Navigating Options

Components

Patterns

Common Use Cases

Lifecycle

Develop

Test

Deploy

Design

Legacy Apps?

Q/A

[obligatory camel pics]

Camel Intro

Why Use Camel

Apache Camel Business Patterns

[my notes on Camel]

presented by Ben O'Day

Apache Camel

Prezi

Apache Camel Business Patterns

[my notes on Camel]

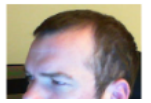
presented by Ben O'Day

presentation goals...

- quick intro to Camel
- when to use it

my background...

- IT Consultant from San Diego, CA
- 15 years of consulting



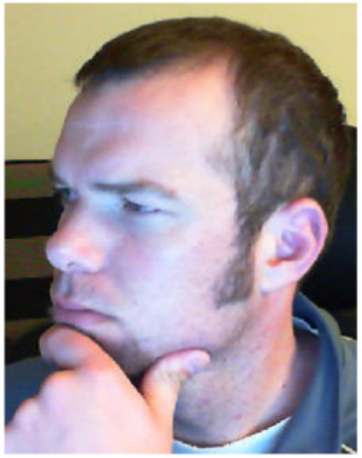
presentation goals...

- quick intro to Camel
- when to use it
- how to navigate options
  - components, patterns
  - common use cases
- SDLC with Camel
- refactoring legacy apps

- overall
  - how to use Camel to address common business problems

*side note - a PDF is available if you are not a Prezi fan: http://bit.ly/1qltYsx

my background...



- IT Consultant from San Diego, CA
- 15 years of consulting
- 5+ years with Apache technologies
  - Camel (committer), ActiveMQ, CXF, Karaf

why I use Camel...

- it simplifies design/dev...
  - common integration issues -> EIPs
  - using common technologies -> components

- it promotes...
  - patterns and best practices
  - abstracting tech details to focus on logic
  - visualizing business logic -> routes

Navigating Options

Components

Patterns

Common Use Cases

Lifecycle

Develop

Test

Deploy

Design

Legacy Apps?

Q/A

[obligatory camel pics]

Camel Intro

Why Use Camel

Apache Camel Business Patterns

[my notes on Camel]

presented by Ben O'Day

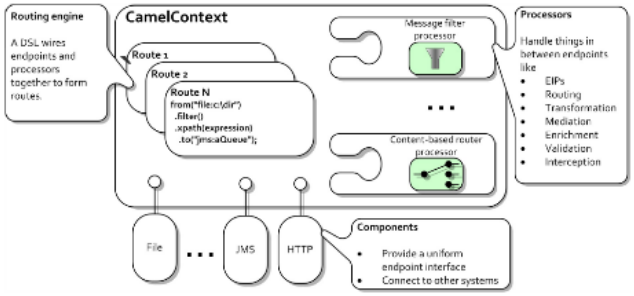
Apache Camel

Prezi

Camel Intro

Executive Summary

- Camel is...
  - Apache 2.0 Licensed
  - an open source message routing framework
  - a large project
    - 100+ components, 50+ EIP implementations
    - 1000s of configurations
  - backed by a large developer community
  - designed to integrate with other Apache projects
    - ActiveMQ, CXF, Karaf, ServiceMix
- Camel is not...
  - an ESB (see Servicemix, Mule)
  - a message broker (see ActiveMQ, RabbitMQ)
  - a runtime container (see Tomcat, Karaf)

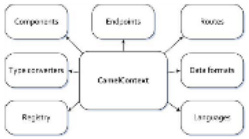


Camel Overview - Getting Started

- first steps...
  - read Camel In Action
  - review articles and blogs
  - download the source code
    - github.com/apache/camel
  - discussion forums
    - camel users, stackoverflow
- learn related technologies...
  - Maven, Spring, JMS, EIPs
- build a demo app..
  - in ActiveMQ (ships with Camel)
    - add routes to camel.xml and start AMQ
  - create a new project -> maven archetype
    - build some unit tests
    - run app
      - maven camel plugin (jar)
      - maven jetty plugin (war)

CamelContext -> Routes

- CamelContext
  - Routes
    - Endpoints
    - Processors



- basic example

```
CamelContext context = new DefaultCamelContext();

context.addRoutes(new RouteBuilder() {
   public void configure() {
      from("file://inbound").to("file://outbound");
   }
});

camelContext.start();
```

Camel Building Blocks...

- Exchange (ID, Properties, In Msg, Out Msg)

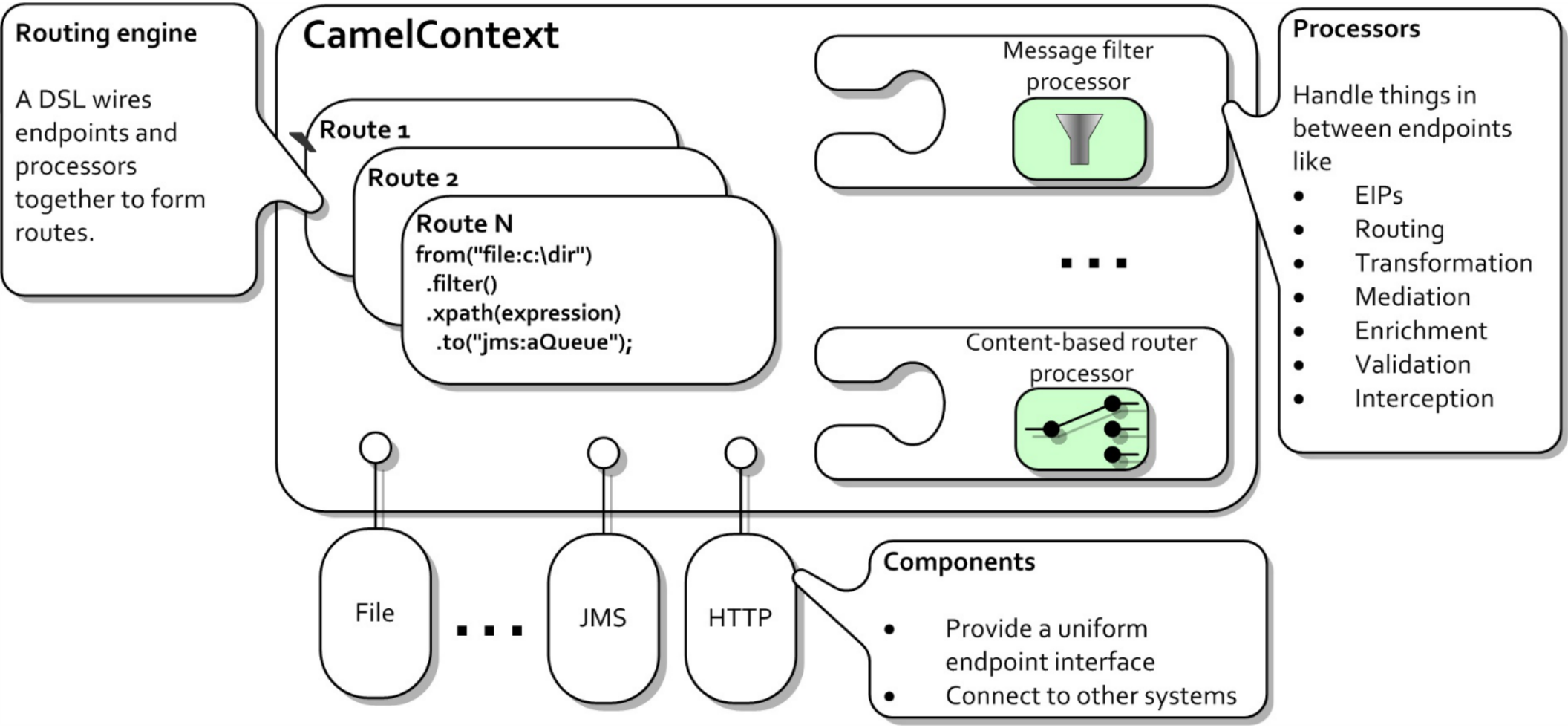
- routes -> DSLs
  - from(...).to(...);
  - consumers/producers

Java DSL - A Java based DSL using the fluent builder style.
Spring XML - A XML based DSL in Spring XML files
Blueprint XML - A XML based DSL in OSGi Blueprint XML files
Groovy DSL - A Groovy based DSL using Groovy programming
Scala DSL - A Scala based DSL using Scala programming
Annotation DSL - Use annotations in Java beans.

- EIPs
  - route enabled integration patterns (filter, router, aggregator)
- components
  - route enabled technologies (file, http, jms, ftp, cxf, hdfs)

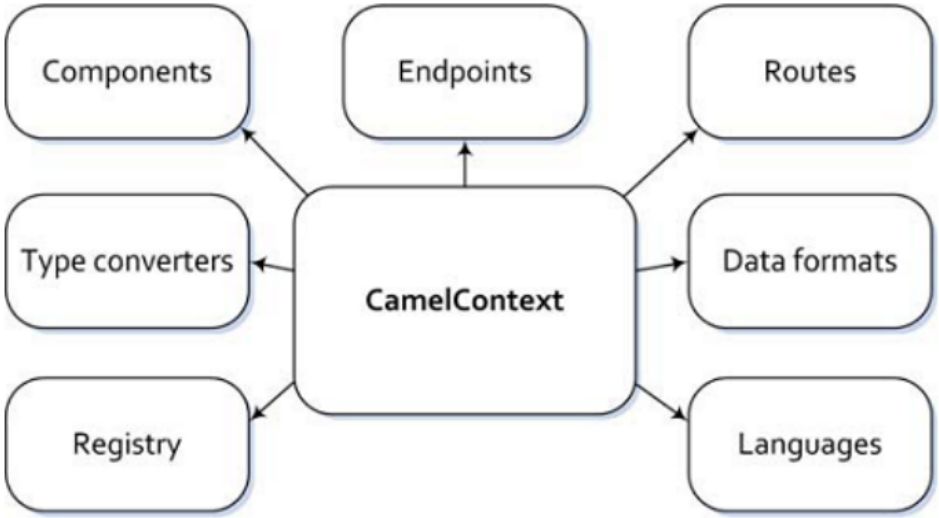
Executive Summary

- Camel is...
  - Apache 2.0 Licensed
  - an open source message routing framework
  - a large project
    - 100+ components, 50+ EIP implementations
    - 1000s of configurations
  - backed by a large developer community
  - designed to integrate with other Apache projects
    - ActiveMQ, CXF, Karaf, ServiceMix
- Camel is not...
  - an ESB (see Servicemix, Mule)
  - a message broker (see ActiveMQ, RabbitMQ)
  - a runtime container (see Tomcat, Karaf)


Routing engine
A DSL wires endpoints and processors together to form routes.
CamelContext
Route 1
Route 2
Route N
from("file:c:\dir")
.filter()
.xpath(expression)
.to("jms:aQueue");
Message filter processor
Content-based router processor
Processors
Handle things in between endpoints like
EIPs
Routing
Transformation
Mediation
Enrichment
Validation
Interception
File
JMS
HTTP
Components
Provide a uniform endpoint interface
Connect to other systems

CamelContext -> Routes

- CamelContext
  - Routes
    - Endpoints
    - Processors

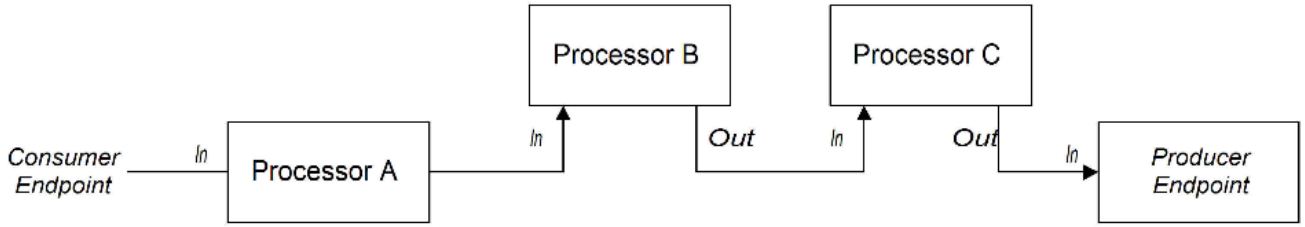


- basic example

```
CamelContext context = new DefaultCamelContext();

context.addRoutes(new RouteBuilder() {
    public void configure() {
        from("file://inbound").to("file://outbound");
    }
});

camelContext.start();
```

Camel Building Blocks...

- Exchange (ID, Properties, In Msg, Out Msg)



- routes -> DSLs
  - from(...).to(...);
  - consumers/producers

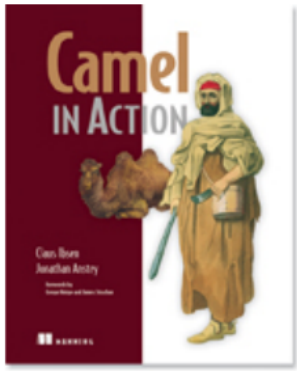
Java DSL - A Java based DSL using the fluent builder style.
Spring XML - A XML based DSL in Spring XML files
Blueprint XML - A XML based DSL in OSGi Blueprint XML files
Groovy DSL - A Groovy based DSL using Groovy programmin
Scala DSL - A Scala based DSL using Scala programming lang
Annotation DSL - Use annotations in Java beans.

- EIPs
  - route enabled integration patterns (filter, router, aggregator)

- components
  - route enabled technologies (file, http, jms, ftp, cxf, hdfs)

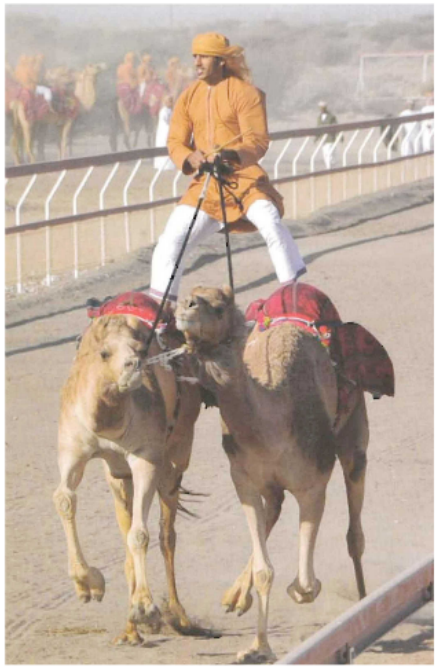
Camel Overview - Getting Started

- first steps...
  - read Camel In Action
  - review articles and blogs
  - download the source code
    - github.com/apache/camel
  - discussion forums
    - camel users, stackoverflow



- learn related technologies...
  - Maven, Spring, JMS, EIPs

- build a demo app..
  - in ActiveMQ (ships with Camel)
    - add routes to camel.xml and start AMQ
  - create a new project -> maven archetype
    - build some unit tests
    - run app
      - maven camel plugin (jar)
      - maven jetty plugin (war)



Why Use Camel

what problems are relevant?

- technology challenges...
  - complex routing
  - technology integrations
  - orchestrating services
- types of requirements...
  - highly event based processes
    - events | messaging
  - exposing multiple endpoints
    - http/jms/file => process
  - complex runtime management
    - auto/manual flow control
    - exception handling

be ready to sell it...

- as with any new technology...
  - choose your battles wisely
  - don't add unnecessary complexity
  - don't use for just one use case
- to management...
  - create a high level design, discuss alternatives
  - position Camel
    - as an integration technology
    - as a lightweight framework
    - increases productivity -> reuse/abstraction
- to technical team...
  - create a POC and demo it
  - highlight code reduction, readability, testability
  - DRY, convention over configuration

alternatives frameworks

- Spring Integration, Mule ESB
- similarities
  - all open source
  - support EIPs, routing, components
  - IDE support, relatively easy deployment
- differences
  - Spring Integration
    - has a narrower component scope
  - Mule
    - CPAL license (more restrictive)
    - has good proprietary components (SAP, Tibco, Seibel, etc)
  - Camel
    - more flexible DSL than Mule (XML, Java, Groovy, Scala)
    - more components than both
    - strong dev community
    - integrates with Apache projects

*for more info, see http://www.slideshare.net/KaiWaehner/spoilt-for-choice-how-to-choose-the

spring

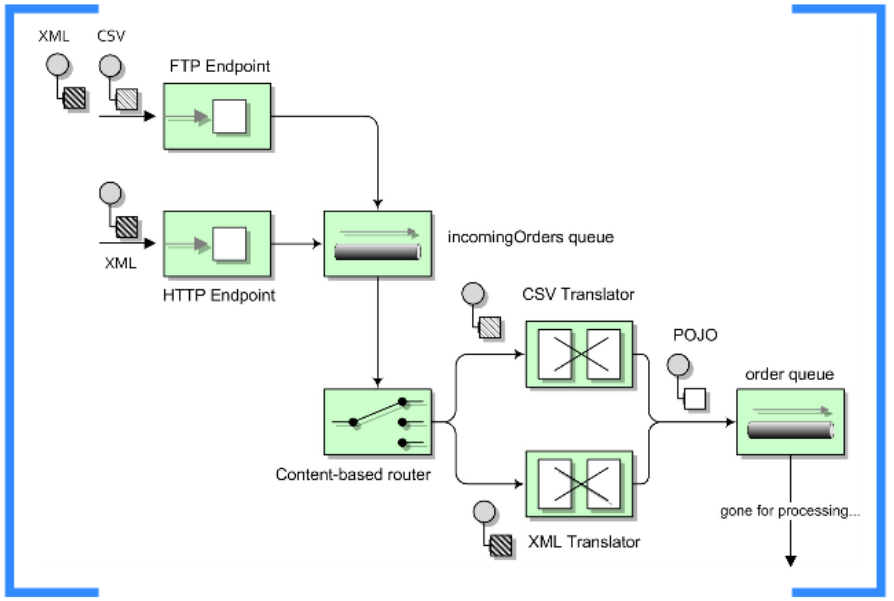
MuleSoft™
connecting the new enterprise

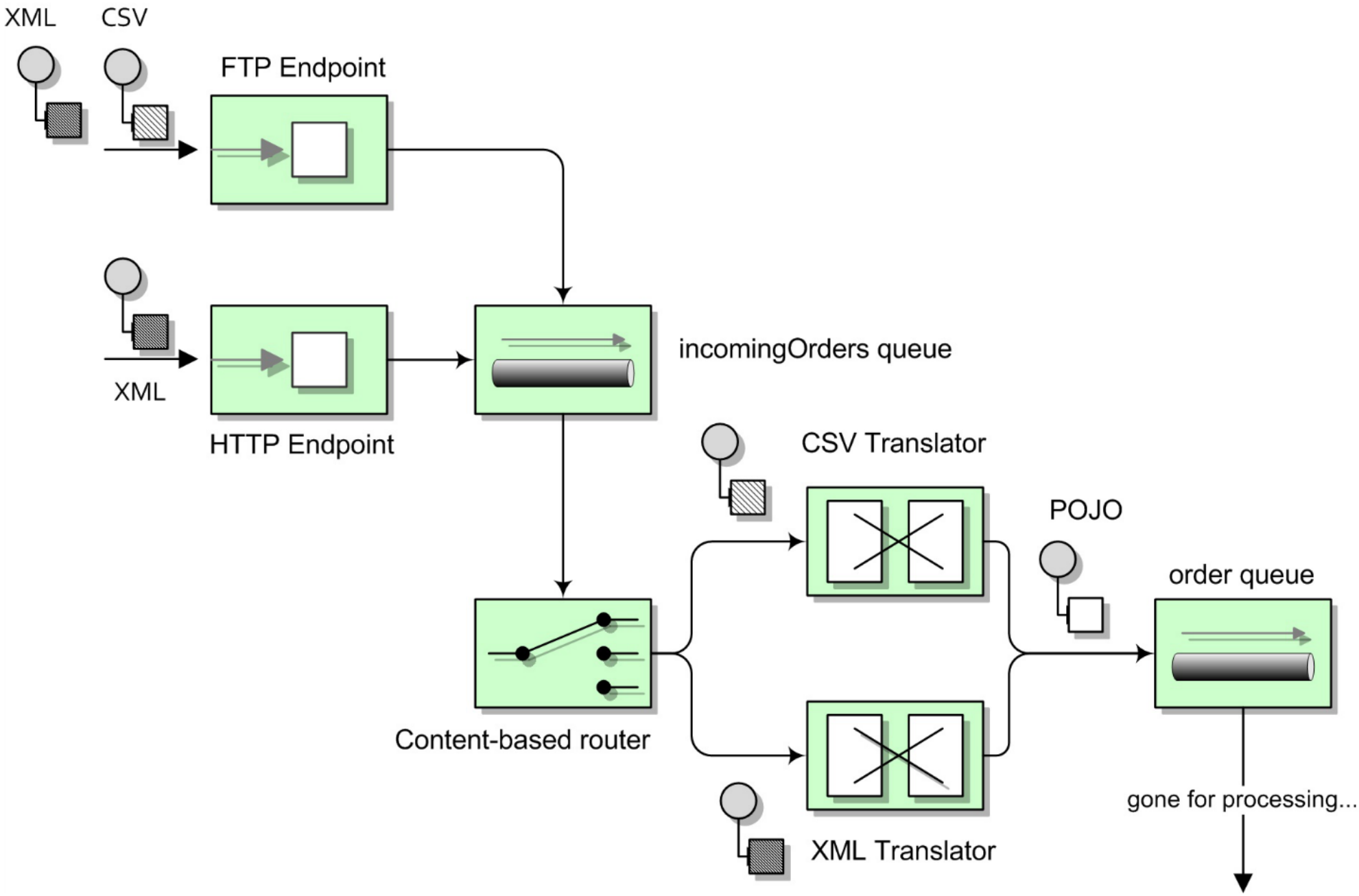
when Camel may not fit

- low level batch processes
  - despite camel-exec for sys level commands
  - better suited for cron/shell scripts (sys admins)
- full blown web applications
  - despite jetty, http, servlet components
  - there are better options
- simple, one off use cases...
  - no wiring/routing needed
  - JMS producing or consuming
  - simple pattern implementation
  - basic component integration

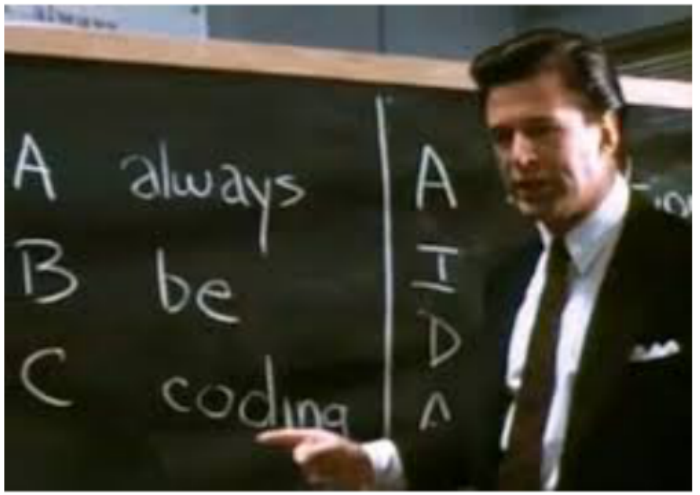
what problems are relevant?

- technology challenges...
  - complex routing
  - technology integrations
  - orchestrating services
- types of requirements...
  - highly event based processes
    - events | messaging
  - exposing multiple endpoints
    - http/jms/file => process
  - complex runtime management
    - auto/manual flow control
    - exception handling




XML
CSV
FTP Endpoint
XML
HTTP Endpoint
incomingOrders queue
CSV Translator
POJO
order queue
Content-based router
XML Translator
gone for processing...

be ready to sell it...



- as with any new technology...
  - choose your battles wisely
  - don't add unnecessary complexity
  - don’t use for just one use case
- to management...
  - create a high level design, discuss alternatives
  - position Camel
    - as an integration technology
    - as a lightweight framework
    - increases productivity -> reuse/abstraction
- to technical team...
  - create a POC and demo it
  - highlight code reduction, readability, testability
  - DRY, convention over configuration

when Camel may not fit

- low level batch processes
  - despite camel-exec for sys level commands
  - better suited for cron/shell scripts (sys admins)
- full blown web applications
  - despite jetty, http, servlet components
  - there are better options
- simple, one off use cases...
  - no wiring/routing needed
  - JMS producing or consuming
  - simple pattern implementation
  - basic component integration

alternatives frameworks

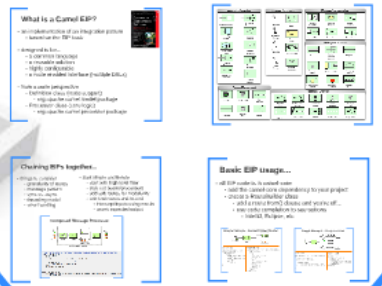
- Spring Integration, Mule ESB
- similarities
  - all open source
  - support EIPs, routing, components
  - IDE support, relatively easy deployment
- differences
  - Spring Integration
    - has a narrower component scope
  - Mule
    - CPAL license (more restrictive)
    - has good proprietary components (SAP, Tibco, Seibel, etc)
  - Camel
    - more flexible DSL than Mule (XML, Java, Groovy, Scala)
    - more components than both
    - strong dev community
    - integrates with Apache projects

*for more info, see http://www.slideshare.net/KaiWaehner/spoilt-for-choice-how-to-choose-the

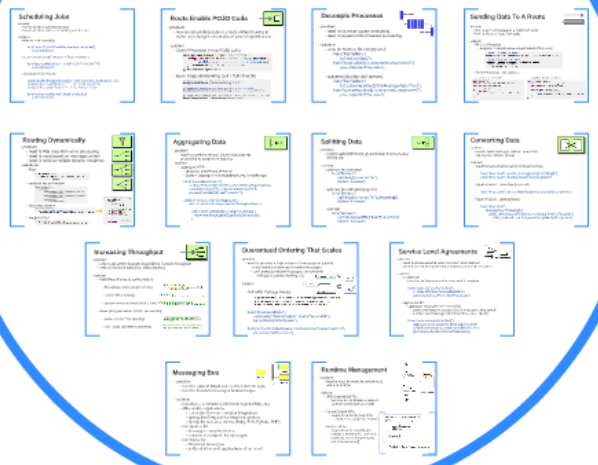
Navigating Options

Components

Patterns



Common Use Cases



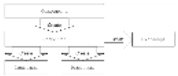
Components

What is a Component?

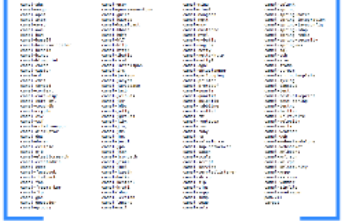
- simply encapsulates an API...to route enable it
- for example
  - camel-file (supports read/write files)
    - wraps java.io.File API

```
from("file:inbound").to("file:outbound")
```

- underlying classes/functions
  - FileComponent
    - createEndpoint() -> FileEndpoint
  - FileEndpoint
    - createProducer() -> FileProducer
    - createConsumer() -> FileConsumer
  - FileConsumer
    - poll() - read files
  - FileProducer
    - process() -> create files



ls ./camel/components



How to Use A Component

- you found a component...now what?
  - review the spec page, samples, unit tests
- include camel-[component].jar in your project
  - add dependency to your pom.xml
  - w/o maven, just add jars to your classpath
- create a basic unit test
  - copy/paste from component's tests
  - sanity test your setup
  - validate desired config options
- integrate your component with your app
  - is your CamelContex setup?
  - is your RouteBuilder wired in?

Evaluating Components

- there are hundreds of components
  - 150+ committed to Camel
  - many more on github, etc.
- how to evaluate components...
  - consider licensing (AL 2.0 vs ?)
  - version supported (parent/pom.xml)
  - supported configuration options
  - producer vs. consumer options
  - review/run unit tests

Custom Components

- first, consider just creating a Bean or Processor
  - simpler way to encapsulate/reuse custom logic
- otherwise, create your own component
  - use an archetype to stub it out

```
mvn archetype:generate
  -DarchetypeGroupId=org.apache.camel.archetypes
  -DarchetypeArtifactId=camel-archetype-component
  -DarchetypeVersion=2.13.0
  -DgroupId=myGroupId -DartifactId=myArtifactId
```

  - manage this project like any other company project
  - include the artifact in your application as a dependency
  - add unit tests to make sure its wired in appropriately
  - optionally publish it for others to use
    - create a Jira and submit a patch file
    - host it on github

Component Categories

- general use
  - direct, seda, file, bean, log, timer
- search, big data, caching
  - lucene, solr, elasticsearch
  - hdfs, s3, twitter, couchDB, mongoDB
  - cache (ehcache), hazelcast
- management
  - jmx, ldap, nagios, splunk, zookeeper
- testing/debugging
  - mock, dataset
- integration
  - http, servlet, cxf, cxfrx, jms
- protocols
  - mina, netty, pop3, imap, smpt, snmp, ftp, ssh, stream, xmpp

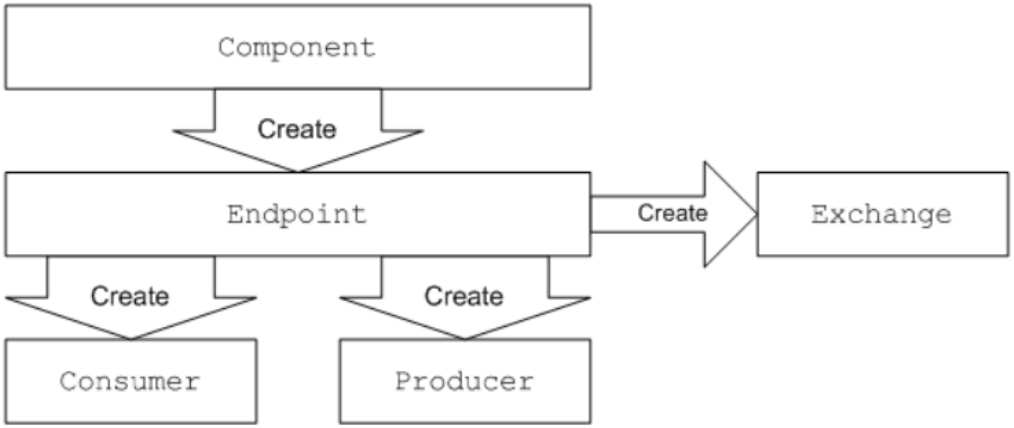
Apache Solr · hazelcast · amazon web services · CouchDB · mongoDB · splunk · EHCACHE · Nagios

What is a Component?

- simply encapsulates an API...to route enable it
- for example
  - camel-file (supports read/write files)
    - wraps java.io.File API

      from("file:inbound").to("file:outbound")

- underlying classes/functions
  - FileComponent
    - createEndpoint() -> FileEndpoint
  - FileEndpoint
    - createProducer() -> FileProducer
    - createConsumer() -> FileConsumer
  - FileConsumer
    - poll() - read files
  - FileProducer
    - process() -> create files



Evaluating Components

- there are hundreds of components
  - 150+ committed to Camel
  - many more on github, etc.

- how to evaluate components...
  - consider licensing (AL 2.0 vs ?)
  - version supported (parent/pom.xml)
  - supported configuration options
  - producer vs. consumer options
  - review/run unit tests

Component Categories

- general use
  - direct, seda, file, bean, log, timer
- search, big data, caching
  - lucene, solr, elasticsearch
  - hdfs, s3, twitter, couchDB, mongoDB
  - cache (ehcache), hazelcast
- management
  - jmx, ldap, nagios, splunk, zookeeper
- testing/debugging
  - mock, dataset
- integration
  - http, servlet, cxf, cxfrx, jms
- protocols
  - mina, netty, pop3, imap, smpt, snmp, ftp, ssh, stream, xmpp

ls ./camel/components

```
camel-ahc
camel-amqp
camel-apns
camel-atom
camel-avro
camel-aws
camel-bam
camel-base64
camel-bean-validator
camel-beanio
camel-bindy
camel-blueprint
camel-cache
camel-castor
camel-cdi
camel-cmis
camel-cometd
camel-context
camel-core-osgi
camel-core-xml
camel-couchdb
camel-crypto
camel-csv
camel-cxf
camel-cxf-transport
camel-disruptor
camel-dns
camel-dozer
camel-eclipse
camel-ejb
camel-elasticsearch
camel-eventadmin
camel-exec
camel-facebook
camel-flatpack
camel-fop
camel-freemarker
camel-ftp
camel-gae
camel-geocoder
camel-groovy
camel-gson
camel-guava-eventbus
camel-guice
camel-hawtdb
camel-hazelcast
camel-hbase
camel-hdfs
camel-hl7
camel-http
camel-http4
camel-ibatis
camel-ical
camel-infinispan
camel-irc
camel-jackson
camel-jasypt
camel-javaspace
camel-jaxb
camel-jclouds
camel-jcr
camel-jdbc
camel-jetty
camel-jgroups
camel-jibx
camel-jing
camel-jms
camel-jmx
camel-josql
camel-jpa
camel-jsch
camel-jsonpath
camel-jt400
camel-juel
camel-jxpath
camel-kafka
camel-kestrel
camel-krati
camel-ldap
camel-leveldb
camel-lucene
camel-mail
camel-mina
camel-mina2
camel-mongodb
camel-mqtt
camel-msv
camel-mustache
camel-mvel
camel-mybatis
camel-nagios
camel-netty
camel-netty-http
camel-netty4
camel-ognl
camel-optaplanner
camel-paxlogging
camel-printer
camel-protobuf
camel-quartz
camel-quartz2
camel-quickfix
camel-rabbitmq
camel-restlet
camel-rmi
camel-routebox
camel-rss
camel-ruby
camel-rx
camel-salesforce
camel-sap-netweaver
camel-saxon
camel-scala
camel-script
camel-servlet
camel-servletlistener
camel-shiro
camel-sip
camel-sjms
camel-smpp
camel-snmp
camel-soap
camel-solr
camel-splunk
camel-spring
camel-spring-batch
camel-spring-integration
camel-spring-javaconfig
camel-spring-ldap
camel-spring-redis
camel-spring-security
camel-spring-ws
camel-sql
camel-ssh
camel-stax
camel-stomp
camel-stream
camel-stringtemplate
camel-syslog
camel-tagsoup
camel-test
camel-test-blueprint
camel-test-spring
camel-testng
camel-twitter
camel-urlrewrite
camel-velocity
camel-vertx
camel-weather
camel-web
camel-web-standalone
camel-websocket
camel-xmlbeans
camel-xmljson
camel-xmlrpc
camel-xmlsecurity
camel-xmpp
camel-xstream
camel-yammer
camel-zipfile
camel-zookeeper
pom.xml
target
```

How to Use A Component

- you found a component...now what?
  - review the spec page, samples, unit tests
- include camel-[component] jar in your project
  - add dependency to your pom.xml
  - w/o maven, just add jars to your classpath
- create a basic unit test
  - copy/paste from component's tests
  - sanity test your setup
  - validate desired config options
- integrate your component with your app
  - is your CamelContex setup?
  - is your RouteBuilder wired in?

Samples

Read from a directory and write to another directory

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

Read from a directory and write to another directory using a overrule dynamic name

```
from("file://inputdir/?delete=true").to("file://outputdir?overruleFile=copy-of-${filename}")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the outputdir and

Reading recursively from a directory and writing to another

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the outputdir and
Will scan recursively into sub-directories. Will lay out the files in the same directory structure in the outputdir a
directories.

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

Will result in the following output layout:

```
outputdir/foo.txt
outputdir/sub/bar.txt
```

```
public class FileConsumerProducerRouteTest extends ContextTestSupport {

    @Override
    protected void setUp() throws Exception {
        deleteDirectory("target/file-test");
        super.setUp();
        template.sendBodyAndHeader("file://target/file-test/a", "Hello World", Exchange.FILE_NAME, "hello.txt");
        template.sendBodyAndHeader("file://target/file-test/a", "Bye World", Exchange.FILE_NAME, "bye.txt");
    }

    public void testFileRoute() throws Exception {
        MockEndpoint result = resolveMandatoryEndpoint("mock:result", MockEndpoint.class);
        result.expectedMessageCount(2);

        result.assertIsSatisfied();
    }

    @Override
    protected RouteBuilder createRouteBuilder() {
        return () -> {
                from("file:target/file-test/a").to("file:target/file-test/b");
                from("file:target/file-test/b").to("mock:result");
        };
    }
}
```

Samples

Read from a directory and write to another directory

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

Read from a directory and write to another directory using a overrule dynamic name

```
from("file://inputdir/?delete=true").to("file://outputdir?overruleFile=copy-of-${file:name}")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the `outputdir` and

Reading recursively from a directory and writing to another

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the `outputdir` and
Will scan recursively into sub-directories. Will lay out the files in the same directory structure in the `outputdir` a
directories.

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

Will result in the following output layout:

```
outputdir/foo.txt
outputdir/sub/bar.txt
```

```
outputdir/foo.txt
outputdir/sub/bar.txt
```

```java
public class FileConsumerProducerRouteTest extends ContextTestSupport {

    @Override
    protected void setUp() throws Exception {
        deleteDirectory("target/file-test");
        super.setUp();
        template.sendBodyAndHeader("file://target/file-test/a", "Hello World", Exchange.FILE_NAME, "hello.txt");
        template.sendBodyAndHeader("file://target/file-test/a", "Bye World", Exchange.FILE_NAME, "bye.txt");
    }

    public void testFileRoute() throws Exception {
        MockEndpoint result = resolveMandatoryEndpoint("mock:result", MockEndpoint.class);
        result.expectedMessageCount(2);

        result.assertIsSatisfied();
    }

    @Override
    protected RouteBuilder createRouteBuilder() {
        return () -> {
                from("file:target/file-test/a").to("file:target/file-test/b");
                from("file:target/file-test/b").to("mock:result");
        };
    }
}
```

Custom Components

- first, consider just creating a Bean or Processor
  - simpler way to encapsulate/reuse custom logic

- otherwise, create your own component
  - use an archetype to stub it out

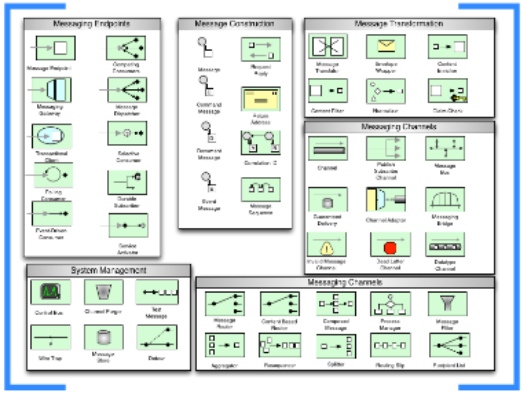
```
mvn archetype:generate
  -DarchetypeGroupId=org.apache.camel.archetypes
  -DarchetypeArtifactId=camel-archetype-component
  -DarchetypeVersion=2.13.0
  -DgroupId=myGroupId -DartifactId=myArtifactId
```

  - manage this project like any other company project
  - include the artifact in your application as a dependency
  - add unit tests to make sure its wired in appropriately
  - optionally publish it for others to use
    - create a Jira and submit a patch file
    - host it on github

Patterns

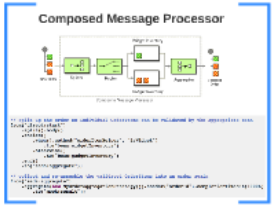
What is a Camel EIP?

- an implementation of an integration pattern
  - based on the EIP book
- designed to be...
  - a common language
  - a reusable solution
  - highly configurable
  - a route enabled interface (multiple DSLs)
- from a code perspective
  - Definition class (route support)
    - org.apache.camel.model package
  - Processor class (core logic)
    - org.apache.camel.processor package



Chaining EIPs together...

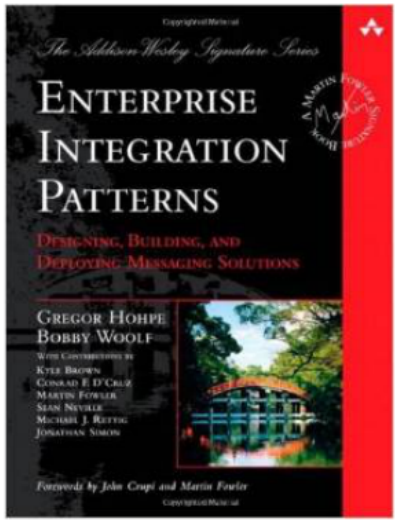
- things to consider
  - granularity of routes
  - message pattern
  - sync vs. async
  - threading model
  - error handling
- start simple and iterate
  - start with high level flow
  - stub out beans/processors
  - add sub routes for modularity
  - unit test routes end-to-end
    - intercept inputs using mocks
    - assert expected output



Basic EIP usage...

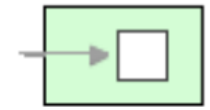
- all EIP code is in camel-core
  - add the camel-core dependency to your project
  - create a RouteBuilder class
    - add a route from() clause and you're off...
    - use code completion to see options
      - IntelliJ, Eclipse, etc

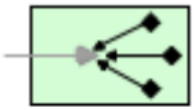
What is a Camel EIP?

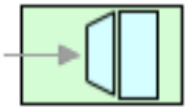


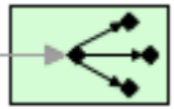
- an implementation of an integration pattern
  - based on the EIP book
- designed to be...
  - a common language
  - a reusable solution
  - highly configurable
  - a route enabled interface (multiple DSLs)
- from a code perspective
  - Definition class (route support)
    - org.apache.camel.model package
  - Processor class (core logic)
    - org.apache.camel.processor package

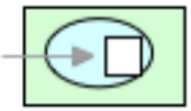
Messaging Endpoints

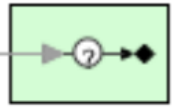

Message Endpoint


Competing Consumers

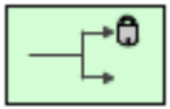

Messaging Gateway

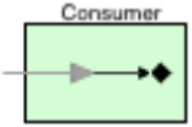

Message Dispatcher

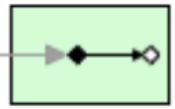

Transactional Client


Selective Consumer


Polling Consumer

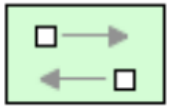

Durable Subscriber


Event-Driven Consumer

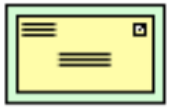

Service Activator

Message Construction

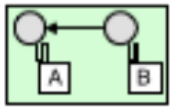

Message


Request Reply

Command Message

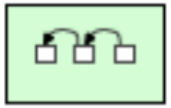

Return Address

Document Message

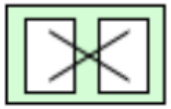


Correlation ID

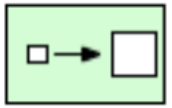
Event Message

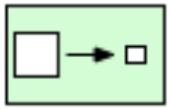

Message Sequence

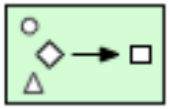
Message Transformation

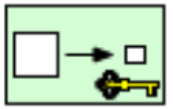

Message Translator


Envelope Wrapper

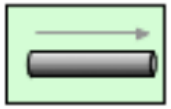

Content Enricher

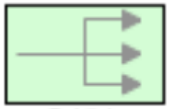

Content Filter

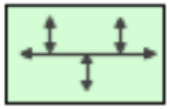

Normalizer

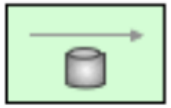

Claim Check

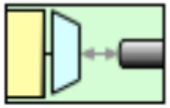
Messaging Channels

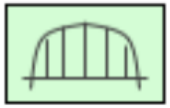

Channel


Publish Subscribe Channel

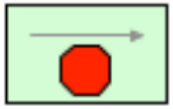

Message Bus

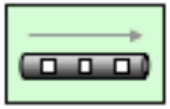

Guaranteed Delivery


Channel Adapter


Messaging Bridge


Invalid Message Channel

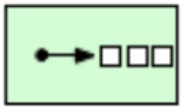

Dead Letter Channel


Datatype Channel

System Management

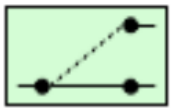

Control Bus


Channel Purger


Test Message


Wire Ttap


Message Store


Detour

Messaging Channels


Message Router

Content Based Router

Composed Message

Process Manager

Message Filter

Aggregator

Resequencer

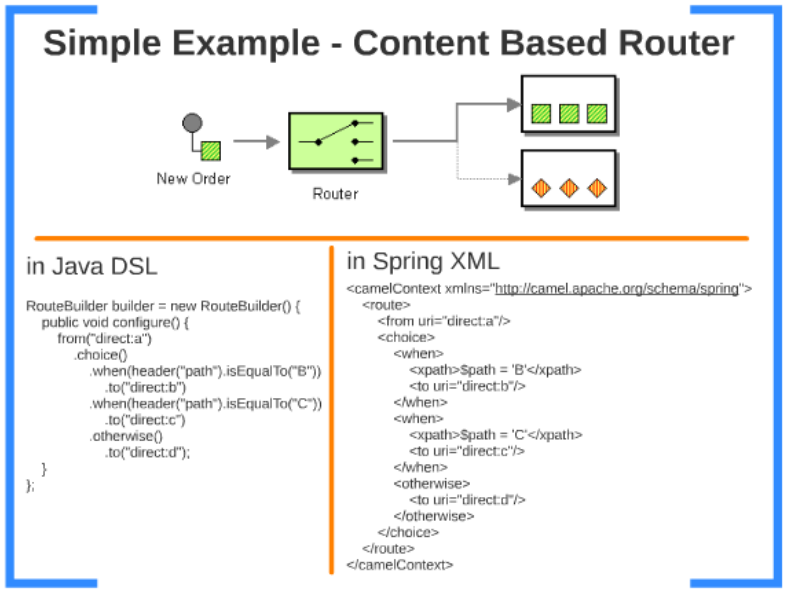
Splitter

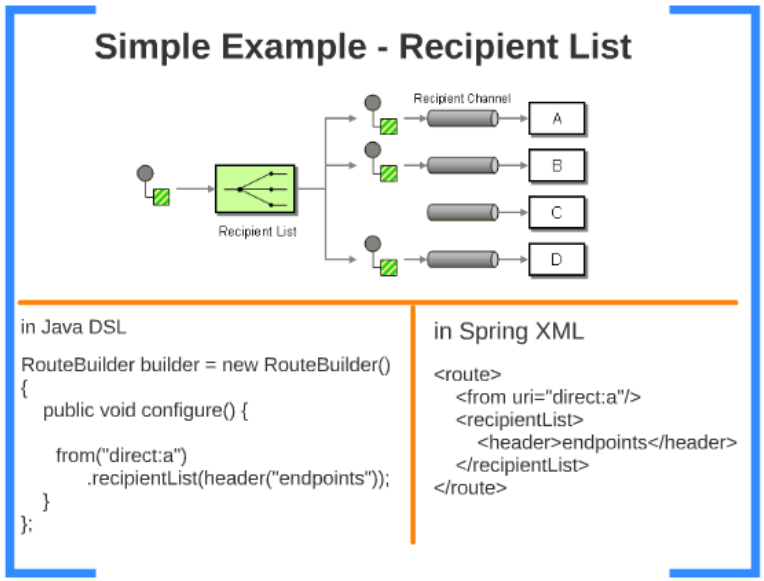
Routing Slip

Recipient List

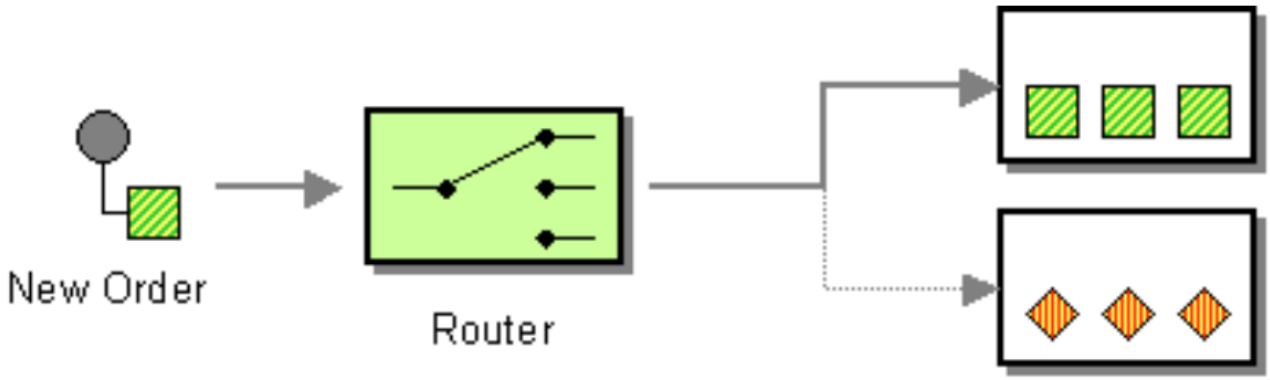
Basic EIP usage...

- all EIP code is in camel-core
  - add the camel-core dependency to your project
  - create a RouteBuilder class
    - add a route from() clause and you're off...
    - use code completion to see options
      - IntelliJ, Eclipse, etc





Simple Example - Content Based Router



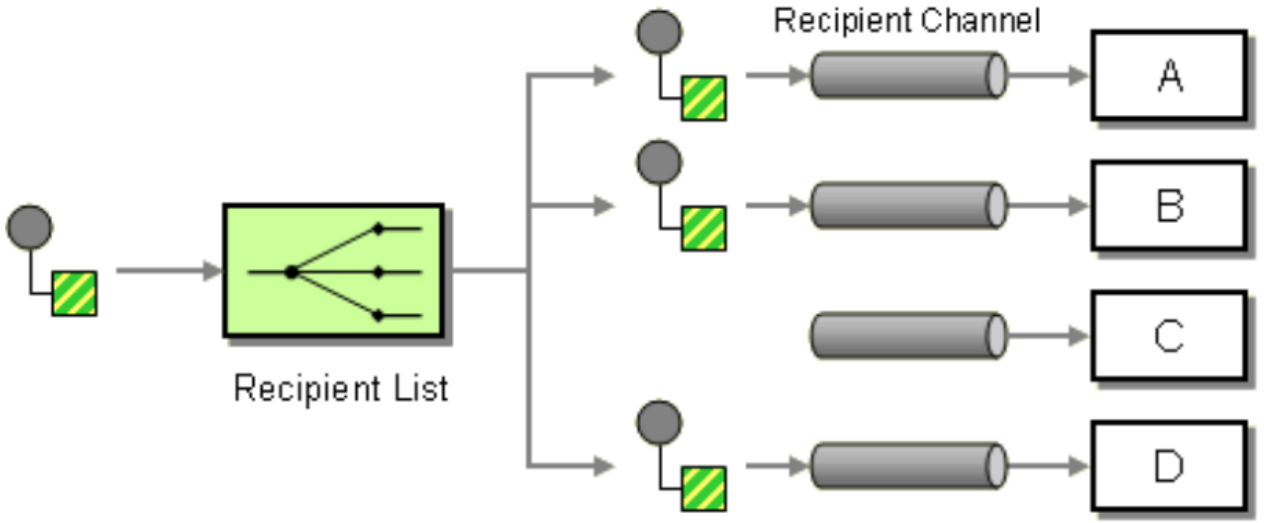
in Java DSL

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:a")
            .choice()
                .when(header("path").isEqualTo("B"))
                    .to("direct:b")
                .when(header("path").isEqualTo("C"))
                    .to("direct:c")
                .otherwise()
                    .to("direct:d");
    }
};
```

in Spring XML

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:a"/>
        <choice>
            <when>
                <xpath>$path = 'B'</xpath>
                <to uri="direct:b"/>
            </when>
            <when>
                <xpath>$path = 'C'</xpath>
                <to uri="direct:c"/>
            </when>
            <otherwise>
                <to uri="direct:d"/>
            </otherwise>
        </choice>
    </route>
</camelContext>
```

Simple Example - Recipient List



in Java DSL

```
RouteBuilder builder = new RouteBuilder()
{
    public void configure() {

      from("direct:a")
          .recipientList(header("endpoints"));
    }
};
```

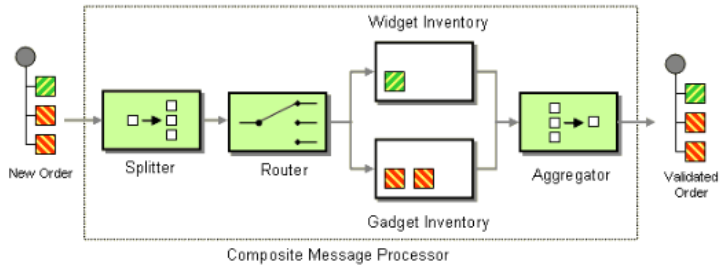
in Spring XML

```
<route>
    <from uri="direct:a"/>
    <recipientList>
        <header>endpoints</header>
    </recipientList>
</route>
```

Chaining EIPs together...

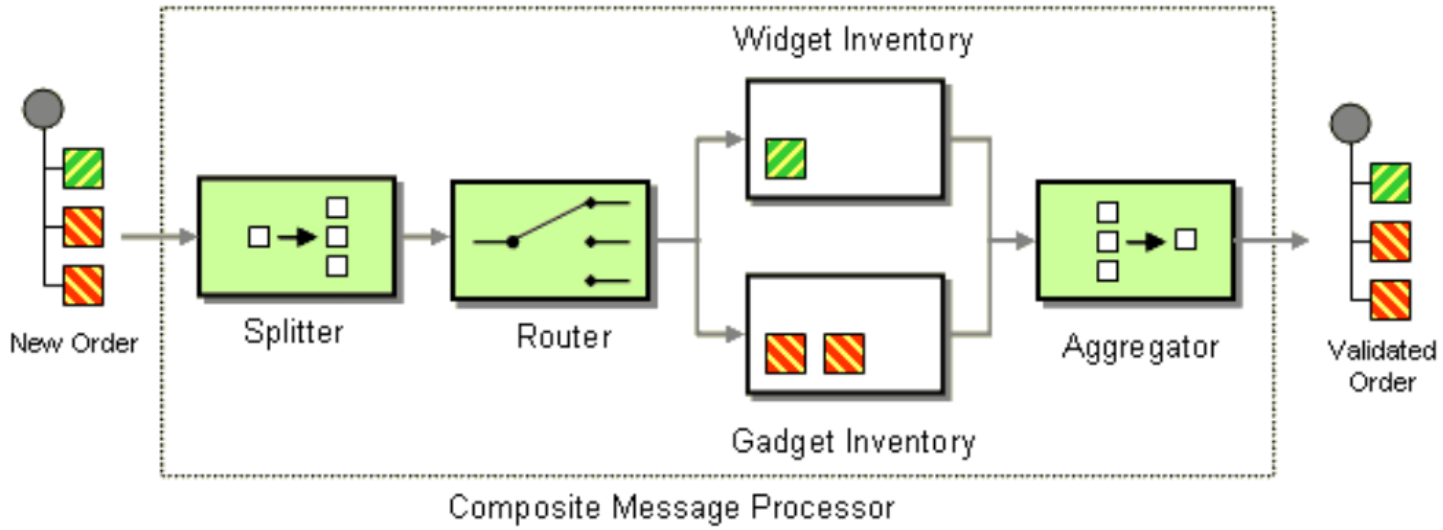
- things to consider
  - granularity of routes
  - message pattern
  - sync vs. async
  - threading model
  - error handling
- start simple and iterate
  - start with high level flow
  - stub out beans/processors
  - add sub routes for modularity
  - unit test routes end-to-end
    - intercept inputs using mocks
    - assert expected output

Composed Message Processor



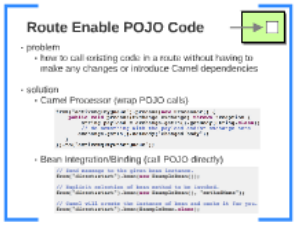

```
// split up the order so individual OrderItems can be validated by the appropriate bean
from("direct:start")
    .split().body()
    .choice()
        .when().method("orderItemHelper", "isWidget")
            .to("bean:widgetInventory")
        .otherwise()
            .to("bean:gadgetInventory")
    .end()
    .to("seda:aggregate");

// collect and re-assemble the validated OrderItems into an order again
from("seda:aggregate")
    .aggregate(new MyOrderAggregationStrategy()).header("orderId").completionTimeout(1000L)
        .to("mock:result");
```

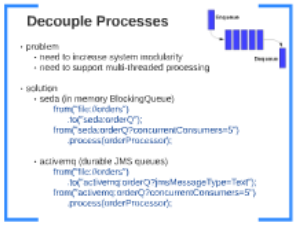
Composed Message Processor

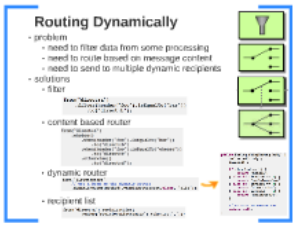


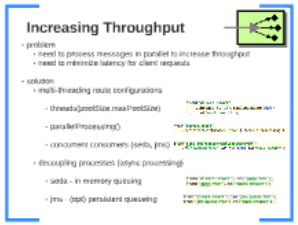

```
// split up the order so individual OrderItems can be validated by the appropriate bean
from("direct:start")
    .split().body()
    .choice()
        .when().method("orderItemHelper", "isWidget")
            .to("bean:widgetInventory")
        .otherwise()
            .to("bean:gadgetInventory")
    .end()
    .to("seda:aggregate");

// collect and re-assemble the validated OrderItems into an order again
from("seda:aggregate")
    .aggregate(new MyOrderAggregationStrategy()).header("orderId").completionTimeout(1000L)
        .to("mock:result");
```

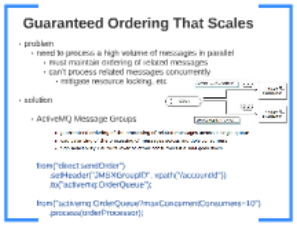
Common Use Cases

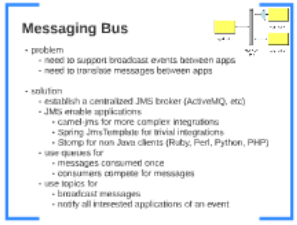


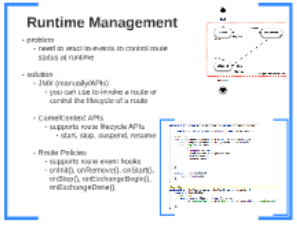












Scheduling Jobs

- problem
  - how to schedule periodic processes
  - how to schedule routes to run during specific times

- solutions
  - timer (run job every 60s)

```
from("timer://MyJob?fixedRate=true&period=60000")
    .process(runJob);
```

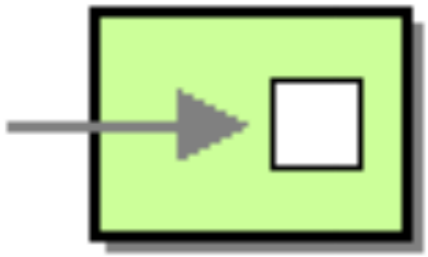
  - quartz (run job every 5 minutes, 8-5pm, weekdays)

```
from("quartz://MyJob?cron=0+0/5+8-17+?+*+MON-FRI")
    .process(runJob);
```

  - Scheduled Route Policies

```
CronScheduledRoutePolicy policy = new CronScheduledRoutePolicy();
policy.setRouteStartTime("0 * * * * ?");  //start every hour
policy.setRouteStopTime("5 * * * * ?");  //stop at 5 after

from("activemq:jobQueue").routePolicy(policy)
    .process(runJob);
```

Route Enable POJO Code



- problem
  - how to call existing code in a route without having to make any changes or introduce Camel dependencies

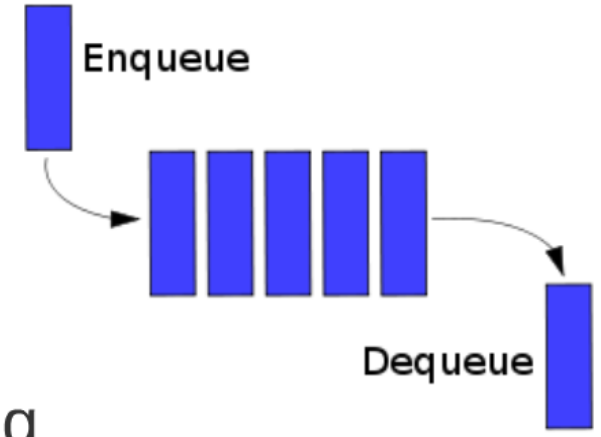
- solution
  - Camel Processor (wrap POJO calls)

```
from("activemq:myQueue").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String payload = exchange.getIn().getBody(String.class);
        // do something with the payload and/or exchange here
       exchange.getIn().setBody("Changed body");
   }
}).to("activemq:myOtherQueue");
```

  - Bean Integration/Binding (call POJO directly)

```
// Send message to the given bean instance.
from("direct:start").bean(new ExampleBean());

// Explicit selection of bean method to be invoked.
from("direct:start").bean(new ExampleBean(), "methodName");

// Camel will create the instance of bean and cache it for you.
from("direct:start").bean(ExampleBean.class);
```

Decouple Processes



- problem
  - need to increase system modularity
  - need to support multi-threaded processing

- solution
  - seda (in memory BlockingQueue)

```
from("file://orders")
    .to("seda:orderQ");
from("seda:orderQ?concurrentConsumers=5")
    .process(orderProcessor);
```

  - activemq (durable JMS queues)

```
from("file://orders")
    .to("activemq:orderQ?jmsMessageType=Text");
from("activemq:orderQ?concurrentConsumers=5")
    .process(orderProcessor);
```

Sending Data To A Route

- problem
  - how to send a message to a route from code
  - how to initiate a route from code

- solution
  - ProducerTemplate
    - template = camelContext.createProducerTemplate();

```
// send to a specific queue
template.sendBody("activemq:MyQueue", "<hello>world!</hello>");

// send with a body and header
template.sendBodyAndHeader("activemq:MyQueue",
   "<hello>world!</hello>",
   "CustomerRating", "Gold");
```

  - POJO Producing - two options...

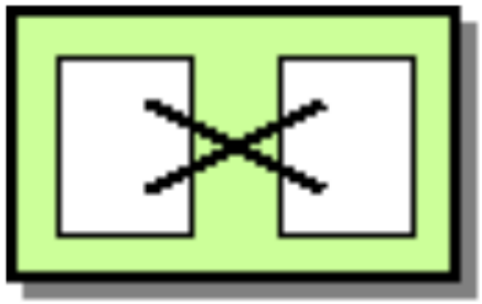
1. using Camel annotation

```
public class Foo {
  @EndpointInject(uri="activemq:foo.bar")
  ProducerTemplate producer;

  public void doSomething() {
    if (whatever) {
      producer.sendBody("<hello>world!</hello>");
    }
  }
}
```

2. Spring remoting annotation

```
public class MyBean {
    @Produce(uri = "activemq:foo")
    protected MyListener producer;

    public void doSomething() {
        // lets send a message
        String response = producer.sayHello("James");
    }
}
```

Converting Data



- problem
  - need to send messages between processes that expect different formats

- solutions
  - DataFormat (convert between common formats)

```
from("direct:start").marshal().string().to("jms://stringQ");
from("direct:start").marshal().json().to("jms:jsonQ");
```

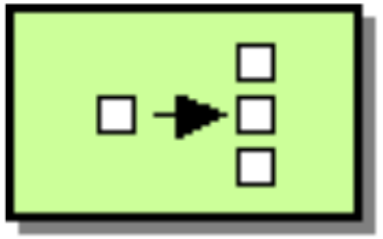
  - TypeConverter - convertBodyTo(class)

```
from("direct:start").convertBodyTo(String.class).to("jms://stringQ");
```

  - TypeConverter - getBody(class)

```
from("direct:start")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            String payload = exchange.getIn().getBody(String.class);
            ...
```

Splitting Data



- problem
  - need to split delimited or grouped data for processing individually

- solution
  - split new line delimited

```
from("file:input")
    .split(body().tokenize("\n"))
    .to(itemProcessor);
```

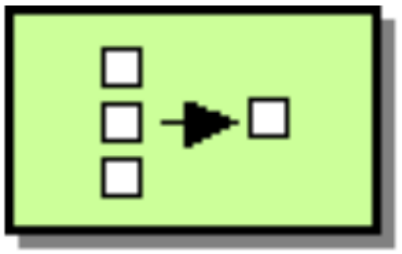
  - split new line delimited (large file)

```
from("file:input")
    .split(body().tokenize("\n")).streaming()
    .to(itemProcessor);
```

  - split XML

```
from("file:input")
    .split().tokenizeXML("item").streaming()
    .to(itemProcessor);
```

Aggregating Data



- problem
  - need to combine related data for processing or sending to systems in batches
- solution
  - aggregator EIP
    - group by size/timeout/interval
    - custom aggregator strategy/predicate, leveldb repo

```
from("jms:inboundOrders")
    .aggregate(header("orderId"), new OrderListAggregator())
    .completionSize(100).completionInterval(10000)
    .process(new BatchOrderProcessor());

public final class OrderListAggregator
    extends AbstractListAggregationStrategy<Order> {

      public Order getValue(Exchange exchange) {
          return exchange.getIn().getBody(Order.class);
      }
  }
```

Routing Dynamically

- problem
  - need to filter data from some processing
  - need to route based on message content
  - need to send to multiple dynamic recipients
- solutions
  - filter

```
from("direct:a")
    .filter(header("foo").isEqualTo("bar"))
        .to("direct:b");
```

  - content based router

```
from("direct:a")
    .choice()
        .when(header("foo").isEqualTo("bar"))
            .to("direct:b")
        .when(header("foo").isEqualTo("cheese"))
            .to("direct:c")
        .otherwise()
            .to("direct:d");
```

  - dynamic router

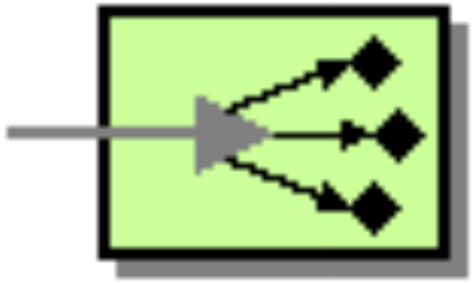
```
from("direct:start")
    // use a bean as the dynamic router
    .dynamicRouter(method(DynamicRouterTest.class, "slip"));
```

```
public String slip(String body) {
    bodies.add(body);
    invoked++;

    if (invoked == 1) {
        return "mock:a";
    } else if (invoked == 2) {
        return "mock:b,mock:c";
    } else if (invoked == 3) {
        return "direct:foo";
    } else if (invoked == 4) {
        return "mock:result";
    }

    // no more so return null
    return null;
}
```

  - recipient list

```
from("direct:a").recipientList(
        header("recipientListHeader").tokenize(","));
```

Increasing Throughput



- problem
  - need to process messages in parallel to increase throughput
  - need to minimize latency for client requests
- solution
  - multi-threading route configurations
    - threads(poolSize,maxPoolSize)

```
from("direct:start")
    .threads(5, 10).maxQueueSize(2000)
    .to("mock:result");
```

    - parallelProcessing()

```
from("direct:start")
    .recipientList(header("foo")).parallelProcessing();
```

    - concurrent consumers (seda, jms)

```
from("seda:foo?concurrentConsumers=10")
    .to("mock:before").delay(2000).to("mock:result");
```

  - decoupling processes (async processing)
    - seda - in memory queuing

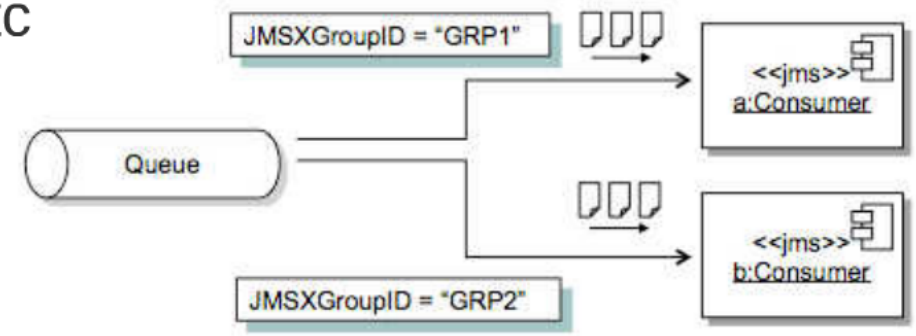
```
from("direct:start").to("seda:foo");
from("seda:foo").to("mock:result");
```

    - jms - (opt) persistent queueing

```
from("direct:start").to("jms:queue:foo");
from("jms:queue:foo").to("mock:result");
```

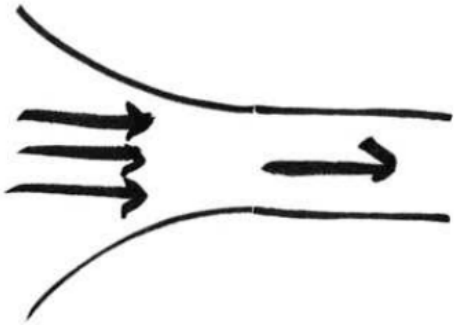
Guaranteed Ordering That Scales

- problem
  - need to process a high volume of messages in parallel
    - must maintain ordering of related messages
    - can't process related messages concurrently
      - mitigate resource locking, etc
- solution
  - ActiveMQ Message Groups
    - guarranteed ordering of the processing of related messages across a single queue
    - load balancing of the processing of messages across multiple consumers
    - high availability / auto-failover to other consumers if a JVM goes down




```
from("direct:sendOrder")
    .setHeader("JMSXGroupID", xpath("/accountId"))
    .to("activemq:OrderQueue");

from("activemq:OrderQueue?maxConcurrentConsumers=10")
    .process(orderProcessor);
```

Service Level Agreements



- problem
  - need to ensure external resources aren't overwhelmed
  - need to not exceed agreed upon frequency of use of a resource

- solution
  - throttler EIP
    - provides configurable control over route throughput

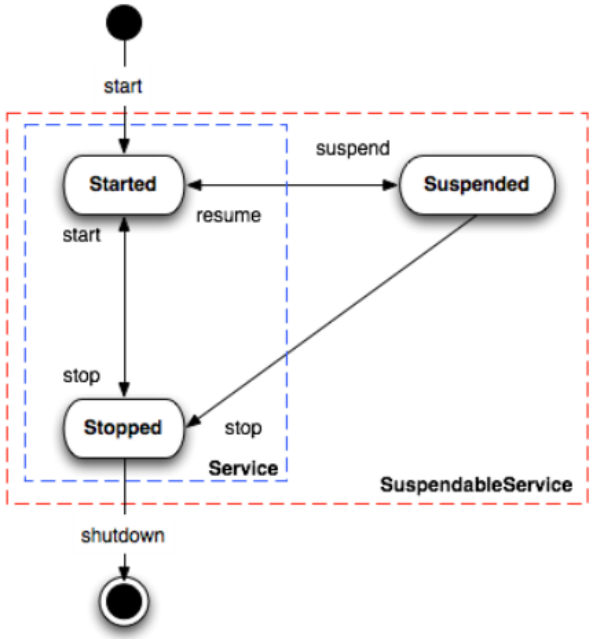
```
from("seda:GetDataFromClient")
    .throttle(100).timePeriodMillis(60000)
    .process(GetDataFromClientProcessor());
```

  - aggregator EIP
    - aggregate messages over time period
      - send fewer/larger messages (List of message during period)
      - just the latest message (short-lived data...stock quote)

```
from("seda:GetDataFromClient")
    .aggregate(constant(true), MyListAggregator())
    .completionSize(100).completionInterval(60000)
    .process(GetDataFromClientProcessor());
```

Runtime Management

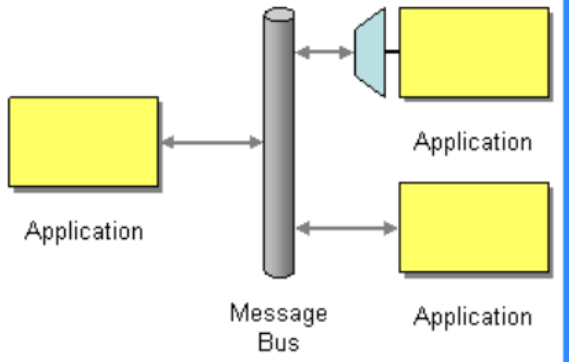
- problem
  - need to react to events to control route status at runtime
- solution
  - JMX (manually/APIs)
    - you can use to invoke a route or control the lifecycle of a route
  - CamelContext APIs
    - supports route lifecycle APIs
      - start, stop, suspend, resume
  - Route Policies
    - supports route event hooks
    - onInit(), onRemove(), onStart(), onStop(), onExchangeBegin(), onExchangeDone()




```
private static class MyCustomRoutePolicy extends RoutePolicySupport {

    private volatile AtomicBoolean stopped = new AtomicBoolean();

    @Override
    public void onExchangeDone(Route route, Exchange exchange) {
        String body = exchange.getIn().getBody(String.class);
        if ("stop".equals(body)) {
            try {
                stopped.set(true);
                stopConsumer(route.getConsumer());
            } catch (Exception e) {
                handleException(e);
            }
        }
    }

    public boolean isStopped() {
        return stopped.get();
    }
}

@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:foo").routeId("foo").routePolicy(policy).to("mock:result");
        }
    };
}
```

```
private static class MyCustomRoutePolicy extends RoutePolicySupport {

    private volatile AtomicBoolean stopped = new AtomicBoolean();

    @Override
    public void onExchangeDone(Route route, Exchange exchange) {
        String body = exchange.getIn().getBody(String.class);
        if ("stop".equals(body)) {
            try {
                stopped.set(true);
                stopConsumer(route.getConsumer());
            } catch (Exception e) {
                handleException(e);
            }
        }
    }

    public boolean isStopped() {
        return stopped.get();
    }
}

@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:foo").routeId("foo").routePolicy(policy).to("mock:result");
        }
    };
}
```

Messaging Bus



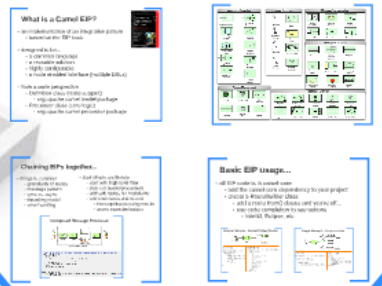
- problem
  - need to support broadcast events between apps
  - need to translate messages between apps

- solution
  - establish a centralized JMS broker (ActiveMQ, etc)
  - JMS enable applications
    - camel-jms for more complex integrations
    - Spring JmsTemplate for trivial integrations
    - Stomp for non Java clients (Ruby, Perl, Python, PHP)
  - use queues for
    - messages consumed once
    - consumers compete for messages
  - use topics for
    - broadcast messages
    - notify all interested applications of an event

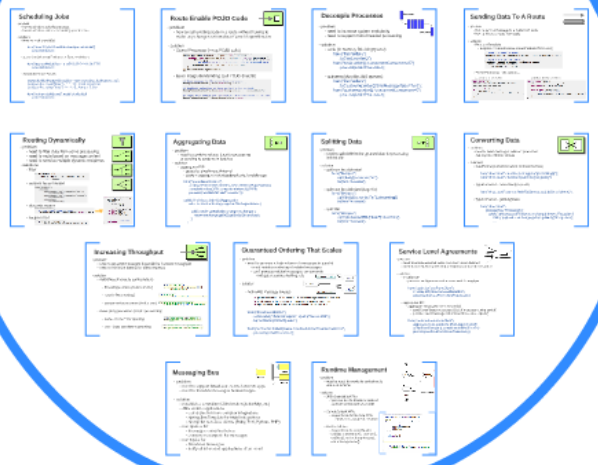
Navigating Options

Components

Patterns



Common Use Cases



Lifecycle

Develop

create a project and POC tests...

divide routes into groups...

route exception handling options...

start adding routes...

how to handle errors

Test

how to test Camel apps

POC route testing...

Write a failing test

Make the test pass

Refactor

performance testing routes...

functional testing routes...

Deploy

deployment options...

deployment options...

DEPLOY!

how to monitor

Design

requirements/high level design

analyze use cases...

refine the design...

Design

requirements/high level design

- where to start?
  - create a diagram to visualize the following:
    - each independent system or component (box)
    - each interaction (lines with arrows)
  - consider relevant requirements
    - SLAs
      - downtime, latency
    - performance
      - expected throughput (min/avg/max/peak)
    - reliability
      - guaranteed delivery, strict ordering
  - error handling
    - manual triaging vs. automated

analyze use cases...

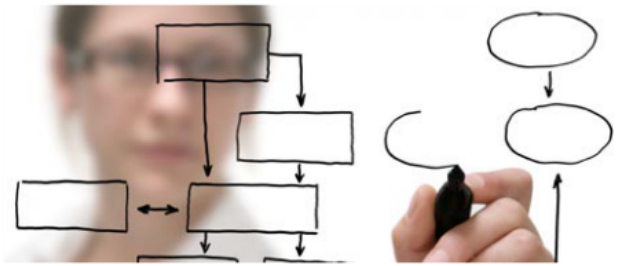
- break down use cases to…
  - identify processes
    - data flows, event triggers, periodic events
  - categorize processes
    - synchronous (request/reply)
    - asynchronous (fire/forget, in-only)
    - high vs. low volume
    - small vs. large payloads
  - identify technology interfaces
    - file, jms, hdfs, solr, jdbc, etc
  - create pseudo routes
    - file -> split -> validate -> save
    - http -> validate -> save
    - jms -> validate -> save

refine the design...

- across all use cases
  - identify routes that have common operations
  - refactor routes to normalize common operations
- consider exception handling requirements
  - affects the structure and granularity of routes
- consider the size/scope of the applications
  - how granular should deployments be
- divide routes into independently deployable groups

requirements/high level design

- where to start?
  - create a diagram to visualize the following:
    - each independent system or component (box)
    - each interaction (lines with arrows)
  - consider relevant requirements
    - SLAs
      - downtime, latency
    - performance
      - expected throughput (min/avg/max/peak)
    - reliability
      - guaranteed delivery, strict ordering
  - error handling
    - manual triaging vs. automated



analyze use cases...

- break down use cases to…
  - identify processes
    - data flows, event triggers, periodic events
  - categorize processes
    - synchronous (request/reply)
    - asynchronous (fire/forget, in-only)
    - high vs. low volume
    - small vs. large payloads
  - identify technology interfaces
    - file, jms, hdfs, solr, jdbc, etc
  - create pseudo routes
    - file -> split -> validate -> save
    - http -> validate -> save
    - jms -> validate -> save

refine the design...

- across all use cases
  - identify routes that have common operations
  - refactor routes to normalize common operations
- consider exception handling requirements
  - affects the structure and granularity of routes
- consider the size/scope of the applications
  - how granular should deployments be
- divide routes into independently deployable groups

- original routes
  - file -> split -> **validate -> save**
  - http -> **validate -> save**
  - jms -> **validate -> save**
- normalized
  - file -> split -> **process order**
  - http -> **process order**
  - jms -> **process order**
  - **process order -> validate -> save**

- original routes
  - file -> split -> **validate -> save**
  - http -> **validate -> save**
  - jms -> **validate -> save**
- normalized
  - file -> split -> **process order**
  - http -> **process order**
  - jms -> **process order**
  - **process order -> validate -> save**

Develop

create a project and POC tests...

- use a Camel archetype to create a project skeleton
  - assumes you know your deployment container
    - camel-archetype-java (JAR)
    - camel-archetype-web (WAR)
    - camel-archetype-blueprint (BUNDLE JAR)
- create POC unit tests
  - validate any complex pattern/technology integration in the design
  - start by reviewing the Camel source unit tests
    - create your own tests
    - make incremental changes to explore technologies
  - leverage Camel EIPs/Components as needed
  - explore new patterns/components until you settle on an approach

divide routes into groups..

- create RouteBuilder classes
  - for each logical/functional area
  - this is more art than science
  - can be as granular as you want
  - depends on scope/requirements of your app
- scoping considerations
  - single CamelContext per app
  - multiple Route Builders
    - onException handling is RB specific
  - for inter app communication use
    - JMS, WS, REST, OSGI services, etc
  - this is all easy to misuse
    - start simple, refactor, test, keep it simple

route exception handling options...

- onException
  - can be route scoped or global (all routes in RouteBuilder)
  - features: configurable redelivery counts/delays, propagation

```
onException(Exception.class)
    .handled(true).maximumRedeliveries(5).redeliveryDelay(10000)
    .to("activemq:generalErrors");

onException(MyBusinessException.class)
    .handled(true).maximumRedeliveries(0)
    .to("activemq:businessErrors");
```

- try-catch-finally
  - modeled after Java's try/catch/finally
  - defined within a specific route and applies only to that route

```
from("direct:start")
    .doTry()
        process(new ProcessorFail())
        .to("mock:result")
    .doCatch(IOException.class, IllegalStateException.class)
        .to("mock:catch")
    .doFinally()
        .to("mock:finally")
    .end();
```

start adding routes...

- create routes
  - based on your designed pseudo routes/POC
- encapsulate core business logic in Beans
  - use bean binding to minimize Camel APIs
  - promotes reusability, independent testability
- start high level
  - coarse grained (readable) routing steps
  - start with the happy path
    - add route exception handling later
- keep routing rules...in routes
  - don't reinvent the wheel in POJO code
  - don't bury complex rules/routing in POJO code
    - use ProducerTemplate sparingly

how to handle errors

- consider the requirements and options...
  - propagate errors back to callers?
  - retry automatically? how many times? what frequency?
  - send to an error queue? send an email?
- best practices
  - keep it simple
    - don't over analyze up front -> build/test/refactor
  - minimize retries
    - don't blindly retry everything
    - retry only if required and errors are temporary
    - retries can be expensive (tie up threads)
  - monitor early and often
    - don't rely on watching log files
    - use a log monitoring tool (Splunk, etc)
    - roll your own email notifications
      - use an aggregator to reduce noise

how to test Ca

- should be an ongoing ef
  - starts in early design
  - continues throughou
  - ends with performan
- types of tests
  - POC tests
    - standlone tests to
  - functional route tests

create a project and POC tests...

- use a Camel archetype to create a project skeleton
  - assumes you know your deployment container
    - camel-archetype-java (JAR)
    - camel-archetype-web (WAR)
    - camel-archetype-blueprint (BUNDLE JAR)

- create POC unit tests
  - validate any complex pattern/technology integration in the design
  - start by reviewing the Camel source unit tests
    - create your own tests
    - make incremental changes to explore technologies
  - leverage Camel EIPs/Components as needed
  - explore new patterns/components until you settle on an approach

divide routes into groups..

- create RouteBuilder classes
  - for each logical/functional area
  - this is more art than science
  - can be as granular as you want
  - depends on scope/requirements of your app

- scoping considerations
  - single CamelContext per app
  - multiple Route Builders
    - onException handling is RB specific
  - for inter app communication use
    - JMS, WS, REST, OSGI services, etc
  - this is all easy to misuse
    - start simple, refactor, test, keep it simple

start adding routes...

- create routes
  - based on your designed pseudo routes/POC
- encapsulate core business logic in Beans
  - use bean binding to minimize Camel APIs
  - promotes reusability, independent testability
- start high level
  - coarse grained (readable) routing steps
  - start with the happy path
    - add route exception handling later
- keep routing rules...in routes
  - don’t reinvent the wheel in POJO code
  - don't bury complex rules/routing in POJO code
    - use ProducerTemplate sparingly

how to handle errors

- consider the requirements and options…
  - propagate errors back to callers?
  - retry automatically? how many times? what frequency?
  - send to an error queue? send an email?

- best practices
  - keep it simple
    - don’t over analyze up front -> build/test/refactor
  - minimize retries
    - don’t blindly retry everything
    - retry only if required and errors are temporary
    - retries can be expensive (tie up threads)
  - monitor early and often
    - don’t rely on watching log files
    - use a log monitoring tool (Splunk, etc)
    - roll your own email notifications
      - use an aggregator to reduce noise

route exception handling options...

- onException
  - can be route scoped or global (all routes in RouteBuilder)
  - features: configurable redelivery counts/delays, propagation

```
onException(Exception.class)
    .handled(true).maximumRedeliveries(5).redeliveryDelay(10000)
    .to("activemq:generalErrors");

onException(MyBusinessException.class)
    .handled(true).maximumRedeliveries(0)
    .to("activemq:businessErrors");
```

- try-catch-finally
  - modeled after Java's try/catch/finally
  - defined within a specific route and applies only to that route

```
from("direct:start")
      .doTry()
         .process(new ProcessorFail())
         .to("mock:result")
      .doCatch(IOException.class, IllegalStateException.class)
         .to("mock:catch")
      .doFinally()
         .to("mock:finally")
      .end();
```

exception handling later

s...in routes
the wheel in POJO code
mplex rules/routing in POJO code
cerTemplate sparingly

Test

how to test Camel apps

- should be an ongoing effort
  - starts in early design
  - continues throughout development
  - ends with performance and acceptance tests

- types of tests
  - POC tests
    - standlone tests to validate routing logic
  - functional route tests
    - standup your real routes to validate flow
    - use adviceWith to inject mock endpoints/assertions
  - route performance tests
    - use camel-dataset to drive/gather stats
  - business logic tests
    - prefer Bean Integration to isolate/test w/o Camel
    - use a mock framework (Mockito, etc) to isolate

POC route testing...

```
public class BasicRouteTest extends CamelTestSupport
{
    @Test
    public void test() throws Exception
    {
        MockEndpoint mockA = getMockEndpoint("mock:mock");
        mockA.setExpectedMessageCount(1);
        template.sendBody("direct:start", "message1");
        assertMockEndpointsSatisfied();
    }

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception
    {
        return new RouteBuilder()
        {
            @Override
            public void configure() throws Exception
            {
                from("direct:start").to("mock:mock");
            }
        };
    }
}
```

Write a failing test

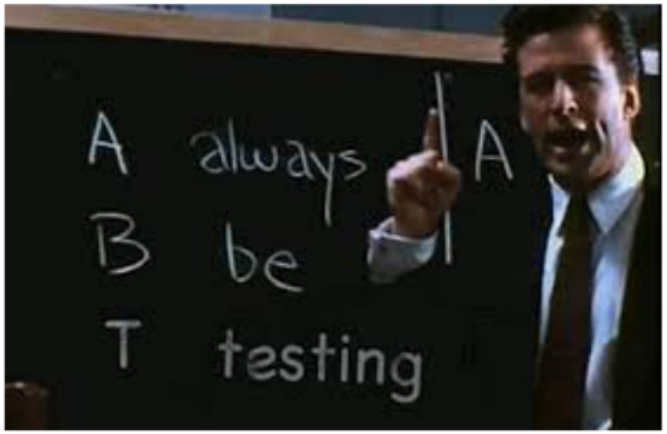
Make the test pass

Refactor

performance testing routes...

functional testing routes...

how to test Camel apps



- should be an ongoing effort
  - starts in early design
  - continues throughout development
  - ends with performance and acceptance tests

- types of tests
  - POC tests
    - standlone tests to validate routing logic
  - functional route tests
    - standup your real routes to validate flow
    - use adviceWith to inject mock endpoints/assertions
  - route performance tests
    - use camel-dataset to drive/gather stats
  - business logic tests
    - prefer Bean Integration to isolate/test w/o Camel
    - use a mock framework (Mockito, etc) to isolate

POC route testing...

```java
public class BasicRouteTest extends CamelTestSupport
{
    @Test
    public void test() throws Exception
    {
        MockEndpoint mockA = getMockEndpoint("mock:mock");
        mockA.setExpectedMessageCount(1);
        template.sendBody("direct:start", "message1");
        assertMockEndpointsSatisfied();
    }

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception
    {
        return new RouteBuilder()
        {
            @Override
            public void configure() throws Exception
            {
                from("direct:start").to("mock:mock");
            }
        };
    }
}
```

functional testing routes...

```java
public class MyRouteBuilder extends RouteBuilder
{
    public static final String INBOUND_DIRECT = "direct:inbound";
    public static final String INBOUND_QUEUE = "seda:received";

    @Override
    public void configure() throws Exception
    {
        from(INBOUND_DIRECT).routeId(INBOUND_DIRECT)
                .to(INBOUND_QUEUE);

        from(INBOUND_QUEUE).routeId(INBOUND_QUEUE)
                .process(new MyProcessor());
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
            http://camel.apache.org/schema/spring http://camel.apache.org/schema/s
            http://www.springframework.org/schema/beans http://www.springframework.

    <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
        <routeBuilder ref="myRouteBuilder"/>
    </camelContext>

    <bean id="myRouteBuilder" class="camel.webapp.MyRouteBuilder"/>
</beans>
```

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:/test-camel-context.xml" })
public class MyRouteBuilderTest
{
    @Autowired
    private ModelCamelContext camelContext;

    @DirtiesContext
    @Test
    public void testInboundEndpoint() throws Exception {

        camelContext.getShutdownStrategy().setTimeout(10);

        camelContext.getRouteDefinition(MyRouteBuilder.INBOUND_DIRECT).adviceWith(camelContext,
                new AdviceWithRouteBuilder()
        {
            @Override
            public void configure() throws Exception
            {
                interceptSendToEndpoint(MyRouteBuilder.INBOUND_QUEUE)
                        .skipSendToOriginalEndpoint()
                        .to("mock:received");
            }
        });

        String inboundMessage = "<message>message1</message>";
        camelContext.createProducerTemplate().sendBody(MyRouteBuilder.INBOUND_DIRECT, inboundMessage);

        MockEndpoint mock = (MockEndpoint) camelContext.getEndpoint("mock:received");
        mock.setExpectedMessageCount(1);
        mock.expectedBodiesReceived(inboundMessage);
        mock.assertIsSatisfied();
    }
}
```

```java
public class MyRouteBuilder extends RouteBuilder
{
    public static final String INBOUND_DIRECT = "direct:inbound";
    public static final String INBOUND_QUEUE = "seda:received";

    @Override
    public void configure() throws Exception
    {
        from(INBOUND_DIRECT).routeId(INBOUND_DIRECT)
                .to(INBOUND_QUEUE);

        from(INBOUND_QUEUE).routeId(INBOUND_QUEUE)
                .process(new MyProcessor());
    }
}
```

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:/test-camel-c
```

routes...

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
            http://camel.apache.org/schema/spring http://camel.apache.org/schema/sp
            http://www.springframework.org/schema/beans http://www.springframework.

    <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
        <routeBuilder ref="myRouteBuilder"/>
    </camelContext>

    <bean id="myRouteBuilder" class="camel.webapp.MyRouteBuilder"/>
</beans>
```

```
xt.xml" })
```

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:/test-camel-context.xml" })
public class MyRouteBuilderTest
{
    @Autowired
    private ModelCamelContext camelContext;

    @DirtiesContext
    @Test
    public void testInboundEndpoint() throws Exception {

        camelContext.getShutdownStrategy().setTimeout(10);

        camelContext.getRouteDefinition(MyRouteBuilder.INBOUND_DIRECT).adviceWith(camelContext,
                new AdviceWithRouteBuilder()
        {
            @Override
            public void configure() throws Exception
            {
                interceptSendToEndpoint(MyRouteBuilder.INBOUND_QUEUE)
                        .skipSendToOriginalEndpoint()
                        .to("mock:received");
            }
        });

        String inboundMessage = "<message>message1</message>";
        camelContext.createProducerTemplate().sendBody(MyRouteBuilder.INBOUND_DIRECT, inboundMessage);

        MockEndpoint mock = (MockEndpoint) camelContext.getEndpoint("mock:received");
        mock.setExpectedMessageCount(1);
        mock.expectedBodiesReceived(inboundMessage);
        mock.assertIsSatisfied();
    }
}
```

performance testing routes...

```java
public class MyRouteBuilder extends RouteBuilder
{
    public static final String INBOUND_DIRECT = "direct:inbound";
    public static final String INBOUND_QUEUE = "seda:received";

    @Override
    public void configure() throws Exception
    {
        from(INBOUND_DIRECT).routeId(INBOUND_DIRECT)
            .to(INBOUND_QUEUE);

        from(INBOUND_QUEUE).routeId(INBOUND_QUEUE)
            .process(new MyProcessor());
    }
}
```

```xml
<bean id="myDataSet" class="camel.webapp.MyDataSet">
    <property name="size" value="1000"/>
</bean>

<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="myRouteBuilder"/>

    <route>
        <from uri="dataset:myDataSet?produceDelay=-1"/>
        <to uri="direct:inbound"/>
    </route>

</camelContext>

<bean id="myRouteBuilder" class="camel.webapp.MyRouteBuilder"/>
```

```java
public class MyDataSet extends SimpleDataSet
{
    @Override
    public void populateMessage(Exchange exchange, long l) throws Exception
    {
        exchange.getIn().setBody("<message>" + l + "</message>");
    }
}
```

```
Apache Camel 2.12.3 (CamelContext: camelContext) started in 0.443 seconds
Sent: 200 messages so far. Last group took: 44 millis which is: 4,545.455 messages per second. average: 4,545.455
Sent: 400 messages so far. Last group took: 39 millis which is: 5,128.205 messages per second. average: 4,819.277
Sent: 600 messages so far. Last group took: 28 millis which is: 7,142.857 messages per second. average: 5,405.405
Sent: 800 messages so far. Last group took: 23 millis which is: 8,695.652 messages per second. average: 5,970.149
Sent: 1000 messages so far. Last group took: 23 millis which is: 8,695.652 messages per second. average: 6,369.427
Apache Camel 2.12.3 (CamelContext: camelContext) is shutting down
```

```
public class MyRouteBuilder extends RouteBuilder
{
    public static final String INBOUND_DIRECT = "direct:inbound";
    public static final String INBOUND_QUEUE = "seda:received";

    @Override
    public void configure() throws Exception
    {
        from(INBOUND_DIRECT).routeId(INBOUND_DIRECT)
                .to(INBOUND_QUEUE);

        from(INBOUND_QUEUE).routeId(INBOUND_QUEUE)
                .process(new MyProcessor());
    }
}
```

ng routes...

```
<bean id="myDataSet" class="camel.webapp.MyDataSet">
    <property name="size" value="1000"/>
</bean>

<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="myRouteBuilder"/>

    <route>
        <from uri="dataset:myDataSet?produceDelay=-1"/>
        <to uri="direct:inbound"/>
    </route>

</camelContext>

<bean id="myRouteBuilder" class="camel.webapp.MyRouteBuilder"/>
```

```
m(INBOUND_DIRECT).routeId(INBOUND_DIRECT)
.to(INBOUND_QUEUE);

m(INBOUND_QUEUE).routeId(INBOUND_QUEUE)
.process(new MyProcessor());
```

```
        <to uri="direct:inbound"/>
    </route>

</camelContext>

<bean id="myRouteBuilder" class="camel.webapp.MyRouteBuilder"/>
```

```
public class MyDataSet extends SimpleDataSet
{
    @Override
    public void populateMessage(Exchange exchange, long l) throws Exception
    {
        exchange.getIn().setBody("<message>" + l + "</message>");
    }
}
```

```
Apache Camel 2.12.3 (CamelContext: camelContext) started in 0.443 seconds
Sent: 200 messages so far. Last group took: 44 millis which is: 4,545.455 messages per second. average: 4,545.455
Sent: 400 messages so far. Last group took: 39 millis which is: 5,128.205 messages per second. average: 4,819.277
Sent: 600 messages so far. Last group took: 28 millis which is: 7,142.857 messages per second. average: 5,405.405
Sent: 800 messages so far. Last group took: 23 millis which is: 8,695.652 messages per second. average: 5,970.149
Sent: 1000 messages so far. Last group took: 23 millis which is: 8,695.652 messages per second. average: 6,369.427
Apache Camel 2.12.3 (CamelContext: camelContext) is shutting down
```

Deploy

deployment options...

- embedded in existing application
  - bootstrap CamelContext (Spring, Main API)
- standalone

```
main = new Main();
main.addRouteBuilder(new MyRouteBuilder());
```

- maven (dev mode)
  - maven-camel-plugin
  - maven-jetty-plugin
- in ActiveMQ
  - ships with Camel
  - simple deployment
    - add JAR to /lib
    - bootstrapp in activemq.xml
    - add routes to camel.xml

deployment options...

- in a web container (Tomcat, Jetty, etc)
  - add a /WEB-INF/web.xml
  - load the Spring ContextLoaderListener
  - add camel.xml in contextConfigLocation param
  - build a war, deploy to /tomcat/webapps
- in OSGi (Karaf, ServiceMix, etc)
  - leverage OSGi's lightweight/explicit classloading
  - build bundle (jar with an OSGi manifest file)
  - the maven-bundle-plugin will generate the bundle
  - more complex and powerful than others deployment options
  - well suited for
    - deploying multiple applications in the same JVM
    - complex deployments (clustering, HA, etc) using Fuse Fabric or Zookeeper

how to monitor

- consider functional use cases
  - need to manage/monitor at runtime?
    - JMX to stop/start routes, gather stats
  - need to view/triage exceptions at runtime?
    - use JMS error queues for visibility/durability
    - use JMX to copy/move/delete messages
- consider non functional use cases
  - system level monitoring
    - JVM stats from JMX
    - 3rd party tools (Nagios, AppDynamics)
  - error monitoring/handling
    - Camel onException() -> JMS error queues
- monitoring tools
  - VisualVM (JMX)
  - ActiveMQ web console (browse queues)
  - Hawt.io web console (AMQ/Camel plugins)

deployment options...

- embedded in existing application
  - bootstrap CamelContext (Spring, Main API)
- standalone

```
main = new Main();
main.addRouteBuilder(new MyRouteBuilder());
```

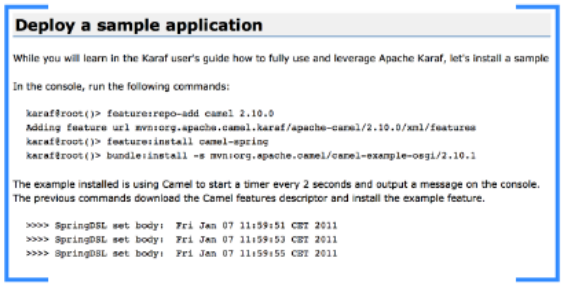
- maven (dev mode)
  - maven-camel-plugin
  - maven-jetty-plugin
- in ActiveMQ
  - ships with Camel
  - simple deployment
    - add JAR to /lib
    - bootstrapp in activemq.xml
    - add routes to camel.xml

deployment options...

- in a web container (Tomcat, Jetty, etc)
  - add a /WEB-INF/web.xml
  - load the Spring ContextLoaderListener
  - add camel.xml in contextConfigLocation param
  - build a war, deploy to /tomcat/webapps

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>My Web Application</display-name>

  <!-- location of spring xml files -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camel-config.xml</param-value>
  </context-param>

  <!-- the listener that kick-starts Spring -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- Camel servlet -->
  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Camel servlet mapping -->
  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/camel/*</url-pattern>
  </servlet-mapping>

</web-app>
```

- in OSGi (Karaf, ServiceMix, etc)
  - leverage OSGi's lightweight/explicit classloading
  - build bundle (jar with an OSGi manifest file)
  - the maven-bundle-plugin will generate the bundle
  - more complex and powerful than others deployment options
  - well suited for
    - deploying multiple applications in the same JVM
    - complex deployments (clustering, HA, etc) using Fuse Fabric or Zookeeper



Deploy a sample application

While you will learn in the Karaf user's guide how to fully use and leverage Apache Karaf, let's install a sample

In the console, run the following commands:

```
karaf@root()> feature:repo-add camel 2.10.0
Adding feature url mvn:org.apache.camel.karaf/apache-camel/2.10.0/xml/features
karaf@root()> feature:install camel-spring
karaf@root()> bundle:install -s mvn:org.apache.camel/camel-example-osgi/2.10.1
```

The example installed is using Camel to start a timer every 2 seconds and output a message on the console. The previous commands download the Camel features descriptor and install the example feature.

```
>>>> SpringDSL set body:  Fri Jan 07 11:59:51 CET 2011
>>>> SpringDSL set body:  Fri Jan 07 11:59:53 CET 2011
>>>> SpringDSL set body:  Fri Jan 07 11:59:55 CET 2011
```

```xml
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>My Web Application</display-name>

  <!-- location of spring xml files -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camel-config.xml</param-value>
  </context-param>

  <!-- the listener that kick-starts Spring -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- Camel servlet -->
  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Camel servlet mapping -->
  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/camel/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Deploy a sample application

While you will learn in the Karaf user's guide how to fully use and leverage Apache Karaf, let's install a sample

In the console, run the following commands:

```
karaf@root()> feature:repo-add camel 2.10.0
Adding feature url mvn:org.apache.camel.karaf/apache-camel/2.10.0/xml/features
karaf@root()> feature:install camel-spring
karaf@root()> bundle:install -s mvn:org.apache.camel/camel-example-osgi/2.10.1
```

The example installed is using Camel to start a timer every 2 seconds and output a message on the console. The previous commands download the Camel features descriptor and install the example feature.

```
>>>> SpringDSL set body:  Fri Jan 07 11:59:51 CET 2011
>>>> SpringDSL set body:  Fri Jan 07 11:59:53 CET 2011
>>>> SpringDSL set body:  Fri Jan 07 11:59:55 CET 2011
```

how to monitor

- consider functional use cases
  - need to manage/monitor at runtime?
    - JMX to stop/start routes, gather stats
  - need to view/triage exceptions at runtime?
    - use JMS error queues for visibility/durability
    - use JMX to copy/move/delete messages

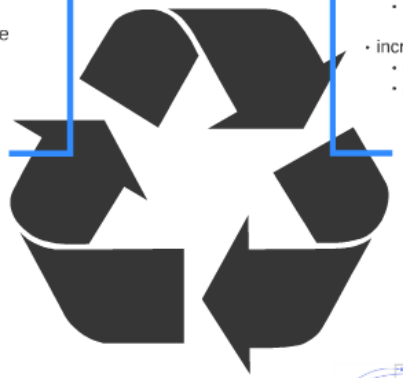
- consider non functional use cases
  - system level monitoring
    - JVM stats from JMX
    - 3rd party tools (Nagios, AppDynamics)
  - error monitoring/handling
    - Camel onException() -> JMS error queues

- monitoring tools
  - VisualVM (JMX)
  - ActiveMQ web console (browse queues)
  - Hawt.io web console (AMQ/Camel plugins)

Legacy Apps?

refactoring considerations

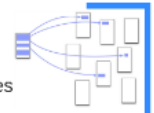
- first, the motivation
  - have specific improvements in mind
  - be prepared to justify the ROI
    - more so than a new application
  - have a clear roadmap defined
    - how is Camel advancing the cause
- next, how invasive the refactoring needs to be
  - should rarely have to scrap and rebuild
  - consider what the end product requirements are
  - perhaps just wrap legacy app with Camel
    - expose new interfaces to old code
      - JMS, WebServices, REST, etc.
    - route enable, rewire existing code



refactor incrementally...

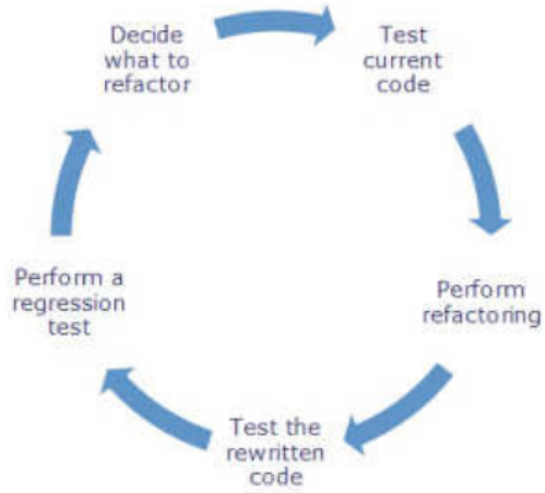
- focus on primary goals first
  - don't snowball it
  - isolate changes to legacy code
  - don't over think exception handling up front
- keep changes/scope to a minimum
  - bootstrap Camel within existing app (Spring, etc)
  - introduce a lightweight container (Tomcat, etc)
  - run standalone during early stages of dev
- incrementally expand scope
  - add timer routes for batch jobs
  - add routes to expose legacy code
    - use bean binding to route enable w/o changes

refactor incrementally...



- refactor event based processes into routes
  - decouple asynchronous operations
    - increases scalability, throughput, flexibility
  - add persistent queues (JMS, etc) if necessary
- monitoring
  - add early to add visibility to your application
    - JMX/VisualVM - raw stats
    - hawt.io - html5 console for AMQ, Camel, etc
  - much easier to improve apps that can be visualized
- consider deployment options early
  - restricted to current container?
  - how will you scale horizontally?

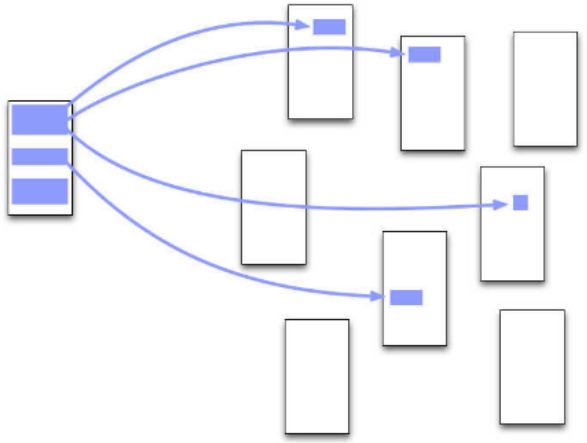
refactoring considerations



- first, the motivation
  - have specific improvements in mind
  - be prepared to justify the ROI
    - more so than a new application
  - have a clear roadmap defined
    - how is Camel advancing the cause

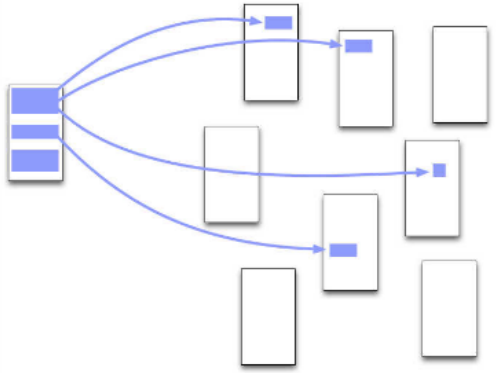
- next, how invasive the refactoring needs to be
  - should rarely have to scrap and rebuild
  - consider what the end product requirements are
  - perhaps just wrap legacy app with Camel
    - expose new interfaces to old code
      - JMS, WebServices, REST, etc.
    - route enable, rewire existing code

refactor incrementally...



- focus on primary goals first
  - don't snowball it
  - isolate changes to legacy code
  - don't over think exception handling up front
- keep changes/scope to a minimum
  - bootstrap Camel within existing app (Spring, etc)
  - introduce a lightweight container (Tomcat, etc)
  - run standalone during early stages of dev
- incrementally expand scope
  - add timer routes for batch jobs
  - add routes to expose legacy code
    - use bean binding to route enable w/o changes

refactor incrementally...



- refactor event based processes into routes
  - decouple asynchronous operations
    - increases scalability, throughput, flexibility
  - add persistent queues (JMS, etc) if necessary

- monitoring
  - add early to add visibility to your application
    - JMX/VisualVM - raw stats
    - hawt.io - html5 console for AMQ, Camel, etc
  - much easier to improve apps that can be visualized

- consider deployment options early
  - restricted to current container?
  - how will you scale horizontally?

Q/A

[obligatory camel pics]

Navigating Options

Components

Patterns

Common Use Cases

Lifecycle

Develop

Test

Deploy

Design

Legacy Apps?

Q/A

[obligatory camel pics]

Camel Intro

Why Use Camel

Apache Camel Business Patterns

[my notes on Camel]

presented by Ben O'Day

Apache Camel

Prezi