



Apache Camel Übungen

Version 2.4 (vom 05.10.2020)

Thomas Bayer, Shaan Jayaratna,
Tobias Polley, Christian Blum

**predic8 GmbH
Koblenzer Straße 65
53173 Bonn**

Inhaltsverzeichnis

1.	Vorbereitung	4
1.1.	Software Downloads	4
1.2.	Entpacken der Kursdisk	4
1.3.	Verwendung eines Proxies	4
1.4.	Eclipse: Sofortiges Aktualisieren des Projekts bei Dateiänderungen.....	5
1.5.	Eclipse: Anzeigen von Dateien, deren Namen mit einem Punkt beginnt	6
2.	Enterprise Integration Patterns	7
2.1.	Der File Endpunkt	7
2.2.	Message Filter Muster.....	8
2.3.	Content Based Router (aka choice)	10
2.4.	Formatierung der Java-Dateien mit Camel-Routen	11
2.5.	Wire Tap.....	12
2.6.	Splitter Muster	13
2.7.	Camel Sprachen am Beispiel der <i>SimpleLanguage</i>	14
2.8.	Serialisierung und Deserialisierung.....	15
2.9.	REST Webservice	18
2.10.	Der Exchange	18
2.11.	Enricher	19
2.12.	Aggregator	20
3.	Hilfsmittel zum Debuggen.....	24
3.1.	Tracing von Routen	24
3.2.	Camel mit <i>hawt.io</i> und <i>jolokia</i>	25
4.	Prozessor.....	26
4.1.	LogProcessor	26
5.	Error Handling.....	27
5.1.	DefaultErrorHandler	27
5.2.	Vorstufe	27
5.3.	Redelivery	28
5.4.	Dead Letter Channel	29
6.	Camel Komponenten	30
	Übung: Offizielle Camel Komponenten auflisten.....	30
	Übung: ActiveMQ installieren	30
	Übung: Hawtio mit ActiveMQ verbinden	31
6.1.	Die JMS Komponente	32
6.2.	Die Jetty Komponente	38
6.3.	Die Camel REST DSL	39
6.4.	Die Bean Komponente	41
6.5.	Bean Binding.....	43
6.6.	JAXB	44
7.	Camel mit Spring Boot.....	47
7.1.	Daten an Camel senden mit dem ProducerTemplate	51
7.2.	Daten von einem Endpunkt lesen	52
8.	Transaktionen	53
8.1.	Nicht-transaktionales Verhalten	53
8.2.	Transaktionen	55
8.3.	Verteilte Transaktionen (XA)	57
9.	JDBC	58

9.1. Apache Derby entpacken und starten	58
9.2. JDBC Komponente	59
10. CXF	61
10.1. Der CXF Endpunkt	61
11. Camel Test Kit	63
11.1. Routen testen	63
11.2. Endpunkte wegmocken	64
11.3. Weitere Erwartungen hinzufügen	65
11.4. Ersetzen der Input-Komponente	66
12. Timer	67
12.1. Timer Komponente	67
13. Micronaut Monitoring	68
13.1. Monitoring im Projekt	68
14. Proxies und Bean Komponente	69
14.1. Proxies für Endpunkte erstellen	69
14.2. Asynchrone Proxies	71

1. Vorbereitung

1.1. Software Downloads

- JDK 11 oder höher
- IntelliJ IDEA 2020 mit Camel Plugin
(Alternativ zu IntelliJ Eclipse : Eclipse IDE for Java EE Developers)
- Maven 3.X
- Git Bash oder cURL (<http://curl.haxx.se/>) Windows Generic Binaries
- Hawt.io 2.10.2 app jar
- Jolokia JVM Agent 1.6.2

- Apache Derby 10.15.1.3 Binaries (für die Camel JDBC Übung)
- Apache ActiveMQ 5.15.9

1.2. Entpacken der Kursdisk

Entpacke die Datei *kursdisk.zip* von der Kursdisk in das Verzeichnis *C:\kurs*. Nach dem Entpacken sollte sich ein Verzeichnis *kursdisk* im *kurs* Ordner befinden.

1.3. Verwendung eines Proxies

Falls kein direkter Internet Zugang vorhanden ist, muss für Maven in *C:\Users\Benutzer\.m2\settings.xml* ein HTTP Proxy Server eingestellt werden:

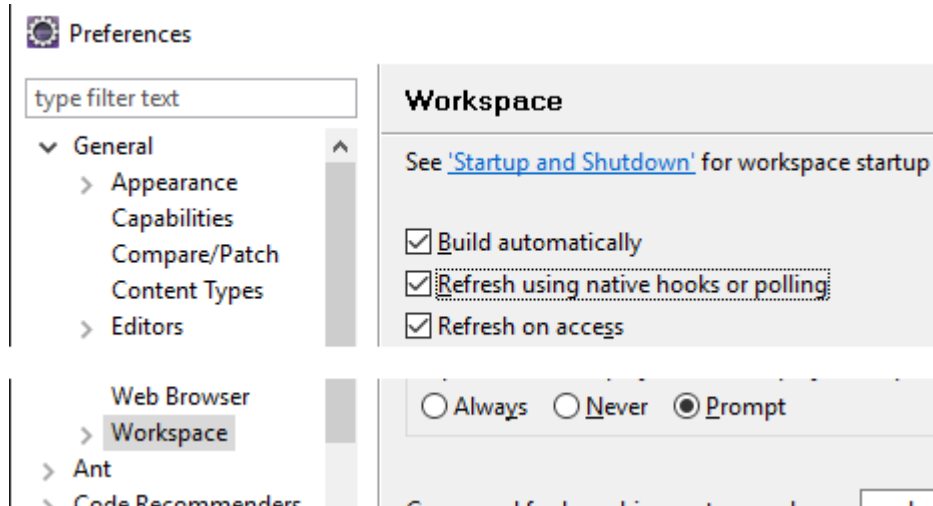
```
<settings>
.
.
<proxies>
  <proxy>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.predic8.de</host>
    <port>3128</port>
    <username>?</username>
    <password>?</password>
    <nonProxyHosts>www.google.com|*.predic8.com</nonProxyHosts>
  </proxy>
</proxies>
.
.
</settings>
```

Listing 1.1: Proxy Einstellungen

1.4. Eclipse: Sofortiges Aktualisieren des Projekts bei Dateiänderungen

Mit IntelliJ kannst Du diese Übung überspringen.

1. Wähle im Menü *Window, Preferences* .
2. Expandiere links *General* und wähle darunter *Workspace* aus.
3. Setze dann das Häkchen bei *Refresh using native hooks or polling*.

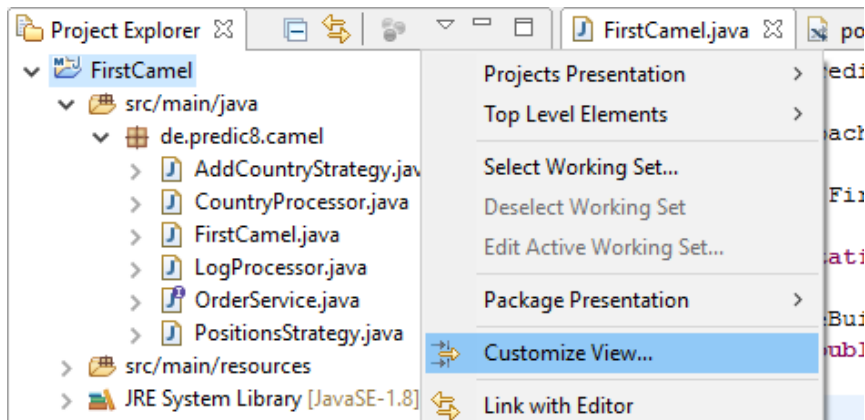


4. Wähle *OK*.

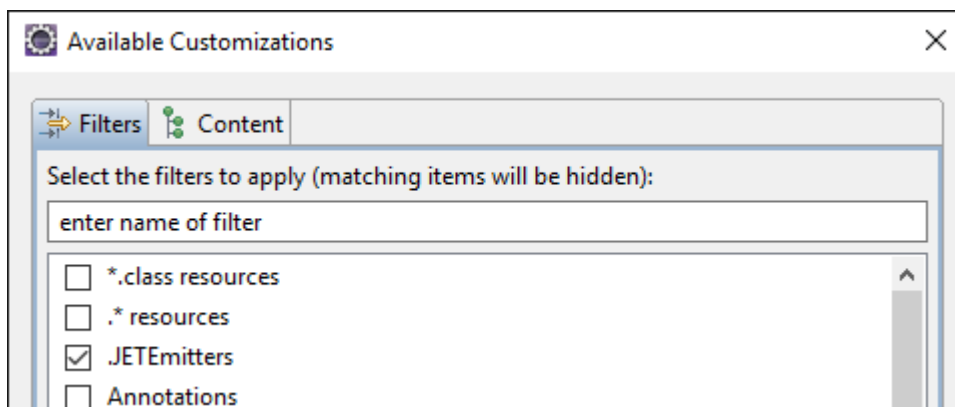
1.5. Eclipse: Anzeigen von Dateien, deren Namen mit einem Punkt beginnt

Mit IntelliJ kannst Du diese Übung überspringen.

1. Klicke über dem *Project Explorer* auf das nach unten zeigende Dreieck, und wähle dann *Customize View...* .



2. Entferne das Häkchen bei *.*resources* .



3. Wähle *OK*.

2. Enterprise Integration Patterns

2.1. Der File Endpunkt

Übung: Erstellen einer Route

1. Öffne / Importiere das Projekt *c:\kurs\kursdisk\Wholesale*.

In IntelliJ öffnest Du am besten die Datei *pom.xml* aus diesem Verzeichnis. IntelliJ importiert das Projekt dann automatisch als Maven-Projekt.

2. Betrachte die Datei *Wholesale.java*.
3. Erweitere den *RouteBuilder* wie folgt:

```
RouteBuilder builder = new RouteBuilder() {  
    public void configure(){  
  
        from("file:in")  
            .to("file:out");  
    }  
};
```

Listing 2.1: Konfiguration einer Route

4. Führe die Klasse aus.
5. Betrachte die Dateien *data/order.csv* und *data/order.xml*.
6. Aktualisiere das gesamte Projekt.

Eclipse: Wähle dafür den Projektordner aus und drücke auf *F5*.

IntelliJ: Öffne das Kontextmenü und wähle "Reload from Disk".

Es sollte ein neuer Ordner mit dem Namen *in* erscheinen.

7. Kopiere die Dateien *order.csv* und *order.xml* in den *in* Ordner.
8. Aktualisiere das Projekt wieder.
9. Betrachte den Inhalt vom *in* und *out* Ordner.
10. Beende das Programm indem du in der Konsole *Enter* drückst.

Hinweis: Camel verwendet eine eigene DSL, *domain-specific language*. Diese ist in Java eingebunden.

Übung: Verhindern des Löschens aus *in*

1. Ändere die Route wie folgt:

```
from("file:in?noop=true")
    .to("file:out");
```

Listing 2.2: URI Optionen

2. Lösche den Inhalt des *out* Ordners.
3. Führe die Klasse erneut aus und kopiere die beiden Dateien *order.csv* und *order.xml* in den *in* Ordner.
4. Betrachte den Inhalt des *in* und *out* Ordners.

2.2. Message Filter Muster**Übung:** Filtern nach Dateiendungen

1. Erweitere die Route wie folgt:

```
...
import static org.apache.camel.Exchange.FILE_NAME;
...
RouteBuilder ...

    from("file:in?noop=true")
        .filter(header(FILE_NAME).endsWith(".xml"))
        .to("file:out");
```

Listing 2.3: Filter

Hinweis: Beim Tippen hilft dir die Auto-Completion. (`.filter(...)`)

2. Lösche den Inhalt des *out* Ordners.
3. Starte das Programm neu. Du ersparst dir das Kopieren in den *in* Ordner, da der Inhalt nicht mehr gelöscht wird.
4. Betrachte den Inhalt des *out* Ordners.

Übung: Filtern nach regulären Ausdrücken (regex)

1. Lasse wieder CSV Dateien zu, indem du die Filterbedingung änderst.

```
from("file:in?noop=true")
    .filter(header(FILE_NAME) .regex(".*\\.csv|.*\\.xml"))
    .to("file:out");
```

Listing 2.4: CSV und XML Dateien zulassen

Hinweis: Reguläre Ausdrücke (regex) kann man unter <https://regex101.com/> auf Vollständigkeit testen.

Reguläre Ausdrücke sind eine eigene DSL

2.3. Content Based Router (aka choice)

Übung: Aufteilen nach Bedingungen

1. Erweitere die Route wie folgt:

```
from("file:in?noop=true")
.filter(header(FILE_NAME).regex(".*\\.csv|.*\\.xml"))
.choice()
    .when(header(FILE_NAME).endsWith("xml"))
        .to("file:out/xml")
    .endChoice()
    .otherwise()
        .to("file:out/csv");
```

Listing 2.5: Router hinzufügen

2. Teste die Änderungen. Vergiss nicht den *out* Ordner zu löschen und das Programm neu zu starten.

Hinweis: Vor *.otherwise()* ist derzeit noch kein *.endChoice()* nötig. Das Ende des *.when(...)*-Bereichs in der Route wird automatisch erkannt.

Im weiteren Verlauf der Übung erweitern wir diesen Bereich der Route umfangreich.

Der Java-Compiler kann dann die Java-DSL für Camel nicht mehr vollständig auflösen.

Daher ist dann ein *.endChoice()* nötig, um das Ende des *.when(...)*-Bereichs festzulegen.

2.4. Formatierung der Java-Dateien mit Camel-Routen

Übung: Auto-Formatieren der Java-Datei

1. Positioniere den Cursor in der Java-Datei.
2. IntelliJ: Drücke Strg+Alt+L.

Die Routen-Einrückung wurde kaput gemacht!

3. Drücke Strg+Z, um die letzte Änderung an der Datei rückgängig zu machen.

Übung: Deaktivieren der Auto-Formatierung

1. Füge vor der Route folgenden Kommentar ein:

```
// @formatter:off  
  
from("file:in?noop=true")
```

2. Füge hinter der Route folgenden Kommentar ein:

```
.to("file:out/csv");  
  
// @formatter:on
```

3. Öffne im Menü *File / Settings*.
4. Navigieren Sie zu *Editor / Code Style*.
5. Setzen Sie den Haken bei *Enable formatter markers in comments*.
6. Wähle *OK*.
7. Drücke erneut Strg+Alt+L.

2.5. Wire Tap

Übung: Anzapfen der Leitung

8. Erweitere dazu die Route wie folgt:

```
from("file:in?noop=true")
  .wireTap("file:log")
  .filter(header(FILE_NAME).regex(".*\\.csv|.*\\.xml"))
  .choice()
    .when(header(FILE_NAME).endsWith("xml"))
      .to("file:out/xml")
    .endChoice()
  .otherwise()
    .to("file:out/csv");
```

Listing 2.6: Anzapfen mit WireTap

9. Teste das Programm. Schaue in den *log* Ordner.

2.6. Splitter Muster

Übung: XML Aufspalten mit XPath

1. Erweitere die Route wie folgt:

```
.choice()  
    .when(header(FILE_NAME).endsWith("xml"))  
        .split(xpath("//item"))  
        .to("file:out/xml")  
    .end()  
.endChoice()
```

Listing 2.7: Route mit Splitter

2. Teste das Programm. Im *out* Ordner sollte ein neuer Ordner *xml* erstellt worden sein. In diesem befindet sich eine Datei.

Frage:

- Warum wurde nur eine Datei erstellt?
- Welcher Artikel befindet sich in der Datei?

2.7. Camel Sprachen am Beispiel der *SimpleLanguage*

Übung: Dateinamen errechnen

1. Ändere den Endpunkt:

```
...  
.split(xpath("//item"))  
.to("file:out/xml?fileName=item-" +  
    "${exchangeProperty.CamelSplitIndex}.xml")  
.end()
```

Listing 2.8: In mehrere Dateien speichern

2. Betrachte nun den Ordner *out*. Was hat sich geändert?

Übung: Camel Sprachen

1. Betrachte die Dokumentation der Camel Sprachen.
2. Betrachte die Dokumentation zur *SimpleLanguage*.
3. Betrachte die *Build-In* Variablen von *Simple*.

2.8. Serialisierung und Deserialisierung

Übung: Deserialisierung nach Java

1. Transformiere das XML Dokument in ein POJO bzw. Java-Objekt. Erweitere dazu die Route wie folgt:

```
JacksonXMLDataFormat xml = new JacksonXMLDataFormat();  
xml.setUnmarshalType(Item.class);
```

RouteBuilder ...

```
.split(xpath("//item"))  
  .unmarshal(xml)  
  .to("file:out/xml?fileName=item-" +  
      "${exchangeProperty.CamelSplitIndex}.xml")  
  .end()
```

```
};
```

Listing 2.9: Deserialisierung hinzufügen

2. Lösche die Dateien im *out* Ordner.
3. Starte das Programm mit den Änderungen. Schaue in den *out* Ordner. Beachte die Formatierung der Dateien nach der Deserialisierung.

Frage: Hat sich etwas geändert?

Übung: Zugriff auf den Body

1. Ändere die Route dazu wie folgt:

```
...
.unmarshal(xml)
.log("${body.class}")
.log("${body.good}")
.log("${body.quantity}")
.to("file:out/xml?fileName=item-" +
    "${exchangeProperty.CamelSplitIndex}.xml")
```

Listing 2.10: Body loggen

2. Betrachte die Ausgabe im Terminal.

Frage: Welchen Objekt-Typ enthält der Body?

3. Beachte die Annotation der Klasse *Item.java*.

```
@XmlElement
public class Item{
    .
    .
    .
}
```

Listing 2.11: Klasse Item annotiert

Übung: Serialisierung in das JSON-Format

1. Ändere die Route wie folgt:

```
...
JacksonDataFormat json = new JacksonDataFormat();

RouteBuilder ...

    .unmarshal(xml)
    .marshal(json)
    .to("file:out/json?fileName=item-
        ${exchangeProperty.CamelSplitIndex}.json")
    .end()

};
```

Listing 2.12: Serialisierung als JSON-Datei

2. Lösche die Dateien im *out* Ordner.
3. Starte das Programm mit den Änderungen. Schau in den *json* Ordner. Beachte die Formatierung der Dateien nach der Serialisierung.

Übung: Für die Schönheit

1. Aktiviere das *PrettyPrinting*:

```
JacksonDataFormat json = new JacksonDataFormat();
json.setPrettyPrint(true);
```

2. Teste die Änderung.

2.9. REST Webservice

Übung: Beef and Potatoes

1. Öffne / Importiere das Projekt *country-service* aus der *kursdisk*.

Mit IntelliJ öffnest Du am besten die Datei *pom.xml* aus diesem Verzeichnis in einem neuen Fenster: IntelliJ importiert das Projekt dann als Maven-Projekt.

2. Starte die Klasse *CountryService.java*.
3. Öffne einen Browser und öffne den Link aus der *README.md*.
4. Ändere *Potatoes* zu *Beef* und öffne den Link erneut.

Zusatzübung: Betrachte den Befehl

`$ curl "http://localhost:8888/supplier/?food=Beef" -v` im Terminal.

2.10. Der Exchange

Übung: Zentrale Datenstruktur

Betrachte die Java Dokumentation zur *Exchange* Klasse

2.11. Enricher

Übung: Aggregieren von zwei Exchanges

1. Betrachte die Klasse *AddCountry*. Erweitere die Klasse wie folgt:

```
public class AddCountry implements AggregationStrategy{
    public Exchange aggregate(Exchange itemExc,
                             Exchange countriesExc {

        Item item = itemExc.getIn().getBody(Item.class);

        item.setCountry(getCode(countriesExc));

        return itemExc;
    }

    private String getCode(Exchange exc){

        return exc.getIn().getBody(String.class);
    }
}
```

Listing 2.13: AddCountry

2. Überarbeite die Route:

```
.unmarshal(xml)
.enrich()
    .simple("http://localhost:8888/supplier/?food=${body.good}")
    .aggregationStrategy(new AddCountry())
.marshall(json)
```

Listing 2.14: Route mit Enricher

3. Teste die Route. Betrachte die Fehlermeldung.
4. Füge der *pom.xml* eine neue *Dependency* hinzu:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-http</artifactId>
        <version>3.5.0</version>
    </dependency>
</dependencies>
```

Listing 2.15: Hinzufügen der Dependency in pom.xml

5. Teste die Route. Betrachte die Dateien im *out* Ordner.
6. Gib vor und nach dem Enrich Muster den Body über das log aus.

2.12. Aggregator

Übung: Erstellen eines Aggregators

1. Betrachte die Klasse *ListStrategy*.
2. Erweitere die Klasse, so dass aus dem neuen Exchange der Inhalt als *List<Item>* lokal gespeichert wird:

```
public class ListStrategy implements AggregationStrategy {

    @Override
    public Exchange aggregate(Exchange aggregate,
                             Exchange newExc) {

        List<Item> items =.getItems(aggregate);

        items.add(getItem(newExc));

        newExc.getIn().setBody(items);

    }

    List<Item> getItems(Exchange exc){

        if(exc == null)
            return new ArrayList<Item>();

        return exc.getIn().getBody(List.class);

    }

    Item getItem(Exchange exc){

        return exc.getIn().getBody(Item.class);

    }

}
```

Listing 2.16: ListStrategy

Übung: Einbinden eines Aggregators

1. Füge nach dem Enricher einen Aggregator ein und passe die Route an:

```
.split(xpath("//item"))
    .unmarshal(xml)
    .enrich()
        .simple("http:localhost:8888/supplier/?food=${body.good}")
        .aggregationStrategy(new AddCountry())

    .aggregate(simple("${body.country}"), new ListStrategy())
        .completionSize(1)

    .marshal(json)
    .to("file:out/list?fileName=item-"+
        "${exchangeProperty.CamelSplitIndex}.json")
.end()
```

Listing 2.17: Route mit Aggregator

2. Stoppe und starte Camel.
3. Betrachte die Dateien im *list* Ordner.

Übung: Verändere die Einstellungen des Aggregators

1. Passe den *Aggregator* an:

```
.aggregate(simple("${body.country}"), new ListStrategy())  
    .completionSize(2)  
.marshal(json)
```

Listing 2.18: Geänderte completionSize

2. Stoppe und starte Camel.
3. Wie lässt sich das Ergebnis auslegen?

Frage: Wo ist das Ketchup?

Übung: Füge eine weitere Bestellung hinzu

1. Betrachte die *data/big-order.xml*.
2. Kopiere die *big-order.xml* ins *in* Verzeichnis.
3. Betrachte die verschiedenen Dateien.
4. Wie lässt sich das Ergebnis auslegen?

Übung: Weitere Optionen des Aggregators

1. Schlage den Aggregator in der Camel Dokumentation nach.

Achtung: Suche die Seite Camel Aggregator2!

2. Ändere die *CompletionSize* auf 3
3. Lösche das *list* Verzeichnis
4. Führe die Route erneut aus
5. Wie lässt sich das Ergebnis auslegen?
6. Setze einen *completionTimeout* von einer Sekunde.
7. Führe die Route erneut aus

Frage: Was hat sich geändert?

Übung: Aggregation ohne eigene Klasse

1. Alternativ zur Erstellung eines Aggregator in einer eigenen Javaklasse, kann man für die Aggregation die *FlexibleAggregationStrategy* verwenden

```
.aggregate(simple("${body.country}"),
    new FlexibleAggregationStrategy()
        .storeInBody()
        .pick(simple("${body}"))
        .accumulateInCollection(ArrayList.class)
    )
.completionSize(2)
```

Listing 2.19: Alternativ die FlexibleAggregationStrategy

3. Hilfsmittel zum Debuggen

3.1. Tracing von Routen

Übung: Tracing von Routen

1. Aktiviere das Tracing für den Context.

```
...  
CamelContext ctx = new DefaultCamelContext();  
  
ctx.setTracing(true);  
ctx.addRoutes(builder);  
...
```

Listing 3.1: Tracing im Camel Context aktivieren

2. Starte Camel und teste die Routen.
3. Betrachte den Trace in der Ausgabe.
4. Benenne die Route um:

```
from("file:in?noop=true")  
    .routeId("OrderRoute")  
    .wireTap("file:log")
```

Listing 3.2: Benennen der Route

5. Betrachte den Trace in der Ausgabe

3.2. Camel mit *hawt.io* und *jolokia*

Hinweis: Unter *libs* ist im *kursdisk*-Order eine Version von *jolokia* und *hawt.io* hinterlegt.

Übung: Management mit *jolokia*

1. Lade den *Jolokia-jvm-Agent* von <https://jolokia.org/download.html> herunter.
2. Wähle im Menü *Run, Run Configurations*. Wähle *Wholesale* aus.
3. Trage unter *Arguments* als VM Arguments folgendes ein:

```
-javaagent:C:\Users\kurs\jolokia-jvm-1.6.2-agent.jar=port=7777,host=localhost
```

Hinweis: `.jar=port=` ist korrekt

4. Füge in der *pom.xml* die Abhängigkeit *camel-jmx* hinzu:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jmx</artifactId>
  <version>3.5.0</version>
</dependency>
```

Übung: Management anzeigen mit *hawt.io*

1. Lade das *hawt.io app jar* von <http://hawt.io/docs/get-started/> herunter
2. Starte es mit:

```
java -jar hawtio-app-2.10.2.jar --port 8090
```

3. Öffne <http://localhost:8090/> im Browser.
4. Wähle *Connect, add connection*.
5. Wähle Name *test*, Scheme *http*, Host *localhost*, Port *7777* und Path */jolokia*.
6. Stelle die Verbindung her.
7. Wähle im Menü *Camel*.

Übung: Debugging mit *hawt.io*

1. Debugging kann angeschaltet werden:

```
getContext().setDebugging(true);
```

4. Prozessor

4.1. LogProcessor

Übung: Erstellen eines *LogProcessor*

1. Betrachte die Datei *LogProcessor.java*.

```
public class LogProcessor implements Processor {  
  
    public void process(Exchange exc) throws Exception {  
        System.out.println("Received: " +  
                           exc.getIn().getHeader("CamelFileName"));  
    }  
  
}
```

Listing 4.1: LogProcessor Implementation

2. Füge den *LogProcessor* in die Route ein:

```
.filter(header(FILE_NAME).regex(".*\\.csv|.*\\.xml"))  
.process(new LogProcessor())  
.choice()
```

Listing 4.2: LogProcessor hinzufügen

3. Teste das Programm. Betrachte die Ausgabe in der Konsole.
Schalte ggf. vorher das Tracing wieder aus, um einen besseren Überblick zu bekommen.

Zusatzübung:

Implementiere einen *Processor* als *Anonyme Innere Klasse*.

5. Error Handling

5.1. DefaultErrorHandler

Übung: Error Handling in Camel

1. Importiere das Maven-Projekt `c:\kurs\kursdisk\CamelErrorHandling`.
2. Betrachte die Datei `CamelErrorHandling.java`.
3. Betrachte die Datei `FailProcessor.java`.

```
public class FailProcessor implements Processor {  
  
    private int i = 0;  
  
    @Override  
    public void process(Exchange exc) throws Exception {  
        if ( i++ < 5 ) {  
            throw new Exception("Something went wrong!");  
        }  
    }  
  
}
```

Listing 5.1: FailProcessor

4. Führe das Programm aus und betrachte die Fehlermeldung.
5. Öffne die Datei `src/main/resources/logging.properties`.
6. Erweitere die Logausgaben, indem du das Kommentarzeichen vor folgender Zeile entfernst:

```
org.apache.camel.processor.level = INFO  
org.apache.camel.component.file.level = INFO
```

Listing 5.2: CamelLogger aktivieren

7. Führe das Programm aus und betrachte die Fehlermeldungen.

5.2. Vorstufe

Baue einen Prozessor am Anfang ein, der „Vorstufe“ auf der Kommandozeile ausgibt.

5.3. Redelivery

Übung: Redelivery in einer Camelroute

1. Erweitere die Konfiguration der Route wie folgt:

```
public void configure() {  
    errorHandler(defaultErrorHandler()  
        .maximumRedeliveries(4)  
        .retryAttemptedLogLevel(LoggingLevel.INFO));  
  
    from("file:in?delay=10000&noop=true")  
        .process(exc -> System.out.println("Vorstufe!"))  
        .process(new FailProcessor())  
        .to("file:out").log("Erfolg!");  
}
```

Listing 5.3: Redelivery aktivieren

2. Teste das Programm.

5.4. Dead Letter Channel

Übung: Erstellen eines Dead Letter Channels

1. Ändere die Konfiguration der Route wie folgt:

```
public void configure() {  
    errorHandler(deadLetterChannel("file:errors")  
        .maximumRedeliveries(3)  
        .retryAttemptedLogLevel(LoggingLevel.INFO));  
  
    from("file:in?consumer.delay=10000&noop=true")  
        .process(exc -> System.out.println("Vorstufe!"))  
        .process(new FailProcessor())  
        .to("file:out").log("Erfolg!");  
}
```

Listing 5.4: Dead Letter Channel

2. Teste das Programm.
3. Schaue in den Ordner *errors*.

6. Camel Komponenten

Übung: Offizielle Camel Komponenten auflisten

1. Öffne die Komponenten Übersichtsseite von Camel.

`http://camel.apache.org/components.html`

2. Schau dir die Dokumentation zur ActiveMQ Komponente an.

Übung: ActiveMQ installieren

1. Lade *ActiveMQ* von <https://activemq.apache.org/components/classic/download/> herunter.
2. Entpacke die Datei *apache-activemq-XXX-bin.zip*.
3. Öffne das Unterverzeichnis *apache-activemq-XXX*.
4. Starte ActiveMQ in der Konsole:

Windows:

`bin\win64\activemq.bat`

Linux

`./bin/activemq.sh`

Hinweis: Eine Version von *ActiveMQ* ist im *libs*-Verzeichnis hinterlegt.

Hinweis: ActiveMQ kann als Windows Dienst installiert werden. Für die Übungen benötigen wir das nicht.

Übung: Hawtio mit ActiveMQ verbinden

1. ActiveMQ zeigt nach dem Start am Ende des Logs in der Konsole folgende Nachricht an:

```
jvm 1 | INFO | ActiveMQ Jolokia REST API available at  
http://0.0.0.0:8161/api/jolokia/
```

2. Kehre in den ersten Hawtio Tab im Browser zurück (oder öffne nochmals <http://localhost:8090/hawtio>).
3. Drücke den *Add connection* Button, um eine neue Verbindung anzulegen.
4. Vergebe als Namen *activemq*, als Host *localhost* als Port *8161*, und als Path */api/jolokia/*.
5. Wähle *add*.
6. Gebe *admin* als Benutzername und *admin* als Passwort ein.
7. Öffne den ActiveMQ Menüpunkt mit dem *Connect Button*.

6.1. Die JMS Komponente

Übung: Zugriff auf JMS Queues

1. Importiere das Maven Projekt *CamelJMS*.
2. Falls *Build* Fehler verbleiben, mache einen Rechts-Klick auf das Projekt und wähle *Maven, Update Project, OK*.
3. Betrachte die *ConsumerRoute* Klasse

```
public class ConsumerRoute extends RouteBuilder {  
  
    public void configure() {  
        from("activemq:documentIn")  
        .to("activemq:documentOut");  
    }  
}
```

Listing 6.1: Camel Route als JMS Consumer und Producer

4. Betrachte die *camel-context.xml* Konfigurationsdatei, die du unter *src/main/resources/META-INF/spring* findest.

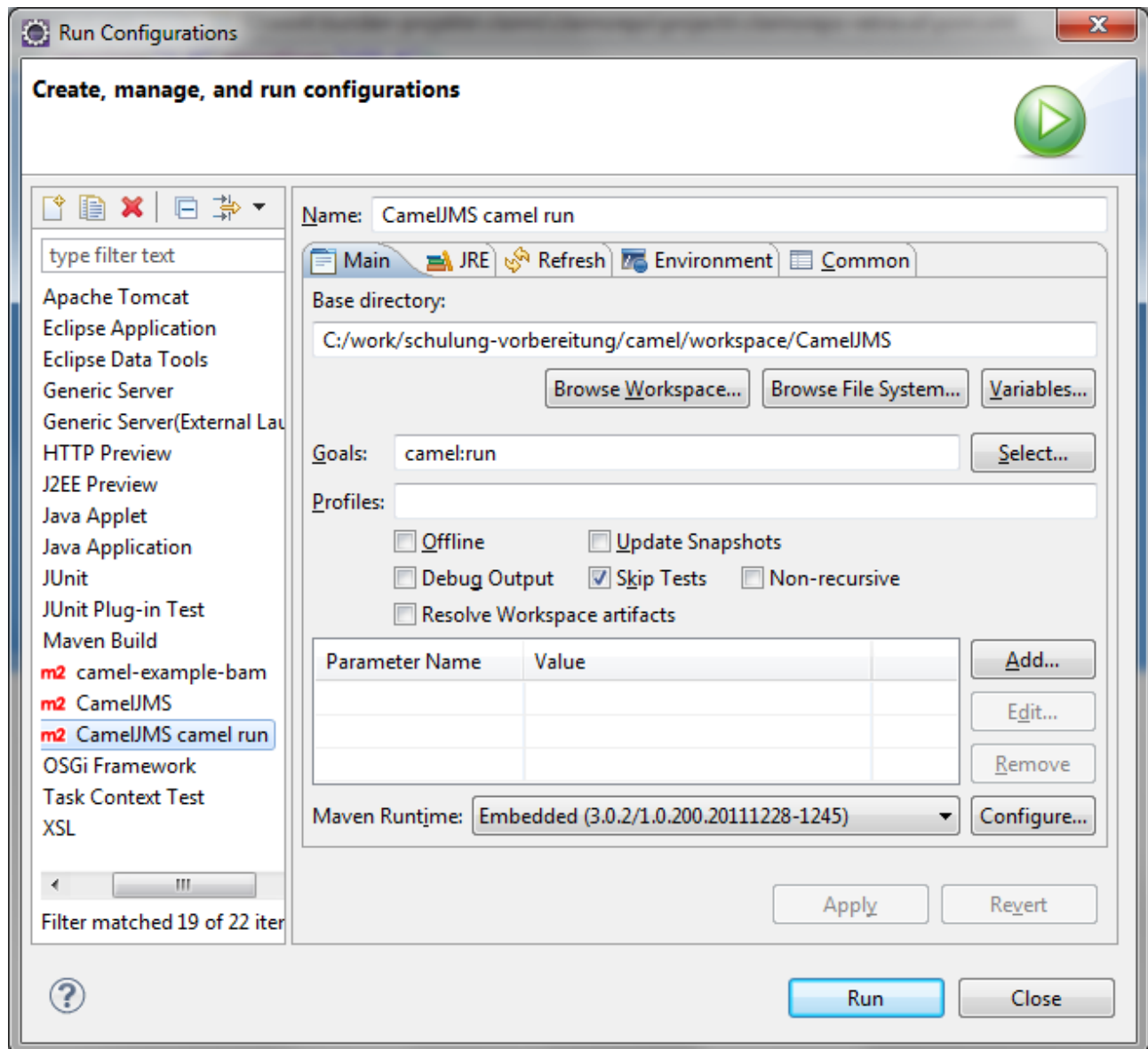
```
<beans ...>  
  <camelContext xmlns="http://camel.apache.org/schema/spring">  
    <packageScan>  
      <package>de.predic8.camel.jms</package>  
    </packageScan>  
  </camelContext>  
  <bean id="activemq"  
    class="org.apache.activemq.camel.component.ActiveMQComponent">  
    <property name="connectionFactory">  
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">  
        <property name="brokerURL"  
          value="tcp://localhost:61616" />  
      </bean>  
    </property>  
  </bean>  
</beans>
```

Listing 6.2: CamelContext und ActiveMQ Komponente

5. Starte das Projekt durch die *Starter*-Klasse.

Zusatzübung: Start mit *Maven Camel Plugin*

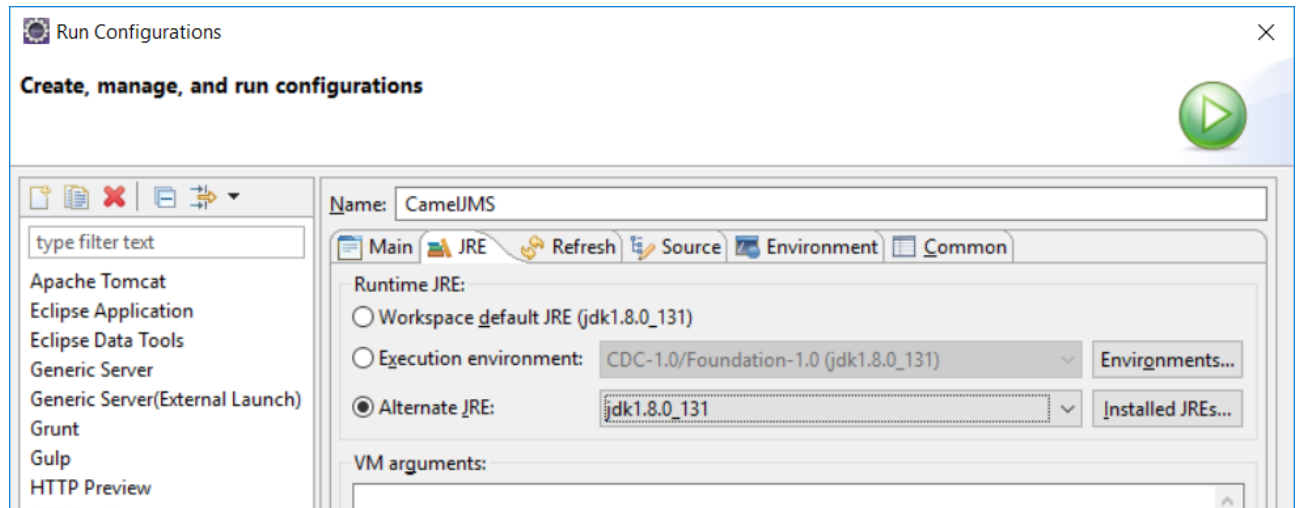
1. Führe das *CamelJMS* Projekt mit dem Maven Ziel *camel:run* aus. Verwende dafür eine *Run Configuration*.



2. Falls das Programm startet, ist die Übung abgeschlossen.

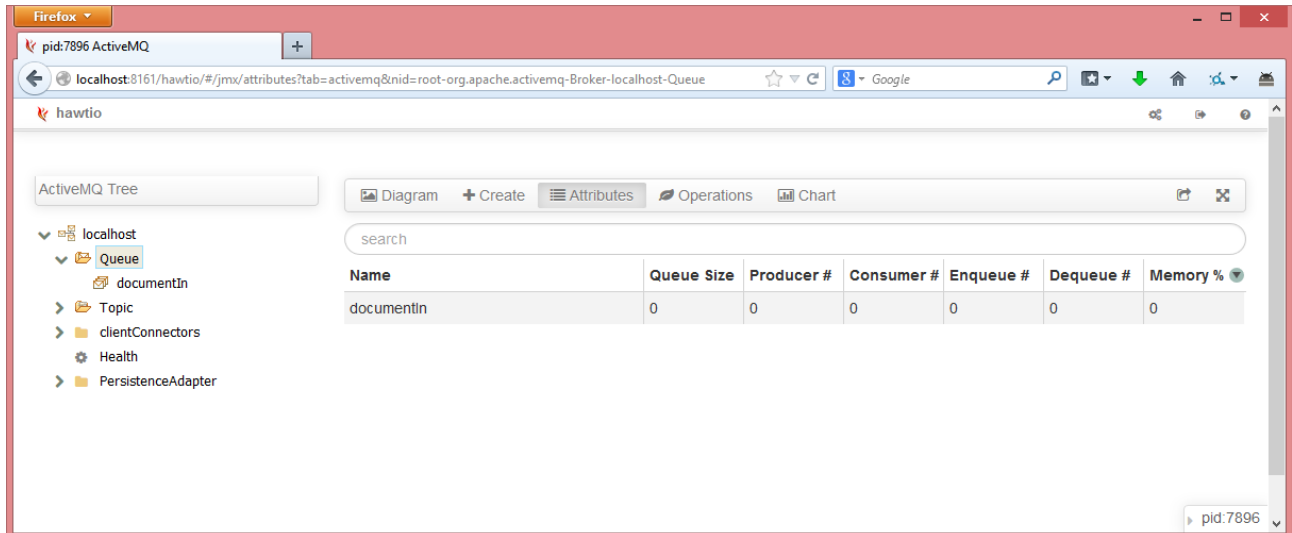
Wenn du die Meldung `No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK?` erhältst, ändere die Konfiguration der Run Configuration so, daß du im JRE Tab ein JDK auswählst.

Steht kein JDK zur Auswahl, kannst du es vorher über den Button *Installed JREs...* registrieren.

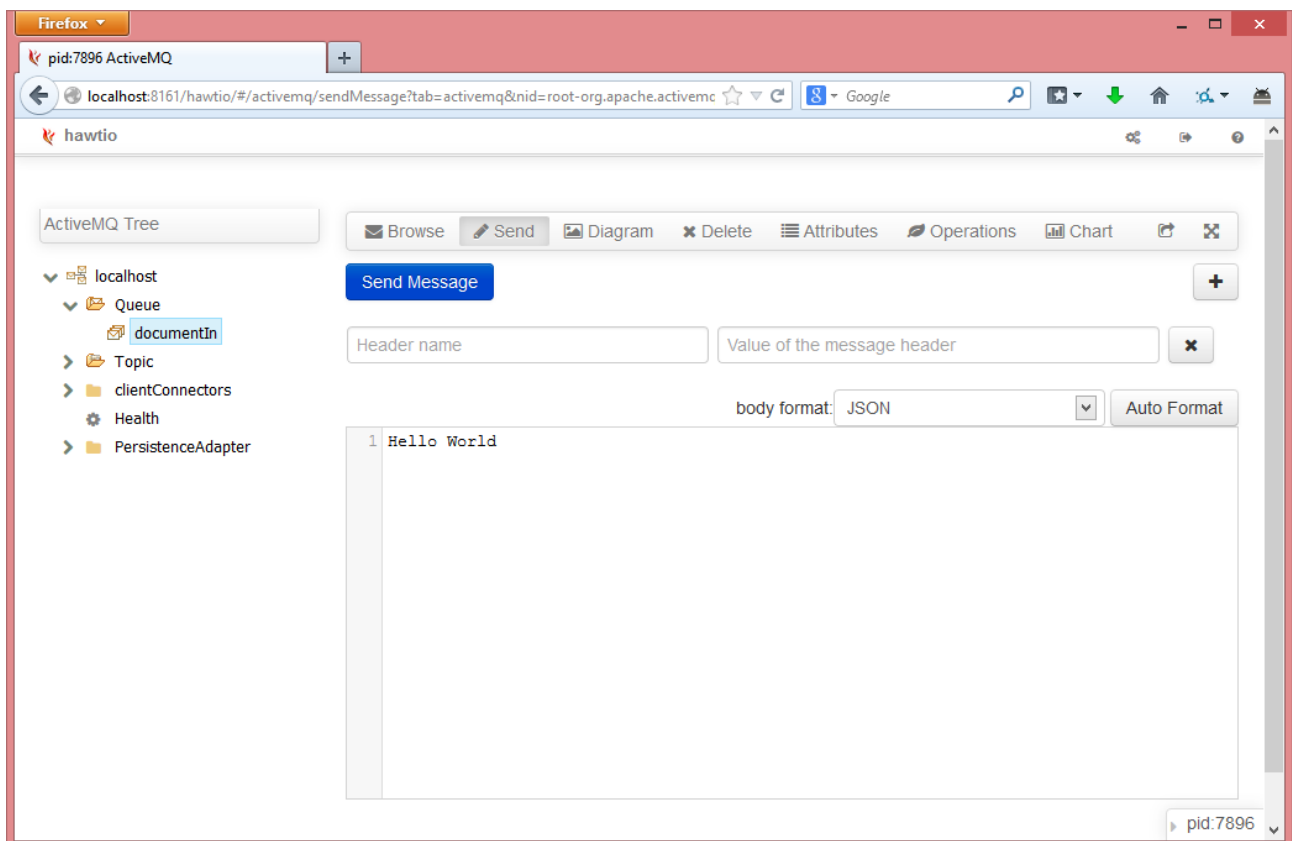


Übung: Nachrichten an die Queue senden

1. Kehre in den Browser zurück und öffne die ActiveMQ Ansicht im Hawtio.
2. Aktualisiere die Ansicht (F5), damit Hawtio den Navigationsbaum neu lädt.
3. Markiere und expandiere den Punkt *Queue* im Navigationsbaum.

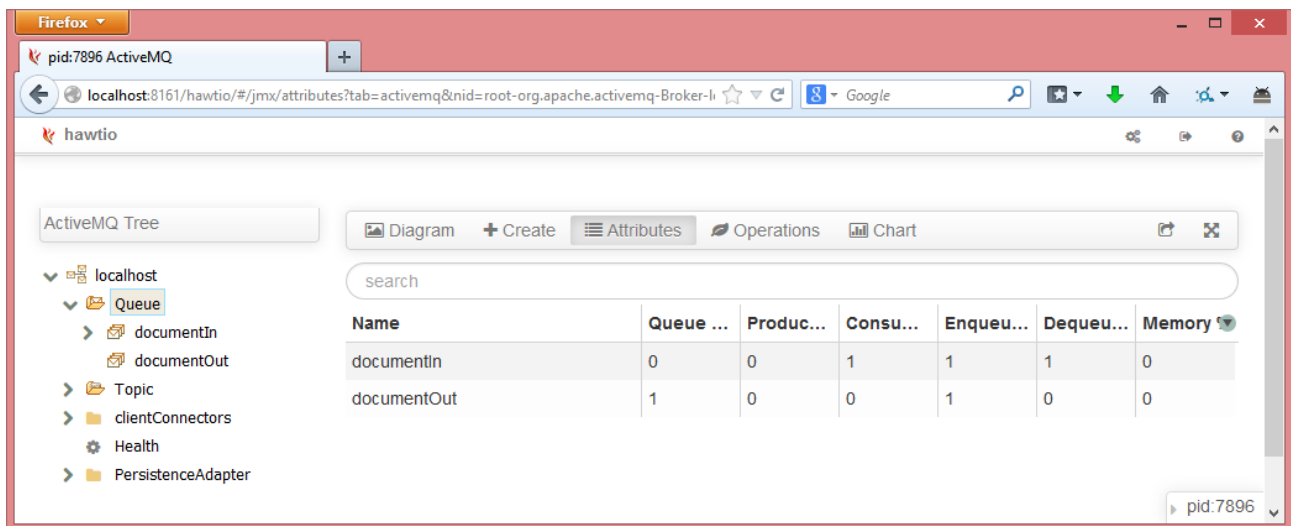


4. Klicke in der Navigation auf *documentIn*, dann in der Mitte auf *Send*, um eine Nachricht an die Queue zu senden.
5. Setze den Nachrichten Inhalt auf *Hello World*.



6. Klicke auf *Send Message*. Navigiere wieder auf die Queue Ansicht. Eventuell musst du kurz warten, bis sich die Ansicht aktualisiert.

Hinweis: Eventuell musst du F5 drücken, damit sich die Navigation aktualisiert und *documentOut* sichtbar ist.



Hinweis: Unsere Camel Route hat die Nachricht von der Queue *documentIn* gelesen und an die Queue *documentOut* versendet.

7. Betrachte die Nachricht indem du zuerst in der Navigation auf *documentOut* klickst, dann in der Mitte *Browse* wählst und die Message ID anklickst.
8. Expandiere die *Headers* Sektion.

The screenshot shows the Hawtio web interface for monitoring an Apache Camel application. The left sidebar, titled 'ActiveMQ Tree', shows a tree structure with 'localhost' expanded, containing 'Queue' (with 'documentIn' and 'documentOut') and 'Topic'. The 'documentOut' queue is selected. The main panel shows the 'Browse' tab for the selected queue. The message ID is 'ID:SANFRANCISCO-52713-1395658421984-0:0:1:1:1'. The 'Headers' section is expanded, displaying a table of headers. The message body is 'Hello World'.

Header	Value
JMSMessageID	ID:SANFRANCISCO-52713-1395658421984-0:0:1:1:1
JMSRedelivered	false
JMSDeliveryMode	PERSISTENT
CamelJmsDeliveryMode	1
JMSXGroupID	null
JMSCorrelationID	null
JMSType	null
JMSExpiration	0
OriginalDestination	null
JMSTimestamp	2014-03-24T11:53:42+01:00
JMSXUserID	null
JMSXGroupSeq	0
JMSPriority	4
JMSReplyTo	null
JMSDestination	queue://documentOut
breadcrumbId	ID:SANFRANCISCO-52647-1395657733677-3:1:1:1:1
BrokerPath	null

1 Hello World

6.2. Die Jetty Komponente

1. jetty (und http4) zu pom.xml hinzufügen
2. `from("jetty:http://localhost:9999/in")
 .to(„activemq:in“)`
3. `curl -d „foo=1“ http://localhost:9999/in -v`
Schlägt fehl
4. `...`
`.to` ersetzen durch `inOnly`
5. `curl -d „{„foo“ : 1}“ -H „Content-Type: application/json“ http://localhost:9999/in`
6. `curl -d „{„foo“ : 1}“ -H „Content-Type: application/json“ http://localhost:9999/in -H
 „Baz: 42“`
7. `...`
`.setBody(constant(„sucess“));`
8. Curl 5.
9. `...`
`.removeHeaders(„*“)`
10. `... -H „Correlation : 42“`

6.3. Die Camel REST DSL

1. Importiere das Projekt `c:\kurs\kursdisk\CamelREST`.
2. Definiere folgendes REST-Mapping sowie die zwei Routen:

```
@Component
public class DemoRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        restConfiguration().component("jetty").host("localhost")
            .port(5555).bindingMode(RestBindingMode.auto);

        rest("/users")
            .get("/").to("direct:getUsers")
            .post("/").to("direct:addUser");

        from("direct:getUsers")
            .setBody(constant("demo"));

        from("direct:addUser")
            .to("file:out");
    }
}
```

Listing 6.3: REST Konfiguration mit Camel

(Lege dazu die Klasse *DemoRoute* an.)

3. Starte das Projekt über die Klasse *MyApplication*.

Caused by: java.lang.IllegalStateException: Cannot find RestConsumerFactory in Registry or as a Component to use

```
at org.apache.camel.component.rest.RestEndpoint.createConsumer(
    RestEndpoint.java:519) ~[camel-core-2.19.0.jar:2.19.0]
at org.apache.camel.impl.EventDrivenConsumerRoute.addServices(
    EventDrivenConsumerRoute.java:69) ~[camel-core-2.19.0.jar:2.19.0]
at org.apache.camel.impl.DefaultRoute.onStartingServices(
    DefaultRoute.java:103) ~[camel-core-2.19.0.jar:2.19.0]
at org.apache.camel.impl.RouteService.doWarmUp(
    RouteService.java:172) ~[camel-core-2.19.0.jar:2.19.0]
at org.apache.camel.impl.RouteService.warmUp(
    RouteService.java:145) ~[camel-core-2.19.0.jar:2.19.0]
... 26 common frames omitted
```

Du erhältst die Fehlermeldung, da in unserem Projekt bisher keine Bibliothek für einen REST-Server eingebunden ist.

4. Füge dem *Project Object Model* (*pom.xml*) eine neue *Dependency* hinzu.

Hinweis: Das passiert korrekterweise unterhalb von `<dependencies>`, nicht unterhalb von `<dependencyManagement>`.

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

Listing 6.4: Jetty Component Dependency

5. Starte *MyApplication* erneut.
6. Rufe per Kommandozeile aus dem *data* Ordner den folgenden Befehl auf:

Hinweis: Tippe den Befehl in eine Zeile.

```
curl -H "Content-Type:text/xml" -d @user.xml
http://localhost:5555/users
```

7. Betrachte die Dateien im *out* Ordner.

Zusatzübung:

Erweitere dein Programm so, dass es *Requests* der folgenden Form verarbeiten kann:

```
http://localhost:5555/users/add?name=jim&password=panse
```

Implementiere einen *Processor* der den Namen und das Password des Users auf der Konsole ausgibt.

Tipp: Die Query Parameter werden im Message Header abgelegt.

6.4. Die Bean Komponente

1. Betrachte die Datei *UserManager.java*.

```
public class UserManager {  
  
    public void addUser(Exchange exc) {  
  
        System.out.println("User angelegt");  
        System.out.println(" Name: "+exc.getIn().getHeader("name"));  
        System.out.println(  
            " Password: "+exc.getIn().getHeader("password"));  
  
    }  
  
}
```

Listing 6.5: userManager

2. Ändere die Route.

```
from("direct:addUser")  
    .to("bean:userBean?method=addUser");
```

Listing 6.68: Bean URI

3. Füge das *UserManager Bean* hinzu, indem du die Klasse *UserManager* mit **@Component** annotierst:

```
import org.apache.camel.Exchange;
```

```
@Component("userBean")
```

```
public class UserManager {
```

```
Listing 6.7: : Annotation für userBean
```

4. Starte das Programm.
5. Rufe *cURL* auf.

```
curl -X POST "http://localhost:5555/users?name=jim&password=panse"
```

6. Betrachte die Ausgabe in der Konsole Ihres Programms.

Hinweis: Beim curl Aufruf die “ nicht Vergissn!

6.5. Bean Binding

1. Ändere die *UserManager* Klasse.

```
@Component("userBean")
public class UserManager {

    public void addUser(
        @Header("name") String name,
        @Header("password") String password) {

        System.out.println("User angelegt");
        System.out.println(" Name: "+name);
        System.out.println(" Password:"+password);

    }

}
```

Listing 6.8: Bean Binding Annotations

2. Teste dein Programm.

6.6. JAXB

1. Füge dem Projekt die JAXB Component hinzu.

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

Listing 41: JAXB Component

2. Ändere die *Route* wie folgt ab:

```
JaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
jaxbDataFormat.setContextPath("de.predic8.camel.advanced");

from("direct:addUser")
  .unmarshal(jaxbDataFormat)
  .to("bean:userBean?method=addUser");
```

Listing 6.9: JAXB Data Format

3. Betrachte die Datei *User.java*.

```
@XmlRootElement
public class User {

    private String name;
    private String password;

    @XmlAttribute
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @XmlAttribute
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Listing 6.10: Annotierte User Klasse

4. Ändere die Klasse *UserManager* wie folgt ab:

```
public void addUser(User user) {  
    System.out.println("User angelegt");  
    System.out.println(" Name: " + user.getName());  
    System.out.println(" Password: " + user.getPassword());  
}
```

Listing 6.11: Methode mit komplexem Parameter

5. Teste das Programm. Verwende den folgenden *cURL* Aufruf aus dem Data-Ordner:

```
curl -H "Content-Type:text/xml" -d @user.xml  
http://localhost:5555/users
```

Zusatzübung:

Erweitere dein Programm so, dass sowohl XML Dokumente als auch Query Parameter verarbeitet werden können.

7. Camel mit Spring Boot

Übung: Erstellen eines Spring Boot Projektes

1. Öffne <https://start.spring.io/> , wähle die aktuellste Spring Boot 2.1er Version aus.
Trage dort folgendes ein:

Group: de.predic8

Artifact: auftrag

unter Options, wähle Java 11

unter *Search dependencies to add:*

Camel, Actuator, Web, ActiveMQ 5

The screenshot shows the Spring Boot Start page. At the top, the 'Spring Boot' version is set to '2.1.10'. Under 'Project Metadata', the 'Group' is 'de.predic8' and the 'Artifact' is 'auftrag'. In the 'Options' section, the 'Name' is 'auftrag', the 'Description' is 'Demo project for Spring Boot', the 'Package Name' is 'de.predic8.auftrag', and the 'Packaging' is 'Jar'. The 'Java' version is set to '11'. In the 'Dependencies' section, 'activemq 5' is searched, and three dependencies are selected: 'Spring Web', 'Spring Boot Actuator', and 'Apache Camel'. The 'Generate' button is highlighted in green.

Spring Boot

2.2.2 (SNAPSHOT) 2.2.1 2.1.11 (SNAPSHOT) **2.1.10**

Project Metadata

Group
de.predic8

Artifact
auftrag

Options

Name
auftrag

Description
Demo project for Spring Boot

Package Name
de.predic8.auftrag

Packaging
Jar War

Java
13 **11** 8

Dependencies

Search dependencies to add

activemq 5

Selected dependencies

Spring Web
Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container. ✓

Spring Boot Actuator
Supports built in (or custom) endpoints that let you monitor
and manage your application - such as application health,
metrics, sessions, etc. ✓

Apache Camel
Apache Camel lets you create the Enterprise Integration
Patterns to implement routing and mediation rules a Java
based Domain Specific Language via Spring. ✓

Generate - Ctrl + G Explore - Ctrl + Space Share...

2. Wähle *Generate*, speichere die Datei und entpacke sie ins *kursdisk*-Verzeichnis.
3. Importiere das Projekt in die IDE.
4. In der *pom.xml*-Datei müssen die fehlenden Abhängigkeiten hinzugefügt werden

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.15.10</version>
</dependency>

<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
</dependency>
```

Listing 7.1: dependencies in der pom.xml

5. Um diese Route zu verwenden, muss dem *CamelContext* *ActiveMQ* hinzugefügt werden. Dazu erweitere die *AuftragApplication* wie folgt:

```
@SpringBootApplication
public class AuftragApplication implements
    CamelContextConfiguration {

    public static void main(String[] args) {
        run(AuftragApplication.class, args);
    }

    @Override
    public void beforeApplicationStart(CamelContext ctx) {
        ActiveMQComponent amq = new ActiveMQComponent();
        ctx.addComponent("activemq", amq);
    }

    @Override
    public void afterApplicationStart(CamelContext ctx) {
    }
}
```

Listing 7.2: Camel Konfiguration

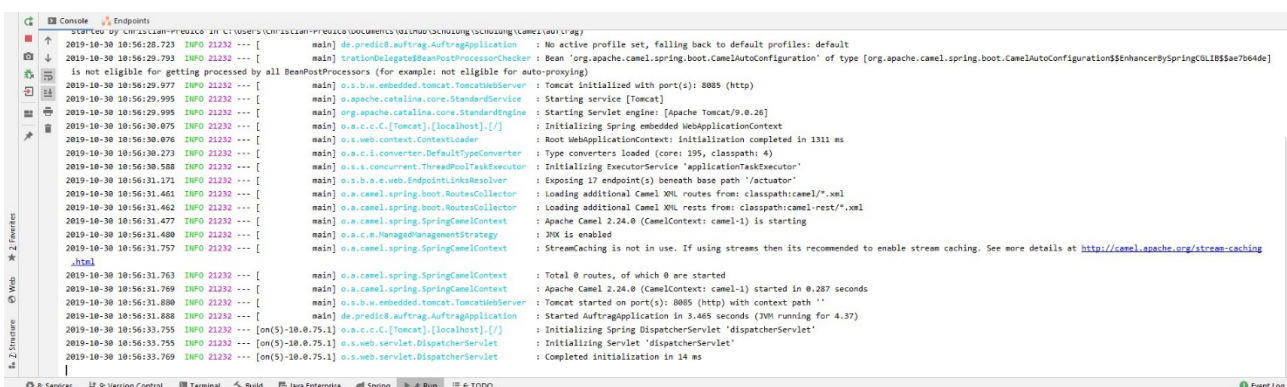
Hinweis: Hier kann der *CamelContext* genauso manipuliert werden wie im *Wholesale*-Projekt.

6. Globale Einstellungen für Spring Boot werden in der *application.context*-Datei verändert. In unserem Fall:

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
server.port=8085
spring.activemq.broker-url=tcp://localhost:61616
```

7. Starte das Programm.

Übung: Erstellen einer Route



```
2019-10-30 10:56:28.723 INFO 21232 --- [main] de.predic8.auftrag.AuftragApplication : No active profile set, falling back to default profiles: default
2019-10-30 10:56:29.793 INFO 21232 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.apache.camel.spring.boot.CamelAutoConfiguration' of type 'org.apache.camel.spring.boot.CamelAutoConfiguration$EnhancerBySpringCol1B9$ae7b64de' is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
2019-10-30 10:56:29.977 INFO 21232 --- [main] o.s.b.w.embedded.tomcat.TomcatStarter : Tomcat initialized with port(s): 8085 (http)
2019-10-30 10:56:29.995 INFO 21232 --- [main] org.apache.catalina.core.StandardEngine : Starting service [Tomcat]
2019-10-30 10:56:29.995 INFO 21232 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.26]
2019-10-30 10:56:30.075 INFO 21232 --- [main] o.s.c.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-10-30 10:56:30.076 INFO 21232 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1311 ms
2019-10-30 10:56:30.273 INFO 21232 --- [main] o.s.c.i.converter.DefaultTypeConverter : Type converters loaded (core: 195, classpath: 4)
2019-10-30 10:56:30.588 INFO 21232 --- [main] o.s.i.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-10-30 10:56:31.172 INFO 21232 --- [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 17 endpoint(s) beneath base path '/actuator'
2019-10-30 10:56:31.465 INFO 21232 --- [main] o.a.camel.spring.boot.RoutesCollector : Loading additional Camel XML routes from: classpath:camel/*.xml
2019-10-30 10:56:31.462 INFO 21232 --- [main] o.a.camel.spring.boot.RoutesCollector : Loading additional Camel XML routes from: classpath:camel-rest/*.xml
2019-10-30 10:56:31.477 INFO 21232 --- [main] o.a.camel.spring.SpringCamelContext : Apache Camel 2.24.0 (CamelContext: camel-1) is starting
2019-10-30 10:56:31.480 INFO 21232 --- [main] o.s.c.m.ManagedManagementStrategy : JMX is enabled
2019-10-30 10:56:31.757 INFO 21232 --- [main] o.a.camel.spring.SpringCamelContext : StreamCaching is not in use. If using streams then its recommended to enable stream caching. See more details at http://camel.apache.org/stream-caching
2019-10-30 10:56:31.763 INFO 21232 --- [main] o.a.camel.spring.SpringCamelContext : Total 0 routes, of which 0 are started
2019-10-30 10:56:31.769 INFO 21232 --- [main] o.a.camel.spring.SpringCamelContext : Apache Camel 2.24.0 (CamelContext: camel-1) started in 0.287 seconds
2019-10-30 10:56:31.880 INFO 21232 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8085 (http) with context path ''
2019-10-30 10:56:31.888 INFO 21232 --- [main] de.predic8.auftrag.AuftragApplication : Started AuftragApplication in 3.465 seconds (JVM running for 4.37)
2019-10-30 10:56:33.755 INFO 21232 --- [on(5)-10.0.0.75-1] o.s.c.c.c.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-10-30 10:56:33.755 INFO 21232 --- [on(5)-10.0.0.75-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2019-10-30 10:56:33.769 INFO 21232 --- [on(5)-10.0.0.75-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 14 ms
```

1. Erzeuge im *de.predic8.auftrag*-Verzeichnis eine neue Klasse *ConsumerRoute*.
2. Erweitere die Klasse um folgenden Code:

```
@Component
public class ConsumerRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception{
    }
}
```

Listing 7.3: RouteBuilder in SpringBoot

Hinweis: Vergiss nicht die Annotation der Klasse.

3. Füge die erste Route, wie folgt, hinzu:

```
@Override
public void configure() throws Exception{
    from("direct:auftrag-eingang")
        .id("auftrag-eingang ")
        .log("${body}")
        .to("activemq:auftrag-eingang ");
}
```

Listing 7.4: Route in die Queue

7.1. Daten an Camel senden mit dem ProducerTemplate

Übung: Erzeugen eines *RestController*s in *SpringBoot*

1. Erstelle eine neue Klasse *BestellController* und annotiere diese mit *@RestController*.
2. Füge folgende Instanzvariablen dem Controller hinzu:

```
final ProducerTemplate pt;  
final ConsumerTemplate ct;
```

Listing 7.5: Instanzvariablen des BestellController

3. Lass dir einen passenden Konstruktor erzeugen:

```
public BestellController(ProducerTemplate pt,  
                        ConsumerTemplate ct) {  
    this.pt= pt;  
    this.ct = ct;  
}
```

Listing 7.6: Konstruktor BestellController

Hinweis: *ALT+Einfg* bei IntelliJ

4. Erstelle das *PostMapping* im *BestellController*:

```
@PostMapping("/auftraege")  
public ResponseEntity beauftragen(@RequestBody JsonObject json) {  
    pt.send("direct:auftrag-eingang", exc ->  
        exc.getIn().setBody(json)  
    );  
    return ResponseEntity.ok("Auftrag erhalten");  
}
```

Listing 7.7: Senden an Camel durch ProducerTemplate

5. Teste das Programm. Verfolge die Queue in *hawtio*.

Verwende den folgenden *cURL* Aufruf in der Konsole:

```
curl -d '{"key1":"value1", "key2":"value2"}' -H "Content-Type:  
application/json" -X POST http://localhost:8085/auftraege
```

6. Betrachte <http://localhost:8085/actuator>

7.2. Daten von einem Endpunkt lesen

Übung: GetMapping ruft Endpunkt einer CamelRoute auf

1. Zur Ausgabe als JSON-Objekt füge folgende Abhängigkeit der *pom.xml* hinzu.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jackson</artifactId>
  <version>2.24.2</version>
</dependency>
```

2. Erweitere *ConsumerRoute* um folgendes:

```
JacksonDataFormat json = new JacksonDataFormat();
json.setPrettyPrint(true);
```

...

```
from("activemq:auftrag-eingang")
  .id("verarbeitung")
  .process( exec -> {
    exec.getIn().getBody(HashMap.class)
    .put("processed", true);
  })
  .marshal(json)
  .log("${body}")
  .to("activemq:bestaetigt");
```

Listing 7.8: Buisness-Logik

3. Erstellen des *GetMapping* in *BestellController*:

```
@GetMapping("/bestaetigungen")
public String bestaetigen() {
    return ct.receiveBody("activemq:bestaetigt", 10 ,
String.class);
}
```

Listing 7.9: Empfangen von Camel durch ConsumerTemplate

4. Starte das Programm. Verwende folgenden *cURL* Aufruf in der Konsole:

```
curl -X GET http://localhost:8085/bestaetigungen
```

8. Transaktionen

8.1. Nicht-transaktionales Verhalten

1. Wir kehren zum Projekt *CamelJMS* zurück.
2. Betrachte die *FailProcessor* Klasse

```
public class FailProcessor implements Processor {
    private int i = 0;
    public void process(Exchange exc) throws Exception {
        if ( i++ < 8 ) {
            throw new Exception("Something went wrong!");
        }
    }
}
```

Listing 8.1: FailProcessor

Hinweis: Je nach Broker und Version kann die Anzahl der vom Broker versuchten Retries variieren. Ggf. musst du entsprechend die Konstante in der *process* Methode anpassen.

3. Erweitere die Route wie folgt:

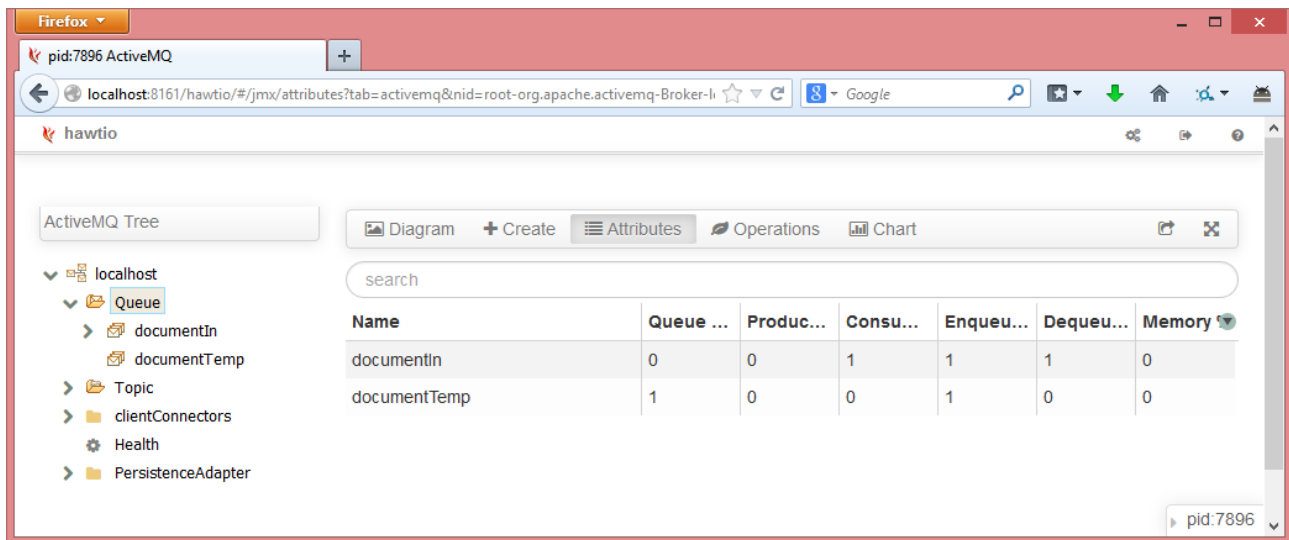
```
public class ConsumerRoute extends RouteBuilder {

    public void configure() {
        from("activemq:documentIn")
        .to("activemq:documentTemp")
        .process(new FailProcessor())
        .to("activemq:documentOut");
    }
}
```

Listing 8.2: Fehler verursachende Route

4. Stoppe und starte Camel.
5. Sende eine Nachricht über die Admin Konsole an die *documentIn* Queue.

6. Betrachte die Queue Ansicht



The screenshot shows the Hawtio web interface for ActiveMQ. The left sidebar displays the 'ActiveMQ Tree' with a hierarchy: localhost > Queue > documentIn, documentTemp, Topic, clientConnectors, Health, and PersistenceAdapter. The main content area is titled 'Attributes' and features a search bar and a table with queue statistics.

Name	Queue ...	Produc...	Consu...	Enqueu...	Dequeu...	Memory
documentIn	0	0	1	1	1	0
documentTemp	1	0	0	1	0	0

Frage: Was wäre wünschenswert?

8.2. Transaktionen

1. Betrachte die Spring Konfiguration.
2. Suche den *TransactionManager* und entferne den Kommentar.

```
<bean id="txManager"
  class="org.springframework.jms.connection.JmsTransactionManager">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616"/>
    </bean>
  </property>
</bean>
```

Listing 8.3: TransactionManager

3. Konfiguriere die ActiveMQ Komponente mit dem *TransactionManager*.

```
<bean id="activemq"
  class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
  </property>
  <property name="transacted" value="true" />
  <property name="transactionManager" ref="txManager" />
</bean>
```

Listing 8.4: Transaktionales ActiveMQ

4. Markiere die Route als *transacted*:

```
from(...)
  .transacted()
...
```

Listing 8.5: RouteDefinition transacted

5. Stoppe und starte Camel.
6. Führe einen *Purge* auf alle Queues durch.

7. Sende eine persistente Nachricht an die *documentIn* Queue.

Name	Queue Size	Producer #	Consumer #	Enqueue #	Dequeue #	Memory %	Dispatch
ActiveMQ.DLQ	1	0	0	1	0	0	0
documentIn	0	0	1	3	3	0	9
documentOut	0	0	0	1	1	0	0

Achtung: Nachricht muss persistent sein, um Transaktionen zu nutzen.

Frage: Wo ist die Nachricht hin?

8. Halte die Camel Route an.

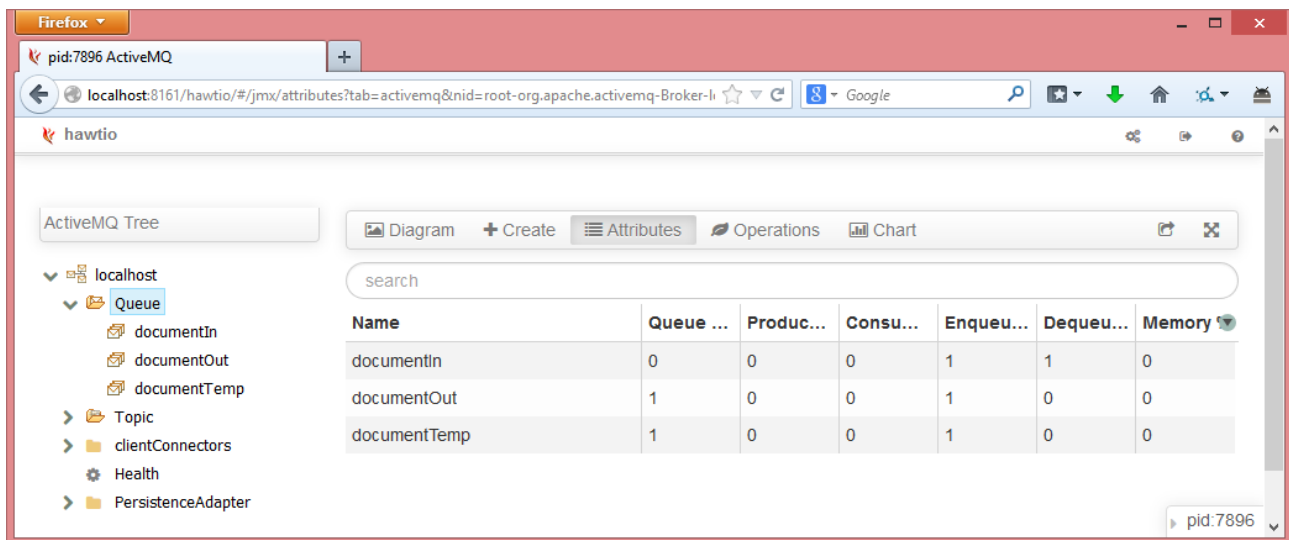
9. Erhöhe die Anzahl der Redelivery Versuche von ActiveMQ, indem du die *Connection URI* der *ConnectionFactory* anpasst.

```
<bean id="activemq"
  class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616?jms.re
deliveryPolicy.maximumRedeliveries=10" />
    </bean>
  </property>
  <property name="transacted" value="true" />
  <property name="transactionManager" ref="txManager" />
</bean>
```

Listing 8.6: Connection URL mit Redelivery Parametern

10. Starte die Camel Route erneut.

11. Sende eine weitere Nachricht an die *documentIn* Queue.



8.3. Verteilte Transaktionen (XA)

Transaktionen in verteilten Systemen können mittels des XA Protokolls von einer zentralen Stelle koordiniert werden.

Das könnte z.B. bei Überweisungen zwischen Banken zum Einsatz kommen, wenn der Stand des Quell- und Ziel-Konto der Überweisung in verschiedenen Datenbanken (oder wahrscheinlich sogar in verschiedenen Datenbanksystemen) gespeichert wird.

Alle beteiligten Systeme müssen XA unterstützen.

Die „Apache Aries Demo“ auf der Kursdisk demonstriert das am Beispiel von ActiveMQ und PostgreSQL, deren lokale Transaktionen in einer globalen Transaktion zusammengefasst werden. Die globale Transaktion wird durch Apache Aries verwaltet.

Das „Apache Aries Demo“ Verzeichnis der Kursdisk enthält zwei Maven-Projekte, die jeweils ein OSGi Bundle ergeben. Die README.txt in diesem Verzeichnis beschreibt das Vorgehen. (Die Bundles werden in ServiceMix 5.3.0 deployed. Das Bundle „com.predic8.route“ enthält eine Camel Route, die ActiveMQ mit Postgres verbindet und die transaktional ist.)

9. JDBC

9.1. Apache Derby entpacken und starten

1. Entpacke die Zip Datei in *c:\kurs*.
2. Öffne eine Kommandozeile und wechsele in folgendes Verzeichnis:

```
C:\kurs\db-derby-10.15.1.3-bin\db-derby-10.15.1.3-bin\bin
```

3. Starte den *Derby Network Server*, indem du die folgende Datei ausführst.

```
startNetworkServer.bat
```

Hinweis: Falls beim Start eine *AccessControlException* „access denied“ auftritt, führe folgende Schritte durch.

4. Starte *notepad.exe* als Administrator.
5. Stelle fest, in welchem Verzeichnis das „java“ Programm liegt, das von Derby verwendet wird.
6. Öffne die Datei *lib\security\java.policy* aus dem Java-Verzeichnis in Notepad, etwa *C:\Program Files\Java\jdk1.8.0_131\jre\lib\security\java.policy*.
7. Füge den Block

```
grant {  
    permission java.net.SocketPermission  
        "localhost:1527", "listen";  
};
```

unten an die Datei an und speichere sie.

8. Wiederhole Schritt 3.

9.2. JDBC Komponente

1. Importiere das Projekt *C:\kurs\kursdisk\CamelJDBC*.
2. Öffne die Datei *XmlToSql.java*.
3. Ändere die Methode *addPerson* wie folgt:

```
public String addPerson(@Header("name") String name,
                       @Header("password") String password) {

    System.out.println("Request");
    System.out.println(" Name: "+name);
    System.out.println(" Password: "+password);

    String sql =
    String.format(
        "insert into person (name, password) values ('%1s', '%2s')",
        name, password);

    System.out.println(" SQL: "+sql);
    return sql;
}
```

Listing 9.1: SQL Insert Anweisung

4. Öffne die Datei *camel-context.xml*.
5. Erweitere die *Route* wie folgt:

```
<beans ...>
    <bean id="xmlToSql" class="de.predic8.camel.jdbc.XmlToSql" />
    <bean id="derbyDS"
        class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/userDB;create=true" />
    </bean>
    <camelContext ...>
        <route>
            <from uri="jetty:http://localhost:5555/persons/add" />
                <to uri="bean:xmlToSql" />
                <to uri="jdbc:derbyDS" />
            </route>
        </camelContext>
    </beans>
```

Listing 9.2: Route mit JDBC Producer

6. Die Konfiguration von Camel ist beendet. Starte Camel, indem du *Run As, Maven Build...* wählst und als Goal *camel:run* eingibst.

7. Öffne eine Kommandozeile im *data* Verzeichnis des Projekts.

8. Führe folgenden Befehl aus. Schreibe ihn in eine Zeile.

```
"c:\kurs\db-derby-10.15.1.3-bin\bin\ij" -p ij.properties  
createPersonTable.sql
```

9. Rufe *cURL* auf.

```
curl "http://localhost:5555/persons/add?name=jim&password=panse"
```

Hinweis: Beachte, dass in der URL im Gegensatz zur letzten Übung *persons* anstatt *users* steht.

10. Führe folgenden Befehl aus:

```
"c:\kurs\db-derby-10.15.1.3-bin\bin\ij" -p ij.properties  
listPersons.sql
```

11. Die Tabelle sollte einen Eintrag mit dem Namen *jim* und dem Password *panse* enthalten.

10. CXF

10.1. Der CXF Endpunkt

1. Importiere das Projekt `c:\kurs\kursdisk\CamelWithCXF`.
2. Wähle auf dem Projekt *Maven, Update Project*.
3. Betrachte die Datei *HelloService.java*.
4. Betrachte die Datei *HelloServiceProcessor.java*.
5. Betrachte das *camel-context.xml* Dokument.
6. Erweitere das Dokument um zwei *Beans* und eine *Route*.

```
<beans ...>

  <bean id="helloServiceProcessor"
        class="de.predic8.camel.webservice.HelloServiceProcessor" />

  <cxf:cxfEndpoint id="helloServiceEndpoint"
        address="http://localhost:5555/services/hello"
        serviceClass="de.predic8.camel.webservice.HelloService"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="cxf:bean:helloServiceEndpoint" />
      <process ref="helloServiceProcessor" />
    </route>
  </camelContext>
</beans>
```

Listing 10.1: CXF Endpoint

7. Starte das Programm mit dem Maven Ziel *camel:run*.
8. Betrachte die *WSDL*:

```
http://localhost:5555/services/hello?wsdl
```

9. Rufe die Operation *sayHello* mit *Soap UI* auf.

Zusatzübung: Web Service aus WSDL ohne ServiceImpl

```
<route>
  <from
uri="cxf:http://localhost:7777/BlzService?wsdlURL=wsdl/blz.wsdl&am
p;dataFormat=MESSAGE&portName={http://thomas-
bayer.com/blz/}BLZServiceSOAP11port_http" />
    <to uri="file:data/blz" />
      <transform>
        <constant>
          OK
        </constant>
      </transform>
    </route>
```

11. Camel Test Kit

11.1. Routen testen

1. Importiere das Projekt `c:\kurs\kursdisk\CamelTestKit`.
2. Füge dem Projekt die folgenden Abhängigkeiten hinzu:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-test</artifactId>
    <version>3.5.0</version>
  </dependency>
  ...
</dependencies>
```

Listing 11.1: Test Kit Abhängigkeiten

3. Betrachte die Datei `SimpleRoute.java` und `SimpleTest.java`.
4. Führe Klasse `SimpleTest` als JUnit Test in IntelliJ aus.

Gibt es ein Problem? Was ist die Lösung?

11.2. Endpunkte wegmocken

1. Ändere die Testmethode in der *SimpleTest* Klasse wie folgt:

```
public void testRoute() throws Exception {
    context.adviceWith(context.getRouteDefinitions().get(0),
        new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
                interceptSendToEndpoint("file://target/outbox")
                    .skipSendToOriginalEndpoint()
                    .to("mock:outbox");
            }
        });

    MockEndpoint mock = getMockEndpoint("mock:outbox");
    mock.expectedMessageCount(1);

    template.sendBody("file://target/inbox", "Hello World");

    assertMockEndpointsSatisfied();
}
```

Listing 11.2: Mock Konfiguration

2. Führe den Test durch.

11.3. Weitere Erwartungen hinzufügen

1. Erweitere die Testmethode wie folgt:

```
MockEndpoint mock = getMockEndpoint("mock:outbox");

mock.expectedMessagesMatches (
    body().convertToString().contains("Hello World"),
    body().convertToString().isEqualTo("Hello You"));

mock.expectedMessageCount(2);

template.sendBody("file://target/inbox", "Hello World");
template.sendBody("file://target/inbox", "Hello You");

assertMockEndpointsSatisfied();
}
```

Listing 11.3: Weitere Erwartungen

2. Führe den Test durch.

Was ist das Problem?

11.4. Ersetzen der Input-Komponente

1. Erweitere die Klasse *SimpleTest* um folgende Methode:

```
@Override
protected CamelContext createCamelContext() throws Exception {
    CamelContext cc = super.createCamelContext();
    cc.addComponent("file", cc.getComponent("direct"));
    return cc;
}
```

Listing 11.4: Veränderter CamelContext

2. Führe den Test erneut aus.

12. Timer

12.1. Timer Komponente

1. Importiere das Maven-Projekt *CamelTimer*.

2. Betrachte die *PingProcessor* Klasse.

```
public class PingProcessor implements Processor {  
  
    public void process(Exchange exc) throws Exception {  
        System.out.println(  
            exc.getProperty(  
                Exchange.TIMER_FIRED_TIME)+": Ping!");  
    }  
  
}
```

Listing 12.1: PingProcessor

3. Erstelle eine Route in der *CamelTimer* Klasse:

```
RouteBuilder builder = new RouteBuilder() {  
    public void configure() {  
        from("timer:ping?period=1000")  
        .process(new PingProcessor());  
    }  
};
```

Listing 12.2: Periodische Route

4. Starte Camel und beobachte die Ausgabe.

```
Wed May 02 14:32:53 CEST 2012: Ping!  
Wed May 02 14:32:54 CEST 2012: Ping!  
Wed May 02 14:32:55 CEST 2012: Ping!  
Wed May 02 14:32:56 CEST 2012: Ping!  
Wed May 02 14:32:57 CEST 2012: Ping!
```

Listing 12.3: Ausgabe des PingProcessors.

13. Micronaut Monitoring

13.1. Monitoring im Projekt

1. Öffne *CamelREST*.
2. Füge dem Projekt die folgenden Abhängigkeiten hinzu:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-micrometer-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
  </dependency>
  ...
</dependencies>
```

Listing 13.1: Abhängigkeiten fürs Überwachen

3. Fügen dem Projekt in *application.properties* folgende Einstellungen hinzu:

```
server.port=8085
management.endpoints.web.exposure.include=health,info,prometheus
```

Listing 13.2: Einstellungen der *application.properties*

4. Starte das Projekt über die Klasse *MyApplication*.
5. Rufe <http://localhost:8085/actuator> auf.

14. Proxies und Bean Komponente

14.1. Proxies für Endpunkte erstellen

1. Importiere das Maven-Projekt *CamelProxyExample*.
2. Betrachte das *OrderService* Interface.

```
public interface OrderService {  
  
    public String order(String string);  
  
}
```

Listing 14.1: OrderService Interface

3. Betrachte die *Shop* Klasse.

```
public class Shop {  
    public String order(String order) {  
        System.out.println("Received: "+ order);  
        return "Accepted";  
    }  
  
}
```

Listing 14.2: Shop Klasse

4. Schicke programmatisch eine Nachricht an die Route, indem du die *main* Methode erweiterst.
Füge die folgenden Anweisung nach dem Aufruf der *ctx.start()* Methode hinzu.

```
ctx.start();  
System.out.println("Camel started.");
```

```
OrderService orderService = new ProxyBuilder(ctx)  
    .endpoint("direct:order")  
    .build(OrderService.class);
```

```
System.out.println("Antwort: "+  
orderService.order("Bestellung #1: 5 Pizzen für die IT"));
```

```
System.out.println("Press any key to stop camel.");
```

Listing 14.3: Route über Proxy Aufrufen

5. Beende und Starte Camel.

6. Betrachte die Ausgaben in der Konsole.

```
...  
Received: Bestellung #1: 5 Pizzen für die IT  
Antwort: Accepted  
...
```

14.2. Asynchrone Proxies

1. Baue eine Wartezeit in die *order* Methode ein.

```
public class Shop {  
  
    public String order(String order) throws Exception {  
        System.out.println("Received: "+ order);  
        Thread.sleep(10000);  
        return "Accepted";  
    }  
  
}
```

Listing 14.4: Order Methode mit Wartezeit

2. Stoppe und starte Camel.
3. Betrachte das Zeitverhalten in der Ausgabe.
4. Ändere das *OrderService* Interface wie folgt:

```
public interface OrderService {  
  
    public Future<String> order(String string);  
  
}
```

Listing 14.5: OrderService für asynchrone Aufrufe

5. Ändere den Aufruf der order Methode wie folgt:

```
Future<String> order = orderService.order("Bestellung #1: 5 Pizzen  
für die IT");  
System.out.println("Bestellung wurde verschickt.");  
System.out.println("Antwort: "+order.get());
```

Listing 14.6: Asynchroner Aufruf

6. Beende und starte Camel.
7. Betrachte das Zeitverhalten in der Ausgabe.

```
...  
Bestellung wurde verschickt.  
Received: Bestellung #1: 5 Pizzen für die IT  
Antwort: Accepted  
...
```

15. (Optional) Camel K

Um diese Übung durchführen zu können wird entweder ein Kubernetes Cluster benötigt auf dem der Teilnehmer administrative Berechtigungen besitzt, oder ein lokal auf dem System laufender Minikube.

Übung: Installiere Camel

Um Camel K verwenden zu können muss auf dem lokalen System Camel installiert sein. Zu Downloaden gibt es die Software unter: <https://github.com/apache/camel-k/releases>. Nach dem Download sollte die Software in den Path aufgenommen werden.

Anschließend kann

```
kamel install
```

Ausgeführt werden, dies installier camel k auf dem Cluster.

Übung: Schreiben einer simplen Route

Eine Route kann angelegt werden, indem wir lokal ein Datei namens *routes.groovy* erzeugen. Anschließend wird dort der folgende Code eingefügt.

```
from("timer:test")  
  .log("ping!")
```

Speichere die Datei ab. Die Route kann dann via

```
kamel run routes.groovy
```

gestartet werden.

Dies erzeugt eine integration Objekt im Cluster. Basierend auf diesem Objekt erzeugt der zuvor installierte camel-k-operator einen laufenden Container.