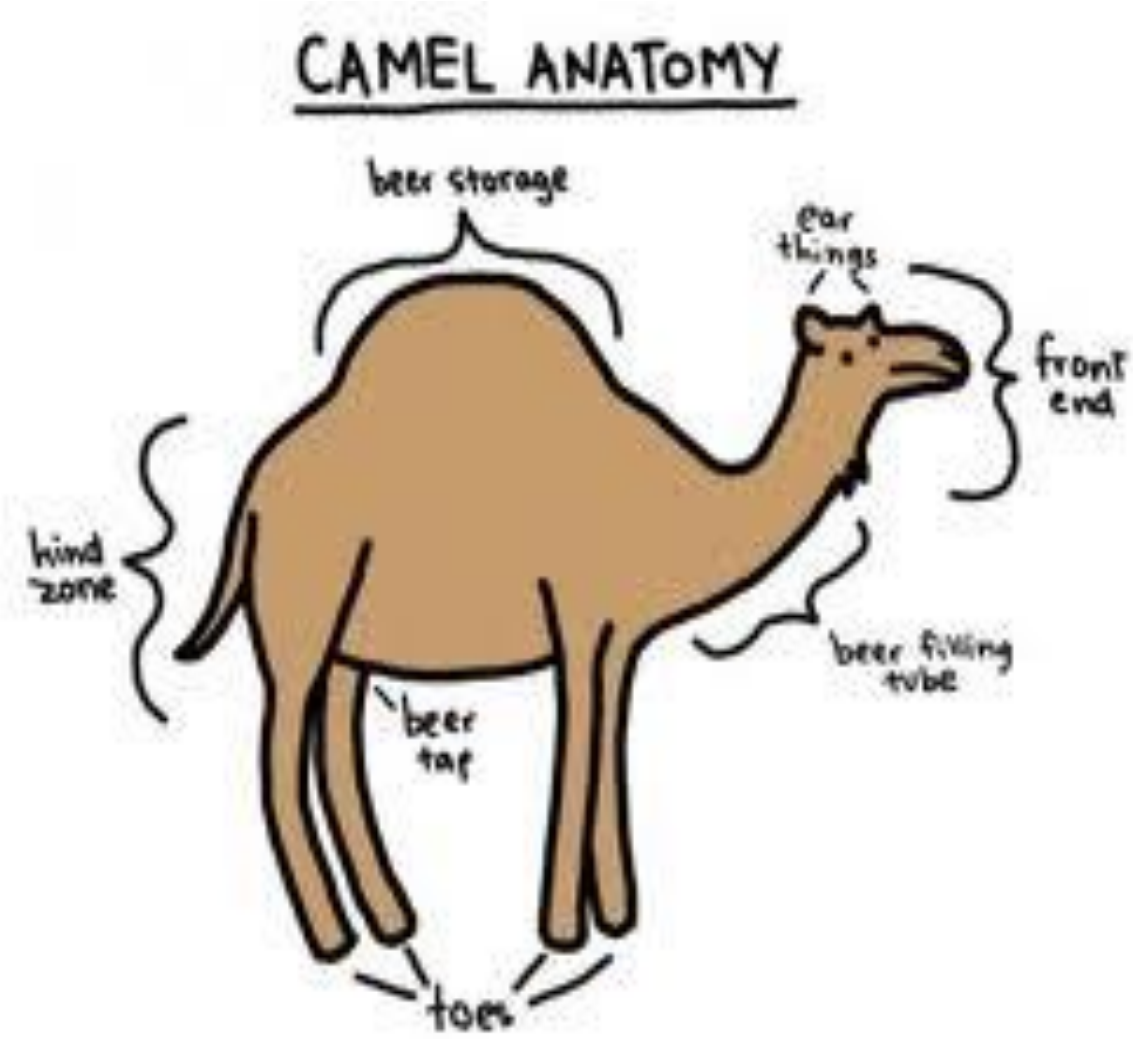
http://camel.apache.org/tutorial-for-camel-on-google-app-engine.html

Noch mehr Code:

Camel in OSGi…

Die Architektur von Camel

- Camels API
- EIPs und die DSLs
- Laufzeitumgebung
- Komponenten



Die API von Camel

Einheitlicher Umgang mit

- Protokollen (Standard oder nicht)
- Datenformaten
- Sprachen für Ausdrücke
- Fehlerbehandlung

Kernkonzepte

- Message und Exchange
- Processor
- Endpoint
- Producer und Consumer
- Language, Expression und Predicate
- TypeConverter
- Route
- CamelContext, Service und Registry

Message und Exchange

Message – repräsentiert die Daten, die in einer Route von einem Processor zum anderen weitergegeben werden

- *Body*
- *Headers* (Metadaten)
- *Attachments*

```
public interface Message {
    String getMessageId();
    Exchange getExchange();
    boolean isFault();

    Map<String, Object> getHeaders();
    Object getBody();
    <T> T getBody(Class<T> type);
    Map<String, DataHandler> getAttachments();
}
```

Exchange – enthält die Messages eines Durchlaufs einer Route

- *in* Message
- *out* Message
- *exception* (falls ausgelöst)
- *properties*
- *id*
- *MEP* (in-out, in-only)

```
public interface Exchange {
    Map<String, Object> getProperties();
    boolean hasProperties();

    Message getIn();
    Message getOut();
    boolean hasOut();
    Exception getException();

    boolean isFailed();
    boolean isTransacted();
    boolean isRollbackOnly();

    CamelContext getContext();
    Endpoint getFromEndpoint();
    String getFromRouteId();
    String getExchangeId();
    ExchangePattern getPattern();
}
```

Processor

- Repräsentiert eine Node, die einen Exchange verarbeiten kann
- Eine Route ist ein Graph aus aneinandergehängten Processors
- Camel enthält eine Vielzahl vordefinierter Processors
- Der Nutzer kann eigene Processors implementieren

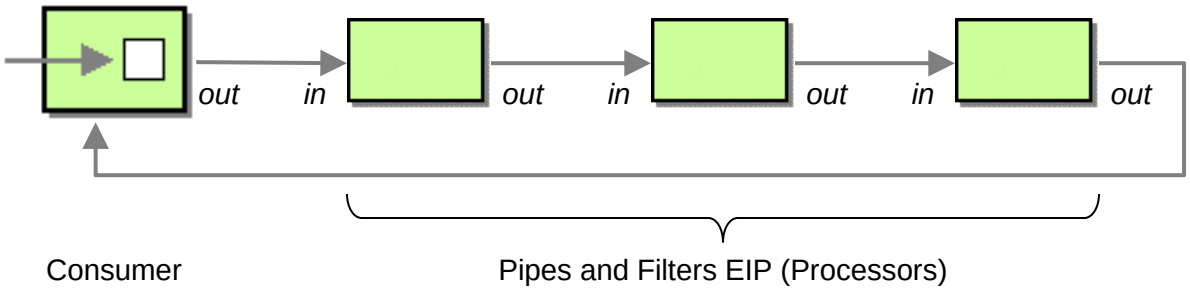
```
public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

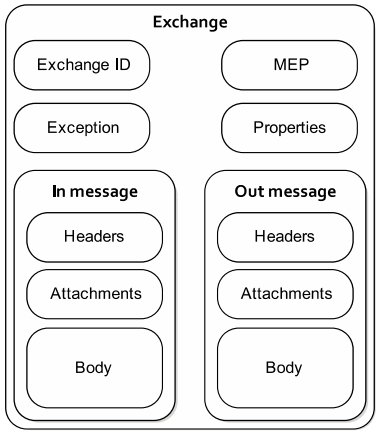
```
from("direct:test").process(new MyProcessor())
```

Konventionen beim Routing

Konvention vor Konfiguration!

- Die out Message des vorherigen Processor ist die in Message des jeweils nächsten
- Gibt der vorherige Processor keine out Message zurück, wird die vorige in Message zur out Message
- Am Ende der Verarbeitung wird die letzte out (oder in) Message zur Rückgabenachricht und vom Consumer zurückgeschickt (in-out MEP)





Exchange Diagramm aus *Camel in Action*, © Manning 2011

Endpoint

- Beschreibt ein Ende eines Kommunikationskanals
- Wird eindeutig durch eine URI identifiziert
- Die URI eines Endpoint dient auch zur Konfiguration
- Endpoints können beide Seiten eines Kommunikationskanals darstellen (Consumer und Producer)

```
public interface Endpoint extends IsSingleton, Service {

    String getEndpointUri();
    String getEndpointKey();
    CamelContext getCamelContext();
    void configureProperties(Map<String, Object> options);

    Exchange createExchange();
    Producer createProducer() throws Exception;
    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

URIs in Endpoints

URIs werden genutzt, um Endpunkte zu konfigurieren

- *Der Präfix (hier `file`) bestimmt die Komponente*
- Camel erkennt und instanziiert Komponenten in JARs automatisch anhand des Präfix
- Komponenten wie z.B. JMS müssen explizit instanziiert werden, der Präfix des Endpunkts ist dann der Beanname (etwa `amq` und `wmq`)
- Der Kontextteil der URI (hier `data/messages`) wird benutzt um den Endpunkt eindeutig zu identifizieren
- Die Parameter der URL (hier `recursive=true`) dienen zur Konfiguration der Komponente

```
from("file://data/messages?recursive=true")…
```

Consumer

- Erstes Glied in der Verarbeitungskette
- Empfängt eine protokollspezifische Nachricht (Server), steckt sie in einen Exchange und beginnt die Verarbeitung in der Route
- Optional wird das Ergebnis zurückgesendet
- Event-Driven Consumer (asynchroner Empfänger) wird durch Events von außen angestoßen
- Polling Consumer (synchroner Empfänger) fragt eine Quelle regelmäßig ab

```
public interface Consumer
    extends Service {

    Endpoint getEndpoint();
}
```

```
public interface PollingConsumer
    extends Consumer {

    Exchange receive();
    Exchange receiveNoWait();
    Exchange receive(long timeout);
}
```

Producer

- Bindeglied zu den zu integrierenden Systemen (z.B. ftp)
- Erzeugt aus Messages protokollspezifische Nachrichten und sendet diese (Client)
- Empfängt optional eine Antwort

```
public interface Producer extends
    Processor, Service, IsSingleton {

    Endpoint getEndpoint();
    Exchange createExchange();
}
```

Language, Expression und Predicate

Language

- Austauschbarer Ausführungs-mechanismus für Ausdrücke
- Erlaubt Entwicklern, Ausdrücke in der von ihnen bevorzugten Sprache zu formulieren

```
public interface Language {
    Predicate createPredicate(String expr);
    Expression createExpression(String expr);
}

public interface Expression {
    <T> T evaluate(Exchange exchange,
Class<T> type);
}

public interface Predicate {
    boolean matches(Exchange exchange);
}
```

Expression

- Ausdruck, anhand dessen ein Exchange ausgewertet wird, liefert einen beliebigen Rückgabewert

Predicate

- Spezielle Ausprägung einer Expression, die einen booleschen Wert zurückliefert

```
…body(simple("${in.header.myHeader}"))…

…filter(xpath("/order/@type"))…

…choice()
   .when(header("myHeader").contains("my
content"))…
```

TypeConverter

- Für die implizite (automatische) Typkonvertierung
- Ein TypeConverter stellt die Konvertierung von einer Klasse zu einer anderen bereit Class → Class

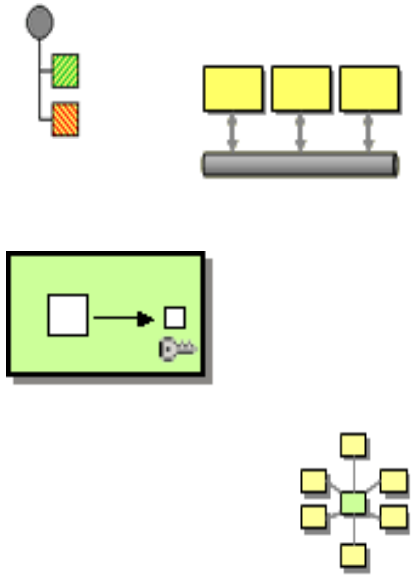
```
from("file://data/messages").to("log:myCategory")
```

Enterprise Integration Patterns in Camel

Integration Patterns decken einen großen Teil der Integrationsaufgaben ab

Kategorien von Integration Patterns

- Messaging Endpoints
- Message Construction
- Messaging Channels
- Message Routing
- Message Transformation
- System Management



Funktionale Einordnung

- Beschreibt ein Artefakt (was es ist, z.B. Message, Message Endpoint)
- Beschreibt eine Aktivität (was es tut, z.B. Content Based Router)

Laufzeitumgebung von Camel

Ausführungsumgebung vorgegeben durch den CamelContext

- Routen
- Registries (für Endpunkte, Beans, Services, Komponenten)
- Routing Engine (synchron und asynchron)
- Monitoring (JMX)
- Events und Notifications

CamelContext wird konfiguriert

- Programmatisch (Java API), Spring oder Blueprint
- Automatische Erkennung von Komponenten in JARs und OSGi Bundles
- RouteBuilders zur Konfiguration von Routen

Komponenten von Camel

Komponenten kapseln jeweils eine Technologie

- Transportprotokolle (file, http, ftp, jms, pop/smtp, rss, …)
- Datenformate (JAXB, JSON, CSV, EDI, HL7, …)
- Expression Languages (Simple, XPath, JavaScript, Ruby, Scala)
- Andere (SQL, Quartz, LDAP, JCR, …)

Die wichtigsten Komponenten

- bean (camel-core)
- file (camel-core)
- camel-spring
- camel-jms
- camel-cxf
- camel-http
- camel-jetty
- …u.v.m.

Fehlerbehandlung

Manchmal entstehen während des Ablaufs einer Route Exceptions (oder Faults).

- *Faults sind dauerhafte Fehler (analog wie bei WSDL), die Wiederholung des Vorgangs würde den gleichen Fehler erzeugen (z.B. der Fehler HTTP 404)*
- *Exceptions sind vorübergehende Fehler. Eine Wiederholung kann erfolgreich sein (z.B. kurzzeitig ausgefallene Services)*

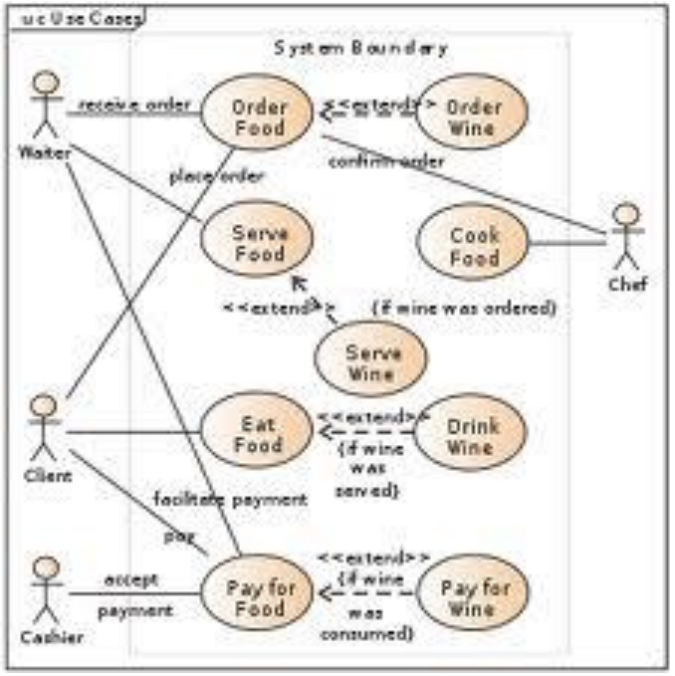
Camel bietet eine mächtige und flexible Fehlerbehandlung:

- *DefaultErrorHandler* – Keine Wiederholung, Exceptions an Aufrufer
- *DeadLetterChannel* – Fehlerhafte Exchanges werden an einen Dead Letter Queue gesendet, Wiederholungen konfigurierbar
- *NoErrorHandler* – Keine Fehlerbehandlung
- *LoggingErrorHandler* – Nur Protokollierung des Fehlers

Noch mehr Praxis…

Typische Anwendungsszenarien

- Adapter für Protokoll- oder Datenformate (Mediation)
- Bestandsapplikationen als Web Services verfügbar machen
- Systeme als Proxy/Gateway entkoppeln
- Services aggregieren (leichtgewichtige Orchestrierung, technische Geschäftsprozesse)
- Zusätzliche Verarbeitungsschritte (z.B. Validierung von Nachrichten)
- Testunterstützung (Services simulieren, Traffic erzeugen, usw.)
- …u.v.m.!



Weitere Informationen

- Camel Website camel.apache.org
- Talend Community Coders : coders.talend.com
- Talendforge Forum talendforge.org/forum
- Christian Schneiders Blog liquid-reality.de
- Christian Schneider auf Twitter @schneider_chris
- Beispiele https://github.com/cschneider/camel-webinar