# Dead Simple Integration with Apache Camel

Aaron Mulder
Chariot Solutions

CHARIOT SOLUTIONS

# Agenda

- Introducing Camel
- Camel Concepts
- Using Camel
- Sample Integration Scenarios with Camel
- Camel Concerns
- Camel Deployments
- Camel Versions

# Introducing Camel

# Camel Basics

- Common code for many integration needs
  - Routing & transformation
  - Addresses many of the standard Enterprise Integration Patterns
  - Supports many transports/protocols
  - Easy integration with Spring and other friends...
- Not really a standalone product
  - More a big set of tools and glue
  - Lightweight and easy to leverage

# Using Camel

- Set up protocol endpoints, transformers, routing rules, bean invocations, etc.
- Define all this in a Spring config file, or a Java (or Scala) DSL, or to some extent using annotations
- Run Camel as part of a Spring application, or an OSGi bundle, or a ServiceMix application, or part of an ActiveMQ broker, or... or... or...
- Pretty lightweight and simple way to add integration to an application

# However…

- Camel is not an ESB
  - No formal “bus” definition
  - No explicit persistence of in-flight messages
  - No deployment model
  - Not based on JBI or other standards
  - No bundled admin tools (coming in next version)
- It's really the lightweight integration solution
  - Handles messaging, files, HTTP services, bean invocations, scheduler, etc., etc., etc.

# With Routing/Transformation

- Can construct routing expressions with various languages (scripting, XPath, etc.)
- Can split/merge messages
- Can do pipelines or parallel execution
- Can apply transformers, either automatic (e.g. body XML to POJOs using JAXB) or manual (calling a bean method to transform a message)

- ...But does not store state like BPEL and friends

# Camel Concepts

# Key Concepts

- A *CamelContext* handles a set of routes
- The routes are defined in XML or a Java* DSL
- They use various *Endpoints* for input and output
  - Which are generated by various *Components*
- They may use expression languages, scripting languages, templating languages, etc.
- The routes are applied to *Exchanges* (composed of in and out *Messages*)
- Messages use various *Data Formats*

# The CamelContext

- The unit of Camel configuration, holding one or more *Routes*
- May be (with some effort) run standalone, or defined in e.g. a Spring or ActiveMQ config file or a ServiceMix service unit or OSGi bundle
- Can be started or stopped as a whole, but individual elements within it generally can't be
- All the Camel logic is executed in the JVM that runs the CamelContext

# Routes

- A route defines the path between one or more *Endpoints*
  - The route may include input, output, and various decisions or processing along the way
    - e.g. Take a JMS message off a certain queue, unmarshall the XML body to Java Beans, pass them to a POJO method selected based on some message header, and generate a reply message from the return value
- Routes may be defined in XML or in Java code (now also a Scala DSL option)
  - Route definitions are processed once when the CamelContext starts

# Components & Endpoints

- A component is a factory for *endpoints*
  - The JMS component manages connectivity to a JMS broker and produces topic/queue endpoints
  - The Bean component manages invoking POJOs, and its endpoints are beans or bean methods
- An endpoint is used in a route (typically the “from” or “to” part of the route)
  - It creates and/or consumes *Message Exchanges*
  - Create an exchange from a message received from Endpoint A, do something, and send the result to Endpoint B

# Endpoint URIs

- An endpoint is defined by a URI
  - Starting with a scheme that identifies which component should create the endpoint
    - activemq:topic:MyTopic
    - bean:SomeSpringBean?methodName=callMe
- Each component includes a default configuration registered under a default scheme (e.g. **bean:**)
- You can also create custom definitions of a component with custom settings, overriding the default scheme or using any other scheme

# Messages & Exchanges

- An Exchange contains an input message, and possibly an output message, an exception, etc.
- A message is a generic concept, and has headers, a body, attachments, etc.
- There are specific implementations for JMS, Files, JMX, etc.
- The message body can be converted between various data formats (XML, Java Beans, File, etc.)

# Languages

- Camel supports many languages for helping to define routes
  - Expression language (like JSP EL)
  - Scripting Languages (Groovy, Python, Ruby, etc.)
  - XPath, XQuery
  - OGNL/JXPath
  - ...
- Useful for expressing conditions (header 'Foo' = 'Bar', body contains 'Baz', etc.)

# Data Formats

- If a message uses a known Data Format (XML, CSV, etc.), then you can use canned transformers on the message body
  - e.g. Convert XML to JavaBeans using JAXB or XMLBeans
  - Then endpoints further along can just expect the new format
    - e.g. a JavaBean method whose argument is a JAXB bean rather than a String or XML document

# Using Camel

# Camel in a Spring Application

- Start by adding the right libraries
  - camel-core-1.6.0.jar and other camel-* JARs as needed (JMS, scripting, etc.)
  - perhaps activemq-core-5.1.0.jar
- Add a <camelContext> element to the Spring config file
  - Uses the Spring 2.x XML syntax to avoid loads of <bean> definitions
- Either define routes in the <camelContext> or point to Java DSL classes

# Camel Logging

- Many problems only manifest in log output
- Camel uses commons logging
  - Defaults to Log4J
- Be sure to add a `log4j.properties` if your project does not already contain one!
  - Trust me, you will want to see the errors when messages aren't delivered properly for some reason

# Note on the DSL

- Route definitions in the DSL are processed *once* at startup
  - Same as XML route definitions – no special abilities just because it's Java code
  - In particular, you can't put dynamic logic in there that should be executed every time the route is executed
  - Instead, put that logic in a bean and have the route invoke the bean
    - Use an endpoint URI like `bean:foo?methodName=callMe` where “foo” is the name of a Spring bean

# Typical Maven POM with Camel

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
    ...
    <dependencies>
        <dependency> <!-- brings camel-core too -->
            <artifactId>camel-jms</artifactId>
            <groupId>org.apache.camel</groupId>
            <version>1.6.0</version>
        </dependency>
        <dependency> <!-- if you're using ActiveMQ -->
            <artifactId>activemq-core</artifactId>
            <groupId>org.apache.activemq</groupId>
            <version>5.2.0</version>
        </dependency>
        <!-- Also maybe camel-script, camel-juel,
             camel-jxpath, camel-ognl, camel-groovy, etc. -->
        ...
```

# Spring Configuration with Camel

```
<beans ...>
  <camelContext id="camel"
       xmlns="http://activemq.apache.org/camel/schema/spring">

    <!-- A package with Java DSL classes defining routes -->
    <package>com.something.camel.dsl</package>

    <!-- A route defined right here in XML -->
    <route>
      <from uri="activemq:queue:InputQueue" />
      <to uri="activemq:topic:OutputTopic" />
    </route>

  </camelContext>
</beans>
```

# Java DSL Route Definition

```
public class MyRoutes extends RouteBuilder {
    public void configure() throws Exception {
        // One Route
        from("activemq:topic:DSL.TestTopic")
            .to("activemq:queue:DSL.TestQueue");
        // Another Route
        from("activemq:topic:DSL.OtherTopic")
            .choice()
                .when().el("${in.header.foo == 'Foo'}")
                    .to("activemq:queue:DSL.FooQueue")
                .when().groovy("in.header.foo == 'Bar'")
                    .to("activemq:queue:DSL.BarQueue")
                .when().ognl("['in'].header.foo == 'Baz'")
                    .to("activemq:queue:DSL.BazQueue");
    }
}
```

# Another Option: Annotated Beans

- The @MessageDriven annotation means you can set up a bean to receive messages without defining an explicit route
  - But no fancy abilities like transformation or filtering
    - Though Camel respects the JMSReplyTo header for JMS
- You can use all the other annotations on beans named explicitly in routes (as well as @MessageDriven beans)
  - For annotating parameters to bean methods to bring in the message body, headers, and expressions, or to set headers on the reply message

# Bean Annotations with Camel

```
public class MyBean {

    @MessageDriven(uri = "activemq:topic:AnnotationTest")
    // Assumes JMSReplyTo is set on the incoming messages
    public String doSomethingAndReply(@Body String body,
            @EL("${in.headers.cost > 1000}") boolean valuable,
            @OutHeaders Map responseHeaders)  {

        String outputMessage = ...;
        responseHeaders.put("valuable", valuable);
        return outputMessage;
    }
}
```

# Camel Capabilities

- Sending messages between endpoints
- Content-based routing
  - Headers easiest, body content possible too
- Filtering
- Transformation (between data formats, or with custom code or templates)
- Manipulating the message flow (aggregating, resequencing, etc.)

# Camel Components

- A long list, including:
  - JMS
  - JavaBeans / Spring Beans
  - CXF (Web Services)
  - Files
  - HTTP/FTP/SMTP
  - Quartz scheduler
  - SQL/JPA
  - Velocity/XSLT

# Working with Routes

- May need to customize certain components (enable transactions on JMS, etc.)
- Figure out what the route should look like, with integration patterns, various beans, expressions and transformations, etc.
- Write a unit test?  (!!)
- Figure out how to express the route in XML or DSL
- Code the expressions, beans, templates, etc.

# Sample Integration Scenarios with Camel

CHARIOT SOLUTIONS

# Shared E-mail Service

- One of the services on an SOA project:
  - Client makes Web Service call with to, from, subject, body, HTML body, etc.
  - Service validates parameters, and returns success or error to the caller
  - Maps arguments to an e-mail message
  - Calls a third-party e-mail provider (XML over HTTP?)
    - Retries if necessary
  - Handles the response from the e-mail provider

# Modeling the E-mail Service

- Part 1
  - CXF accepts client call
  - Camel routes to a JavaBean to validate arguments
  - Camel sends a JMS message to a queue with message arguments
  - Camel replies to the sender
  - When complete, caller is finished, message with e-mail data is on a queue

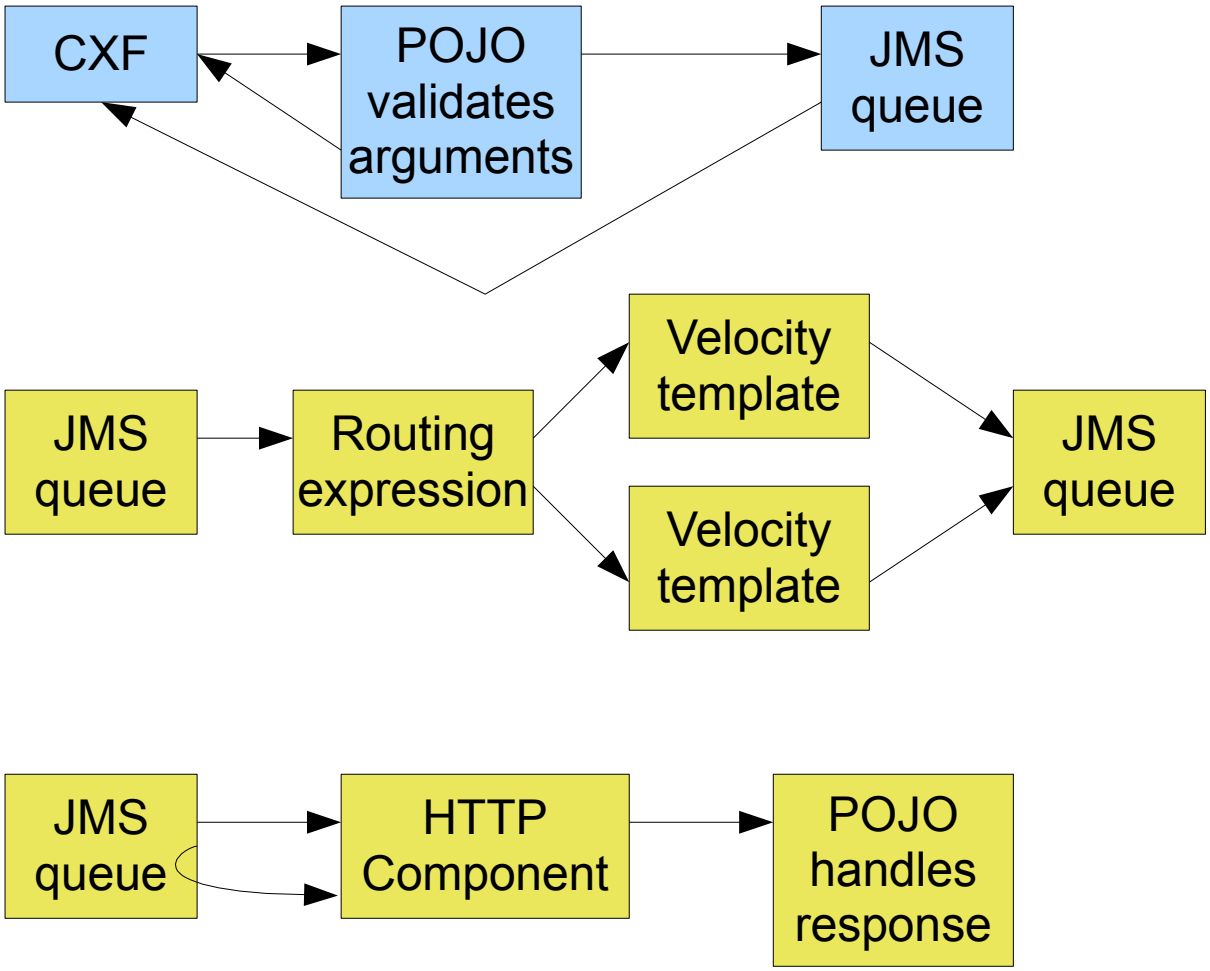
# Modeling the E-mail Service

- Part 2
  - Camel reads e-mail with message data off the queue
  - Camel sends to a Velocity template, which formats the e-mail and puts it into the XML form used for the third-party e-mail service
    - Selecting a template based on whether an HTML body is present
  - The finished e-mail is posted to another queue
- This could be combined with the previous bit if you don't mind holding up the client to get the message formatted

# Modeling the E-mail Service

- Part 3
  - Camel reads the finished message from the queue using a transactional receive
  - Camel calls the third party e-mail service using the HTTP component
  - On a failure, the receive is rolled back and retried
  - On success, response is passed to POJO (if needed)
  - Need to customize the JMS component here to enable transactions
  - Use an error handler to set a custom retry duration (likely, a few minutes)

# E-mail Service in Camel



# E-mail Configuration Excerpt

```
public class MyRoutes extends RouteBuilder {
    public void configure() throws Exception {
        // Last 2/3 of the configuration from previous slide
        from("activemq:queue:IncomingMessages")
            .choice()
                .when().el("${in.header.html}")
                    .to("velocity:templates/HTMLMessage")
                    .to("activemq:queue:OutgoingMessages")
                .when().el("${!in.header.html}")
                    .to("velocity:templates/TextMessage")
                    .to("activemq:queue:OutgoingMessages");
        from("jmswithtx:queue:OutgoingMessages")
            .to("http://mailservice.com/sendMail")
            .to("bean:handleMailServiceResponse");
    }
}
```

CHARIOT SOLUTIONS

# Messaging Configuration Excerpt

```
<beans ...>
  <camelContext id="camel">
    <!-- Specify DSL packages here -->
  </camelConxtext>

  <bean name="jmxwithtx"
class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://foo.com:61616" />
    <property name="transacted" value="true" />
    <property name="transactionManager" ref="tm" />
  </bean>
</beans>
```

# Multiple Business Partners

- From an integration project:
  - Data received from multiple business partners in varying ways
    - XML over FTP, CSV over HTTP, XLS over e-mail
  - Data must be converted to canonical format and posted to a JMS queue
    - Gets sent to an accounting system
  - A JMS response must be received, formatted, and e-mailed to the partner

# Modeling the Business Partners

- Step 1: receive data
  - Set up a route for each partner
    - Use the File component to look for FTP files
    - Use the Jetty component to receive HTTP files
    - Use the Mail component to receive unread messages from a POP or IMAP mailbox
  - The route should send the data to the next step for format conversions

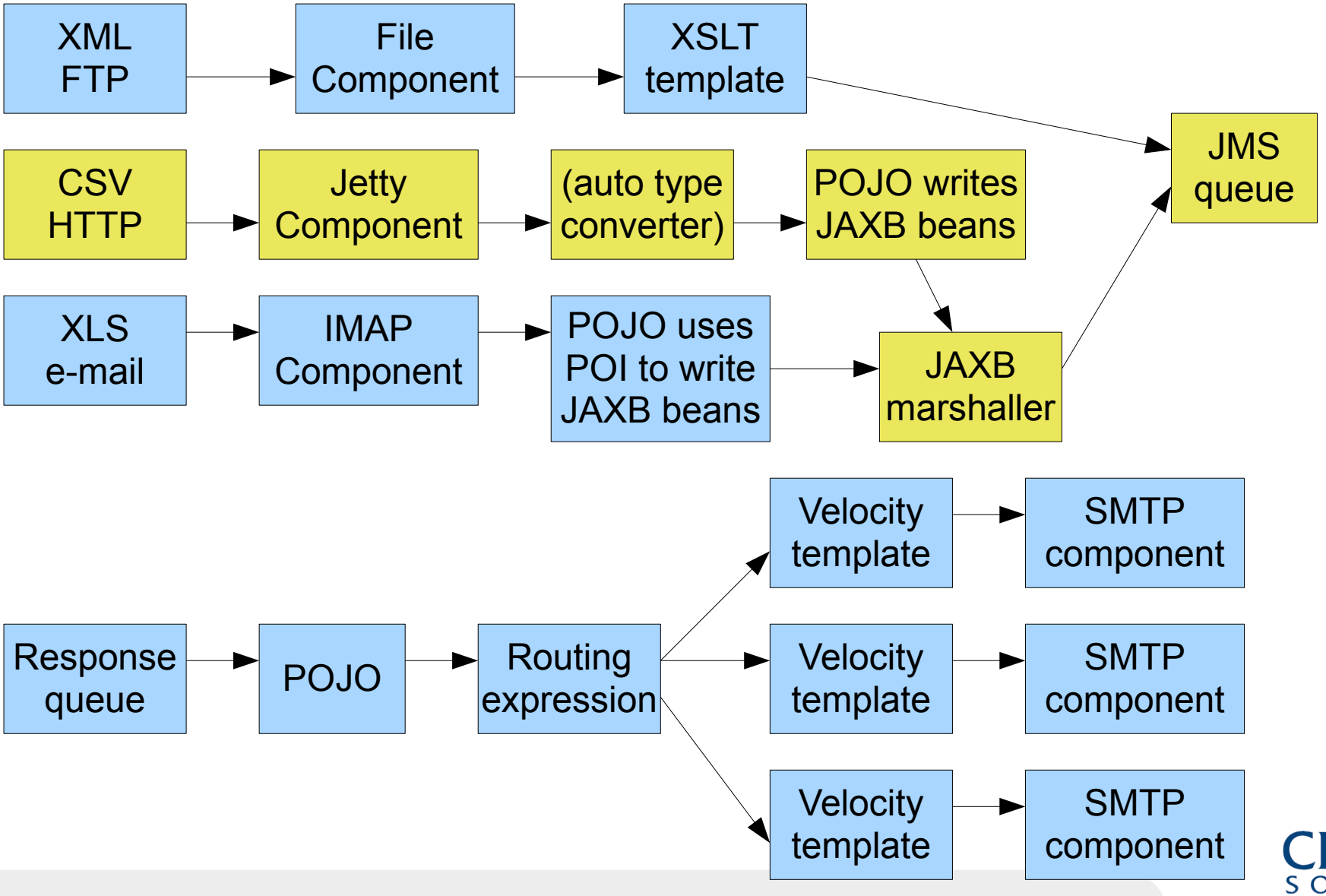
# Modeling the Business Partners

- Step 2: format data
  - Define the canonical format as a set of JAXB POJOs mapping to the canonical XML format
  - Set up a route for each partner, with a JavaBean that converts the input format to the JAXB POJOs
    - Camel can autoconvert CSV to Java objects
    - Could use POI to process the XLS
  - Use the automatic JAXB type converter to turn the POJOs into an XML document
  - Or just use XSLT to convert XML input

# Modeling the Business Partners

- Step 3: Deal with accounting system
  - The message is now in XML format, easy to send to a queue
  - Set up another route to handle responses from the response queue
    - Send it to a bean to do the processing and extract any needed parameters, including a message header for the destination partner
    - Route to a different path based on the partner
    - Then to a Velocity template to format the response e-mail
    - Then to an SMTP component to send the mail

# Business Partners in Camel




CHARIOT SOLUTIONS

# Partners Configuration Excerpt

```
<beans ...>
  <camelContext id="camel"
       xmlns="http://activemq.apache.org/camel/schema/spring">
    <!-- The route for CSV Input -->
    <route>
      <from uri="jetty:http://localhost:5000/CSVInput"/>
      <unmarshal><csv/></unmarshal>
      <to uri="bean:CSVConverter"/>
      <marshal><jaxb prettyPrint="true" /></marshal>
      <to uri="activemq:queue:ToAccounting"/>
    </route>
  </camelConxtext>

  <bean name="CSVConverter" class="com.xyz.CSVConverterBean"/>
</beans>
```

# Testing the CSV Route

```
@ContextConfiguration(locations = "/CSVInputTest-context.xml")
public class ExcelInputTest extends AbstractJUnit38SpringContextTests {
    @Autowired
    protected CamelContext camelContext;
    protected ProducerTemplate<Exchange> template;

    protected void setUp() throws Exception {
        template = camelContext.createProducerTemplate();
    }

    public void testCSVConversion() throws InterruptedException {
        MockEndpoint finish = MockEndpoint.resolve(camelContext, "mock:finish");
        finish.setExpectedMessageCount(1);
        InputStream in = ExcelInputTest.class.getResourceAsStream("/input.csv");
        assertNotNull(in);
        template.sendBody("direct:CSVstart", in);
        MockEndpoint.assertIsSatisfied(camelContext);
        assertEquals(Invoice.class, finish.getExchanges().get(0).getIn().getBody
().getClass());
    }
}
```

CHARIOT SOLUTIONS

# Camel Concerns

# Crash & Recovery

- If the JVM goes down, any exchanges in process are lost
  - A transaction manager can recover in-flight transactions
  - However, only a few of the components support transactions (though they use the Spring TX Manager, so e.g. a Spring Bean could hook in)
- Could put e.g. persistent JMS queues or DB calls between steps in a process?
- Could actually use interceptors to store state of in-flight exchanges?

# Deployment

- Camel itself does not support deployment, hot deployment, or redeployment
  - A CamelContext can be started or stopped, but it has to be something outside of Camel that does it
- Spring does not have a deployment model either
- May look to Spring DM, ServiceMix 3, ServiceMix 4 Kernel + OSGi, or a Spring app deployed in Tomcat or some other container

# Camel Deployments

# Deploying Camel Routes

- Due to the deploy/redeploy issue, Camel routes should typically be deployed in some other container for production
  - ServiceMix
  - Spring DM
  - Spring + Tomcat
  - etc.
- All of these can take a Spring config file as the configuration format

# Testing Camel Routes

- The same Spring config file can be used in unit tests
- May want to separate component definitions from routes so that tests can use different component configurations (e.g. in-VM ActiveMQ)
- Also extensive support for mock destinations for testing, though it may be harder to use the exact route definitions that way
  - But remember, you can redefine the activemq: or other scheme to use the mock component!

# Building Camel App Modules

- Extensive Maven support
  - camel-maven-plugin can run artifacts in Camel, as well as generating visualizations of the routes as part of the Maven docs
  - maven-bundle-plugin can build OSGi bundles, including bundles holding Camel routes
  - jbi-maven-plugin can build JBI service units and service assemblies, including Camel service units
- But all of these formats are zip/jar based at heart, so the regular Maven JAR build works fine too

# Camel Versions

# Camel Releases

- 1.6.0 is the current release, with significant improvements over the early 1.x releases
  - It's built against Spring 2.5
- 2.0 is the next upcoming Camel release
- ActiveMQ 5.2.0 includes Camel 1.5.0 but you can run it fine with Camel 1.6.0 instead
- ServiceMix 3.3 includes Camel 1.4.0
- The ServiceMix 4.0 kernel can run Camel OSGi bundles

# Camel Libraries

- Even the packages that “include” Camel may only include camel-core and maybe camel-jms
- Be prepared to add other Camel JARs for scripting support, various components like the scheduler, test base classes, etc.
- Those Camel JARs depend on third-party JARs for the actual script engines, scheduler services, etc.
  - Though Maven takes care of this for you

# Bundling Camel

- Camel is still pretty lightweight
  - camel-core-1.6.0 is under 1 MB
  - camel-core and all 40+ other camel-* JARs are around 2 MB total
- But ActiveMQ and Quartz and scripting engines and so on will potentially add a noticeable amount
- Still, much lighter than your average ESB or integration server product

# Q&A

CHARIOT SOLUTIONS