Carrying Enterprise on a Little Camel

(using Apache Camel, Apache ActiveMQ as applications to integrate systems and people)
Goals

• Show that Apache Camel, ActiveMQ can be used as stand-alone applications, in non-JVM environment.

• Show that Apache Camel, ActiveMQ can deliver a quick solution within hours. Without coding.

• Show a few examples of how common problems translate into message-centric data processing.
Application, not a framework

Application allows to:

• install, not to deploy,
• configure, not to code,
• fine-tune in production.

Framework requires:

• programming,
• dev tools,
• compilation.
Application, not a framework

Application allows to:

• install, not to deploy,
• configure, not to code,
• fine-tune in production.

Framework requires:

• programming,
• dev tools,
• compilation.
Application, not a framework

Application allows to:

- install, not to deploy,
- configure, not to code,
- fine-tune in production.

Framework requires:

- programming,
- dev tools,
- compilation.
Application, not a framework
Application, not a framework

Standalone executable (JAR)

Bundles

Dzmitry Pletnikau
Seville, Spain 2016
Application, not a framework

Standalone executable (JAR)

- can start from command-line;
- can run as container process;
- can run as Linux daemon;
- can run as Windows service.

Dzmitry Pletnikau
Seville, Spain 2016
Application, not a framework

If you are curious:

• Apache Karaf is OSGI container;
• OSGI is a specification of modular system for JVM;
• OSGI enables “hot” code swapping in JVM;
• OSGI enables “discovery” of components and services;
• Multiple instances of Apache Karaf can be managed with Apache Karaf Cellar;
• Apache Karaf has Spring, Blueprint, and other deployers: building OSGI bundles on-the-fly;
• Apache Karaf can be branded;
• Apache ServiceMix is branded Apache Karaf with Apache ActiveMQ, Camel and CXF already bundled with it.
Application, not a framework

How to start:

- Go to http://servicemix.apache.org/
- Download the default assembly
- Run ./bin/servicemix[.bat]
- Inside Karaf console, start watching logs: log:tail
  (or type help for actually helpful help)

How to start even faster:

- There are several ServiceMix images on Docker Hub:
  docker run -d -t --name smx -p 48101:8101 \
  -v /Users/dimitry/Desktop/deploy.slide9:/deploy \
  mkroli/servicemix
- Karaf console: ssh -p 48101 smx@127.0.0.1
Application, not a framework

ServiceMix is now running, but none of our configuration is present.
Configuration, not code

Apply configuration:

cp ./slides/timer.xml ./deploy/

What happens here:

• Apache Karaf recognizes new Blueprint file;
• Creates and deploys a bundle from it.
Configuration, not code

ServiceMix is now running, and our configuration is present.
Configuration, not code

What happens here:

- Apache Karaf recognizes Blueprint file has changed;
- Apache Karaf updates existing bundle;
- Old context and routes are shut down;
- New context and routes are started.
Speed in business

- **Blue**: acts every time she is 80% confident in the business idea.

- **Green**: acts only if she is 100% confident in the business idea.

- **Blue** acts x2 (twice) as often as green.

- They optimize their investments using Kelly criterion.
Speed in business

For most business, most tasks are not core competencies. These change at least once a year for most companies:

- logistics partners, warehouses;
- payment providers;
- banks;
- marketing channels.

Only few areas are stable enough to develop algorithms in-house.

Apache Camel is good for everything that changes fast.
Configuration, not code

- Apache Camel Components represent event-based side-effects existing in the real world.

- We configure the connections between them. With routes.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route autoStartup="true">
      <from uri="timer:test" />
      <to uri="log:test" />
    </route>
  </camelContext>
</blueprint>
```
“Well, if you think of Camel routes as highways, components are roughly analogous to on and off ramps.”

– Claus Ibsen or Jonathan Anstey
Configuration, not code

- Apache Camel supports >100 side-effects as Components;
- Provides up to 10,000 different two-lines configurations (from/to pairs).
Configuration, not code

- Apache Camel supports >100 side-effects as Components;
- Provides up to 10,000 different two-lines configurations (from/to pairs).

But not every combination is valid:

- some side-effects can only be produced;
- some side-effects can only be consumed;
- can’t build time-machine:

  ```xml
  <from uri="file:/tmp/events/"/>
  <to uri="timer:test"/>
  ```

  Can consume from file

  Can’t produce time change
Configuration, not code

Apache Camel is like Unix-pipes. To redirect program output to a file:

- using Unix pipes
  ```
  date | date.txt
  ```

- using Apache Camel:

  1. Consume time as trigger, once
  2. Produce new process, and
  3. consume process’s output
  4. Produce the file

- **install the exec component first in ServiceMix console:**
  ```
  feature:install camel-exec
  ```

- ```
  cat ./slides/pipes.xml
  ```

  ```xml
  <blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
      <route autoStartup="true">
        <from uri="timer:date?repeatCount=1"/>
        <to uri="exec:date"/>
        <to uri="file:./?fileName=date.txt"/>
      </route>
    </camelContext>
  </blueprint>
  ```
Configuration, not code

Apache Camel routes are versatile.

- trivially handle complex routing scenarios;
- make the power of >100 Components available.
Apache Camel routes are versatile.

```xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
    <property name="webhookUrl" value="https://hooks.slack.com/services/T04FQ2G81/G35A11AB4/zrfRNJC0MaVsxlidsRj8D817"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route autoStartup="true">
      <from uri="timer:live?period=30s" />
      <to uri="http://{{env:MYSERVICE_PORT_80_TCP_ADDR}}:{{env:MYSERVICE_PORT_80_TCP_PORT}}?throwExceptionOnFailure=false" />

      <choice>
        <when><simple>${headers.CamelHttpResponseCode} not in '200,201'</simple>

          <to uri="slack:@dimitry12?username=MyserviceWatcher" />
          <setHeader headerName="CamelAwsCwMetricName"><constant>MyserviceDown</constant></setHeader>
          <setHeader headerName="CamelAwsCwMetricUnit"><constant>Count</constant></setHeader>
          <setHeader headerName="CamelAwsCwMetricValue"><constant>1</constant></setHeader>
          <to uri="aws-cw://integration.unicity.com?accessKey=RAW(T04FQ2G81)\&secretKey=RAW(G35A11AB4)" />
        </when>
        </choice>
      </route>
    </camelContext>
  </blueprint>
```
Configuration, not code

Apache Camel routes are versatile.

✗ ssh -p 48101 smx@127.0.0.1
smx@root> feature:install camel-http camel-slack camel-aws

✗ cat ./deploy.slide9/service-watcher.xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
    <property name="webhookUrl" value="https://hooks.slack.com/services/T04FQ2G81/G35A11AB4/zrfRNJC0MaVsxlidsRj8D817"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route autoStartup="true">
      <from uri="timer:live?period=30s"/>
      <to uri="http:{{env:MYSERVICE_PORT_80_TCP_ADDR}}:{{env:MYSERVICE_PORT_80_TCP_PORT}}?throwExceptionOnFailure=false"/>

      <choice>
        <when><simple>${headers.CamelHttpResponseCode} not in '200,201'</simple>
        <to uri="slack:@dimitry12?username=MyserviceWatcher"/>

        <setHeader headerName="CamelAwsCwMetricName"><constant>MyserviceDown</constant></setHeader>
        <setHeader headerName="CamelAwsCwMetricUnit"><constant>Count</constant></setHeader>
        <setHeader headerName="CamelAwsCwMetricValue"><constant>1</constant></setHeader>
        <to uri="aws-cw://integration.unicity.com?accessKey=RAW(T04FQ2G81)\&secretKey="/>
        </when>
      </choice>
    </route>
  </camelContext>
</blueprint>

Trigger every 30 seconds
Apache Camel routes are versatile.

```
ssh -p 48101 smx@127.0.0.1
smx@root>feature:install camel-http camel-slack camel-aws
```

```
cat ./deploy.slide9/service-watcher.xml
<brp xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
    <property name="webhookUrl" value="https://hooks.slack.com/services/T04FQ2G81/G35A11AB4/zrfRNJC0MaVsxlidsRj8D817"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route autoStartup="true">
      <from uri="timer:live?period=30s" />
      <to uri="http://{{env:MYSERVICE_PORT_80_TCP_ADDR}}:{{env:MYSERVICE_PORT_80_TCP_PORT}}?throwExceptionOnFailure=false" />
      <choice>
        <when><simple>${headers.CamelHttpResponseCode} not in '200,201'</simple>
          <to uri="slack:@dimitry12?username=MyserviceWatcher" />
          <setHeader headerName="CamelAwsCwMetricName"><constant>MyserviceDown</constant></setHeader>
          <setHeader headerName="CamelAwsCwMetricUnit"><constant>Count</constant></setHeader>
          <setHeader headerName="CamelAwsCwMetricValue"><constant>1</constant></setHeader>
          <to uri="aws-cw://integration.unicity.com?accessKey=RAW(T04FQ2G81)&amp;secretKey=" />
        </when>
        </choice>
    </route>
  </camelContext>
</blueprint>
```
Apache Camel routes are versatile.

```
✗ ssh -p 48101 smx@127.0.0.1
smx@root>feature:install camel-http camel-slack camel-aws

✗ cat ./deploy.slide9/service-watcher.xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
        <property name="webhookUrl" value="https://hooks.slack.com/services/T04FQ2G81/G35A11AB4/zrfRNJC0MaVsxlidsRj8D817"/>
    </bean>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route autoStartup="true">
            <from uri="timer:live?period=30s" />
            <to uri="http://{{env:MYSERVICE_PORT_80_TCP_ADDR}}:{{env:MYSERVICE_PORT_80_TCP_PORT}}?throwExceptionOnFailure=false" />

            <choice>
                <when><simple>${headers.CamelHttpResponseCode} not in '200,201'</simple>
                <to uri="slack:@dimitry12?username=MyserviceWatcher"/>
                <setHeader headerName="CamelAwsCwMetricName"><constant>MyserviceDown</constant></setHeader>
                <setHeader headerName="CamelAwsCwMetricUnit"><constant>Count</constant></setHeader>
                <setHeader headerName="CamelAwsCwMetricValue"><constant>1</constant></setHeader>
                <to uri="aws-cw://integration.unicity.com?accessKey=RAW(T04FQ2G81)&amp;secretKey=RAW(G35A11AB4)"/>
            </when>
            </choice>
        </route>
    </camelContext>
</blueprint>
```

Trigger every 30 seconds
Make HTTP-request to the service
Report to Slack
-and-
to AWS CloudWatch
Configuration, not code

Apache Camel and its Components allow to only write code configuration for what’s unique about the problem at hand.

It often results in only a few lines of code configuration.

Benefits:

- fewer bugs;
- can eyeball code/configuration for correctness;
- can quickly experiment;
- self-documenting!
From problem to solution

1. Identify the source of “truth”, or “master” of data.
1. Identify the source of “truth”, or “master” of data.

2. Design the event-trigger around the source of “truth”.
From problem to solution

1. Identify the source of “truth”, or “master” of data.
2. Design the event-trigger around the source of “truth”.
3. Decide where you want the “truth” to propagate.

My service

AWS CloudWatch
From problem to solution

1. Identify the source of “truth”, or “master” of data.
2. Design the event-trigger around the source of “truth”.
3. Decide where you want the “truth” to propagate.
4. Connect with Apache Camel routes and Components.
Example: verify consistency

```xml
<route id="routeCheckNoother11ExigoOrdersUs" autoStartup="true" errorHandlerRef="dlc">
  <from uri="timer://routeCheckNoother11ExigoOrdersUsTimer?f"ixedRate=true&amp;period=60000&amp;delay=0" />

  <setBody>
    <![CDATA[
      select count(*) as records_num
      from Orders
      where
        other11 = '' and country in ('US') and orderstatusid > 6 and customerid <> 3 and
        ordertypeid <> 8
    ]]>"
  </setBody>

  <to uri="jdbc:{{live.Exigo.replicatedDb.connection_id}}" />

  <setHeader headerName="CamelAwsCwMetricName"><constant>ExigoOrdersNoOther11Us</constant></setHeader>
  <setHeader headerName="CamelAwsCwMetricUnit"><constant>Count</constant></setHeader>
  <setHeader headerName="CamelAwsCwMetricValue">
    <simple>${body.get(0).get('records_num')}</simple>
  </setHeader>

  <to uri="aws-cw://consistencychecks.integration.unicity.com?accessKey=RAW{key} &secretKey=RAW(key)" />
</route>
```
Example: serve Wordpress-assets from AWS S3

Each instance in a WordPress cluster also runs a ServiceMix Docker container.

Message-headers are often used to configure what component does

```xml
<route>
  <from uri="file:/var/www/unicity.com/unicityblog.wp/wp-content/uploads/?recursive=true&amp;delete=true"/>

  <setHeader headerName="CamelAwsS3Key">
    <simple>blog/uploads/${in.header.CamelFileName}</simple>
  </setHeader>

  <to uri="aws-s3://unicity.com?accessKey=RAW(key) & secretKey=RAW(key)"/>
</route>
```
Example: handle business-logic

1) If you already have a JVM class implementing your logic, use Camel’s Bean component. How to do it without writing Java-code?

2) Inline business-logic:

```xml
<route id="mobilehydra_log2queue">
    <from uri="file:///var/www/mh/application/logs/?delete=true&amp;include=log.*\.php"/>
    <setBody><javaScript>
        return request.body.substring(request.body.indexOf('\n')+1);
    </javaScript></setBody>
    <to uri="activemq://hydralog_queue"/>
</route>
```

3) Use external process:

```xml
<route id="makeInfotraxRequest" autoStartup="true" errorHandlerRef="dlc">
    <from uri="direct://makeInfotraxRequest_in" />
    <to uri="exec:php?args=local/php/map.php" />
    <setHeader headerName="Exchange.HTTP_URI">
        <simple>{{Infotrax.api.url}}${body}}</simple>
    </setHeader>
    <to uri="http://dummy"/>
</route>
```
Exactly-once message delivery

Apache Camel route connects point A with point B.
Exactly-once message delivery

Scenario:

• get cash from the buyer;
• succeed immediately;
• record to unstable and/or locking master-system later.
Exactly-once message delivery

Scenario:

• get cash from the buyer;
• succeed immediately;
• record to unstable and/or locking master-system later.

Apache ActiveMQ queue is a solution:

• persists messages/events on disk, surviving JVM crashes/restarts;
• delivers message exactly once;
• offers a reliable buffer between real-time publisher and intermittent subscriber.
Exactly-once delivery

<route>
    <from uri="file:/tmp/promises_paid" />
    <setHeader headerName="promise_id">${body}</simple></setHeader>

    <to uri="activemq://promises_paid" />
</route>

<route>
    <from uri="activemq://promises_paid" />
    <to uri="{{Datatrax_update_endpoint}}"/>  
    <setBody>
        <xpath resultType="java.lang.String">//struct/var[@name='errors']/string</xpath>
    </setBody>

    <choice>
        <when><simple>${body} == ""</simple>
            <stop/>
        </when>
    </choice>

    <setHeader headerName="AMQ_SCHEDULED_DELAY">3600000</constant>
    <to uri="activemq://promises_paid" />
</route>

✗ ssh -p 48101 smx@127.0.0.1
smx@root>feature:install camel-activemq
Exactly-once delivery

```xml
<route>
  <from uri="file:/tmp/promises_paid" />
  <setHeader headerName="promise_id"><simple>${body}</simple></setHeader>

  <to uri="activemq://promises_paid" />
</route>

<route>
  <from uri="activemq://promises_paid" />
  <to uri="{{Datatrax_update_endpoint}}"/>
  <setBody>
    <xpath resultType="java.lang.String">//struct/var[@name='errors']/string</xpath>
  </setBody>

  <choice>
    <when><simple>${body} == ""</simple>
      <stop/>
    </when>
  </choice>

  <setHeader headerName="AMQ_SCHEDULED_DELAY"><constant>3600000</constant></setHeader>

  <to uri="activemq://promises_paid" />
</route>

✗
ssh -p 48101 smx@127.0.0.1
smx@root>feature:install camel-activemq

Break the route using **local** ActiveMQ queue:
  - for persistence, and
  - delayed redelivery.
```
Over-the-network message delivery

Scenario:

- there is a network gap between where the event originates, and where it needs to propagate.
Over-the-network message delivery

Scenario:

• there is a network gap between where the event originates, and where it needs to propagate.

Apache ActiveMQ queue is a solution:

• can operate over network of brokers.
Over-the-network message delivery

ActiveMQ network-of-brokers can connect together isolated instances of Apache Camel. With all the same guarantees.

This topic is out-of-scope for this presentation. But star/hub-spoke topology is trivial to setup.

Network gap

Over-the-network message delivery

✗ ssh -p 48101 smx@127.0.0.1
smx@root> feature:install camel-activemq

<route>
  <from uri="file:/tmp/promises_paid" />
  <setHeader headerName="promise_id"><simple>${body}</simple></setHeader>
  <to uri="activemq://promises_paid"/>
</route>

<route>
  <from uri="activemq://promises_paid" />
  <to uri="{{Datatrax_update_endpoint}}"/>
  <setBody>
    <xpath resultType="java.lang.String">//struct/var[@name='errors']/string</xpath>
  </setBody>
  <choice>
    <when><simple>${body} == ""</simple>
      <stop/>
    </when>
  </choice>
  <setHeader headerName="AMQ_SCHEDULED_DELAY"><constant>3600000</constant></setHeader>
  <to uri="activemq://promises_paid"/>
</route>
Example: aggregating logs

```
<route id="mobilehydra_log2queue">
  <from uri="file://var/www/mh/application/logs/?delete=true&amp;include=log.*\..php"/>

  <setBody><javaScript>
    return request.body.substring(request.body.indexOf(\'\n\')+1);
  </javaScript></setBody>

  <to uri="activemq://hydralog_queue"/>
</route>

<route id="mobilehydra_queue2log" autoStartup="true">
  <from uri="activemq://hydralog_queue"/>

  <log message="${body}" logName="net.unicity.mobilehydra_log" loggingLevel="INFO"/>
</route>
```
Event-based paradigm

Big Data is only useful if you can express your business task as split/map/reduce-combination.

Route-based integrations only work if you can express your business process as sequence of messages: moving from one highly specialized piece of software to another.

Most business-process are not request-response: they are fire-and-forget instead.

You always need a trigger for a route. We most often use:

• time;

• file-system changes (is ultimately driven by timer);

• new messages available in ActiveMQ queue.
Event-based paradigm

File-system offers the easiest interface:

- we intercept HTTP traffic between Cobol systems using HTTPD mod_security and drop it on the disk;
Event-based paradigm

File-system offers the easiest interface:

- we intercept HTTP traffic between Cobol systems using HTTPD mod_security and drop it on the disk;
- wizard-like textual UI using Windows-batch script:

```bash
set /p id="Enter the payment ID: 

set FILENAME=voucher-crm-submit-%RANDOM%%RANDOM%%RANDOM%.txt
@echo %id% > C:\Datatrax\Applications\gates\%FILENAME%

:loop

if exist C:\Datatrax\Applications\gates\result-%FILENAME%.html (
    c:\windows\explorer.exe C:\Datatrax\Applications\gates\result-%FILENAME%.html
    goto :end
)
if exist C:\Datatrax\Applications\gates\error-%FILENAME%.txt (
    type C:\Datatrax\Applications\gates\error-%FILENAME%.txt
    set /p id=(Press Enter to close)
    goto :end
)

timeout 1 > NUL
goto :loop

:end
```
Event-based paradigm

File-system offers the easiest interface:

- we intercept HTTP traffic between Cobol systems using HTTPD mod_security and drop it on the disk;
- wizard-like textual UI using Windows-batch script:

```batch
set /p id="Enter the payment ID: "
set FILENAME=voucher-crm-submit-%RANDOM%%RANDOM%%RANDOM%.txt
@echo %id% > C:\Datatrax\Applications\gates\%FILENAME%

:loop
if exist C:\Datatrax\Applications\gates\result-%FILENAME%.html (  
    c:\windows\explorer.exe C:\Datatrax\Applications\gates\result-%FILENAME%.html  
    goto :end
)
if exist C:\Datatrax\Applications\gates\error-%FILENAME%.txt (  
    type C:\Datatrax\Applications\gates\error-%FILENAME%.txt  
    set /p id=(Press Enter to close)  
    goto :end
)
timeout 1 > NUL
goto :loop

:end
```

- users are trained to “drop” CSVs to trigger processing:

<!-- from uri="file:resubmit/Exigo2Infotrax-Order-Sync/?delete=true&amp;exclude=example.csv" /> -->
Event-based paradigm

HTTP offers second easiest (for us) interface. And supports **blocking** and request-response semantics.

Scenario:
- accounting system starts “end-of-day” processing;
- we need to capture some records while system is “frozen” and not accepting new entries;
- we need to process the records and submit to third-party over SFTP - it takes long time and we can’t keep system frozen.

Solution:
- use HTTP-interface to block the accounting system while we extract the data;
- “release” the accounting system waiting for HTTP-request to complete;
- continue with time-consuming submission process.
Event-based paradigm

Accounting system’s batch-file executed as part of end-of-day process:

```bash
... curl.exe http://localhost:8080/settlement_run ...
```

```
<route id="allOrders">
   <from uri="cxfrs:bean:rsServer?bindingStyle=SimpleConsumer" />
   <to uri="http://infoamr.xdbc/export_settlement" />
   <to uri="seda:data_ready?waitForTaskToComplete=Never" />
</route>

<route>
   <from uri="seda:data_ready" />
   <setHeader headerName="Exchange.HTTP_METHOD"><constant>POST</constant></setHeader>
   <setHeader headerName="Exchange.HTTP_URI">
   </setHeader>
   <to uri="http://dummy"/>
   <to uri="file:tmp_buzone?fileName=CFDI_UNICITYFAC.txt"/>
   <to uri="exec:zip?args=-c ./tmp_buzone/CFDI_UNICITYFAC.zip ./tmp_buzone/CFDI_UNICITYFAC.txt"/>
   <pollEnrich>
      <simple>file:camel/tmp_buzone?fileName=CFDI_UNICITYFAC.zip&amp;noop=true</simple>
   </pollEnrich>
   <to uri="sftp://UNICITYFAC@contenedor.buzone.com.mx/UNICITYFAC/xx/ENT?password=RAW(pass)"/>
</route>
```
Apache CXF

Apache CXF allows to expose Java objects and methods over HTTP using annotations.

```
<route id="allOrders">
  <from uri="cxfrs:bean:rsServer?bindingStyle=SimpleConsumer" />
  <to uri="http://infoamr.xdbc/export_settlement" />
  <to uri="seda:data_ready?waitForTaskToComplete=Never" />
</route>

<route>
  <from uri="seda:data_ready" />
  <setHeader headerName="Exchange.HTTP_METHOD"><constant>POST</constant></setHeader>
  <setHeader headerName="Exchange.HTTP_URI">
  </setHeader>
  <to uri="http://dummy"/>
  <to uri="file:tmp_buzone?fileName=CFDI_UNICITYFAC.txt"/>
  <to uri="exec:zip?args=-c ./tmp_buzone/CFDI_UNICITYFAC.zip ./tmp_buzone/CFDI_UNICITYFAC.txt"/>
  <pollEnrich>
    <simple>file:camel/tmp_buzone?fileName=CFDI_UNICITYFAC.zip&amp;noop=true</simple>
  </pollEnrich>
  <to uri="sftp://UNICITYFAC@contenedor.buzone.com.mx/UNICITYFAC/xx/ENT?password=RAW(pass)"/>
</route>
```
<route id="allOrders">
    <from uri="rest:get:/:settlement_run" />
    <to uri="http://infoamr.xdbc/export_settlement" />
    <to uri="seda:data_ready?waitForTaskToComplete=Never" />
</route>

$route>
    <from uri="seda:data_ready" />
    <setHeader headerName="Exchange.HTTP_METHOD"><constant>POST</constant></setHeader>
    <setHeader headerName="Exchange.HTTP_URI">
    </setHeader>
    <to uri="http://dummy"/>
    <to uri="file:tmp_buzone?fileName=CFDI_UNICITYFAC.txt"/>
    <to uri="exec:zip?args=-c ./tmp_buzone/CFDI_UNICITYFAC.zip ./tmp_buzone/CFDI_UNICITYFAC.txt"/>

    <pollEnrich>
        <simple>file:camel/tmp_buzone?fileName=CFDI_UNICITYFAC.zip&amp;noop=true</simple>
    </pollEnrich>
    <to uri="sftp://UNICITYFAC@contenedor.buzone.com.mx/UNICITYFAC/xx/ENT?password=RAW(pass)"/>
</route>
What Apache Camel is not good for?

Anything that’s unique to your business:

• Validation rules. XML-schemas can’t detect two incompatible coupons in the order. Inline logic gets unmanageable. We built in-house microservice.

• Mapping and data-transformation. XSLT and Velocity templates get unmanageable. We built in-house microservice. Check out Dozer: dozer.sourceforge.net.

• Resource-planning and calculation. We built in-house microservice. Check out Optaplanner: optaplanner.org.
What happens here?

```xml
<route>
  <from uri="file:/tmp/promises_paid" />
  <setHeader headerName="promise_id"><simple>${body}</simple></setHeader>

  <to uri="activemq://promises_paid" />
</route>
```

```xml
<route>
  <from uri="activemq://promises_paid" />

  <to uri="{{Datatrax_update_endpoint}}"/>

  <setBody>
    <xpath resultType="java.lang.String">//struct/var[@name='errors']/string</xpath>
  </setBody>

  <choice>
    <when><simple>${body} == ""</simple>
      <stop/>
    </when>
  </choice>

  <setHeader headerName="AMQ_SCHEDULED_DELAY"><constant>3600000</constant></setHeader>

  <to uri="activemq://promises_paid" />
</route>
```
“Do the right thing.”
Type Converters

What happens here?

Apache Camel TypeConverters:
- convert InputStream to Document
- convert NodeList to String
QA and Testing

Parameterized route

```
<route>
  <from uri="activemq://promises_paid" />
  <to uri="{{Datatrax_update_endpoint}}"/>
  <setBody>
    <xpath resultType="java.lang.String">//struct/var[@name='errors']/string</xpath>
  </setBody>
  <choice>
    <when><simple>${body} == ""</simple>
      <stop/>
    </when>
  </choice>
  <setHeader headerName="AMQ_SCHEDULED_DELAY"><constant>3600000</constant></setHeader>
  <to uri="activemq://promises_paid" />
</route>
```

CAT ./etc/register-promises.cfg

Datatrax_update_endpoint=http://172.17.48.112:8081/cgi-bin/runcbldt2.cgi
Parameterized route

```xml
<route>
  <from uri="activemq://promises_paid" />
  <to uri="{{Datatrax_update_endpoint}}"/>
  <setBody>
    <!-- Set the body with XPath expression -->
  </setBody>
  <choice>
    <when><simple>${body} == ""</simple>
      <stop/>
    </when>
  </choice>
  <setHeader headerName="AMQ_SCHEDULED_DELAY"><constant>3600000</constant></setHeader>
  <to uri="activemq://promises_paid" />
</route>

<cm:property-placeholder persistent-id="register-promises-dev" />
```
Mocking

Dzmitry Pletnikau
Seville, Spain 2016

Parameterized route

```xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0">
  <route>
    <from uri="activemq://promises_paid" />
    <to uri="{{Datatrax_update_endpoint}}"/>
    <setBody>
      <xpath resultType="java.lang.String">//struct/var[@name='errors']/string</xpath>
    </setBody>
    <choice>
      <when><simple>${body} == ""</simple>
        <stop/>
      </when>
    </choice>
    <setHeader headerName="AMQ_SCHEDULED_DELAY"><constant>3600000</constant></setHeader>
    <to uri="activemq://promises_paid" />
  </route>
  <cm:property-placeholder persistent-id="register-promises-mock" />
</blueprint>
```
Development life-cycle

We saw how we can easily:

• prototype;
• quickly iterate;
• switch between environments;
• mock external systems.

Once the route is ready to be used in production:

• non-engineers have to manage and monitor it;
• non-engineers should be alerted about anomalies and be able to fix the problem.
Human interface

Apache Camel exposes vital statistics about routes

jconsole - part of JDK (not JRE)
Human interface

Allows to manage routes (start/stop and more)
Questions?

Useful resources:

• https://camel.apache.org/components.html
• http://mail-archives.apache.org/mod_mbox/camel-users/

Contact:

• @spring_stream
• dzmitry@pletnikau.com