

# Apache Camel

## Specification

Provides API , standards, recommended practices, codes and technical publications, reports and studies.

JCP - Java Community Process

JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)

The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)

The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)

Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa

JSR 317 – JPA 2    JSR 318 – EJB 3.1

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws        JSR 311 - Jax-RS

JSR 299 - Context & DI    JSR 330 – DI

JSR 294: Improved Modularity Support in the Java

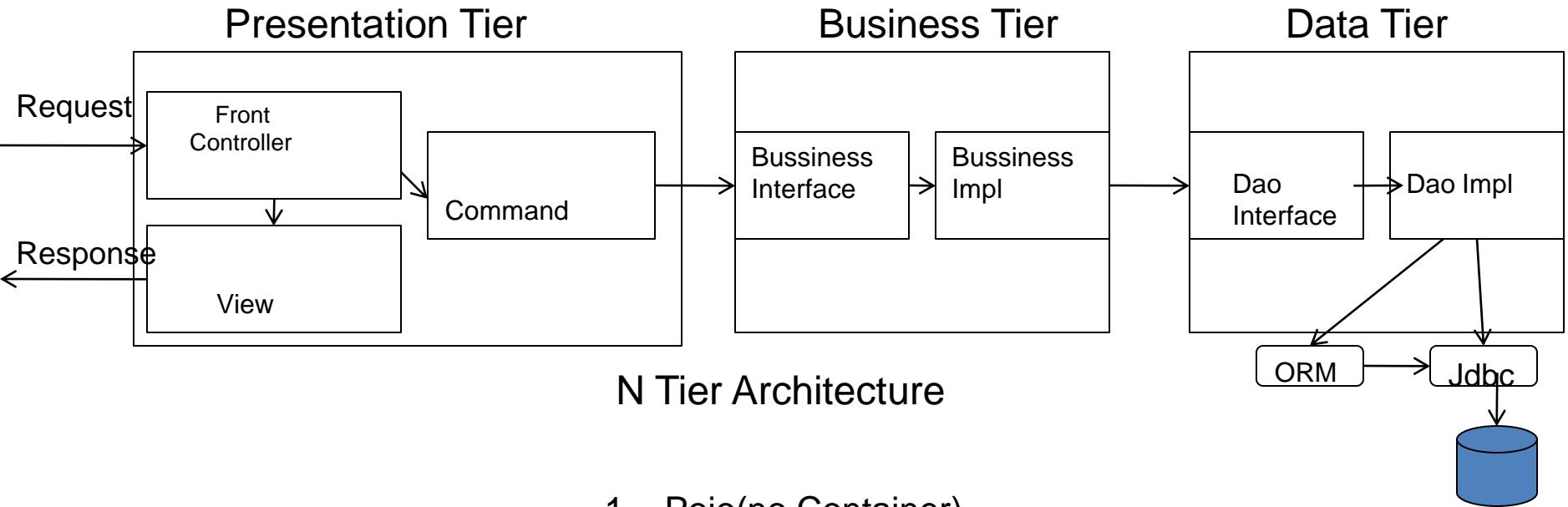
JSR-107 (JCACHE) compatible Cache interface

JSR 170- Message Store

JSR208,312 – JBI (Java Business Integration)

JSR 94 - Java Rule Engine API

OMG (Object Management Group) - BPMN



- 1. Servlet/jsp
- 2. MVC
- Struts
- JSF
- Flex
- Gwt
- Spring MVC
- ...

> Any MVC + Spring

1. Pojo(no Container)
2. Ejb 2.x(HW Container)
  - Session Bean
  - Mdb
3. Pojo + LW Container
  - Spring
  - Microcontainer
  - Xwork
4. Ejb3.x

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
  - Hibernate
  - Kodo
  - Toplink
  - MyBatis
5. JPA (JavaSE/JavaEE)
  - + Spring Templates

# SOA

Service Oriented Architecture (SOA) is an architectural style for implementing business processes as a set of loosely-coupled services.

SOA:

Is an architectural style

Focuses on business processes

Exposes application functionality as services

Uses loose coupling

## Characteristics of a Service

Loosely Coupled

Coarse Grained

Message Oriented

Standards Based

Location Transparent

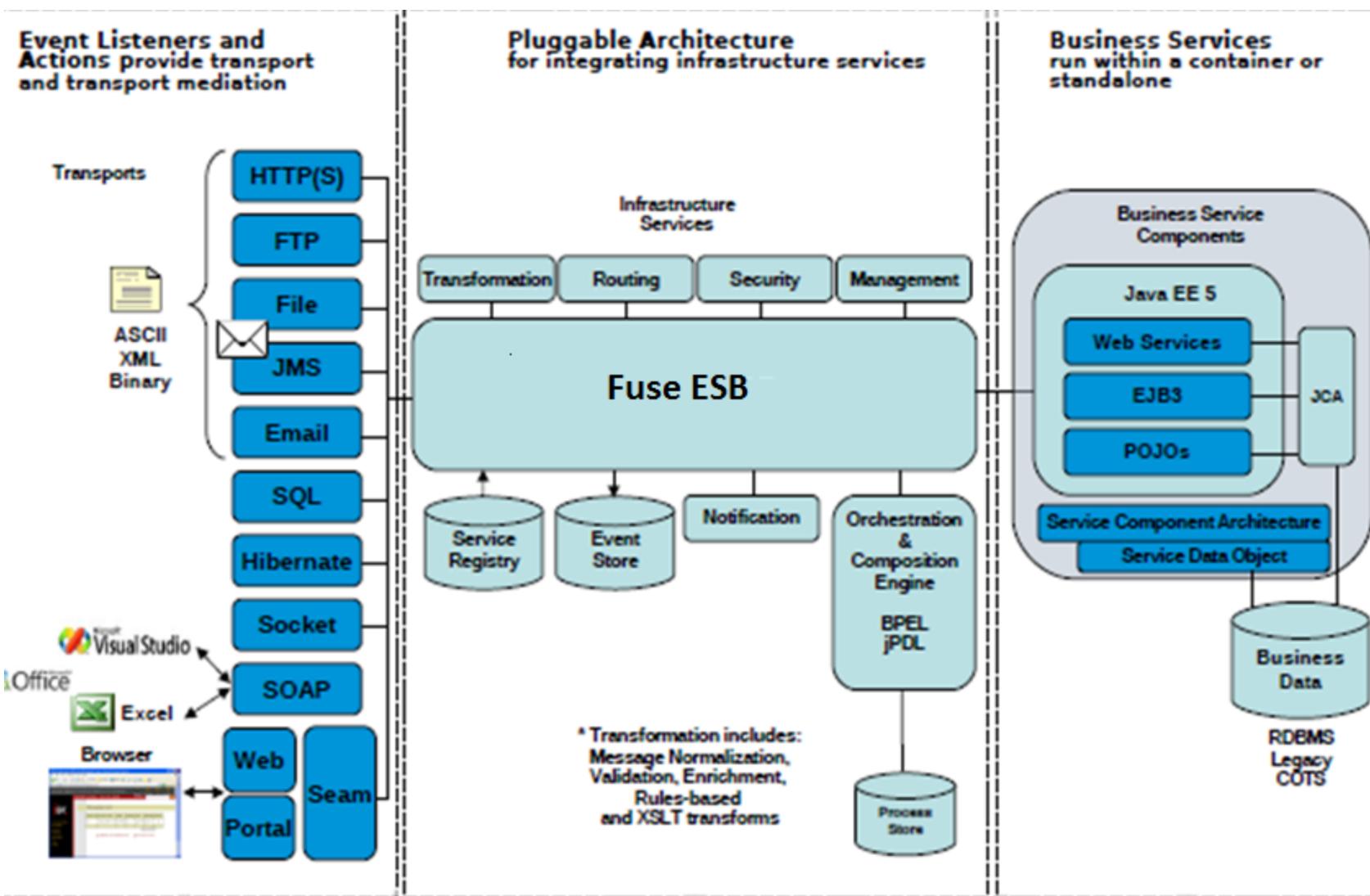
Reusable

Contract Based

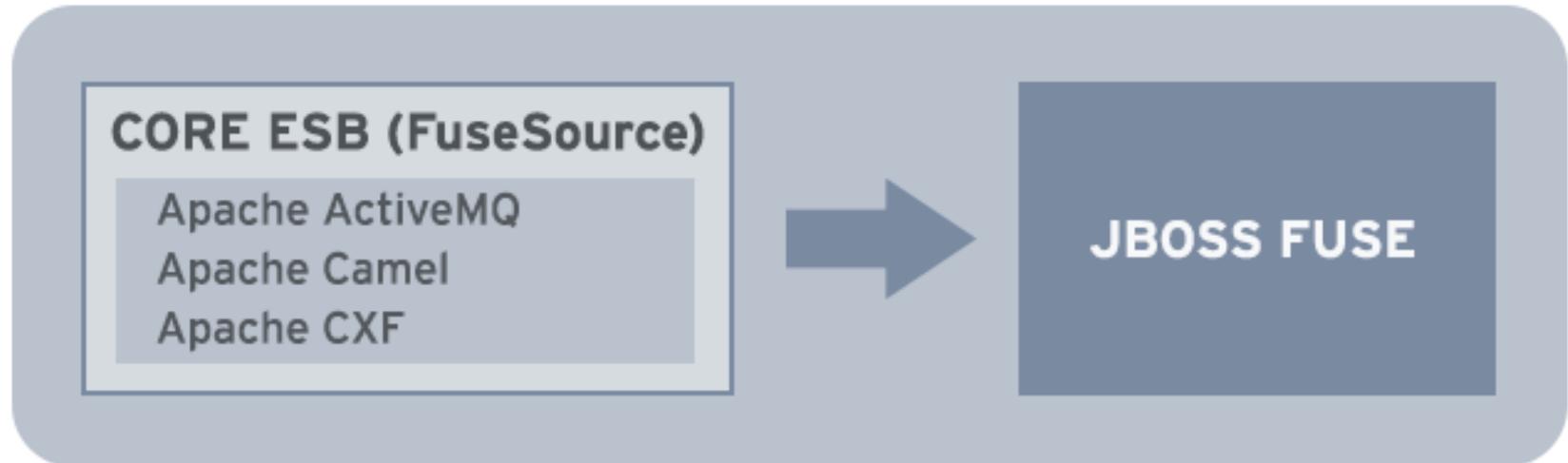
## **Fuse ESB – What is it?**

Fuse ESB provides a service oriented architecture (SOA) platform to integrate business services into automated business processes through the use of messaging, transformation, registries, and related services.

# Enterprise Service Bus



JBoss Fuse combines several technologies like core Enterprise Service Bus capabilities (based on Apache Camel, Apache CXF, Apache ActiveMQ), Apache Karaf and Fabric8 in a single integrated distribution.



Apache Camel

Compose your applications from Enterprise Integration Patterns (EIPs) based on the popular Hohpe and Woolf EIPs.

Apache CXF

Integrate applications with SOAP, XML/HTTP and RESTful HTTP.

Apache ActiveMQ

Provides core messaging within the ESB and for integrating with other applications.

Apache Karaf

Offers a lightweight OSGI-based runtime container for managing the components that compose your applications.

Fabric8

Makes it simple to manage large and distributed, JBoss Fuse deployments from a central location.

# JBoss Fuse

Open source implementation of an Enterprise Service Bus

- Application container for components
- Based on OSGi technology (<http://www.osgi.org>)

## Support for multiple component models

- OSGi bundles
- Spring beans
- Servlets
- Camel routes (<http://camel.apache.org>)
- CXF web services and RESTful services (<http://cxf.apache.org>)
- JBI artifacts
- Extensible support for additional component models, such as EJB

Powerful/extensible management console and command shell

- Based on metrics from JMX MBeans

## **Fuse is Based on Apache ServiceMix**

The ServiceMix project is hosted by Apache

- <http://servicemix.apache.org>

Certified, tested, and distributed by Red Hat

- Each new release is copied from Apache
- New branch created: <apache release>-fuse-00-00
- Tested against customer use-cases and third party products
- Certified and re-released for download from:

<http://www.redhat.com/products/jbossembedded>

## JBoss Fuse architecture

Employs a layered architecture based on OSGi

- **The core** : Karaf - lightweight runtime container
  - Extends OSGi with features for handling/managing OSGI bundles
- **The technology layer** : Layer of component technologies
  - Plugs into the core to support the needs of your applications

JMS

JAX-WS

JAX-RS

Camel

Spring

JBIG

Technology Layer

Core

Console

Logging

Deployer

Provisioning

Admin

SpringDM

OSGi

Karaf

## Core features

Hot deployment support for OSGi bundles

Dynamic configuration of services

- Through OSGi “Configuration Admin” service

Dynamic logging back-end provided by Log4J

- Supports different APIs
  - (SLF4J, Java Utils, JCL, Avalon, Tomcat, OSGi)

Application provisioning through

- File-drop
- Maven repository
- Remote download (<http://>)

Extensible administration shell console

Secure remote access via ssh

Security framework based on JAAS

## **Technology layer features (1)**

### **Spring Framework**

When JBoss Fuse/ESB loads an OSGi bundle into its runtime  
(or)

Generates an OSGi bundle on the fly,

Then

JBoss Fuse instantiates the Spring application context

### **JMS Message Broker – Messaging Middleware**

- JBoss Fuse can deploy the Apache ActiveMQ broker (OSGi-ready)

### **JAX-WS/JAX-RS - Web Services support**

- JBoss Fuse can deploys the Apache CXF runtime (OSGiready)

**The OSGi Blueprint Container** specification allows us to use dependency injection in our OSGi environment, declarative import and export of OSGi services, registering lifecycle listeners and wiring dependencies into our services with a few lines of XML code.

Ex : Apache Aries, Gemini Blueprint

## **Technology layer features (2)**

### **Enterprise Integration Pattern (EIP) support**

- JBoss Fuse deploys the Apache Camel runtime (OSGi-ready)

Bean-based OSGi services and distributed OSGi services supported

### **Java Business Integration (JBI) support**

- JBoss Fuse provides a JBI 1.0 container, to support legacy code implemented as service units / service assemblies and deployed using JBoss Fuse

Can be extended to support integration technologies such as SCA or EJB3

## Using JBoss Fuse / ESB

Using JBoss Fuse in your development, deployment and production processes:

- 1) Start with Camel to solve your integration problem using the Fuse IDE to develop/test. May also include JBoss A-MQ for messaging and CXF for Web/Rest services
- 2) Deploy your Camel application into a JBoss Fuse instance (aka container)
- 3) Provision/manage multiple containers within your enterprise and centrally manage with Fuse Fabric through the Fuse Management Console

# Deploying a JBoss Fuse solution

Typically

- Deploy the core (Karaf)
- Plus one or more technology components

Simple Examples:

- To support EIPs : Karaf runtime + Camel feature
- To support JAX-WS/JAX-RS solutions : Karaf runtime + CXF feature
- To support JMS solutions : Karaf runtime + ActiveMQ feature
- To support JBI-based solutions : Karaf runtime + JBI feature

## **Service Component Architecture (SCA)**

- ✓ OASIS standard for creating service-oriented applications
  - Describe
  - Assemble
  - Build
- ✓ Defines how to create components and how to combine those components into larger structures called composites.
- ✓ Set of specifications for building applications

## Fuse Service Works

Fuse Service Works is a lightweight service delivery framework providing full lifecycle support for developing, deploying, and managing service-oriented applications.

- Based on SwitchYard
- Leverages standards - SCA, BPEL, BPMN2, WebServices, CDI, etc.
- Incorporates complimentary technologies - Apache Camel
- Supported via Red Hat subscription model, includes:
  - SwitchYard (service composition framework)
  - Camel (Enterprise Service Bus)
  - Riftsaw (BPEL)
  - S-RAMP (Design Time Governance)
  - Run Time Governance
  - Integration components for jBPM6 and Drools

# What is OSGi Technology

- It's a module system for the Java platform
- It's dynamic
- It's service-oriented
- A specification of the OSGi Alliance, a non-profit organization

# Specifications

- OSGi Release 1 (R1): May 2000
- OSGi Release 2 (R2): October 2001
- OSGi Release 3 (R3): March 2003
- OSGi Release 4 (R4): October 2005 / September 2006
- OSGi Release 5 (R5): June 2012

# Specifications

- OSGi framework
- Standard service definitions
  - Log Service
  - Http Service
  - Device Service
  - Package Administration Service
  - Permission Administration Service
  - Configuration Administration Service
  - Preferences Service
  - User Administration Service

# Key Benefits

- **Multiple Service Support:**  
OSGi environments should be capable of hosting multiple applications from different Service Providers on a single Service Platform with each application providing an independent set of services to the end user.
- **Service Collaboration Support:**  
An important aspect of the OSGi deployment model is that it allows the core platform to be extended with deployed services. This is not limited to end user oriented services only, which is a limitation of some other deployment models.

# Key Benefits

- **Multiple Network Technology Support:**  
OSGi Service Platforms can work with wide area technologies like xDSL, Cable modems, Satellite, ISDN and PSTN and local area networks like Bluetooth, USB, IEEE 1394 Firewire.
- Popular service discovery techniques like UPnP, Jini, Salutation and several others work very well in conjunction with an OSGi Service Platform and can even potentially interwork transparently.

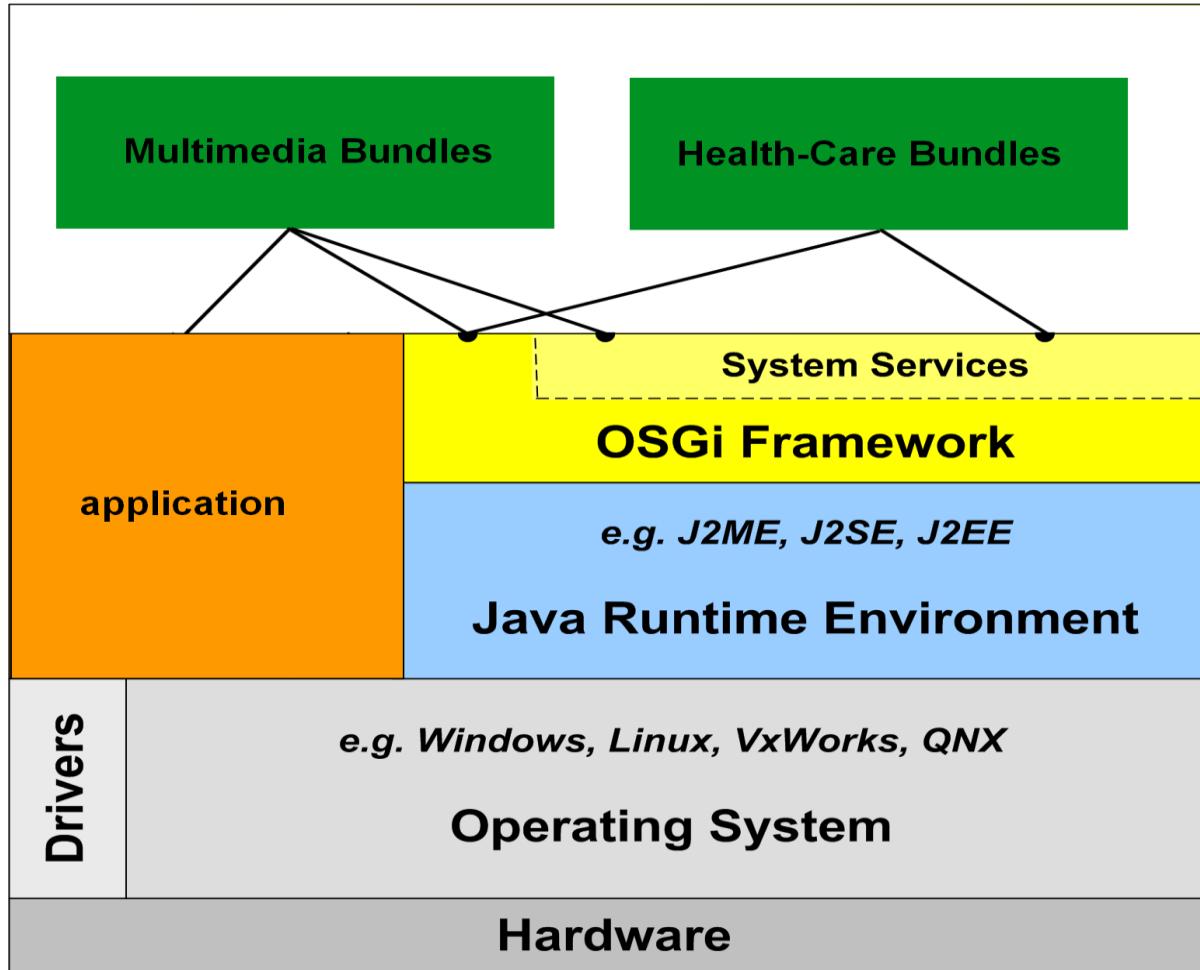
# Key Benefits

- **Security:**  
The OSGi offers a fine grained security architecture that limits the potential harm a malicious or badly written application can do.
- **Simplicity:**  
The OSGi environment offers a service environment for everybody by removing much of the complexity and putting it into the hands of professionals. The environment can be remotely managed by a professional organization, if so desired.

# OSGi Framework

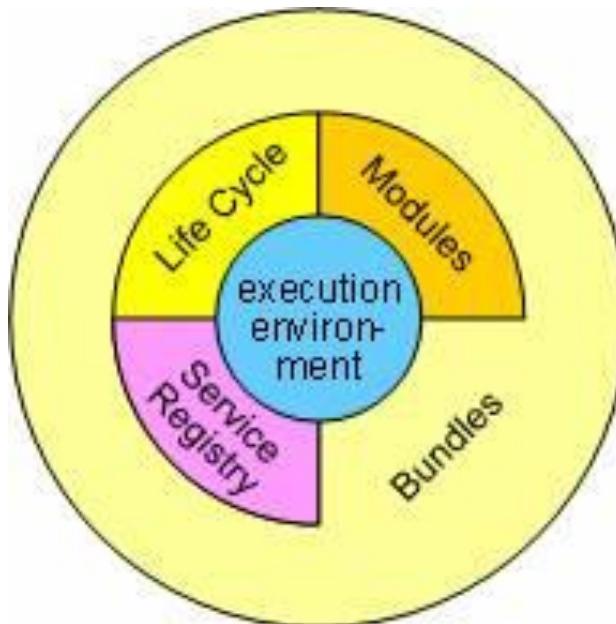
- Services gateway
- Generic application framework
- Lightweight framework
  - Simple component model
  - Service registry
  - Support for deployment

# OSGi Framework



# OSGi Framework

- The Framework is divided in a number of layers: Execution Environment, Modules, Life Cycle Management and Service Registry. Additionally, there is a security system that is deeply intertwined with all the layers.



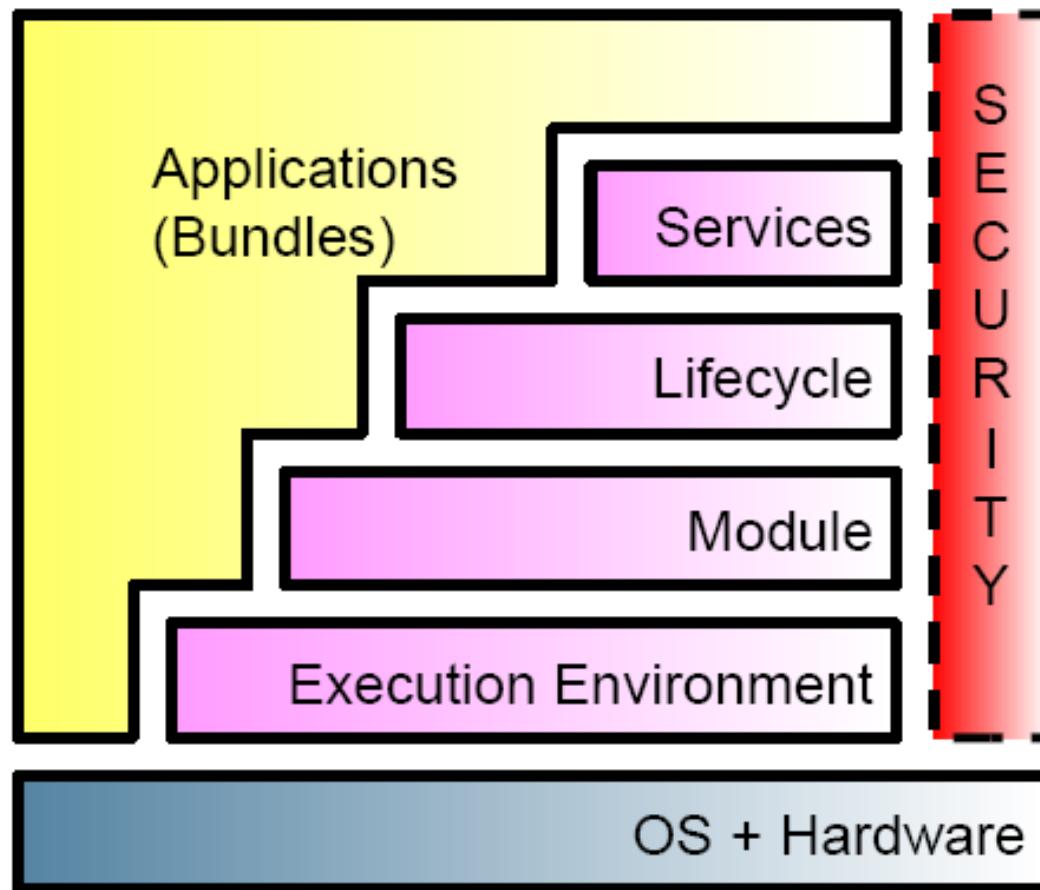
# OSGi Framework

- **Execution environment:**  
the specification of the Java environment. Java 2 Configurations and Profiles.
- **Modules:**  
defines the class loading policies. The OSGi Framework is a powerful and strictly specified class loading model. It is based on top of Java but adds modularization.
- **Life Cycle Management:**  
adds bundles that can be dynamically installed, started, stopped, updated, and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in run time.

# OSGi Framework

- **Service Registry:**  
The service registry provides a cooperation model for bundles that takes the dynamics into account. The service registry provides a comprehensive model to share objects between bundles. A number of events are defined to handle the coming and going of services.
- **Security** is based on Java and the Java 2 security model.

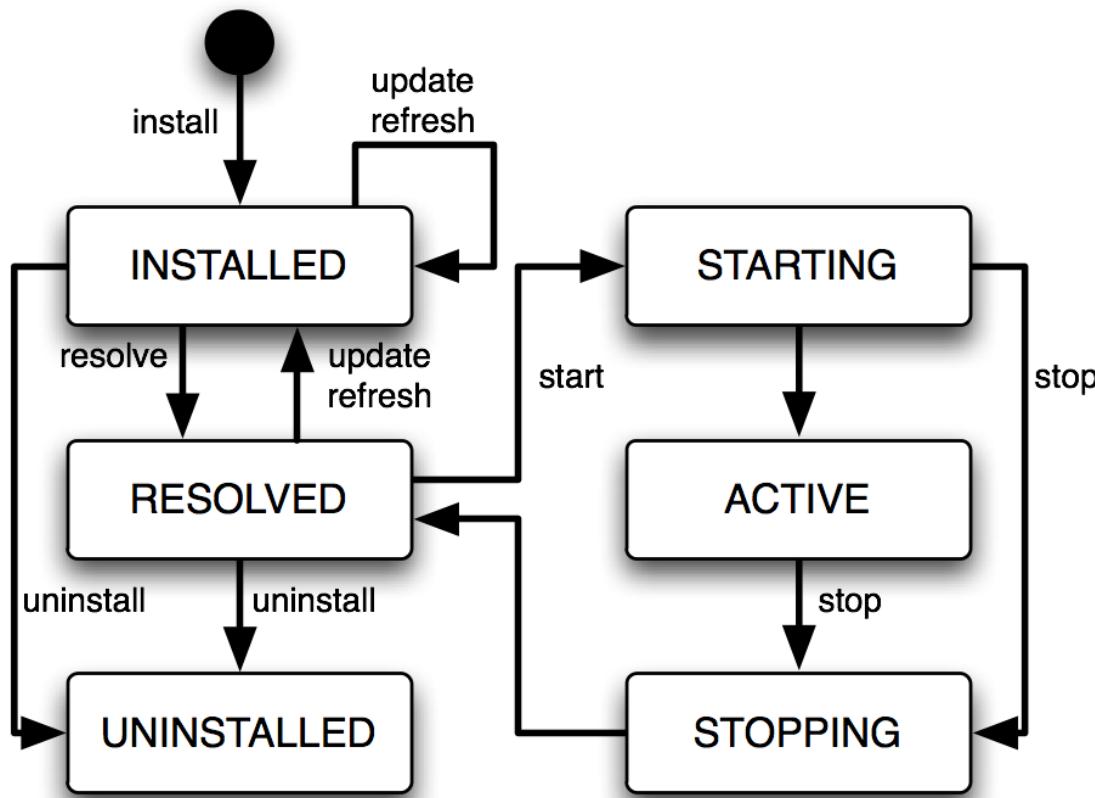
# OSGi Framework

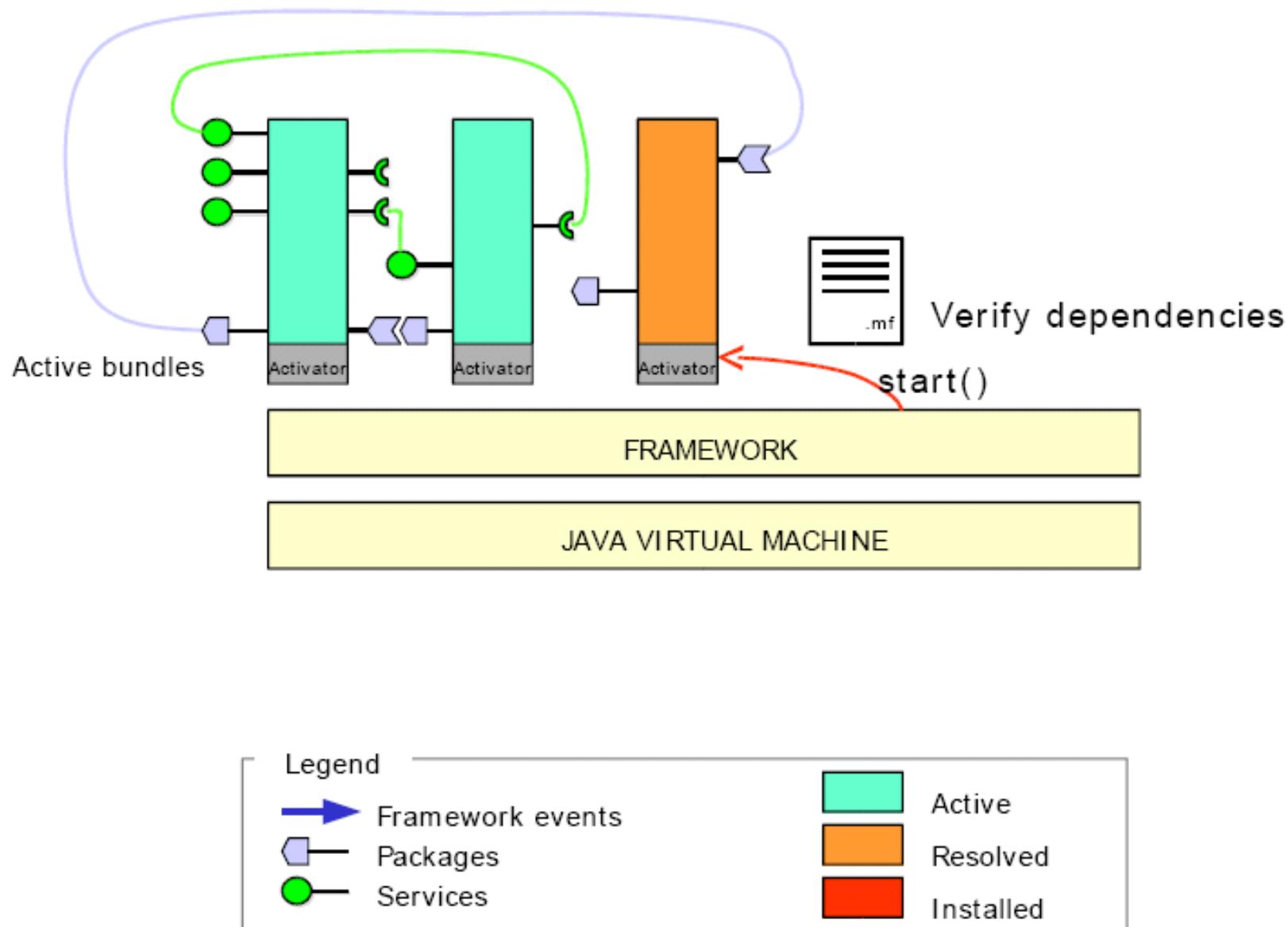


# Service & Bundle

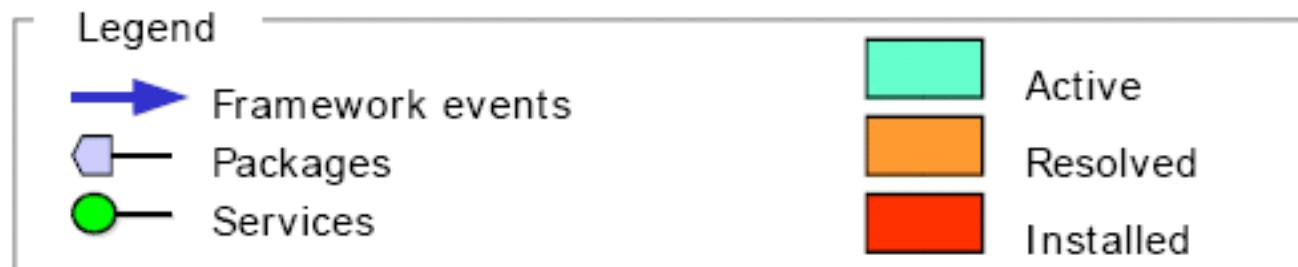
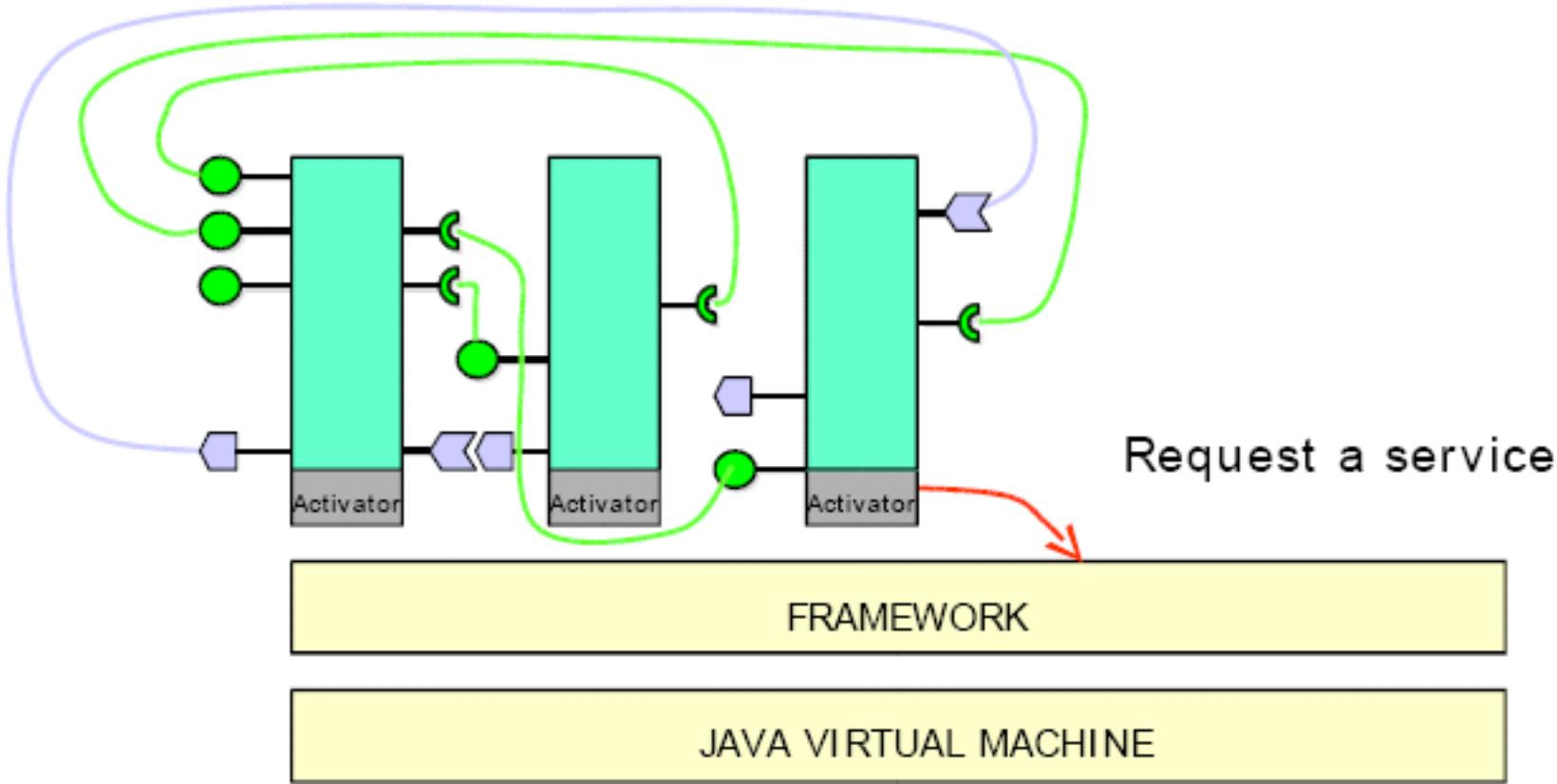
- Services
  - Provide applications
- Bundles (JAR file)
  - Provide (export) and reuse (import) services via the framework
  - Identify Java packages (classes)
  - Implement specified interface (services)
  - Register services with the Service Registry

# Bundle Life Cycle









# Bundle Activator

- Activator Class ([implements BundleActivator](#))
  - start()
    - This method implements what should the bundle do when it starts.  
The same as main() method in common Java program.
  - stop()
    - This method implements what should the bundle do when it is going to stop.

# A Simple Bundle Implement

```
public class Activator implements BundleActivator
{
    public void start(BundleContext context)
    {
        System.out.println("Hello World.");
    }
    public void stop(BundleContext context)
    {
        System.out.println("Bye World.");
    }
}
```

# Service Export

- Activator Class ([implements BundleActivator](#))
  - start()
    - We have to implement a class and a service method for this class, and register this service in the start() method.
    - `BundleContext.registerService` (`java.lang.String` class, `java.lang.Object` service, `java.util.Dictionary` properties)
  - stop()
    - Some services should be unregistered before the bundle stopped.
    - `ServiceRegistration.unregister()`

# Service Export

```
public void start(BundleContext context)
    throws Exception
{
.....
    this.logReg = this.context.registerService(
        LogService.class.getName(), this.logServ, null);
    this.readerReg = this.context.registerService(
        LogReaderService.class.getName(), this.readerServ, null);
.....
}
```

# Service Export

- Manifest file

Bundle-Name: Log Service

Bundle-Description: OSGi compliant log service.

Bundle-Version: 1.0.0

.....

Import-Package: org.osgi.framework

**Export-Package:** org.osgi.service.log; specification-version=1.1

**Export-Service:** org.osgi.service.log.LogService,  
org.osgi.service.log.LogReaderService

# Service Import

```
public void start(BundleContext context) throws Exception
{
    .....
    m_context.addServiceListener(this,
        "(&(objectClass=" + DictionaryService.class.getName() + ")" +
        "(Language=*))");
    // Query for any service references matching any language.
    ServiceReference[] refs = m_context.getServiceReferences(
        DictionaryService.class.getName(), "(Language=*)");
    .....
}
```

# Service Import

```
public void serviceChanged(ServiceEvent event)
{
    switch (event.getType())
    {
        case ServiceEvent.REGISTERED:
            registerServlet();
            break;

        case ServiceEvent.UNREGISTERING:
            unregisterServlet();
            break;
    }
}
```

# Service Import

- Manifest file

Bundle-Name: Homebox Logger

.....

Bundle-Activator: ismp.homebox.Activator

Bundle-Description: collect bundle information

Import-Package: org.osgi.framework,  
org.osgi.service.http,  
javax.servlet,  
javax.servlet.http,  
org.ungoverned.osgi.service.shell

**osgi:install** means install a bundle from a file or stream, and it maps to the `BundleContext.installBundle` method in the API.

**osgi:refresh** performs a "refresh packages" operation, which allows exports/imports to be rewired after installing or updating a set of bundles. For example, bundles that are currently wired to a particular exporter of a package may be rewired to a newly installed bundle that exports the same package

**osgi:resolve** is similar to refresh, but it only wires up bundles that are currently in the INSTALLED state. I.e. it will not rewire existing wires belonging to bundles that are already in the RESOLVED state.

**osgi:restart** stops and restarts a specific bundle. This does not cause the bundle implementation to be updated, it simply stops and starts.

**osgi:update** requests for a single bundle to be updated (i.e. reloaded from its original location). This may involve stopping, re-resolving and starting the bundle, depending on what state it was in before the update.

## **OSGI Bundles - Advanced :**

If a bundle imports (Import-Package) a specific package, that package must be made available by another bundle's exports (Export-Package).

If bundle A has Import-Package: org.apache.foo then there must be a bundle deployed that has an Export-Package: org.apache.foo

For every Import-Package package declaration, there must be a corresponding Export-Package with the same package

Bundles can also attach other attributes to the packages it imports or exports. What if we added a version attribute to our example:

Bundle-Name: Bundle A

Import-Package: org.apache.foo;version="1.2.0"

This means, Bundle A has a dependency on package org.apache.foo with a minimum version of 1.2.0. Although with OSGI you can specify a range of versions, if you don't specify a range but rather use a fixed version, it will result in a meaning of "a minimum" of the fixed value. If there is a higher version for that same package, the higher version will be used. So bundle A will not resolve correctly unless there is a corresponding bundle B that exports the required package:

Bundle-Name: Bundle B

Export-Package: org.apache.foo;version="1.2.0"

Import-Package dictates exactly what version (or attribute) it needs, and a corresponding Export-Package with the same attribute must exist

What happens if you have a scenario where Bundle A imports a package and it specifies a version that is provided by two bundles:

Bundle-Name: Bundle A

Import-Package: org.apache.foo;version="1.2.0"

Bundle-Name: Bundle B

Export-Package: org.apache.foo;version="1.2.0"

Bundle-Name: Bundle C

Export-Package: org.apache.foo;version="1.2.0"

## **Which one bundle does Bundle A use?**

The answer is it depends on which bundle (B or C) was installed first.

Bundles installed first are used to satisfy a dependency when multiple packages with the same version are found :

Things can get a little more complicated when hot deploying bundles after some have already been resolved. What if you install Bundle B first, then try to install Bundle A and the following Bundle D together:

Bundle-Name: Bundle D

Export-Package: org.apache.foo;version="1.3.0"

As we saw from above, the version declaration in Bundle A (1.2.0) means a minimum version of 1.2.0; so if a higher version was available then it would select that (version 1.3.0 from Bundle D in this case). However, that brings us to another temporal rule for the bundle resolution:

**Bundles that have already been resolved have a higher precedence than those not resolved :**

The reason for this is the OSGI framework tends to favor reusability for a given bundle. If it's resolved, and new bundles need it, then it won't try to have many other versions of the same package if it doesn't need to.

## Bundle “uses” directive

The above rules for bundle resolution are still not enough and the wrong class could still be used at runtime resulting in a class-cast exception or similar. Can you see what could be missing?

What if we had this scenario. Bundle A exports a package, org.apache.foo, that contains a class, FooClass. FooClass has a method that returns an object of type BarClass, but BarClass is not defined in the bundle’s class space, it’s imported like this:

```
public class FooClass {  
    public BarClass execute(){ ... }  
}
```

Bundle-Name: Bundle A  
Import-Package: org.apache.bar;version="3.6.0"  
Export-Package: org.apache.foo;version="1.2.0"

So far everything is fine as long as there is another bundle that properly exports org.apache.bar with the correct version.

Bundle-Name: Bundle B

Export-Package: org.apache.bar;version="3.6.0"

These two bundles will resolve fine. Now, if we install two more bundles, Bundle C and Bundle D that look like this:

Bundle-Name: Bundle C

Import-Package: org.apache.foo;version="1.2.0", org.apache.bar;version="4.0.0"

Bundle-Name: Bundle D

Export-Package: org.apache.bar;version="4.0.0"

We can see that Bundle C imports a package, org.apache.foo from Bundle A. Bundle C can try to use FooClass from org.apache.foo, but when it gets the return value, a type of BarClass, what will happen? Bundle A expects to use version 3.6.0 of BarClass, but bundle C is using version 4.0.0. So the classes used are not consistent within bundles at runtime (i.e., you could experience some type of mismatch or class cast exception), but everything will still resolve just fine at deploy time following the rules from above.

What we need is to tell anyone that imports org.apache.foo that we use classes from a specific version of org.apache.bar, and if you want to use org.apache.foo you must use the same version that we import. That's exactly what the uses directive does. Let's change bundle A to specify exactly that:

Bundle-Name: Bundle A

Import-Package: org.apache.bar;version="3.6.0"

Export-Package: org.apache.foo;version="1.2.0";uses:=org.apache.bar

# Export Jar

- Eclipse Export
- Install Bundle
- Uninstall Bundle

## Apache Felix Karaf

Karaf started life as the ServiceMix Kernel

- Moved to a sub-project of Apache Felix
- Now a top-level project (TLP)
- <http://karaf.apache.org>

ServiceMix 4 was released before Karaf became a TLP

- The current version of this course uses ServiceMix 4 = JBoss Fuse
- Based on Apache ServiceMix 4
- References in this course are to “Felix Karaf”

## **Apache Felix Karaf**

Enables management of the OSGi runtime environment

- Command line interface wraps an OSGi container
- Configured to use Apache Felix as an OSGi container:  
<http://felix.apache.org>
- Also supports Equinox:  
<http://www.eclipse.org/equinox>
- Simple, extensible interface:  
<http://felix.apache.org/site/61-extending-the-console>

## Features (1 of 2)

Hot deployment mechanism for OSGi bundles

Spring configuration

JBI service assemblies, and more

Dynamic configuration, propagated to services/bundles

- Via OSGi ConfigurationAdmin service

Unified logging back-end

- Combines output from various logging APIs
  - (Log4j, SLF4j, Java Utils, JCL, Avalon, Tomcat, OSGi)
- Allows users to manage logs using simple Log4j-style configuration

Web console provides easy access for command entry

## **Features (2 of 2)**

Application provisioning

- Hot deploy via the JBoss Fuse's deploy folder or
- Maven lookup

Camel routes can be directly loaded

Administration

- Via an extensible shell console Remote access
- Via ssh

Security framework

- Based on JAAS

## Deployer Subsystem (1 of 2)

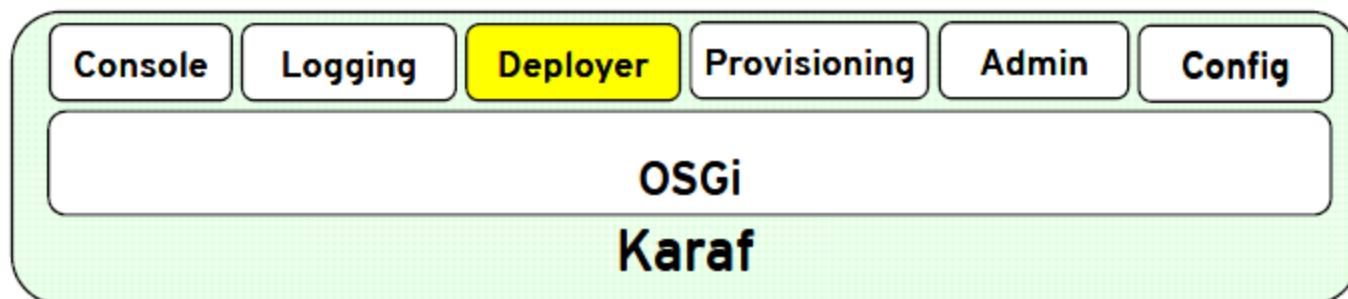
Hot-deploys artifacts found in the deploy directory

- OSGi bundles
- Spring application context files

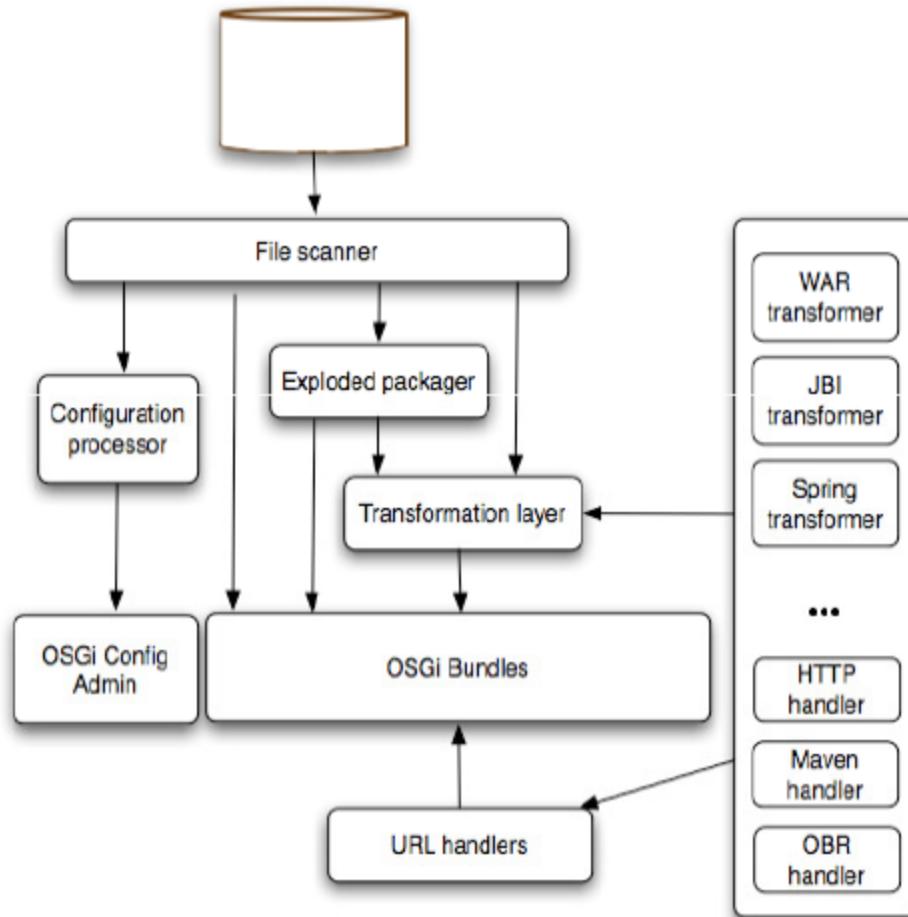
Provides additional deployment capabilities

- Through different URL handlers
- mvn: | file: | http: | wrap: | jbi: | etc...

Extensible architecture enables deployment support for different kinds of artifacts



## Deployer Subsystem (2 of 2)

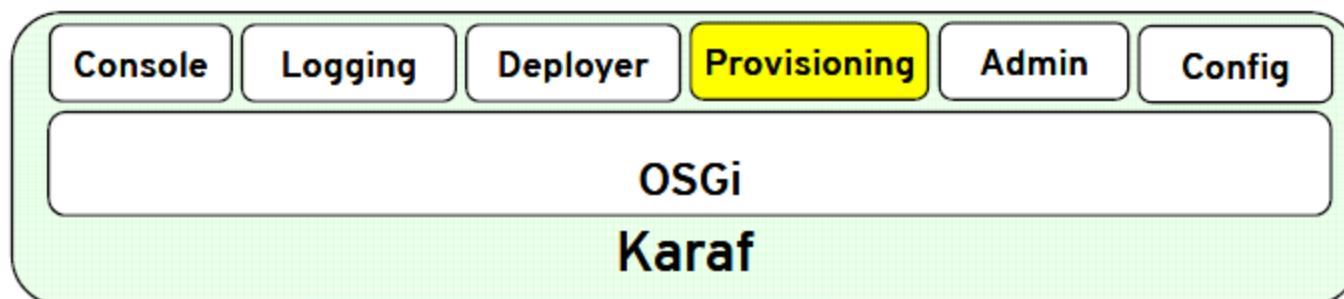


## Provisioning Subsystem

Provides a simple interface to manage OSGi bundles individually,

- Plus a simple, flexible mechanism to manage applications as "features"
  - A 'feature' is a collection of bundles, and may include other features
  - Features are defined in a simple XML file, which is registered with Karaf
- Felix Karaf provides commands to register, lookup, install, and uninstall features...
- i.e. to manage many OSGi bundles, one at a time, in a specific order
- URL handlers facilitate local/remote bundles

Features enable users to define an application as a set of OSGi bundles, and deploy these sets together in one unit



## Logging Subsystem

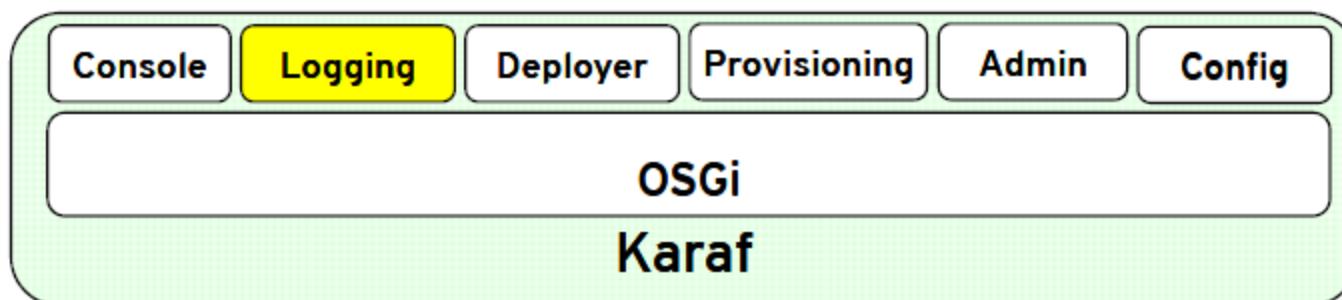
Combines output from loggers to provide a standard OSGi Log service

- Based on OPS4j Pax Logging

<http://www.ops4j.org/projects/pax/logging>

- Supports API for Apache Commons Logging, SLF4J, Log4j, Java Util Logging
- Combines all output into one synchronized log
- Uses Log4j configuration for easy management

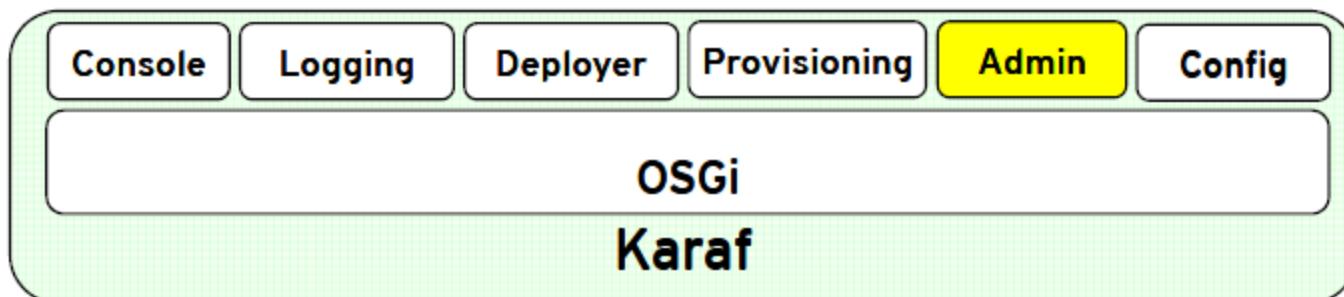
Felix Karaf also provides a set of console commands to display, view and change the log levels at runtime



## Administration Subsystem

Provides commands to administer other instances of Felix Karaf

- Each instance is a separate JVM instance with its own:
  - Configuration and state information
  - Logs and temporary files
- Typically
  - A number of instances on the same machine can share installation files such as systems bundles and bootstrap JARs



## Console Subsystem

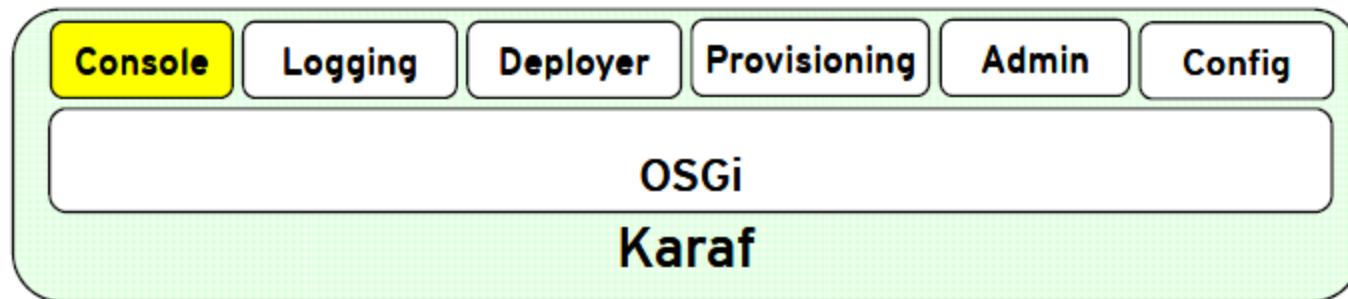
Provides a command line interface for administrative tasks

Commands take the form {subshell}:{command} [options]

- A sub-shell is a group of related commands
- e.g. commands related to the OSGi framework begin with “osgi:”

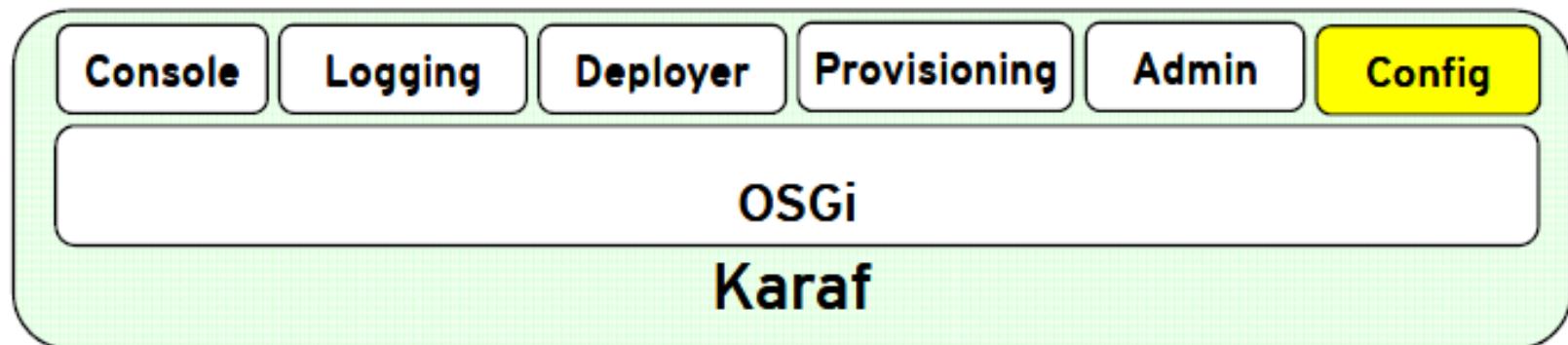
The console provides a mechanism for connecting to and issuing commands in a remote instance of the kernel

- The ssh connection can be made secure



## Configuration Subsystem

- Uses the OSGi Configuration Admin service to facilitate dynamic configuration of all OSGi bundles
- Users can view and edit the configuration via the console
  - These changes are propagated immediately to all bundles
- Users can also edit the property files in the **/etc directory**
  - Changes detected automatically, using a directory polling mechanism, and propagated immediately to all bundles
  - Changes are persistent, if instance restarted with a clean database, it will use the configuration settings in its /etc directory



## **JBoss Fuse Directory Layout**

The directory layout of JBoss Fuse is as follows:

- /bin – start and stop scripts
- /data – cached information and log files
  - /cache – cached bundles loaded by the OSGi container at runtime
  - /generated-bundles – generated bundles created by JBoss Fuse
  - /log – log files
- /deploy – hot deployment directory
- /etc – configuration files for JBoss Fuse and all other applications; also activemq.xml configuration
- /system – bundles implementing JBoss Fuse and all other applications
- /lib – bootstrapped libraries

## Shell Console Commands

shell:cat — displays the contents of a file or URL

shell:clear — clears the console buffer

shell:each — execute a closure on a list of arguments

shell:echo — prints arguments to the standard output

shell:exec — executes system processes

shell:grep — displays lines matching a regular expression

shell:head — displays the first lines of a file

shell:history — prints the command history

shell:if — executes an if/then/else block

shell:info — displays system information and statistics about the container

shell:java — execute a Java application

shell:logout — disconnects the shell from the current session

shell:more — displays output as pages of a specified length

shell:new — creates a new Java object of the specified class

shell:printf — formats and prints the specified output

shell:sleep — sleeps for a specified time, then wakes up

shell:sort — writes a sorted concatenation of the specified files to standard output

shell:source — run a shell script

shell:tac — captures the STDIN and returns it as a string and optionally writes the content to a file

shell:tail — displays the last lines of a file

shell:watch — watches and refreshes the output of a command

# Apache Camel

## **What is Apache Camel?**

- Quote from the website

Apache Camel is a  
powerful Open Source  
Integration Framework  
based on known  
Enterprise Integration Patterns

**Apache Camel** is a rule-based routing and mediation engine that provides a Java object-based implementation of the Enterprise Integration Patterns using an API (or declarative Java Domain Specific Language) to configure routing and mediation rules.

Camel provides *fluent builders* for creating routing and mediation rules

# Camel

Integration Engine And Router

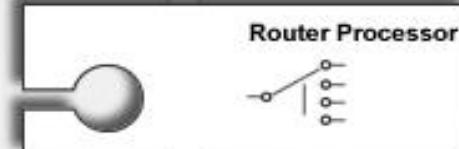
## Camel Endpoints

- Camel can send messages to them
- Or Receive Messages from them

Filter Processor



Router Processor



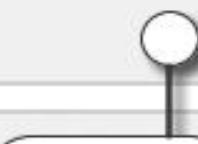
## Camel Processors

- Are used to wire Endpoints together
- Routing
- Transformation
- Mediation
- Interception
- Enrichment
- Validation
- Tracking
- Logging



JMS Component

JMS API



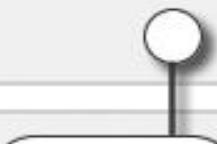
HTTP Component

Servlet API



Web Container  
Jetty | Tomcat | ...

HTTP Client



File Component

File System



Local File System

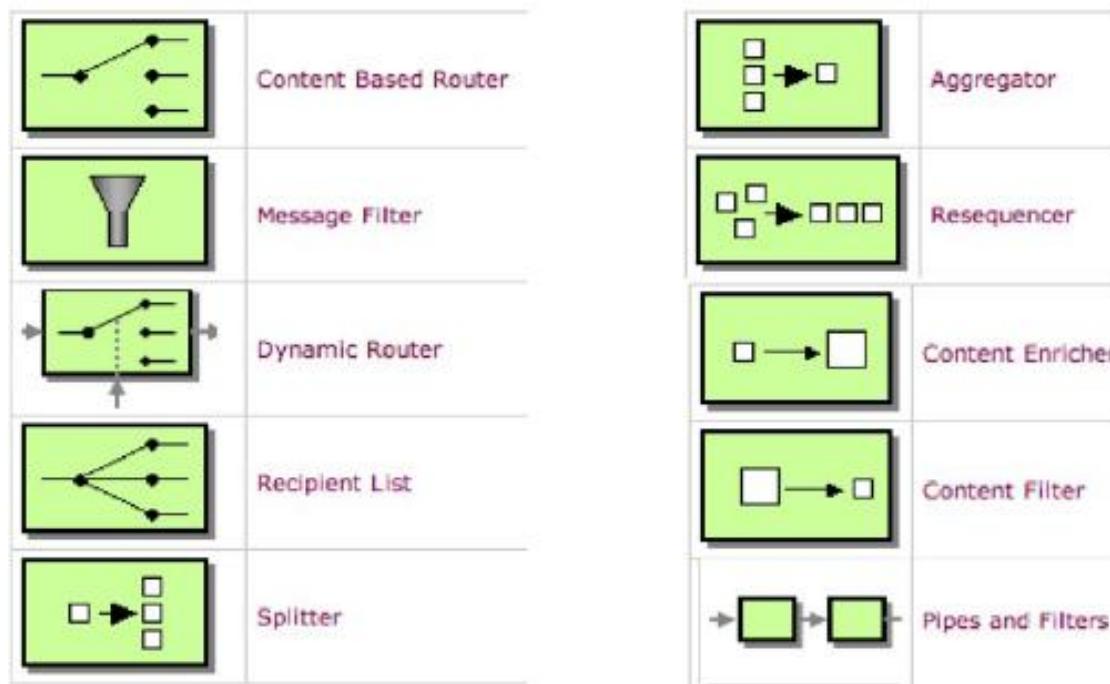
JMS Provider  
ActiveMQ | IBM |  
Tibco | Sonic ...

## Camel Components

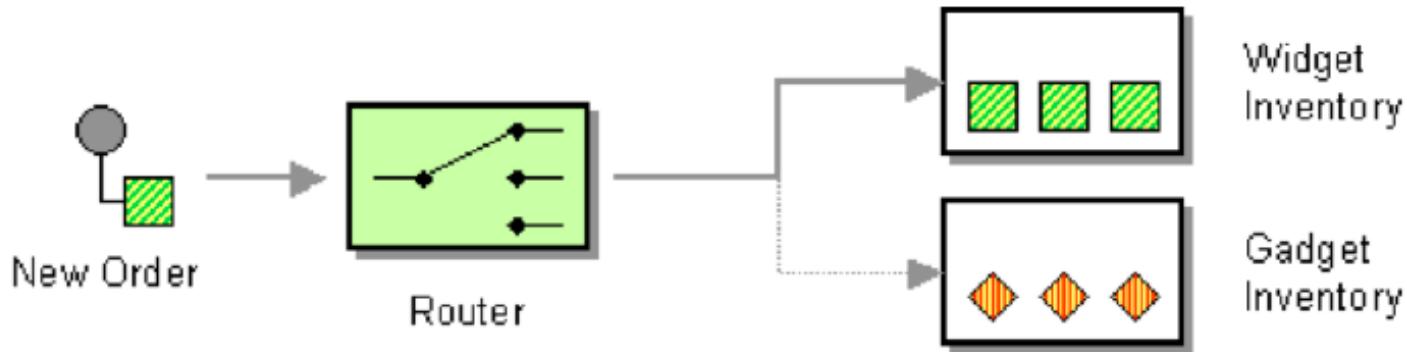
- Provide a uniform Endpoint Interface
- Act as connectors to all other systems

# What is Apache Camel?

- Enterprise Integration Patterns



## What is Apache Camel?



```
Endpoint newOrder = endpoint("activemq:queue:newOrder");
Predicate isWidget = xpath("/order/product = 'widget'");
Endpoint widget = endpoint("activemq:queue:widget");
Endpoint gadget = endpoint("activemq:queue:gadget");

from(newOrder)
.choice()
.when(isWidget).to(widget)
.otherwise().to(gadget);
```

# What is Apache Camel?

- Java Code

```
import org.apache.camel.Endpoint;
import org.apache.camel.Predicate;
import org.apache.camel.builder.RouteBuilder;

public class MyRoute extends RouteBuilder {

    public void configure() throws Exception {
        Endpoint newOrder = endpoint("activemq:queue:newOrder");
        Predicate isWidget = xpath("/order/product = 'widget'");
        Endpoint widget = endpoint("activemq:queue:widget");
        Endpoint gadget = endpoint("activemq:queue:gadget");

        from(newOrder)
            .choice()
                .when(isWidget).to(widget)
                .otherwise().to(gadget)
            .end();
    }
}
```

# What is Apache Camel?

- Camel Java DSL

```
import org.apache.camel.builder.RouteBuilder;

public class MyRoute extends RouteBuilder {

    public void configure() throws Exception {
        from("activemq:queue:newOrder")
            .choice()
                .when(xpath("/order/product = 'widget'"))
                    .to("activemq:queue:widget")
                .otherwise()
                    .to("activemq:queue:gadget")
            .end();
    }
}
```

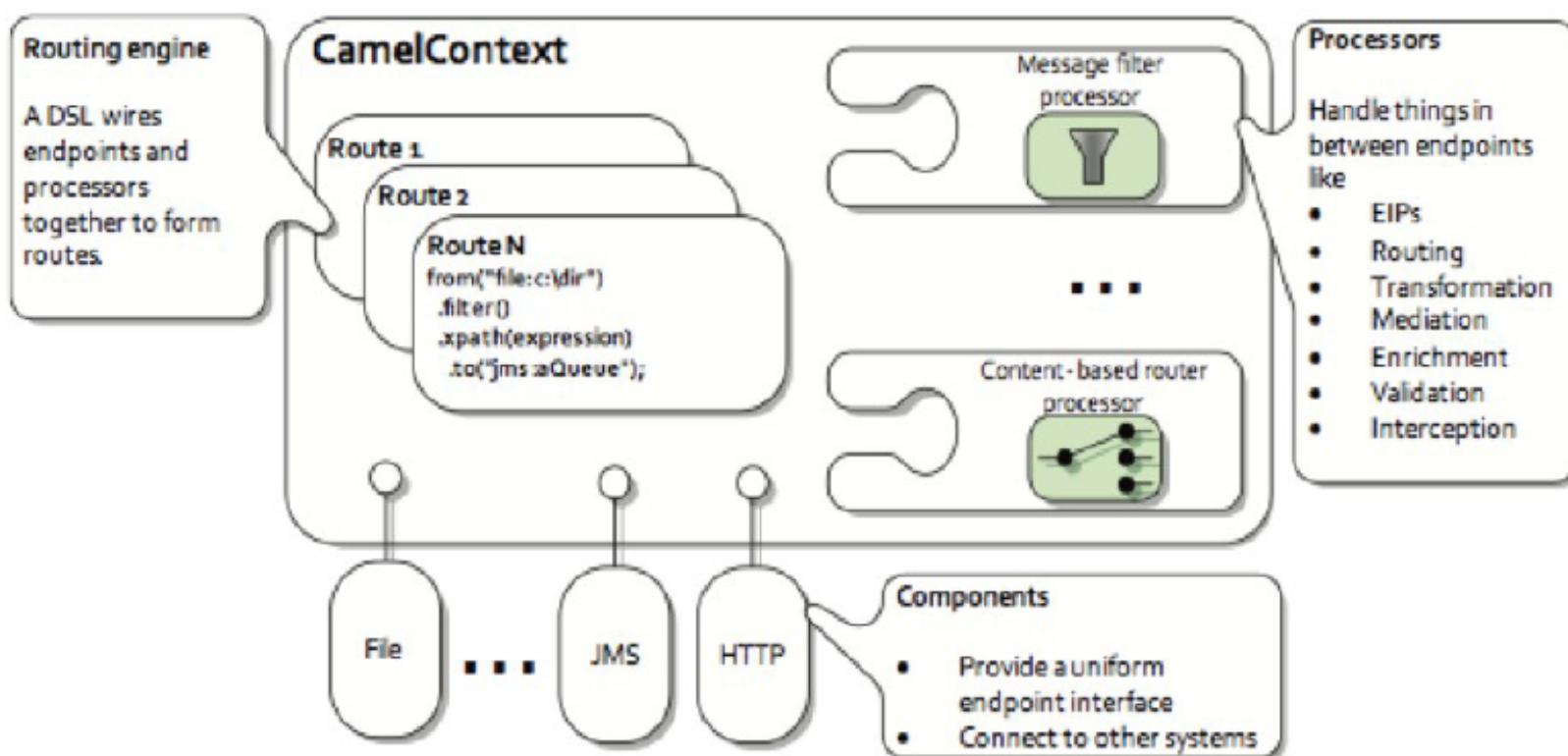
# What is Apache Camel?

- Camel XML DSL

```
<route>
    <from uri="activemq:queue:newOrder"/>
    <choice>
        <when>
            <xpath>/order/product = 'widget'</xpath>
            <to uri="activemq:queue:widget"/>
        </when>
        <otherwise>
            <to uri="activemq:queue:gadget"/>
        </otherwise>
    </choice>
</route>
```

# What is Apache Camel?

- Camel's Architecture



# What is Apache Camel?

120+ Components

activemq	cxf	flatpack	jasypt
activemq-journal	cxfrs	freemarker	javaspace
amqp	dataset	ftp/ftps/sftp	jbi
atom	db4o	gae	jcr
bean	direct	hdfs	jdbc
bean validation	ejb	hibernate	jetty
browse	esper	hl7	jms
cache	event	http	jmx
cometd	exec	ibatis	jpa
crypto	file	irc	jt/400

# What is Apache Camel?

- Summary
  - Integration Framework
  - Enterprise Integration Patterns (EIP)
  - Routing (using DSL)
  - Easy Configuration (endpoint as uri's)
  - Payload Agnostic
  - No Container Dependency
  - A lot of components



## **Spring Support**

Apache Camel is designed to work nicely with the Spring Framework in a number of ways.

Camel uses Spring Transactions as the default transaction handling in components like JMS and JPA.

Camel works with Spring Xml Configuration.

Camel supports a powerful version of Spring Remoting .

Camel provides powerful Bean Integration with any bean defined in a Spring ApplicationContext.

## **Spring Support**

Camel integrates with various Spring helper classes; such as providing Type Converter support for Spring Resources etc

Allows Spring to dependency inject Component instances or the CamelContext instance itself and auto-expose Spring beans as components and endpoints.

Allows you to reuse the Spring Testing framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's powerful Mock and Test endpoints

## **Using Spring to configure the CamelContext**

We can configure a CamelContext inside any spring.xml using the CamelContextFactoryBean. This will automatically start the CamelContext along with any referenced Routes along any referenced Component and Endpoint instances.

- ✓ Adding Camel schema
- ✓ Configure Routes in two ways:
  - Using Java Code
  - Using Spring XML

## A Camel Route (1 of 2)

Step-by-step processing of a message:

- 1.Consumer endpoint - listens for an incoming message...
- 2.Route through zero or more processors  
Apply enterprise integration patterns / custom processing code / interceptor patterns / more
- 3.Route to a Producer endpoint  
Sends an outgoing message (optional)

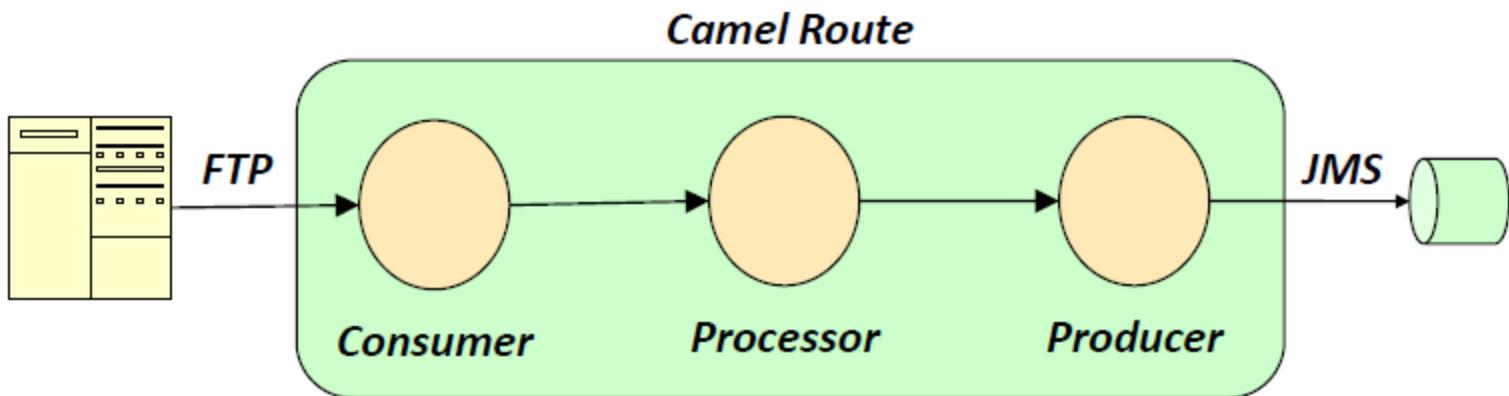
## **A Camel Route (2 of 2)**

- > May involve any number of processing components
  - Modify the original message
  - And/or redirect it
- > Can be defined in Java or XML
  - Camel DSL (Domain Specific Language) is implemented in Java
  - XML schema used to define the route in a Spring XML configuration file

## Example Route

Scenario:

1. Consume an XML file from an FTP server
2. Process/transform it using XSLT
3. Produce a JMS message and place on a queue



## Endpoints

Can create or receive messages

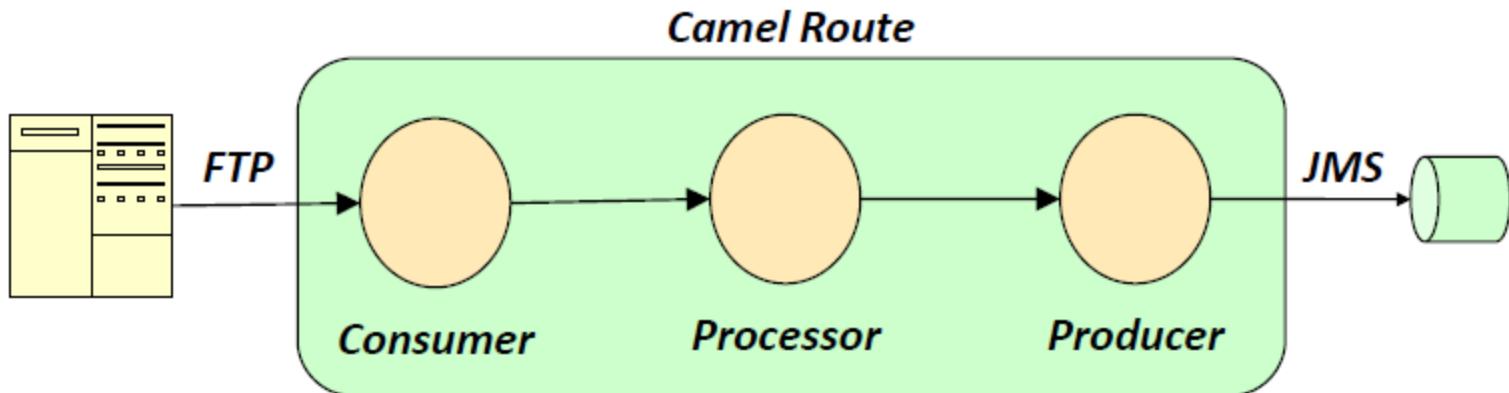
- Examples: an FTP server, a Web Service, or a JMS broker

Camel defines endpoints using URI syntax

**“scheme://specific-part?key=value&key=value ...”**

ftp://john@localhost?password=doe

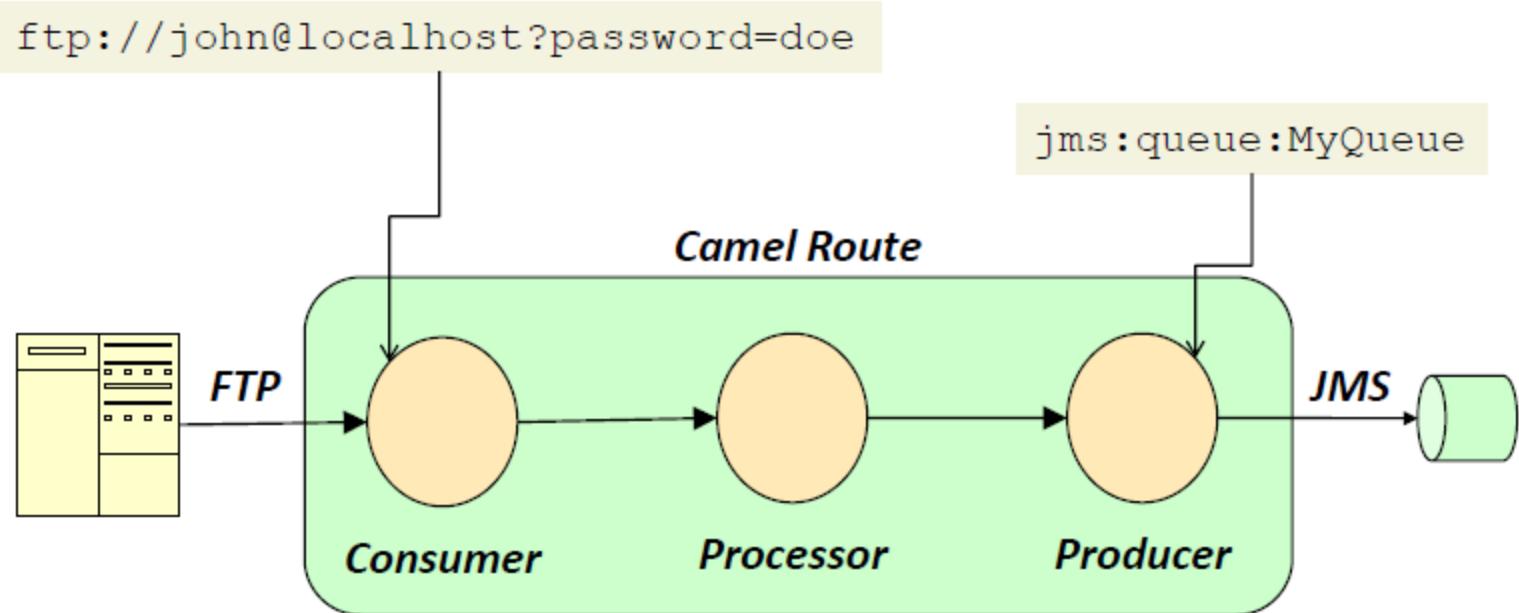
jms:queue:MyQueue



## Consumers and Producers

Created from endpoints

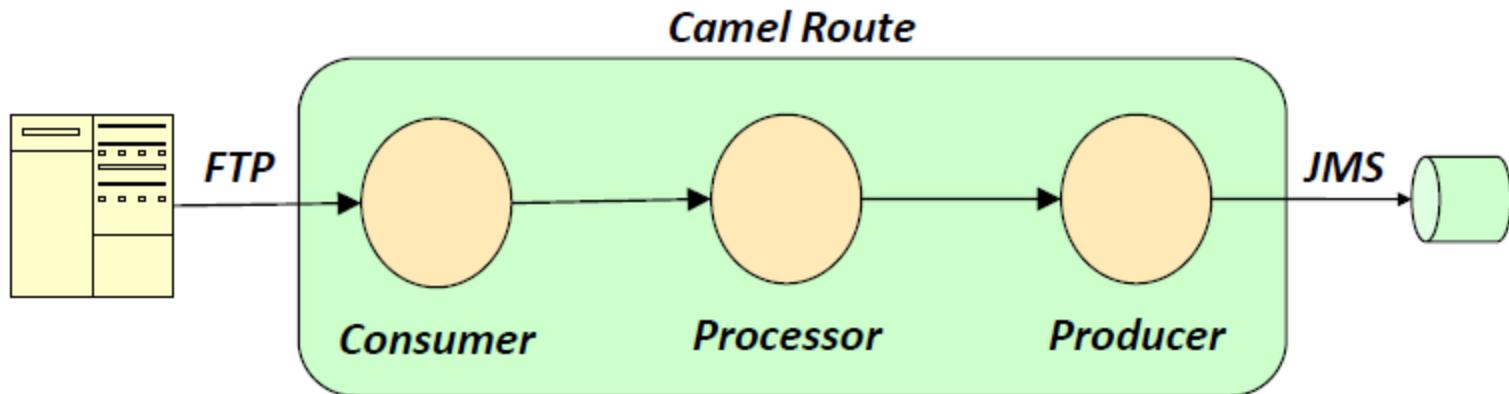
- Some create producers and consumers
- Some support the creation of either a producer or a consumer



## Camel DSL

Camel provides a DSL to allow you to easily define routes

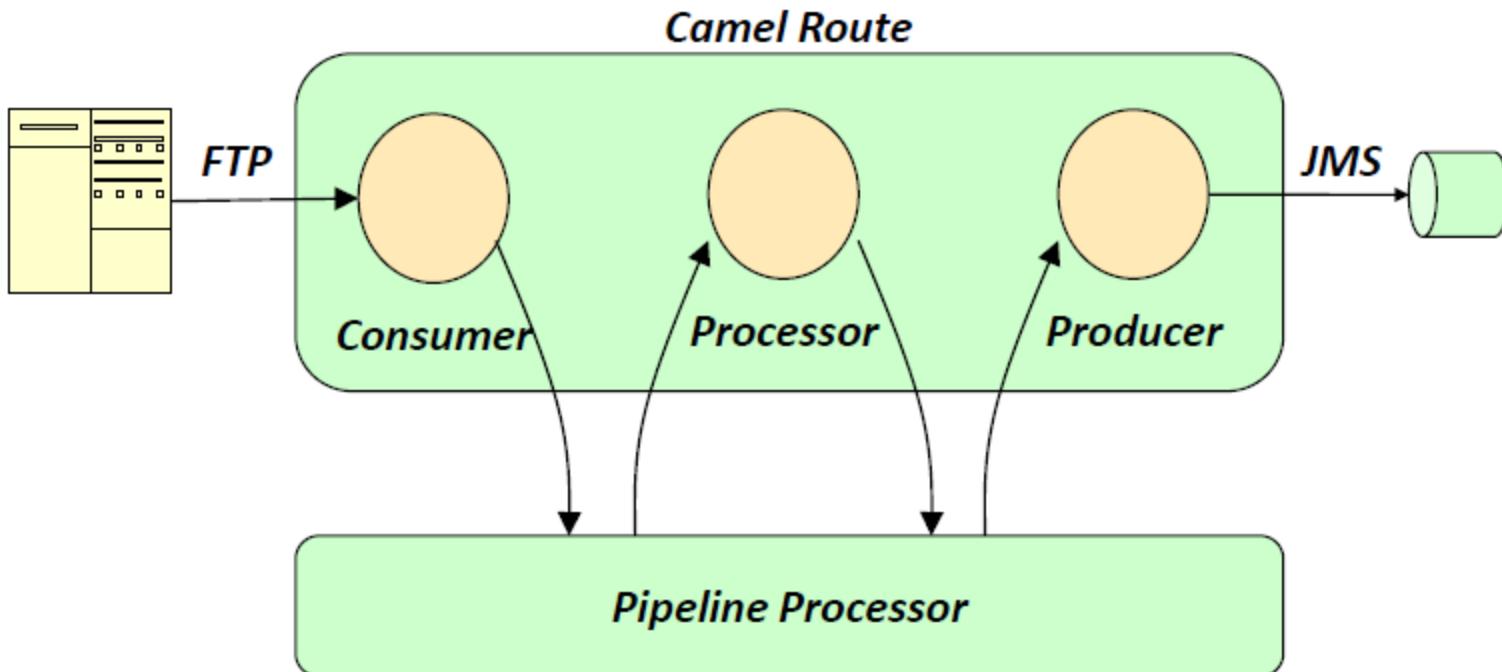
```
from("ftp://john@localhost?password=doe")
    .process("xslt:MyTransform.xslt")
    .to("jms:queue:MyQueue")
```



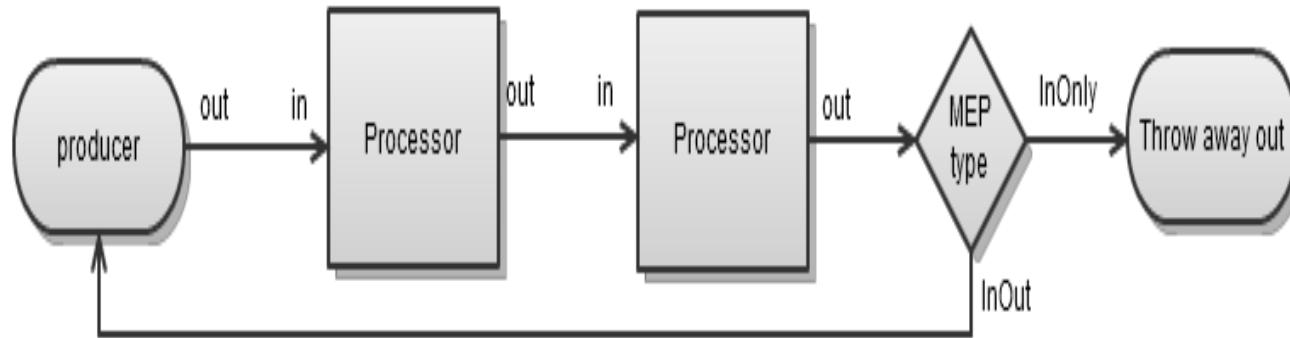
## Processors and Pipelines

Camel creates a "pipeline" of processors between a consumer and the producers in a route

- Passes incoming messages to a processor
- Sends output from the processor to the next process in the chain



## Flow of an exchange through a route



- The out message from each step is used as the in message for the next step
  - > if there is no out message then the in message is used instead
  - > For the InOut MEP the out from the last step in the route is returned to the producer. In case of InOnly the last out is thrown away

## Using getIn or getOut methods on Exchange

Now suppose we want to use a Camel Processor to adjust a message. This can be done as follows:

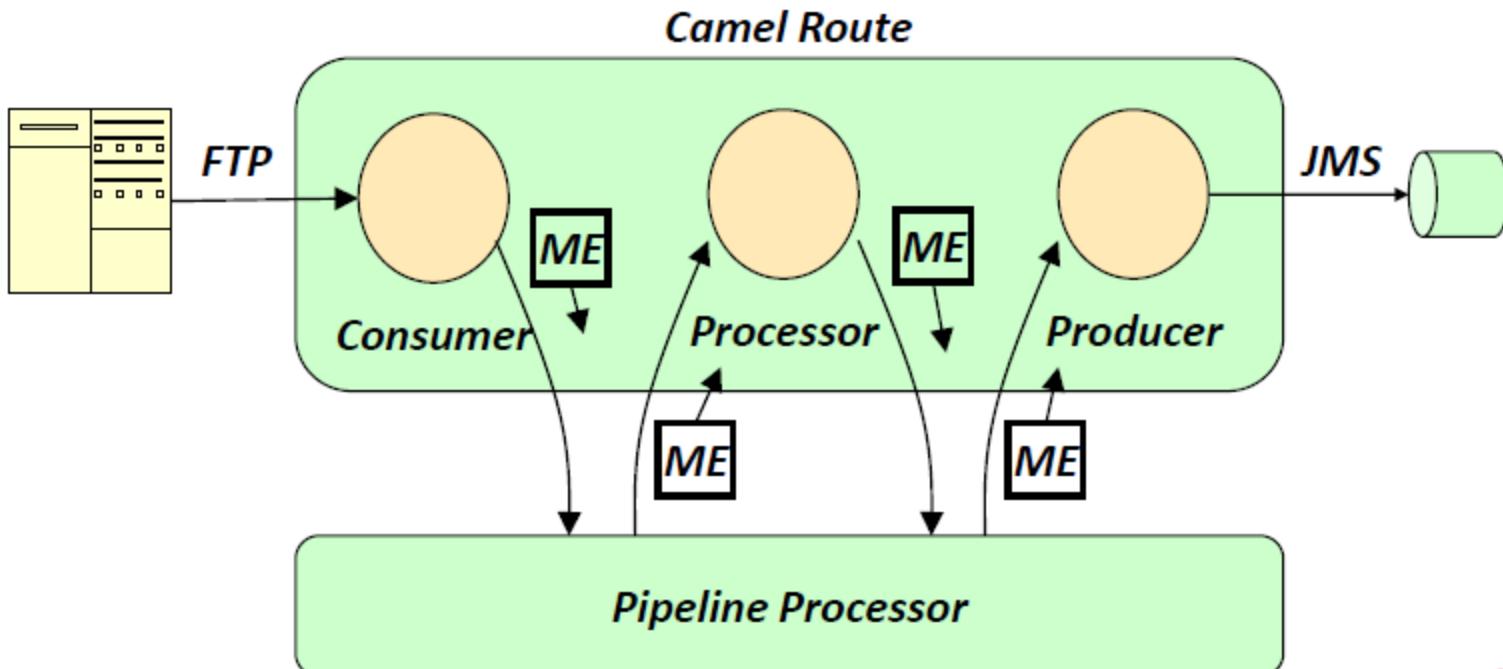
```
public void process(Exchange exchange) throws Exception
{
    String body = exchange.getIn().getBody(String.class);
    // change the message to say Hello
    exchange.getOut().setBody("Hello " + body);
}
```

## Sample Processors

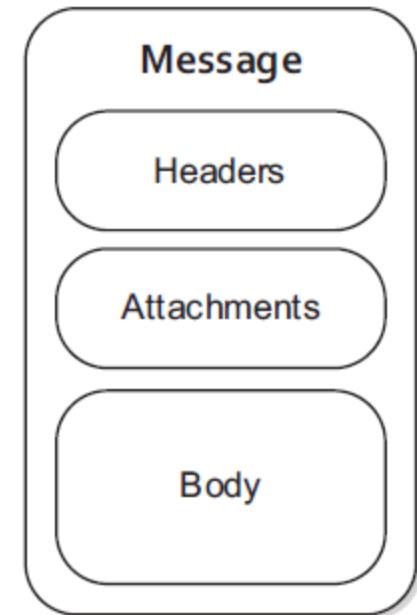
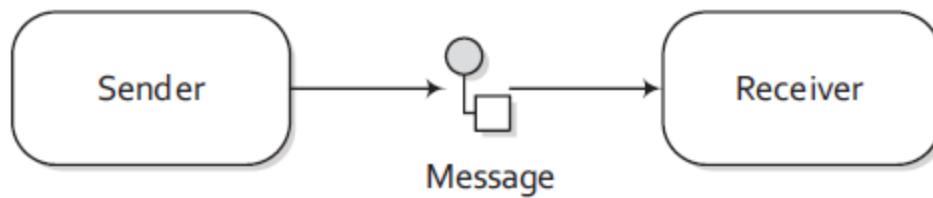
- **Choice** `choice()`
  - Used to route incoming messages to alternative producer endpoints
  - Each alternative producer endpoint is preceded by a `when()` method, which takes a predicate argument
- **Filter** `filter()`
  - Used to prevent uninteresting messages from reaching the producer endpoint
  - Takes a single predicate argument
- **Throttler** `throttle()`
  - Ensures that a producer endpoint does not get overloaded
  - Works by limiting the number of messages per second
- **Custom processor**
  - define a class that implements the `org.apache.camel.Processor` interface and overrides the `process()` method

# Message Exchanges

- Runtime sends a Message Exchange along the pipeline
  - The exchange is a holder for input and output messages
  - Messages are not canonical.
  - Unless explicitly transformed, they maintain original format, until exit



## Message



Messages are the entities used by systems to communicate with each other when using messaging channels. Messages flow in one direction from a sender to a receiver.

Messages have a body (a payload), headers, and optional attachments.

Messages are uniquely identified with an identifier of type `java.lang.String`. For protocols that don't define a unique message identification scheme, Camel uses its own UID generator.

The body is of type `java.lang.Object`.

## Exchange

An exchange in Camel is the message's container during routing. An exchange also provides support for the various types of interactions between systems, also known as message exchange patterns (MEPs).

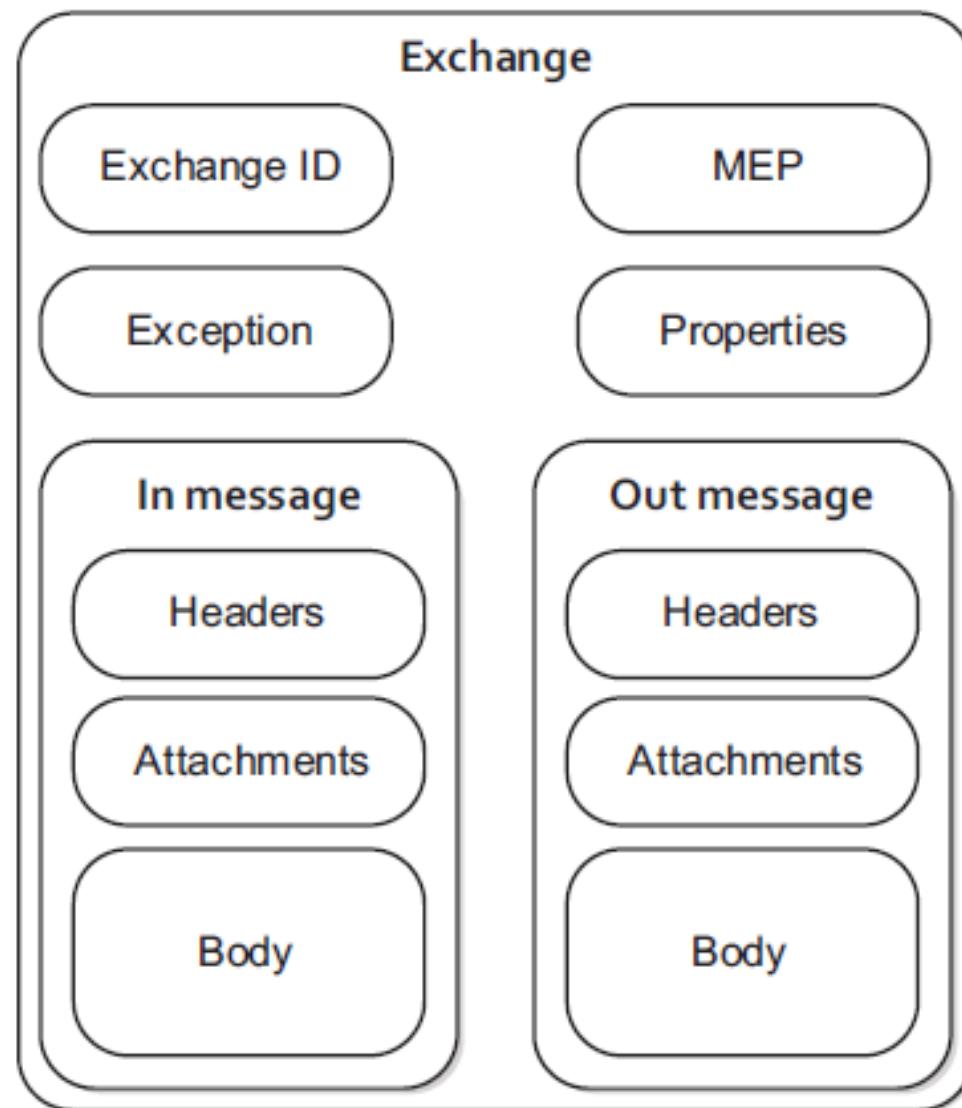
MEPs are used to differentiate between one-way and request-response messaging styles.

The Camel exchange holds a pattern property that can be either

>> InOnly—A one-way message (also known as an Event message).

For example, JMS messaging is often one-way messaging.

>> InOut—A request-response message. For example, HTTP-based transports are often request reply, where a client requests to retrieve a web page, waiting for the reply from the server.

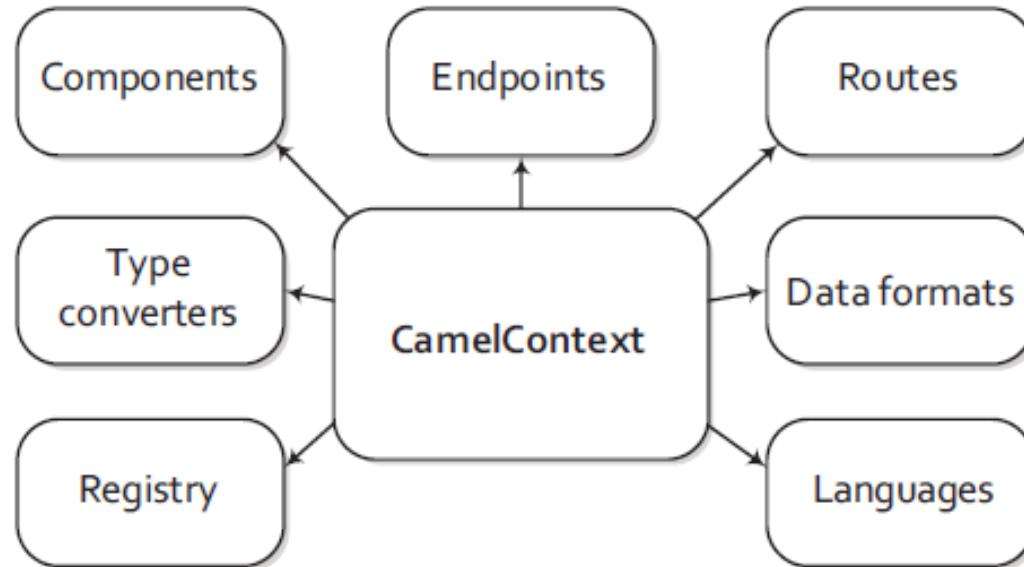


## JBoss Camel

- **The JBoss Camel architecture consists of:**
  - CamelContext **objects**
  - **Which provide the runtime environment for**
    - RouteBuilders
    - which encapsulate rules, endpoints, and components

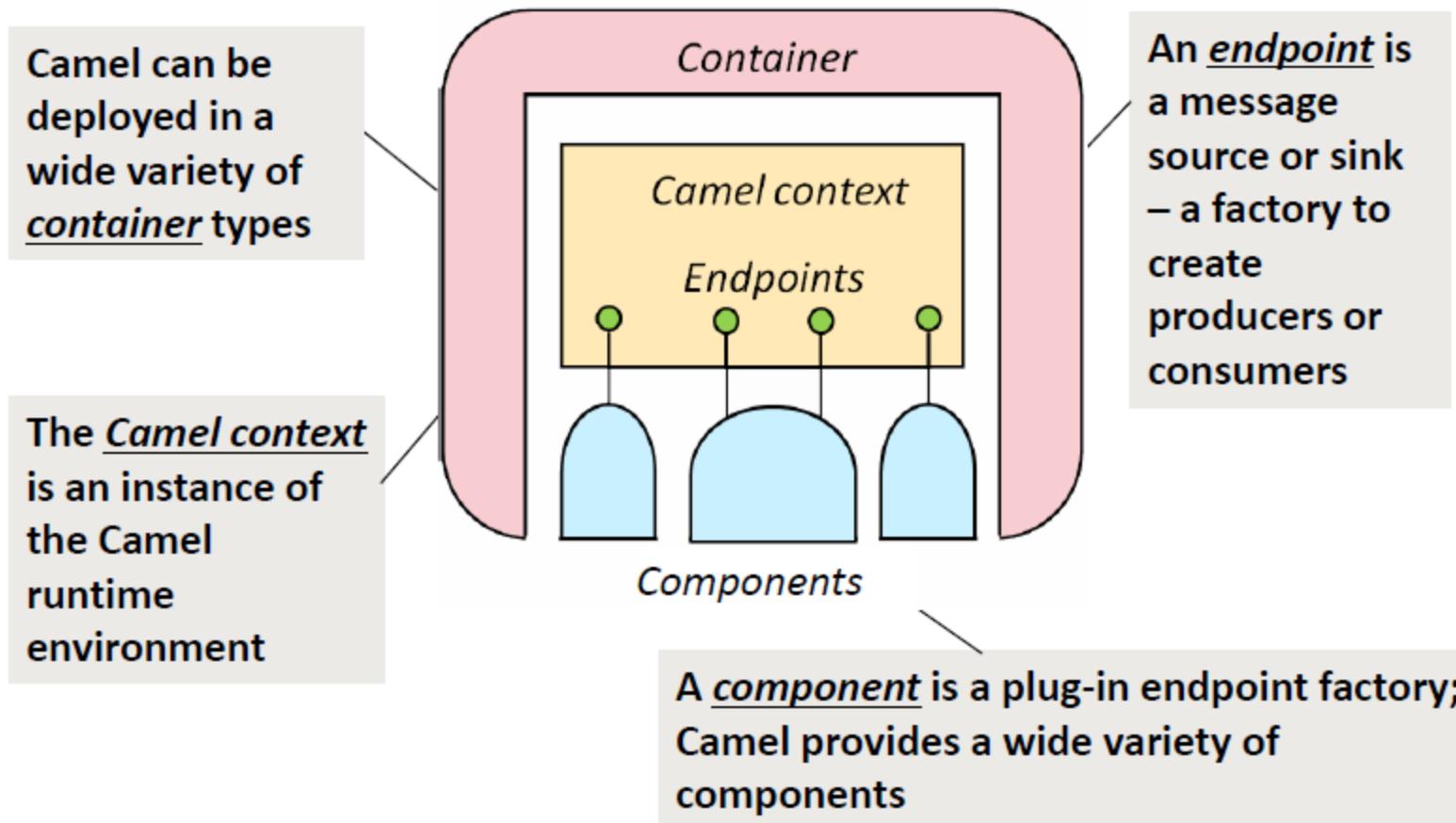
CamelContext :

This will provide Camel's runtime environment.



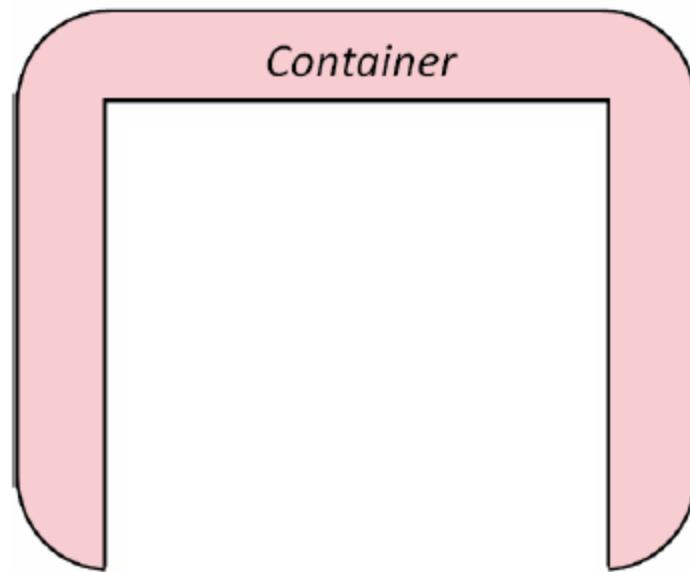
Service	Description
Components	Contains the components used. Camel is capable of loading components on the fly either by autodiscovery on the classpath or when a new bundle is activated in an OSGi container.
Endpoints	Contains the endpoints that have been created.
Routes	Contains the routes that have been added.
Type converters	Contains the loaded type converters. Camel has a mechanism that allows you to manually or automatically convert from one type to another.
Data formats	Contains the loaded data formats.
Registry	Contains a registry that allows you to look up beans. By default, this will be a JNDI registry. If you're using Camel from Spring, this will be the Spring ApplicationContext. It can also be an OSGi registry if you use Camel in an OSGi container.
Languages	Contains the loaded languages. Camel allows you to use many different languages to create expressions. You'll get a glimpse of the XPath language in action when we cover the DSL.

# Camel Architecture



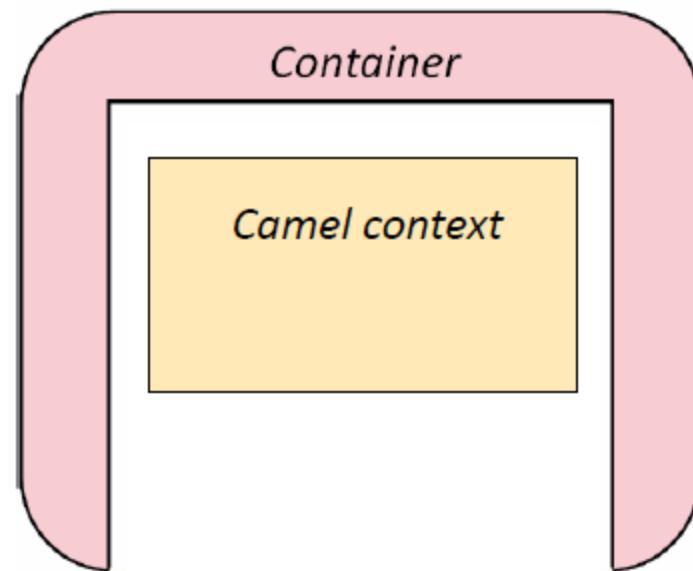
## Container

- Runs in a JVM and acts as a server runtime
- Camel routes deployed
  - In any Java based container:
    - ServiceMix 4.x (OSGi)
    - ServiceMix 3.x (JBI)
    - ActiveMQ (JMS)
    - Tomcat (Servlet)
    - Other JEE servers
  - Also in a simple JVM
    - Useful for development and ad-hoc system testing

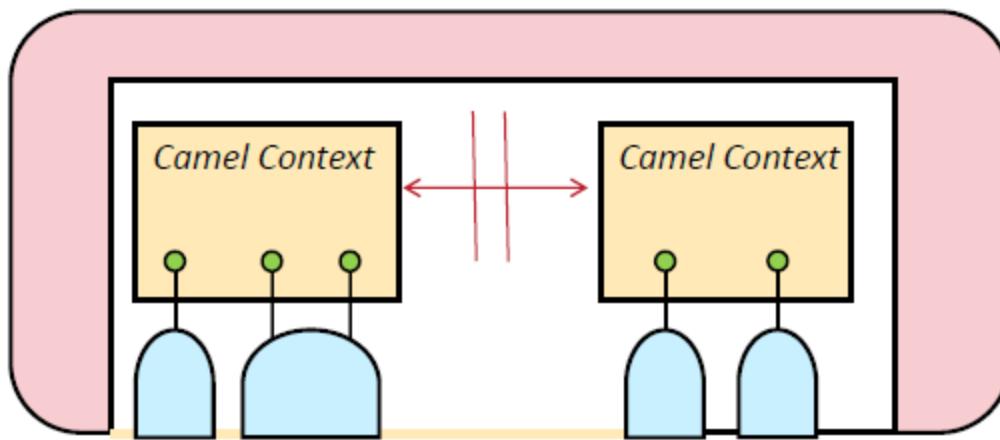


## Camel Context (1 of 2)

- **Acts as a runtime environment managing...**
  - Routes
  - Components
  - Endpoints
  - Messages
  - JMX for remote management
  - Tracing
- **Create a CamelContext:**
  - Programmatically in Java
  - OR
  - Define it in a Spring XML configuration file
- **There can be more than one context in a container**



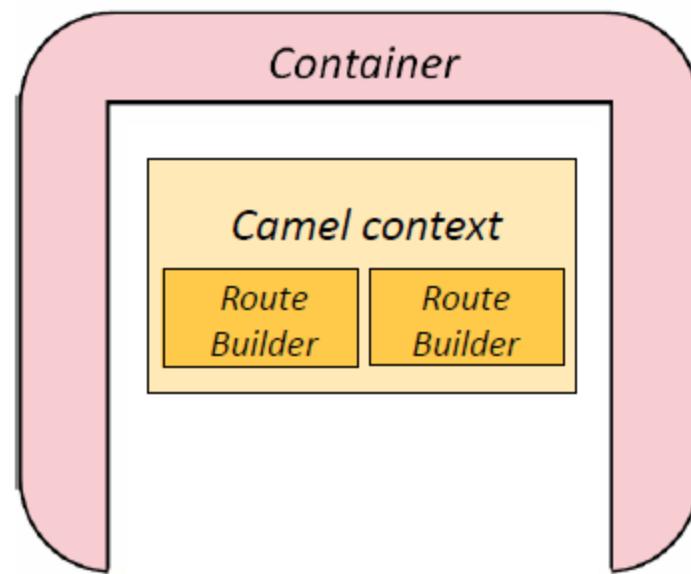
## Camel Context (2 of 2)



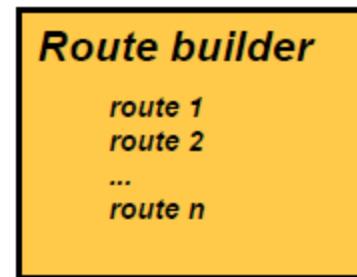
- Camel Contexts are isolated from each other
- Endpoints cannot communicate between CamelContexts
  - A transport layer will be required in this case (JMS, NMR, VM)
- CamelContext has a lifecycle. It can be started or stopped.

## Route Builders (1 of 2)

- **Each** CamelContext **contains**  
**one or more** RouteBuilders
  - Java classes
  - Facilitate creation of process flows and EIPs
  - Encapsulate routing rules
- **A developer:**
  - Defines custom classes that implement the RouteBuilder **interface**
  - Adds instances of these classes to the CamelContext



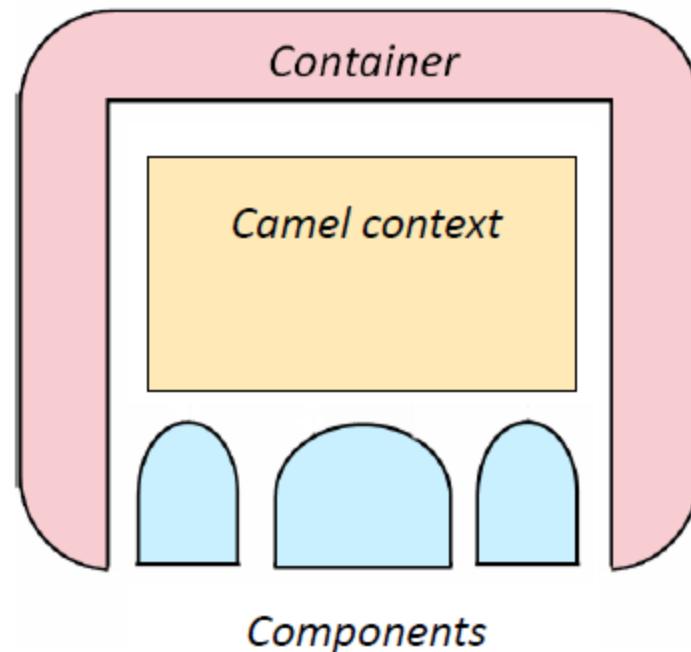
## Route Builders (2 of 2)



```
package com.acme.quotes;  
import org.apache.camel.builder.RouteBuilder;  
  
public class MyRouteBuilder extends RouteBuilder {  
    public void configure() {  
        from("file:/inputDir").to("jms:outputQueue");  
    }  
}
```

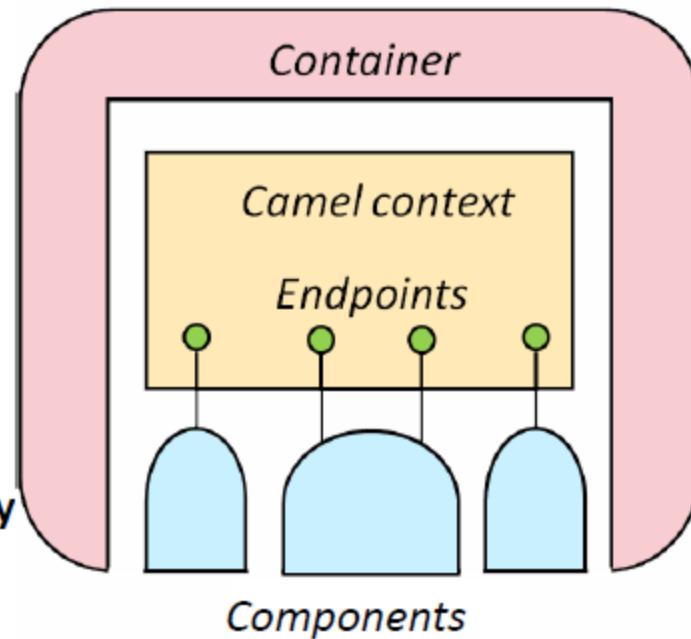
# Components

- Operationally endpoint factories
- Create endpoints of a given type
  - For example, an **HTTP component** is used to create **HTTP endpoints**
- Camel uses
  - Connection pools and lazy initialization techniques
  - Lowers the cost of endpoint creation



# Endpoints

- Interpret the specified URI to:
  - Set up a listener  
(consumer endpoint)  
OR
  - Send a message  
(producer endpoint)
- Refer to
  - A location/address (URL)  
OR
  - A named resource/software entity  
(URN)
- For example, a  
`tcp://host:port address`  
represents TCP-based  
communication



## Developing within the Spring Framework

- **Configure a CamelContext inside Spring XML configuration file**
  - Spring automatically starts the CamelContext
  - Camel then initializes all routes defined in the CamelContext
- **We recommend Spring**
  - Widespread adoption in the Camel community
  - Configuration driven rather than Code driven
  - Simple and easy to read
  - ServiceMix 4.x supports Spring Dynamic Modules (DM) & Blueprint
- **The Spring framework supports Camel routes defined:**
  - In Java DSL
  - Directly in Spring XML

## Configuring a CamelContext

- 1. Add the Camel schema to the `schemaLocation` declaration**
- 2. Use the `<camelContext>` tag to create a CamelContext**
- 3. Instantiate Camel components for the endpoints you plan to use in your routes**
  - All components in `camel-core` instantiated automatically
  - Other components are instantiated as Spring beans
  - Use a meaningful 'id' in each camel route
  - Add configuration (server host/port, resource classes, etc.)
  - Your CamelContext will register the instantiated component
  - The component can then be used to initialize route endpoints
- 4. Configure Camel routes**
  - Using Java DSL or Spring XML

## Configuring a CamelContext Example

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
        beans.xsd
        http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-
        spring.xsd">
</beans>
```

*Add the Camel schema location*

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

*Instantiate a Camel context*

# Referencing a Camel Component

- **First we must configure a component**
  - In this example we enable and configure a JMS component:

```
<bean name="jmsprovider"
    class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <bean class="...">
            <property name="brokerURL" value="..." />
        </bean>
    </property>
</bean>
```

- **Use the component to create endpoints in our Camel routes, by defining the endpoint URI as follows:**

"jmsprovider:queue:MyQueue"

## Defining Camel Routes in Java DSL

- **Create a RouteBuilder subclass and implement configure()**
  - **Example routing messages from the filesystem to a JMS queue:**

```
import org.apache.camel.builder.RouteBuilder;
public class MyRouter extends RouteBuilder {
    public void configure() throws Exception {
        from("file:" + inputDir)
            .to("jmsprovider:queue:" + outputQueue);
    }
}
```

- **Then instantiate the route in the Spring configuration file:**

```
<bean id="MyRoute" class="com.example.MyRouter">
    <property name="inputDir" value="/tmp/MyDir" />
    <property name="outputQueue" value="MyQueue" />
</bean>
```

## Defining Camel Routes in Java DSL

- And provide the `routeBuilder` reference to the `CamelContext` using `<routeBuilder>` tag

```
<camelContext  
xmlns="http://camel.apache.org/schema/spring">  
    <routeBuilder ref="myBuilder" />  
</camelContext>  
  
<bean id="myBuilder"  
class="org.myproject.routes.MyRouteBuilder"/>
```

## Defining Camel Routes in Java DSL

- **Or, let CamelContext to discover them using <package/> and/or <packageScan/>**

```
<camelContext  
    xmlns="http://camel.apache.org/schema/spring">  
        <packageScan>  
            <package>org.myproject.scan</package>  
            <excludes>**/*Excluded*</excludes>  
            <includes>**/*</includes>  
        </packageScan>
```

## Defining Camel Routes in Spring XML

- **Or, you can define your route in the Spring configuration file, nested inside your CamelContext:**

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

    <route id="MyRoute">
        <from uri="file:/tmp/MyDir"/>
        <to uri="jmsprovider:queue:MyQueue"/>
    </route>

</camelContext>
```

# Java DSL versus Spring XML

- You can choose whether to use Java DSL or Spring XML
  - Below is a comparison to help you decide
- We will focus on Java DSL examples in this chapter!
  - Students can transfer these examples to XML as a practical exercise

## Java DSL

- Pro:
  - More options for custom development, embedded solutions in legacy applications/frameworks
- Con:
  - Developer needs to take control of instantiation and route lifecycles

## Spring XML

- Pro:
  - Configuration over coding, which is simple, and means you can see your route and its resources / configuration, all in one place
- Con:
  - More coarse-grained and verbose

## Defining a RouteBuilder

- 1. Create a router that extends the `RouteBuilder` class**
  - `configure()` method provides a place to define routes
- 2. The router will implement `InitializingBean` interface**
  - The `afterPropertiesSet()` method provides a hook to validate and post-process bean properties and to initialize backend dependencies
- 3. The router will implement `DisposableBean` interface**
  - The `destroy()` method provides a hook to clean up prior to shutdown
- 4. Log debug messages throughout the router life cycle using `slf4j`**
  - Avoid the bad practice of using `System.out.println()`

## The RouteBuilder subclass

```
public class MyRouter extends RouteBuilder
    implements InitializingBean, DisposableBean {

    public void configure() throws Exception {
        Define your Camel routes here
    }

    public void afterPropertiesSet() throws Exception {
        Validate and post-process bean properties,
        and initialize backend dependencies
    }

    public void destroy() throws Exception {
        Release resources and clean up prior to
        shutdown
    }
}
```

## Implementing `configure()`

- **Define your Camel routes here**
  - Place core of your routing application code here
- **Example:**

```
public class MyRouter extends RouteBuilder {  
    public void configure() throws Exception {  
        from("file:MyDir")  
            .to("jmsprovider:queue:MyQueue");  
    }  
}
```

## Implementing `afterPropertiesSet()`

- This method is part of the `InitializingBean` interface
  - Called by Spring, once the bean properties are set
- Used to
  - Validate inputs
    - Detect a required property value that is null
  - Post-process inputs
    - Construct an endpoint URI, so routes can be used in different environments
  - Initialize backend dependencies
    - Configure a database connection
- Throw `BeanInitializationException` if any of the above operations fail

## Implementing `destroy()`

- **This method is part of the `DisposableBean` interface**
  - Called by Spring, prior to shutdown (of Camel routes & context)
- **Used to release shared resources, close connections, etc.**
- **Example:**

```
public void destroy() throws Exception {  
    logger.debug("Shutting down route.");  
    dbConnection.close();  
}
```

## Creating Endpoints with CamelContext

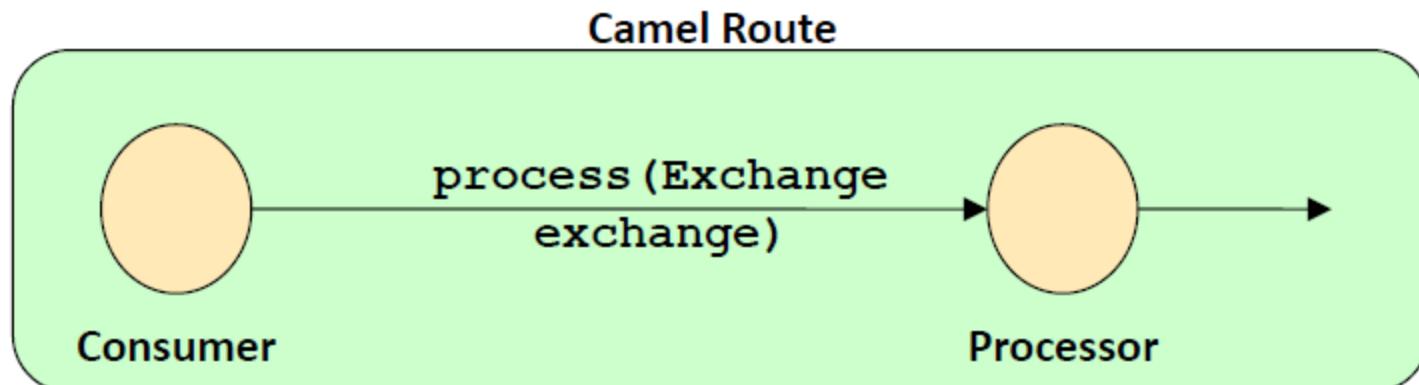
- **Spring alternative**
- **Create an endpoint with the CamelContext**
  - Can ( or not) use propertiesPlaceHolder
  - Pass the reference to the route class

```
<camelContext trace="true"
    xmlns="http://camel.apache.org/schema/spring">
    <endpoint id="sourceDirectoryXml"
        uri="${sourceDirectoryXmlUri}" />

public class RouteByCurrencyRouter extends RouteBuilder
{
    @EndpointInject(ref = "sourceDirectoryXml")
    Endpoint sourceUri;
```

## Creating Custom Processors

- Use the `Processor` interface to inject custom code into a route
  - Validation of incoming messages
  - Transformation of the message data
  - Implementing business rules as custom patterns
- Implement the `process()` method to access the `Exchange`, which contains the incoming message and outgoing message



## Custom Processor Example

```
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
public class MyProcessor implements Processor {
    public void process(Exchange exchange) throws Exception
    {

        String payload =
exchange.getIn().getBody(String.class);
        payload = payload.toLowerCase();
        exchange.getIn().setBody(payload);
    }
}
```

*Get the incoming message payload string*

*Transform to a lowercase string*

*Update the incoming message*

## Camel-CXF Web Service (3 of 3)

```
<bean name="demolmpl" class="com.example.webservice.DemoImpl"/>
```

*Instantiate the JAX-WS implementation class as a Spring bean*

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

```
  <route id="DemoRoute">
```

```
    <from uri="cxn:bean:webConsumer"/>
```

```
    <to uri="log:com.example?showAll=true&multiline=true"/>
```

```
    <bean ref="demolmpl" method="doWork"/>
```

```
  </route>
```

```
  </camelContext>
```

```
</beans>
```

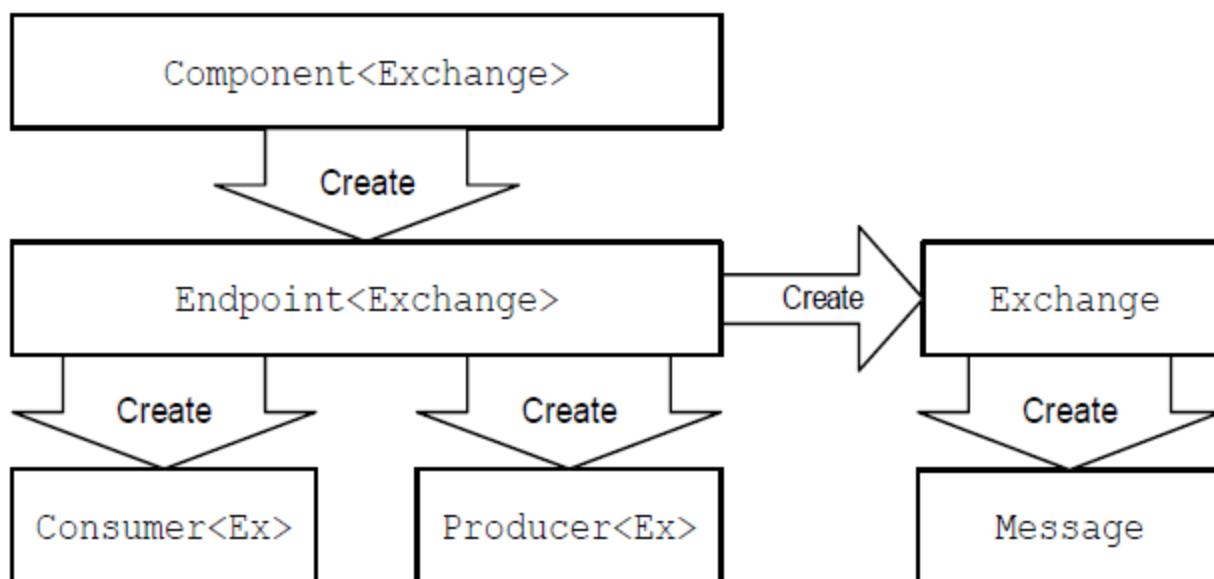
*Define a Camel route to accept web service requests, log their contents, invoke the JAX-WS bean, and return a response to the original client*

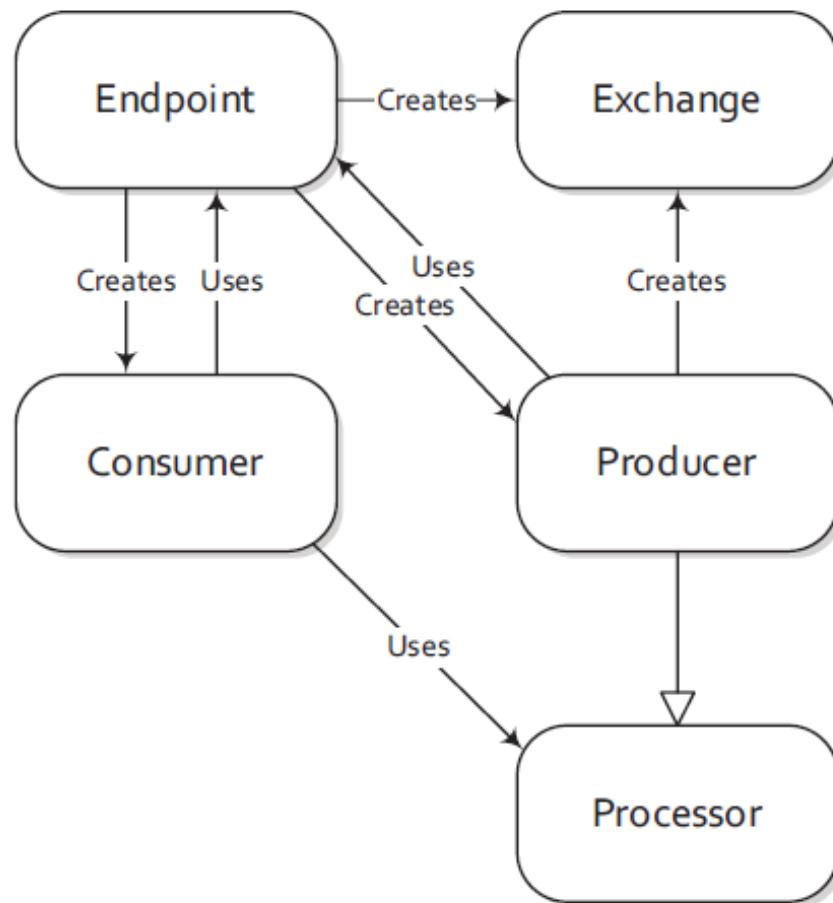
*Done!*

# **Custom Components**

## Component Classes

- Once resolved from the Spring registry, the component class is used to execute a set of factory patterns:
  - Component is a factory for Endpoint objects
  - Endpoint is a factory for Consumer, Producer, and Exchange objects
  - Exchange is a factory for Message objects





**How endpoints work with  
producers, consumers, and an exchange**

## Implementing a Component

- 1. Define a new Component class**
  - Creates instances of Endpoint
- 2. Define a new Endpoint class**
  - Encapsulates the endpoint URI
  - Creates instances of Consumer, Producer, Exchange
- 3. Define a new Consumer class**
  - Choose a suitable threading model
- 4. Define a new Producer class**
  - Choose synchronous / asynchronous implementation
- 5. Define a new Exchange class, and a new Message class**
  - Only necessary when the default exchange does not work

# Component Class

- **Extend class** `org.apache.camel.impl.DefaultComponent`
  - Base class parses the URI and loads endpoint parameters
  - Must:
    - Implement `createEndpoint()`
    - Invoke `setProperties()` to inject the endpoint parameters
  - Example: a simple file consumer

```
public class MyFileComponent<E extends Exchange>
extends DefaultComponent<E> {
    protected Endpoint<E> createEndpoint(
        String uri, String remaining, Map parameters)
        throws Exception {
        File file = new File(remaining);
        MyFileEndpoint result =
            new MyFileEndpoint(file, uri, this);
        setProperties(result, parameters);
        return result;
    }
}
```

## Endpoint Class

- Extend one of these classes:
  - DefaultEndpoint - event-driven consumer threading
  - ScheduledPollEndpoint - scheduled poll consumer threading
  - DefaultPollingEndpoint - custom poll consumer threading
- Example: file endpoint uses the scheduled polling model
  - Implement `createConsumer()` `createProducer()` `isSingleton()`
  - Always pass this into both constructors
  - Always call `configureConsumer()` to inject consumer parameters (`consumer.*`)
- Adding new options to your endpoint URI
  - Add standard bean methods to the Endpoint implementation
  - Example, to add bean methods `getMax()` and `setMax()`
  - Use the option to your URIs:

```
myfile://foo/bar?max=2048
```

## Endpoint Class Example

```
public class MyFileEndpoint<E extends Exchange>
extends ScheduledPollEndpoint<E> {
    private File file;
    private int max;
    public boolean getMax() { return this.max; }
    public boolean setMax(int max) { this.max = max; }
    public boolean isSingleton() { return true; }
    public Consumer<E> createConsumer(Processor processor)
throws Exception {
    Consumer<E> result = new MyFileConsumer(this, processor);
    configureConsumer(result);
    return result;
}
public Producer<E> createProducer() throws Exception {
    Producer<E> result = new MyFileProducer(this);
    return result;
}
}
```

## Consumer Class

- Extend one of these classes:
  - `DefaultConsumer` for event-driven threading
  - `ScheduledPollConsumer` for scheduled poll threading
  - `PollingConsumerSupport` for custom poll threading
- Example: file consumer uses the scheduled polling model
  - The constructor must take a reference to a Processor object
    - The next processor in the chain
  - Implement the `poll()` method;
    - Always call `getProcessor().process()` to delegate to the first processor

## Consumer Class Example

```
public class MyFileConsumer<E extends Exchange>
extends ScheduledPollConsumer<E> implements Consumer<E> {
    protected synchronized void poll() throws Exception {
        File file = endpoint.getFile();
        Exchange exchange = endpoint.createExchange();
        exchange.setIn(new MyFileMessage(file));
        try {
            getProcessor().process(exchange);
        } catch (Exception e) {
            e.printStackTrace();
        }
        File newFile = new File(
                file.getParentFile(),
                MyFileEndpoint.RENAME_PREFIX + file.getName());
        file.renameTo(newFile);
    }
}
```

## Producer Class

- Extend DefaultProducer for all producers
- Implement AsyncProcessor for asynchronous producers
- Example: file producer will use the synchronous pattern
  - Implement the `process()` method,
    - Responsible for sending and receiving messages to and from the physical endpoint

## Producer Class Example

```
public class MyFileProducer<E extends Exchange>
    extends DefaultProducer<E> {
    public void process(Exchange exchange) throws
Exception {
        File target = createFileName();
        InputStream in =
            ExchangeHelper.getMandatoryInBody(
                exchange, InputStream.class);
        writeToFile(in, target); // helper method
    }
}
```

## The Exchange class

- Not always necessary to implement
- The `DefaultExchange` implementation is often adequate
  - Stores In, Out, and Fault messages
  - Stores the message exchange policy
  - Stores exchange properties in a hash map
- Reasons for implementing a custom exchange class:
  - Adding support for lazy instantiation of In, Out, and Fault messages
  - Adding helper methods to provide convenient, type-safe access to exchange properties or message content

## Auto-Discovery

- **Enables Camel to load component implementations on demand,**

- Based on the URI prefix

META-INF/services/org/apache/camel/component/myfile]

- **Configuring auto-discovery:**

1. **Create a Java properties file**

- Named after the component prefix, in the following sub-directory of your new component project:

META-INF/services/org/apache/camel/component

1. **Enter a single property class**

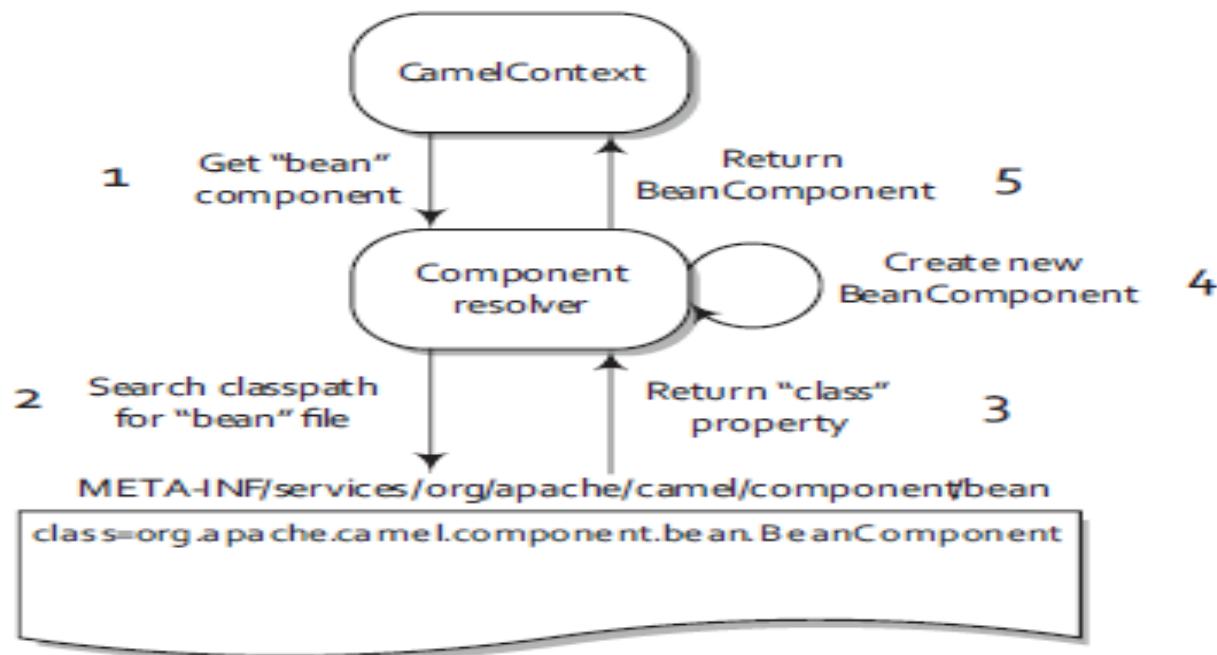
- The new component's class name

2. **Package your component as an OSGi bundle, and make sure that the new properties file is included**

META-INF/services/org/apache/camel/component/myfile]

4. **Example:**

class=com.example.MyFileComponent



To autodiscover a component named “bean”, the component resolver searches for a file named “bean” in a specific directory on the classpath. This file specifies that the component class that will be created is BeanComponent.

# Consumer Lifecycle

When you define a route that uses your new component as a consumer, like this :

```
from("helloworld:foo").to("log:result");
```

It does the following:

- ✓ creates a HelloWorldComponent instance (one per CamelContext)
- ✓ calls HelloWorldComponent createEndpoint() with the given URI
- ✓ creates a HelloWorldEndpoint instance (one per route reference)
- ✓ creates a HelloWorldConsumer instance (one per route reference)
- ✓ register the route with the CamelContext and call doStart() on the Consumer
  
- ✓ consumers will then start in one of the following modes:
  - event driven - wait for message to trigger route
  - polling consumer - manually polls a resource for events
  - scheduled polling consumer - events automatically generated by timer
  - custom threading - custom management of the event lifecycle

## Producer Lifecycle

When you define a route that uses your new component as a producer, like this

```
from("direct:start").to("helloworld:foo");
```

It does the following:

- ✓ creates a HelloWorldComponent instance (one per CamelContext)
- ✓ calls HelloWorldComponent createEndpoint() with the given URI
- ✓ creates a HelloWorldEndpoint instance (one per route reference)
- ✓ creates a HelloWorldProducer instance (one per route reference)
- ✓ register the route with the CamelContext and start the route consumer
- ✓ the Producer's process(Exchange) method is then executed
  - (generally, this will decorate the Exchange by interfacing with some external resource (file, jms, database, etc)

## **Interconnecting Routes (Direct vs Seda)**

## Interconnecting Routes

Camel supports breaking routes up into re-usable sub-routes, and synchronously or asynchronously calling those sub-routes.

## **In-memory messaging (Direct, SEDA, and VM components)**

Camel provides three main components in the core to handle in-memory messaging.

For synchronous messaging, there is the Direct component.

For asynchronous messaging, there are the SEDA and VM components.

The only difference between SEDA and VM is that the SEDA component can be used for communication within a single CamelContext, whereas the VM component is a bit broader and can be used for communication within a JVM.

If you have two CamelContexts loaded into an application server, you can send messages between them using the VM component.

# Interconnecting Routes

	<b>Within Context</b>	<b>Within JVM</b>
<b>Synchronous</b>	Direct	Direct-VM
<b>Asynchronous</b>	SEDA	VM

# Interconnecting Routes

## Direct and Direct-VM

### CamelContext-1

```
from("activemq:queue:one").to("direct:one");
from("direct:one").to("direct-vm:two");
```

### CamelContext-2

```
from("direct-vm:two").log("Direct Excitement!");
```

Run on same thread as caller

direct-vm **within JVM, including other OSGi Bundles**

# Interconnecting Routes

## SEDA and VM

### CamelContext-1

```
from("activemq:queue:one").to("seda:one");  
from("seda:one").to("vm:two");
```

### CamelContext-2

```
from("vm:two").log("Async Excitement!");
```

Run on different thread from caller

concurrentConsumers=1 controls thread count

vm within JVM, including other OSGi Bundles

# Interconnecting Routes

## SEDA and VM

```
from("activemq:queue:one") . to("seda:one") ;  
  
from("seda:one?multipleConsumers=true") . log("here") ;  
  
from("seda:one?multipleConsumers=true") . log("and there")
```

Publish / Subscribe like capability

Each route gets its own copy of the Exchange

Multicast EIP better for known set of routes

# Camel - Error Handler

## Error handling in Camel

Error handling in Camel can roughly be separated into two distinct types:

- > non transactional
- > Transactional

Where non transactional is the most common type that is enabled out-of-the-box and handled by Camel itself.

The transaction type is handled by a backing system such as a J2EE application server.

## default error handler (non transactional)

In Camel 2.0 onwards a global DefaultErrorHandler is setup as the Error Handler by default. It's configured as:

- > no redeliveries
- > no dead letter queue
- if the exchange failed an exception is thrown and propagated back to the client
- > The original caller wrapped in a RuntimeCamelException.

## Dead Letter Channel Error Handler (non transactional)

Camel supports the Dead Letter Channel from the EIP patterns using the DeadLetterChannel processor which is an Error Handler.

- Uses AOP interceptors

Inside java code :

```
errorHandler(deadLetterChannel("jms:queue:dead")
    .maximumRedeliveries(3).redeliveryDelay(5000));
```

Using the Spring XML Extensions:

```
<route errorHandlerRef="myDeadLetterErrorHandler">
    ...
</route>
```

## Transactional

Camel leverages Spring transactions. Usually you can only use this with a limited number of transport types such as JMS or JDBC based, that yet again requires a transaction manager such as a Spring transaction, a J2EE server or a Message Broker.

## TransactionErrorHandler (default for transactions)

This is the new default transaction error handler in Camel 2.0 onwards, used for transacted routes.

It uses the same base as the DefaultErrorHandler so it has the same feature set as this error handler. By default any exception thrown during routing will be propagated back to the caller and the Exchange ends immediately.

However you can use the Exception Clause to catch a given exception and lower the exception by marking it as handled. If so the exception will not be sent back to the caller and the Exchange continues to be routed.

## TransactionErrorHandler (ex- propagated back to the caller)

In this route below, any exception thrown in eg the validateOrder bean will be propagated back to the caller, and its the jetty endpoint. It will return a HTTP error message back to the client.

```
from("jetty:http://localhost/myservice/order").transacted()  
.to("bean:validateOrder").to("jms:queue:order");
```

## TransactionErrorHandler (ex-handling exception)

We can add a `onException` in case we want to catch certain exceptions and route them differently, for instance to catch a `ValidationException` and return a fixed response to the caller.

```
onException(ValidationException.class).handled(true)
.transform(body(constant("INVALID ORDER")));
```

```
from("jetty:http://localhost/myservice/order").transacted()
.to("bean:validateOrder").to("jms:queue:order");
```

## **Exception handling - Different Approaches**

## default handling

The default mode uses the DefaultErrorHandler strategy which simply propagates any exception back to the caller and ends the route immediately.

This is rarely the desired behavior, at the very least, you should define a **generic/global exception handler** to log the errors and put them on a queue for further analysis (during development, testing, etc).

```
onException(Exception)
    .to("log:GeneralError?level=ERROR")
    .to("activemq:GeneralErrorQueue");
```

## **try-catch-finally**

This approach mimics the Java for exception handling and is designed to be very readable and easy to implement. It inlines the try/catch/finally blocks directly in the route and is useful for route specific error handling.

```
from("direct:start")
    .doTry()
        .process(new MyProcessor())
    .doCatch(Exception.class)
        .to("mock:error");
    .doFinally()
        .to("mock:end");
```

## **onException**

This approach defines the exception clause separately from the route. This makes the route and exception handling code more readable and reusable. Also, the exception handling will apply to any routes defined in its CamelContext.

```
from("direct:start")
    .process(new MyProcessor())
    .to("queue:end");
```

```
onException(Exception.class)
    .to("queue:error");
```

## handled/continued

These APIs provide valuable control over the flow. Adding handled(true) tells Camel to not propagate the error back to the caller (should almost always be used). The continued(true) tells Camel to resume the route where it left off (rarely used, but powerful). These can both be used to control the flow of the route in interesting ways, for example...

```
from("direct:start")
    .process(new MyProcessor())
    .to("queue:end");
```

```
//send the exception back to the client (rarely used, clients need a
meaningful response)
onException(ClientException.class)
    .handled(false) //default
    .log("error sent back to the client");
```

## **handled/continued**

```
// handle the error internally
onException(HandledException.class)
    .handled(true)
    .setBody(constant("error"))
    .to("queue:error");

//ignore the exception and continue the route
onException(ContinuedException.class)
    .continued(true);
```

## using a processor for more control

If you need more control of the handler code, you can use an inline Processor to get a handle to the exception that was thrown and write your own handle code...

```
onException(Exception.class)
    .process(new Processor() {
        public void process(Exchange exchange) throws
Exception {
            Exception exception = (Exception)
exchange.getProperty(Exchange.EXCEPTION_CAUGHT);
            //log, email, reroute, etc.
        }
    });
});
```

```
onException(Exception.class)
    .process(
        new Processor() {
            public void process(Exchange exchange) throws Exception {
                SoapFault fault = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, SoapFault.class);
                System.out.println("Fault: " + fault); // --> This returns NULL

                Exception excep = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
                System.out.println("excep: " + excep);
                System.out.println("excep message: " + excep.getMessage());
                System.out.println("excep cause: " + excep.getCause());

                SoapFault SOAPFAULT = new SoapFault(excep.getMessage(), SoapFault.FAULT_CODE_CLIENT);
                Element detail = SOAPFAULT.getOrCreateDetail();
                Document doc = detail.getOwnerDocument();
                Text tn = doc.createTextNode("this is a test");
                detail.appendChild(tn);

                exchange.getOut().setFault(true);
                exchange.getOut().setBody(SOAPFAULT);

                exchange.setProperty(Exchange.ERRORHANDLER_HANDLED, false);
                exchange.removeProperty("CamelExceptionCaught");
            }
        })
    .handled(true)
    .end();
```

# Declarative Transactions

Transaction policies are instantiated as beans in Spring XML. You can then reference a transaction policy by providing its bean ID as an argument to the `transacted()` DSL command. For example, if you want to initiate transactions subject to the behavior, `PROPAGATION_REQUIRES_NEW`, you could use the following route:

```
from("file:src/data?noop=true")
    .transacted("PROPAGATION_REQUIRES_NEW")
    .beanRef("accountService","credit")
    .beanRef("accountService","debit")
    .to("file:target/messages");
```

```
<bean id="PROPAGATION_REQUIRES_NEW"
      class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
    <property name="propagationBehaviorName"
      value="PROPAGATION_REQUIRES_NEW"/>
</bean>
```

## Sample route with PROPAGATION\_NEVER policy in Spring XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ... >

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="file:src/data?noop=true"/>
            <bean ref="accountService" method="credit"/>
            <transacted ref="PROPAGATION_REQUIRED"/>
            <bean ref="accountService" method="debit"/>
        </route>
    </camelContext>

</beans>
```

```
<transacted/>
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:okay"/>
    <!-- we mark this route as transacted. Camel will lookup the spring
        transaction manager and use it by default. We can optimally pass in
        arguments to specify a policy to use that is configured with a spring
        transaction manager of choice. However Camel supports
        convention over configuration as we can just use the defaults out of the
        box and Camel that suites in most situations -->

    <transacted/>
    <setBody>
      <constant>Tiger in Action</constant>
    </setBody>
    <bean ref="bookService"/>
    <setBody>
      <constant>Elephant in Action</constant>
    </setBody>
    <bean ref="bookService"/>
  </route>
</camelContext>
```

# AspectJ with Camel

```
@Aspect  
@Component
```

```
public class CustomAspect {  
  
    @Around("execution(*  
com.account..*.process(org.apache.camel.Exchange))  
        && args(exchange) && target(org.apache.camel.Processor)")  
  
    public Object copyHeadersAndBody(ProceedingJoinPoint pjp,  
        Exchange exchange)  
        throws Throwable {  
        .....  
  
        Object retVal = pjp.proceed();  
  
        .....  
  
        return retVal;  
    }  
}
```

Spring configuration file :

Add the below element :

```
<aop:aspectj-autoproxy />
```

# Simple Expression Language

## Simple Expression Language

The Simple Expression Language was a really simple language you can use, but has since grown more powerful. Its primarily intended for being a really small and simple language for evaluating Expression and Predicate without requiring any new dependencies or knowledge of Xpath.

To get the body of the in message: "body",  
or "in.body" or "\${body}".

A complex expression must use \${ } placeholders, such as: "Hello \${in.header.name} how are you?".

Variables :

camelId  
CaemelContext  
exchangeld  
Id  
Body  
In.body  
Out.body  
Header.foo  
In.header.foo  
Out.header

## OGNL expression support

The Simple and Bean language now supports a Camel OGNL notation for invoking beans in a chain like fashion.

Suppose the Message IN body contains a POJO which has a getAddress() method.

Then you can use Camel OGNL notation to access the address object:

```
simple("${body.address}")
simple("${body.address.street}")
simple("${body.address.zip}")
```

Camel understands the shorthand names for getters, but you can invoke any method or use the real name such as:

```
simple("${body.address}")
simple("${body.getAddress.getStreet}")
simple("${body.address.getZip}")
simple("${body.doSomething}")
```

The simple language also includes file language out of the box which means the following expression is also supported:

file:name to access the file name

file:name.noext to access the file name with no extension

file:name.ext to access the file extension

file:ext to access the file extension

file:onlyname to access the file name (no paths)

file:onlyname.noext to access the file name (no paths) with no extension

file:parent to access the parent file name

file:path to access the file path name

file:absolute is the file regarded as absolute or relative

file:absolute.path to access the absolute file path name

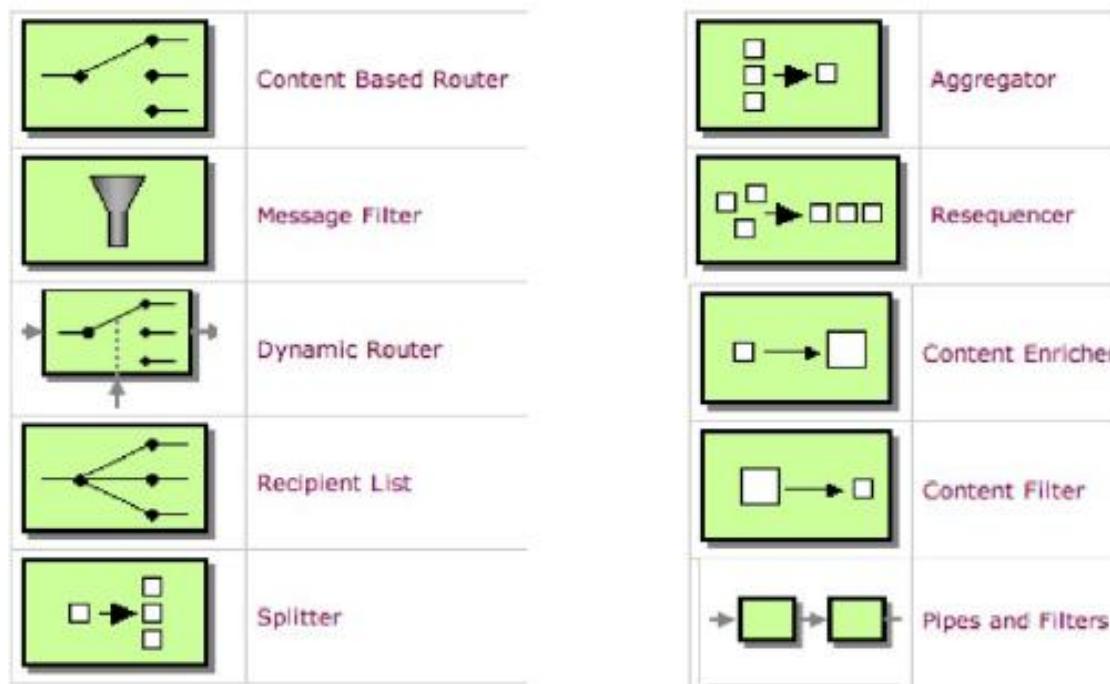
file:length to access the file length as a Long type

file:size to access the file length as a Long type

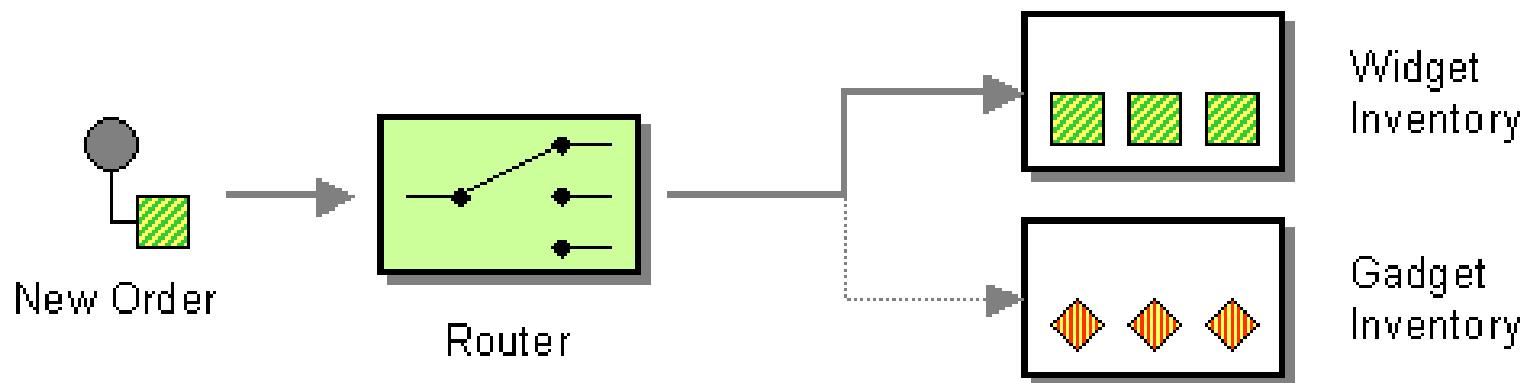
file:modified to access the file last modified as a Date type

# What is Apache Camel?

- Enterprise Integration Patterns

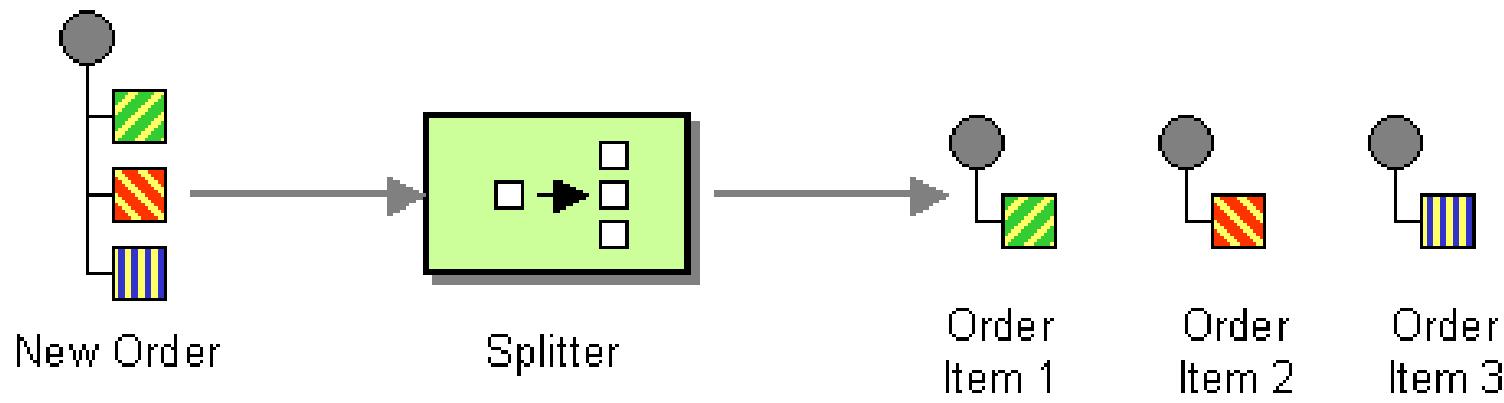


**The Content Based Router** allows you to route messages to the correct destination based on the contents of the message exchanges.



## Splitter

The Splitter from the EIP patterns allows you split a message into a number of pieces and process them individually



You need to specify a Splitter as split()

## Using the Spring XML Extensions:

```
<camelContext errorHandlerRef="errorHandler"
xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="direct:a"/>
    <split>
        <xpath>/invoice/lineItems</xpath>
        <to uri="direct:b"/>
    </split>
</route>
</camelContext>
```

## Using the Fluent Builders:

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

        from("direct:a")
            .split(body(String.class).tokenize("\n"))
            .to("direct:b");
    }
};
```

Splitting files by grouping N lines together:

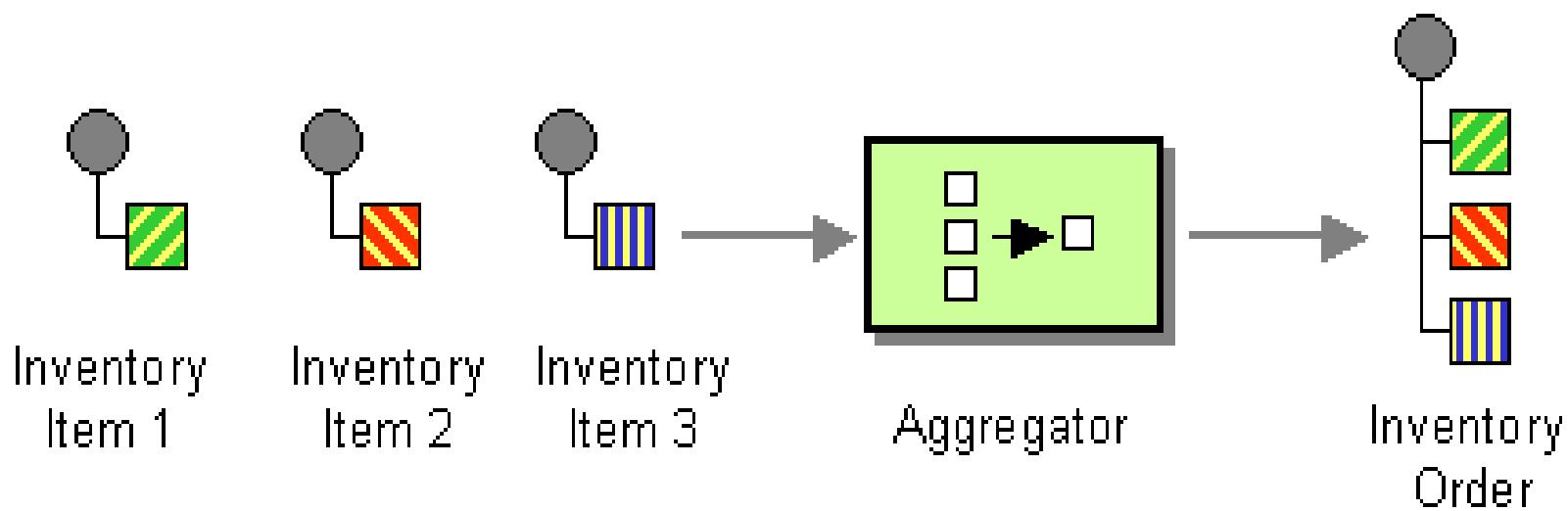
The Tokenizer language has a new option group that allows you to group N parts together, for example to split big files into chunks of 1000 lines.

```
from("file:inbox")
    .split().tokenize("\n", 1000).streaming()
        .to("activemq:queue:order");
```

And in XML DSL

```
<route>
    <from uri="file:inbox"/>
    <split streaming="true">
        <tokenize token="\n" group="1000"/>
        <to uri="activemq:queue:order"/>
    </split>
</route>
```

**The Aggregator** from the EIP patterns allows you to combine a number of messages together into a single message.



A correlation Expression is used to determine the messages which should be aggregated together. If you want to aggregate all messages into a single message, just use a constant expression.

An AggregationStrategy is used to combine all the message exchanges for a single correlation key into a single message exchange.

Ex1:

```
from("activemq:abc").aggregator(header("JMSDestination"))
.to("activemq:xyz");
```

Ex2:

```
from("seda:start").aggregate().xpath("/stockQuote/@symbol",
String.class).batchSize(5).to("mock:result");
```

## **completion-predicate**

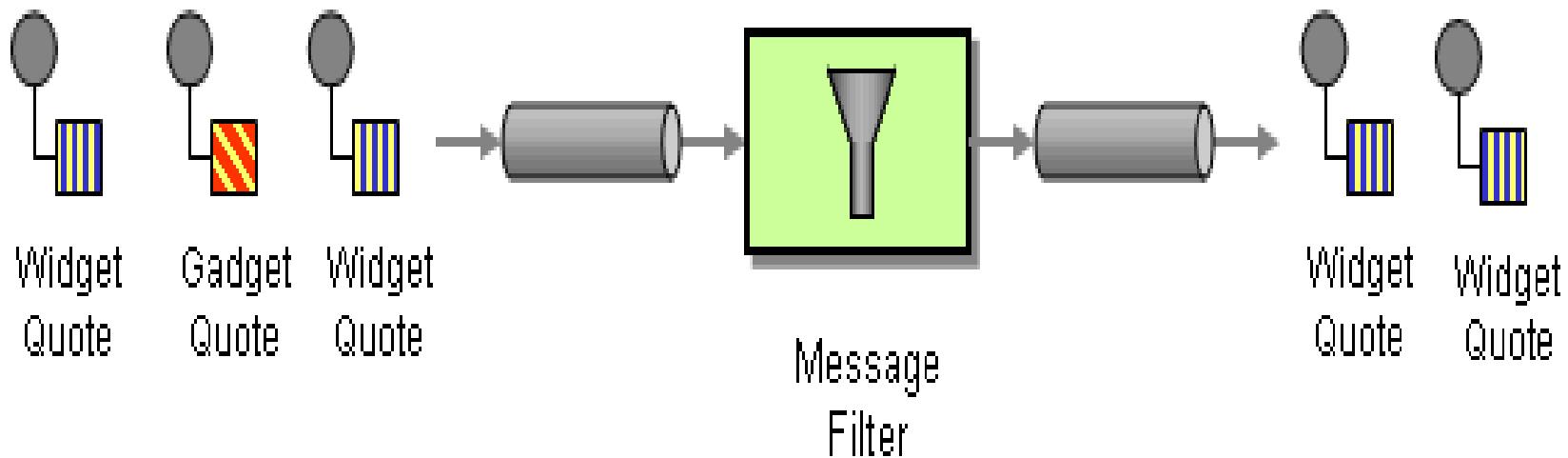
```
<route>
    <from uri="direct:aggregator" />
    <aggregate completionSize="500" completionInterval="60000"
eagerCheckCompletion="true">
        <correlationExpression>
            <xpath>/fizz/buzz</xpath>
        </correlationExpression>

        <completion-predicate>
            <simple>${property.fireNow} == 'true'</simple>
        </completion-predicate>

        <to uri="bean:postProcessor?method=run" />
    </aggregate>
</route>
```

## Message Filter

The Message Filter from the EIP patterns allows you to filter messages



Inside java code :

```
RouteBuilder builder = new RouteBuilder() {  
    public void configure() {  
        errorHandler(deadLetterChannel("mock:error"));  
  
        from("direct:a")  
            .filter(header("foo").isEqualTo("bar"))  
            .to("direct:b");  
    }  
};
```

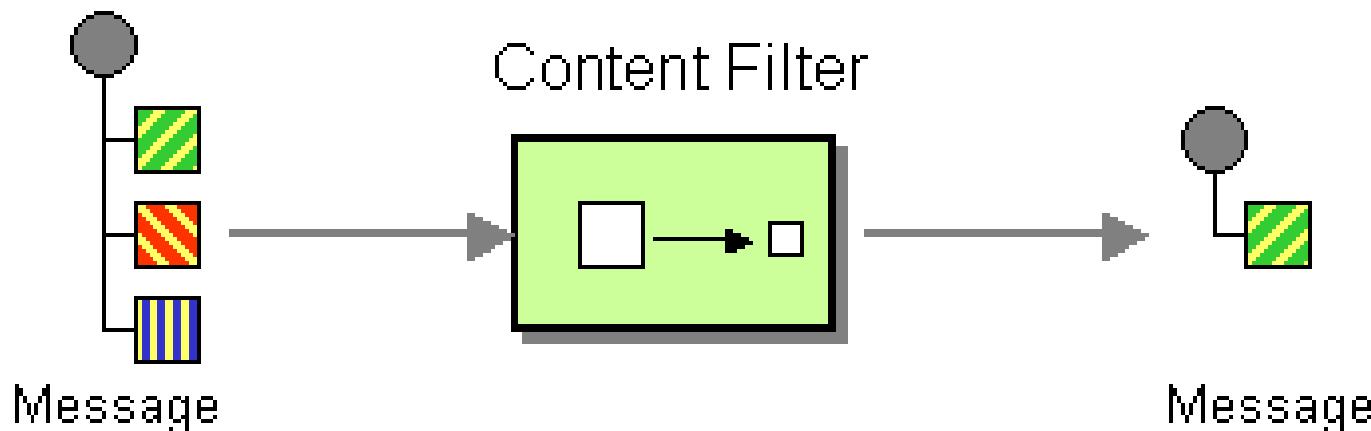
## Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
xmlns="http://camel.apache.org/schema/spring">
<route>
    <from uri="direct:a"/>
    <filter>
        <xpath>$foo = 'bar'</xpath>
        <to uri="direct:b"/>
    </filter>
</route>
</camelContext>
```

## Content Filter

Camel supports the Content Filter from the EIP patterns using one of the following mechanisms in the routing logic to transform content from the inbound message.

- Message Translator
- invoking a Java bean
- Processor object



simple example using the DSL directly

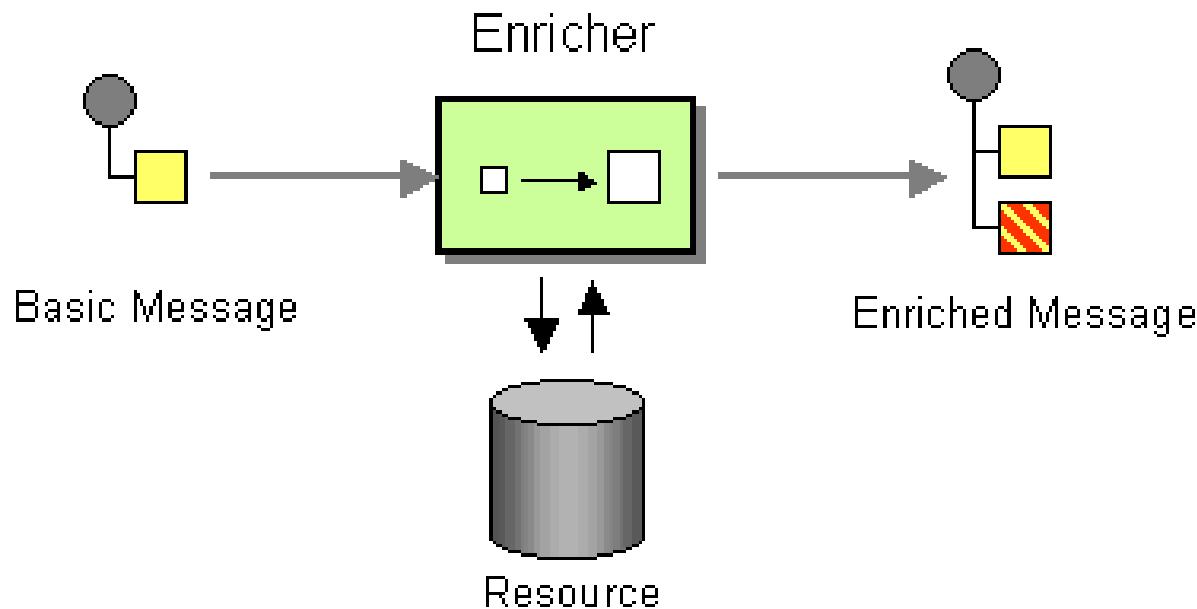
```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

## Custom Processor

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

## Content Enricher

Camel supports the Content Enricher from the EIP patterns using a Message Translator, an arbitrary Processor in the routing logic, or using the enrich DSL element to enrich the message.



## **Content enrichment using the enrich DSL element**

Camel comes with two flavors of content enricher in the DSL

enrich

pollEnrich

enrich uses a Producer to obtain the additional data. It is usually used for Request Reply messaging, for instance to invoke an external web service.

pollEnrich on the other hand uses a Polling Consumer to obtain the additional data. It is usually used for Event Message messaging, for instance to read a file or download a FTP file.

```
AggregationStrategy aggregationStrategy = ...
```

```
from("direct:start")
.enrich("direct:resource", aggregationStrategy)
.to("direct:result");
```

```
from("direct:resource")
```

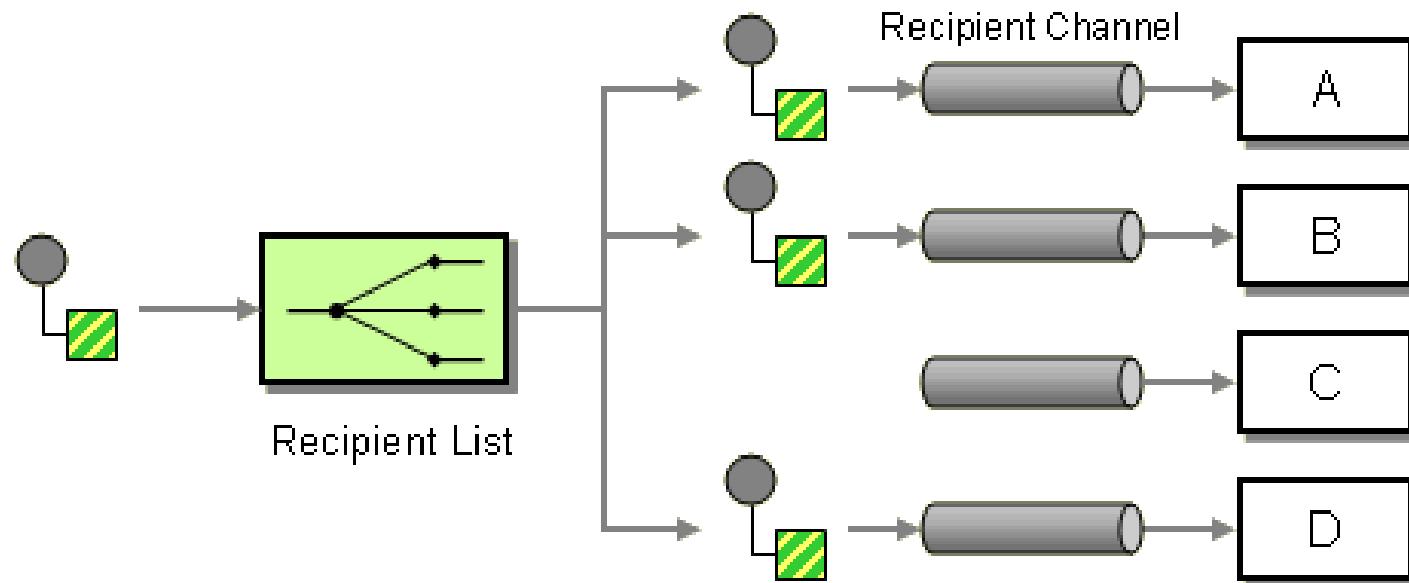
```
...
```

```
public class ExampleAggregationStrategy implements AggregationStrategy {  
  
    public Exchange aggregate(Exchange original, Exchange resource) {  
        Object originalBody = original.getIn().getBody();  
        Object resourceResponse = resource.getIn().getBody();  
        Object mergeResult = ... // combine original body and resource response  
        if (original.getPattern().isOutCapable()) {  
            original.getOut().setBody(mergeResult);  
        } else {  
            original.getIn().setBody(mergeResult);  
        }  
        return original;  
    }  
}
```

## Recipient List

The Recipient List from the EIP patterns allows you to route messages to a number of dynamically specified recipients.

The recipients will receive a copy of the same Exchange, and Camel will execute them sequentially



## **Static Recipient List**

```
<camelContext id="buildStaticRecipientList"
xmlns="http://camel.apache.org/schema/spring">
<route>
<from uri="seda:a"/>
<to uri="seda:b"/>
<to uri="seda:c"/>
<to uri="seda:d"/>
</route>
</camelContext>
```

## **Static Recipient List (using multicast)**

```
<camelContext errorHandlerRef="errorHandler"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:a"/>
    <multicast>
      <to uri="direct:b"/>
      <to uri="direct:c"/>
      <to uri="direct:d"/>
    </multicast>
  </route>
</camelContext>
```

## Dynamic Recipient List

The following example shows how to extract the list of destinations from a message header called recipientListHeader, where the header value is a comma-separated list of endpoint URIs:

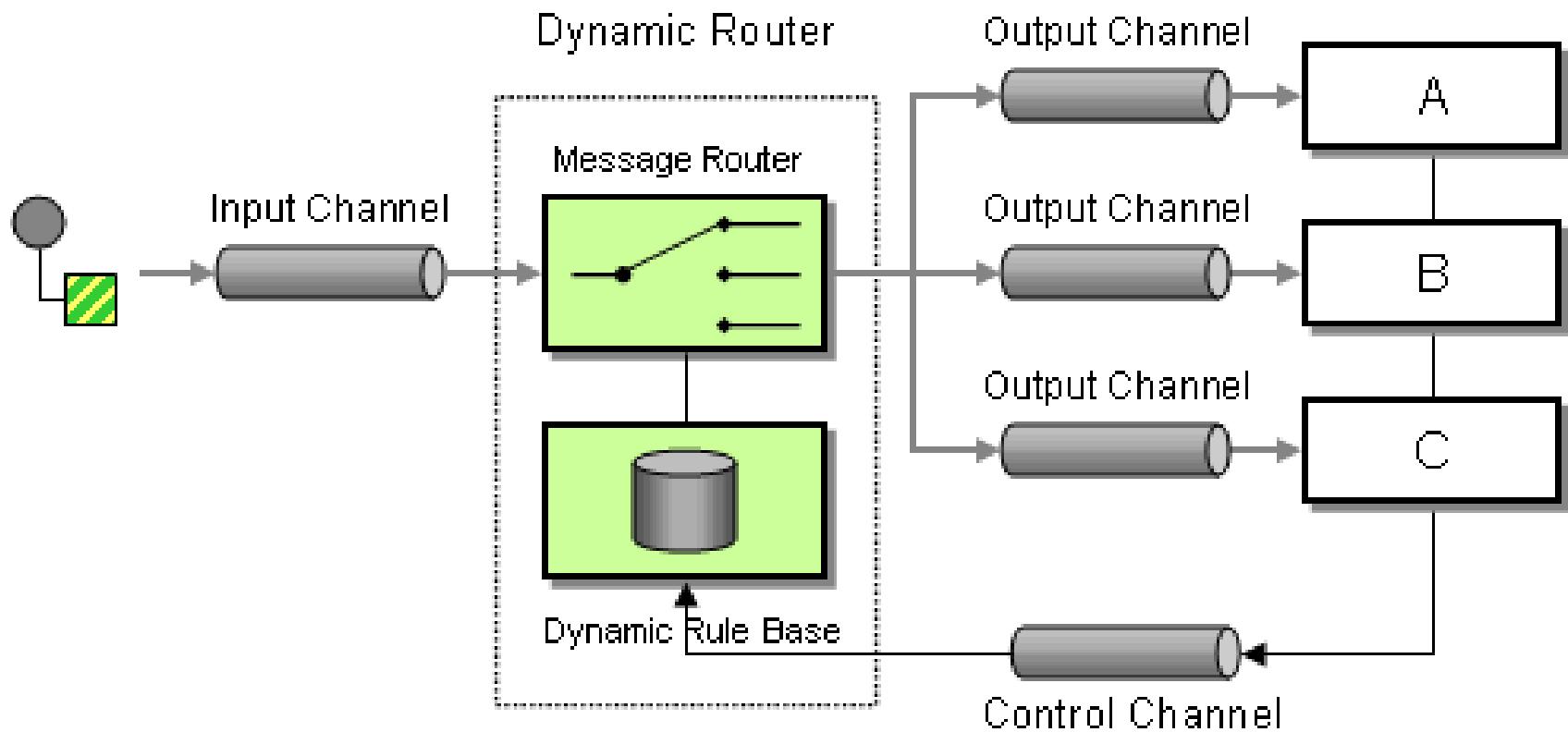
```
from("direct:a")
    .recipientList(header("recipientListHeader"))
        .tokenize(",");
```

## **Dynamic Router**

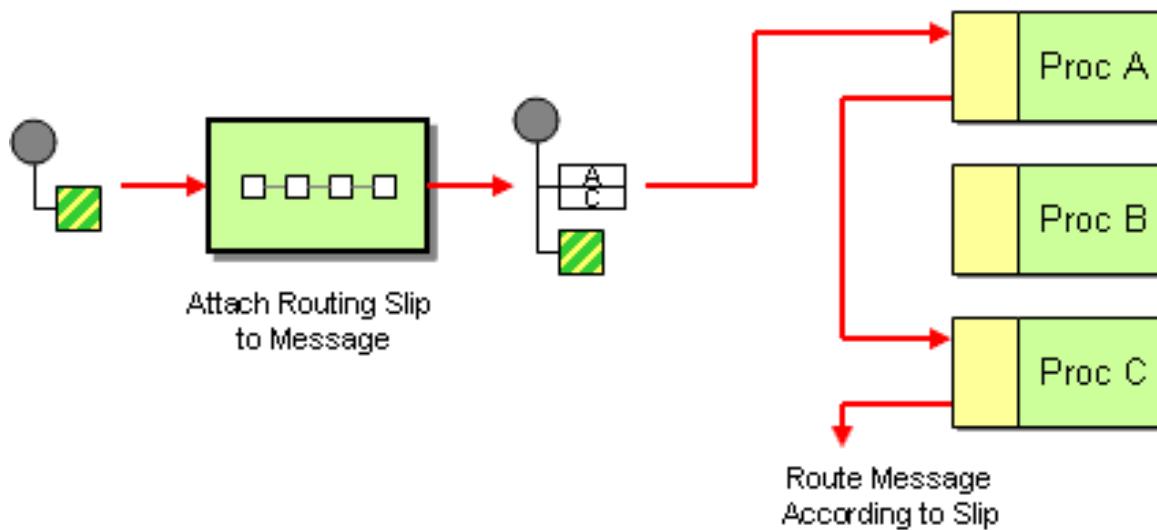
This pattern enables you to route a message consecutively through a series of processing steps, where the sequence of steps is not known at design time.

The list of endpoints through which the message should pass is calculated dynamically at run time.

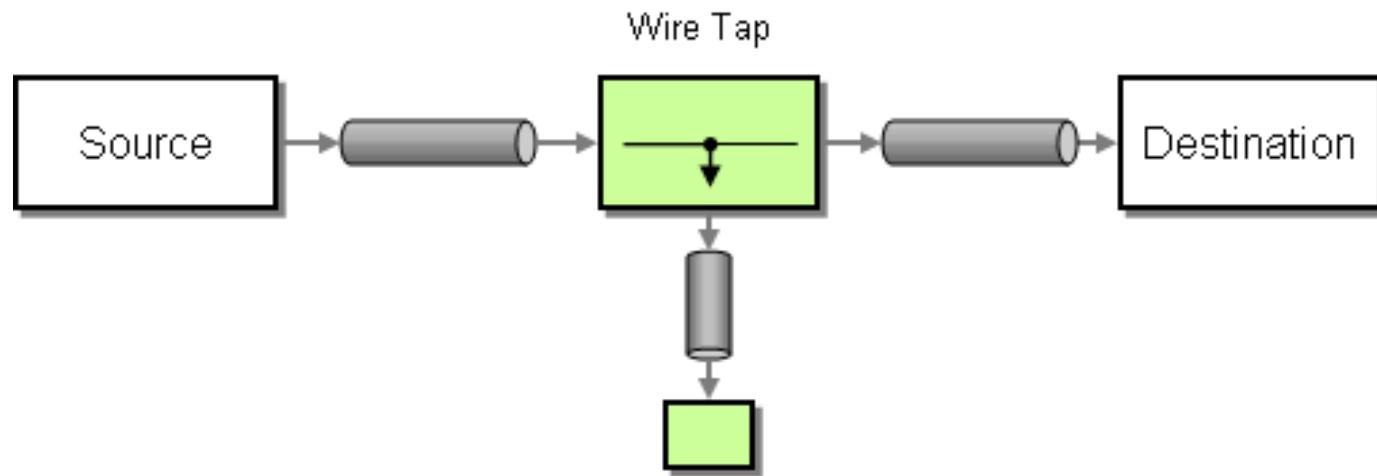
Each time the message returns from an endpoint, the dynamic router calls back on a bean to discover the next endpoint in the route.



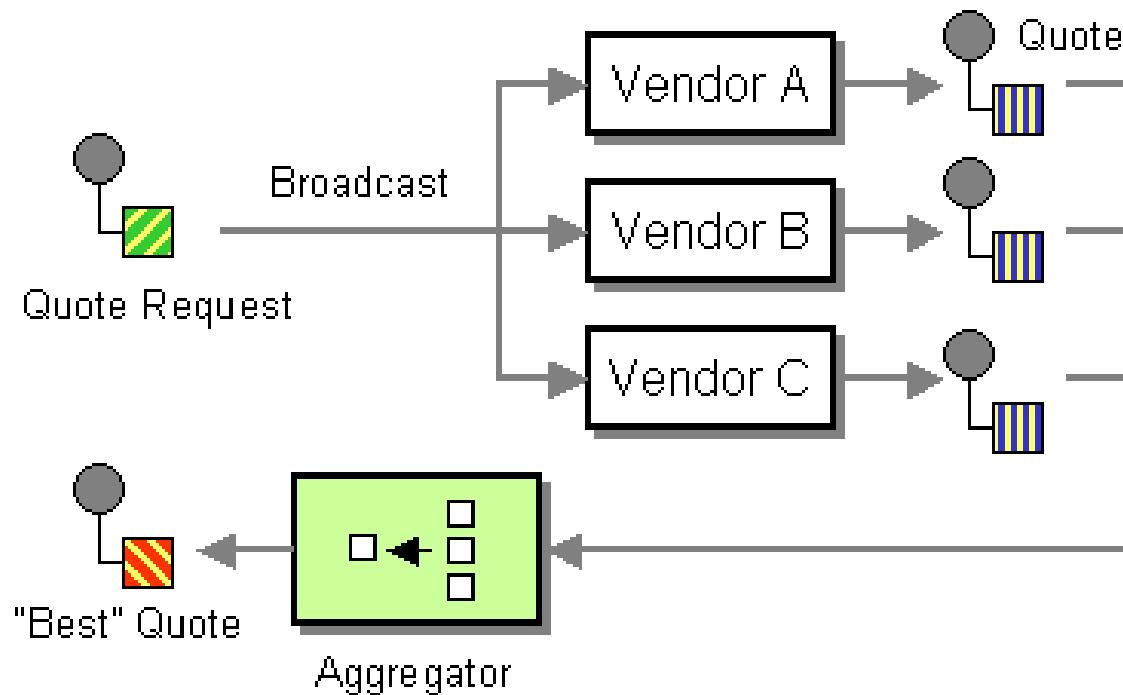
**The Routing Slip** from the EIP patterns allows you to route a message consecutively through a series of processing steps where the sequence of steps is not known at design time and can vary for each message.



**Wire Tap** allows you to route messages to a separate location while they are being forwarded to the ultimate destination



**The Scatter-Gather** allows you to route messages to a number of dynamically specified recipients and re-aggregate the responses back into a single message.



**The Throttler Pattern** allows you to ensure that a specific endpoint does not get overloaded, or that we don't exceed an agreed SLA with some external service.

```
<route>
  <from uri="seda:a"/>
  <!-- throttle 3 messages per 10 sec -->
  <throttle timePeriodMillis="10000">
    <constant>3</constant>
    <to uri="mock:result"/>
  </throttle>
</route>
```

## Load Balancer

The Load Balancer Pattern allows you to delegate to one of a number of endpoints using a variety of different load balancing policies.

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
<route>
  <from uri="direct:start"/>
  <loadBalance>
    <roundRobin/>
    <to uri="mock:x"/>
    <to uri="mock:y"/>
    <to uri="mock:z"/>
  </loadBalance>
</route>
</camelContext>
```

# **Multi-Threaded Routes**

## **Threading :**

In a Camel route, the consumer initiates the message flow by :

- Setting up a listener against the endpoint, and determining the MEP
- Creating and populating a Message Exchange
- Choosing a threading model
- Invoking the first processor

Routes are usually single-threaded by default

## **Synchronous and Asynchronous Processing :**

Sync/Async depends on the consumer's endpoint

- Synchronous (e.g. webservice) processing uses a single thread to process the message request from start to finish and delivers a response, even through multiple route segments.
- Asynchronous (e.g. JMS) processing assigns a thread to each route segment which reads a request from a BlockingQueue, processes it and since there is no response expected the thread returns earlier than in the Synchronous processing.

## Threading Mechanisms :

Camel provide explicit threading control which allows you to switch between Synchronous and asynchronous processing. This can make the Consumer to be multi-threaded.

To siwtch between sync/async, we can set the MEP :

- Using the camel processors inOut, inOnly, setExchangePattern
- Using the option exchangePattern on some consumer endpoints
- Explicitly setting the provider's MEP as a parameter in the processor

For example :

```
From ("jms:queueA")
    .inOut()
        .to("cxf:service")
            .to(ExchangePattern.InOnly, "seda:results")
```

## Multi-Threaded Consumers :

Most consumers are single-threaded by default. However many of the Consumer endpoints provide an option concurrentConsumers.

When this option is set, each concurrent consumer gets an independent Thread. Then in each thread, the route's entire logic can execute. Thus, the route now becomes multi-threaded.

Ex :

From ("jms:queueA?concurrentConsumers=5").to(...)

From("seda:test?concurrentConsumers=5").to(..)

Dynamic thread pool :

```
From("seda:input")
    .threads(8)
    .coreSize(5)
    .maxSize(20)....
```

## Apache Camel thread pool API

The Apache Camel thread pool API builds on the Java concurrency API by providing a central factory (of org.apache.camel.spi.ExecutorServiceStrategy type) for all of the thread pools in our Apache Camel application.

Centralizing the creation of thread pools in this way provides several advantages, including:

- > Simplified creation of thread pools, using utility classes.
- > Integrating thread pools with graceful shutdown.
- > Threads automatically given informative names, which is beneficial for logging and management.

## **Component threading model**

Some Apache Camel components—such as SEDA, JMS, and Jetty—are inherently multi-threaded.

These components have all been implemented using the Apache Camel threading model and thread pool API.

## **Default thread pool profile settings:**

The default thread pools are automatically created by a thread factory that takes its settings from the default thread pool profile.

The default thread pool profile has the following settings :

Thread Option	Default Value
---------------	---------------

maxQueueSize	1000
--------------	------

poolSize	10
----------	----

maxPoolSize	20
-------------	----

keepAliveTime	60 (seconds)
---------------	--------------

rejectedPolicy	CallerRuns
----------------	------------

## Changing the default thread pool profile

It is possible to change the default thread pool profile settings, so that all subsequent default thread pools will be created with the custom settings. we can change the profile either in Java or in Spring XML.

For example, in the Java DSL, we can customize the poolSize option and the maxQueueSize option in the default thread pool profile, as follows:

```
// Java
import org.apache.camel.spi.ExecutorServiceStrategy;
import org.apache.camel.spi.ThreadPoolProfile;
...
ExecutorServiceStrategy strategy = context.getExecutorServiceStrategy();
ThreadPoolProfile defaultProfile = strategy.getDefaultThreadPoolProfile();

// Now, customize the profile settings.
defaultProfile.setPoolSize(3);
defaultProfile.setMaxQueueSize(100);
...
```

In the Spring DSL, you can customize the default thread pool profile, as follows:

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <threadPoolProfile
        id="changedProfile"
        defaultProfile="true"
        poolSize="3"
        maxQueueSize="100"/>
    ...
</camelContext>
```

## **Creating a custom thread pool:**

A custom thread pool can be any thread pool of `java.util.concurrent.ExecutorService` type. The following approaches to creating a thread pool instance are recommended in Apache Camel:

- > Use the `org.apache.camel.builder.ThreadPoolBuilder` utility to build the thread pool class.
  
- > Use the `org.apache.camel.spi.ExecutorServiceStrategy` instance from the current `CamelContext` to create the thread pool class.

In Java DSL, you can define a custom thread pool using the ThreadPoolBuilder, as follows:

```
// Java
import org.apache.camel.builder.ThreadPoolBuilder;
import java.util.concurrent.ExecutorService;
...
ThreadPoolBuilder poolBuilder = new ThreadPoolBuilder(context);
ExecutorService customPool =
poolBuilder.poolSize(5).maxPoolSize(5).maxQueueSize(100).build("customPool");
...
from("direct:start")
.multicast().executorService(customPool)
.to("mock:first")
.to("mock:second")
.to("mock:third");
```

In Spring DSL:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <threadPool id="customPool"
        poolSize="5"
        maxPoolSize="5"
        maxQueueSize="100" />

    <route>
        <from uri="direct:start"/>
        <multicast executorServiceRef="customPool">
            <to uri="mock:first"/>
            <to uri="mock:second"/>
            <to uri="mock:third"/>
        </multicast>
    </route>
</camelContext>
```

## **Processor threading model**

Some of the standard processors in Apache Camel create their own thread pool by default.

These threading-aware processors are also integrated with the Apache Camel threading model and they provide various options that enable you to customize the thread pools that they use.

Ex : aggregate,multicast,recipientList,split,threads,wireTap

Processor	Java DSL	Spring DSL
aggregate	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
multicast	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
recipientList	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
split	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef

	executorService() executorServiceRef() poolSize() maxPoolSize() keepAliveTime() timeUnit() maxQueueSize() rejectedPolicy()  wireTap(String uri, ExecutorService executorService) wireTap(String uri, String executorServiceRef)	@executorServiceRef @poolSize @maxPoolSize @keepAliveTime @timeUnit @maxQueueSize @rejectedPolicy
threads		
wireTap		@executorServiceRef

# **Apache CXF webservices**

Wsdl document consists of :

- Service : Combines a number of related endpoints.
- Port : Is a binding and network endpoint. Port is also called an endpoint as it defines the information necessary to access and invoke a webservice.
- Binding : defines a concrete protocol and data format for some port type

- Porttype : is a collection of operations (interface)
- Operation : is an action that can be performed on a webservice
- Message : The messages used to exchange the data with a webservices. Data includes input parameters, return values and faults.
- Types : Defines data types available to the webservices, these types are typically defined using xsd

## SOAPMessage (an XML document)

SOAPPart

SOAPEnvelope

SOAPHeader (optional)

Headers (if any)

SOAPBody

XML Content  
or SOAPFault

AttachmentPart

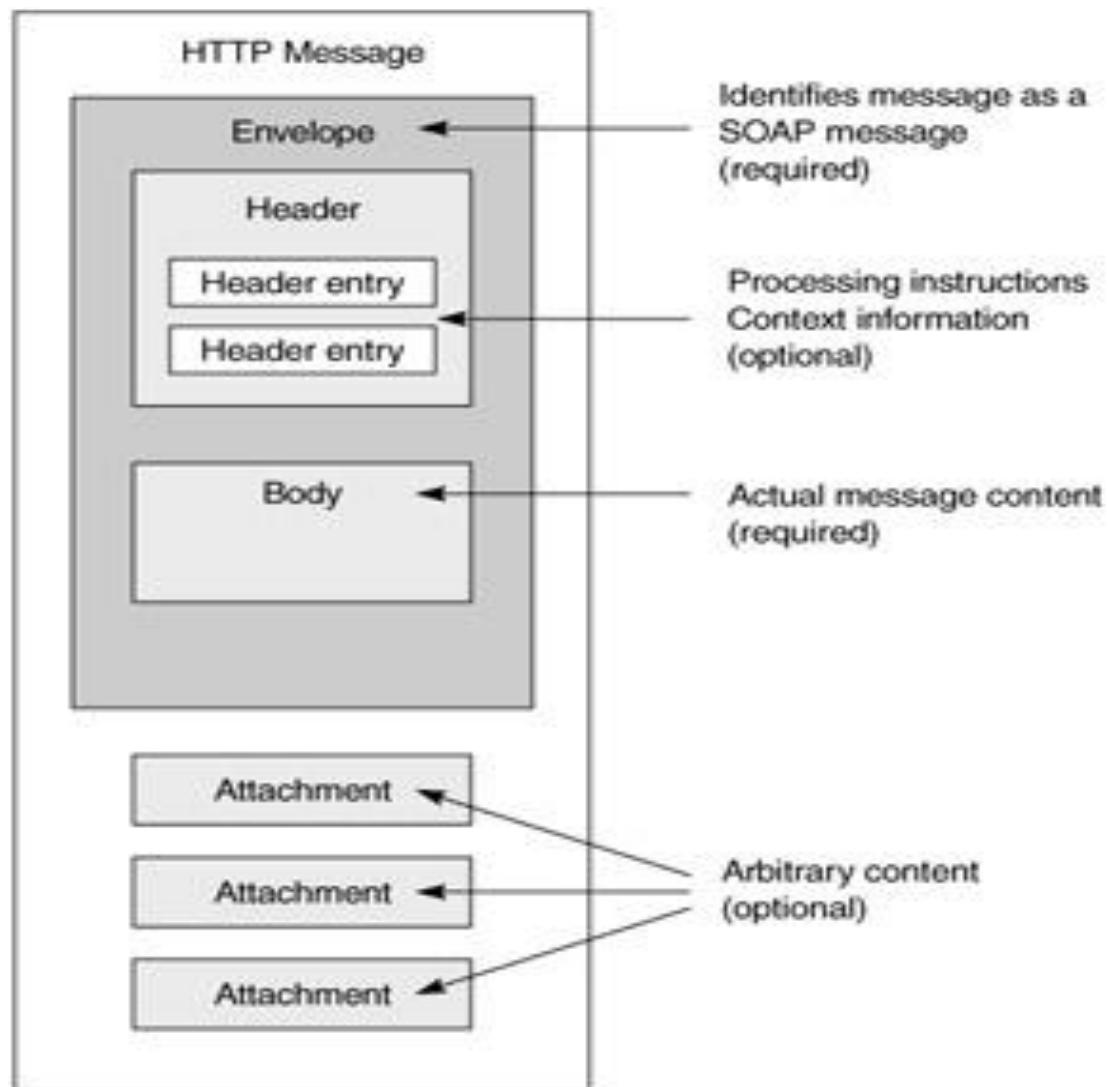
MIME Headers

Content (XML or non-XML)

AttachmentPart

MIME Headers

Content (XML or non-XML)



## CXF Component for Web Service access

- **Implements integration with CXF Web services framework,**
  - Enables definitions of
    - Web services (consumer endpoints)
    - Web clients (producer endpoints)
- **Endpoint URI format:**
  - **Address style URI:** cxf:address [ ?options ]
  - **Bean style URI:** cxf:bean:cxfEndpointId
- **Styles of endpoint URI:**
  - **Address style:** All endpoint configuration is in the URI
    - No bean necessary
    - But the URI is verbose and configuration options are limited
  - **Bean style:** References a Spring bean
    - Compact URI format
    - Flexible configuration supports CXF interceptors, features, and more

## CXF Example (1 of 2)

- Sample route with CXF endpoints:
  - The web consumer receives a SOAP/HTTP request from an external client
  - The web producer sends a SOAP/HTTP request to an external service
  - The web producer receives a SOAP/HTTP response from the external service
  - The web consumer sends a SOAP/HTTP response to the external client

```
<beans ... >
    <camelContext
        xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxft:bean:webService" />
            <to uri="cxft:bean:webClient" />
        </route>
    </camelContext>
</beans>
```

The web consumer

The web producer

This route is a simple web service proxy

## CXF Example (2 of 2)

- CXF endpoints are defined for use in Camel routes as follows:

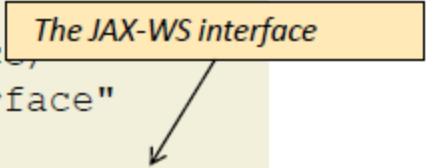
```
<beans xmlns:cxf="http://camel.apache.org/schema/cxf">

    <cxf:cxfEndpoint id="webService"
        address="http://localhost:9001/camel/webservice",
        serviceClass="com.example.webservice.DemoInterface"
        wsdlURL="wsdl/demo.wsdl"
        serviceName="tns:DemoService"
        endpointName="tns:SoapOverHttpEndpoint"
        xmlns:tns="http://example.com/webservice">
    </cxfEndpoint>

    <cxf:cxfEndpoint id="webClient"
        // identical structure to 'webservice' above...

</beans>
```

The JAX-WS interface



## CXF JAX-WS Interface

- **Setting the `serviceClass` attribute:**
  - In CXF, we set `serviceClass` to
    - the JAX-WS interface in web client endpoints
    - the JAX-WS implementation class in web service endpoints, because we want CXF to dispatch requests to our code
  - In Camel routes, we set `serviceClass` to the JAX-WS interface in all endpoints, because we want Camel to process the requests explicitly
- **Using JAX-WS annotations:**
  - Can omit the `wsdlURL`, `serviceName`, and `endpointName` attributes, provided `serviceClass` class is annotated with the relevant information using JAX-WS annotations (e.g. using the `@WebService` annotation)

## Camel-CXF Web Service Example (1 of 3)

- **Example:**  
**Implementing a web service as a Camel route**

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
        beans.xsd
        http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-spring.xsd
        http://camel.apache.org/schema/cxf
        http://camel.apache.org/schema/cxf/camel-cxf.xsd">
    ...

```

*Import the camel-spring and camel-cxf schemas*

## Camel-CXF Web Service (2 of 3)

```
<import resource="classpath:META-INF/cxf/cxf.xml"/>
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
<import resource
    ="classpath:META-INF/cxf/cxf-extension-http-jetty.xml"/>
```

*Import CXF configuration files to enable the  
SOAP, HTTP, Jetty, and CXF plug-ins*

```
<cx:cxEndpoint id="webConsumer"
    address="http://localhost:9001/camel/webservice/"
    serviceClass="com.example.webservice.DemoInterface"
    wsdlURL="wsdl/demo.wsdl"
    serviceName="tns:DemoService"
    endpointName="tns:SoapOverHttpEndpoint"
    xmlns:tns="http://example.com/webservice">
</cx:cxEndpoint>
```

...

*Define a CXF endpoint with the JAX-WS  
interface as its service class*

## CXF Payload

- The `dataFormat` option can have one of the following values:
  - **POJO**: Message body is an `Object []` array of { `return value, parameters` }
  - **PAYOUTLOAD**: Message body is same as contents of `soap:body`
  - **RAW** : Message body is raw message, represented by `InputStream`
- Set `dataFormat` option using the `cxf:properties` element:

```
<cxf:cxfEndpoint id="testEndpoint" ... >
    <cxf:properties>
        <entry key="dataFormat" value=" RAW " />
    </cxf:properties>
</cxf:cxfEndpoint>
```

Default is : RAW

## Transform the response

```
<transform>
    <constant>
        <![CDATA[
            <soap:Envelope
                xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
                <soap:Body>
                    <ns2:orderResponse
                        xmlns:ns2="http://test.code.cxf.camel.mycompany.com/">
                        <return>OK</return>
                    </ns2:orderResponse>
                </soap:Body>
            </soap:Envelope>
        ]]>
    </constant>
</transform>
```

**Select the PAYLOAD format**, if we want to access the SOAP message body in XML format, encoded as a DOM object (that is, of org.w3c.dom.Node type).

One of the advantages of the PAYLOAD format is that no JAX-WS and JAX-B stub code is required, which allows your application to be dynamic, potentially handling many different WSDL interfaces.

The SOAP body is marshalled as follows:

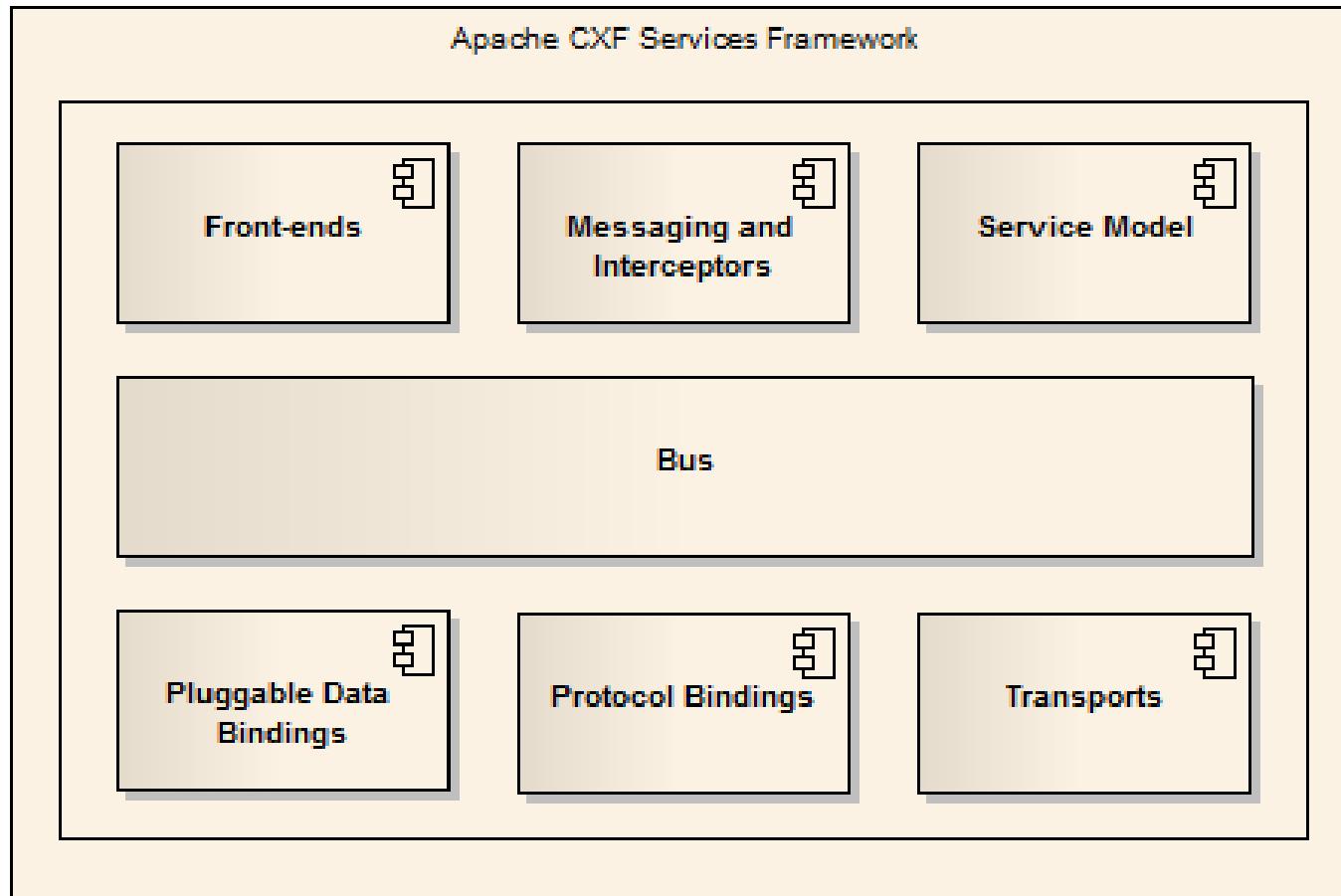
The message body is effectively an XML payload of org.w3c.dom.Node type (wrapped in a CxfPayload object).

The type of the message body is  
org.apache.camel.component.cxf.CxfPayload.

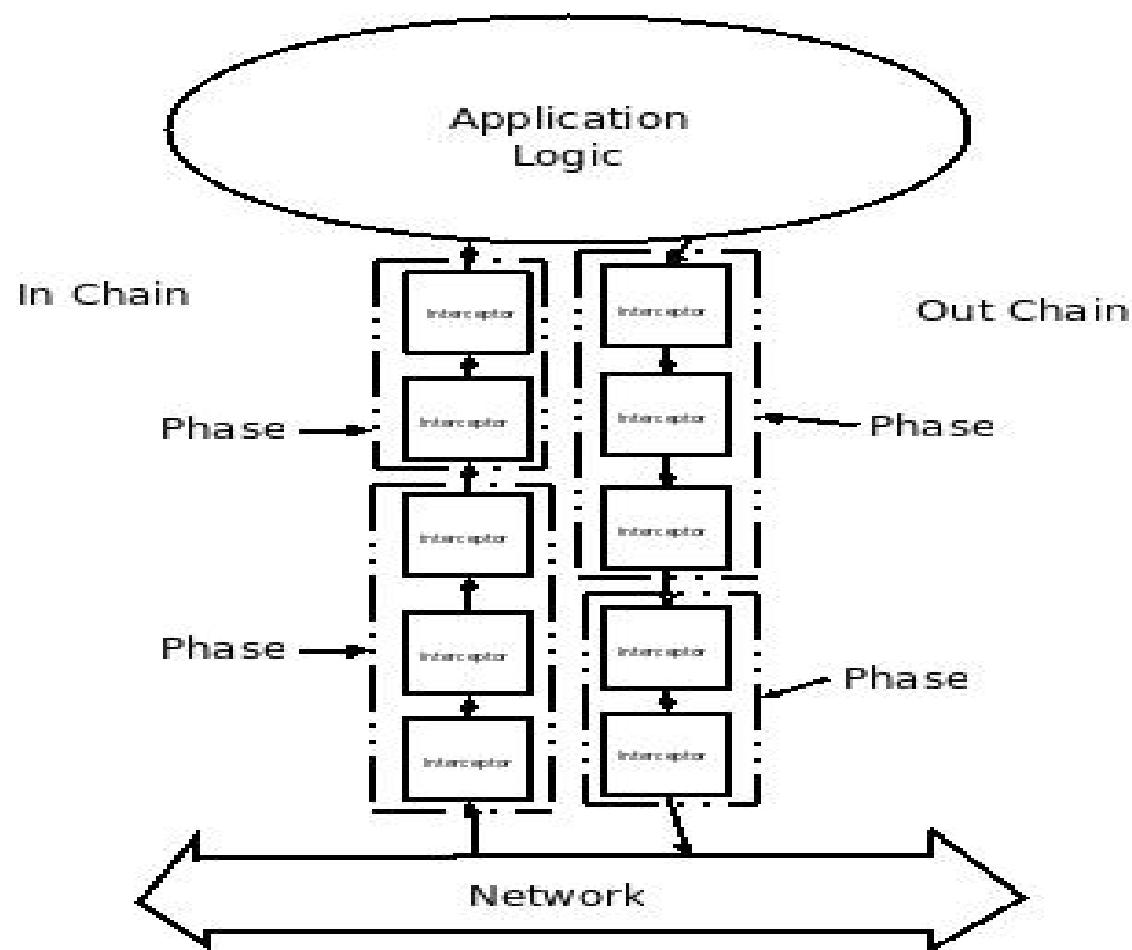
The overall CXF architecture is primarily made up of the following parts:

- Bus: Contains a registry of extensions, interceptors and Properties
- Front-end: Front-ends provide a programming model to create services.
- Messaging & Interceptors: These provide the low level message and pipeline layer upon which most functionality is built.

- Service Model: Services host a Service model which is a WSDL-like model that describes the service
- Pluggable Data Bindings: ...
- Protocol Bindings: Bindings provide the functionality to interpret the protocol.
- Transports: Transport factory creates Destinations (Receiving) and Conduits (Sending)



## *Apache CXF interceptor chains*



**Interceptors** are the fundamental processing unit inside CXF. When a service is invoked, an InterceptorChain is created and invoked.

Each interceptor gets a chance to do what they want with the message.

Ex : reading the message, transforming it, processing headers, validating the message, etc.

Interceptors are used with both CXF clients and CXF servers. When a CXF client invokes a CXF server, there is an outgoing interceptor chain for the client and an incoming chain for the server.

When the server sends the response back to the client, there is an outgoing chain for the server and an incoming one for the client.

Additionally, in the case of SOAPFaults, a CXF web service will create a separate outbound error handling chain and the client will create an inbound error handling chain.

Some examples of interceptors inside CXF include:

**SoapActionInterceptor** - Processes the SOAPAction header and selects an operation if it's set.

**Attachment(In/Out)Interceptor** - Turns a multipart/related message into a series of attachments.

## **Custom Interceptor :**

```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class AttachmentInInterceptor extends
AbstractPhaseInterceptor<Message> {
    public AttachmentInInterceptor() {
        super(Phase.RECEIVE);
    }

    public void handleMessage(Message message) {

    }

    public void handleFault(Message messageParam) {
    }
}
```

```
public class SoapActionInInterceptor extends AbstractSoapInterceptor {  
  
    public void handleMessage(SoapMessage message) throws Fault {  
  
    }  
  
    private void getAndSetOperation(SoapMessage message, String  
action) {  
  
    }  
}
```

## Applying interceptors :

```
@org.apache.cxf.interceptor.InInterceptors (interceptors =
{"com.example.Test1Interceptor" })
@org.apache.cxf.interceptor.InFaultInterceptors (interceptors =
{"com.example.Test2Interceptor" })
@org.apache.cxf.interceptor.OutInterceptors (interceptors =
{"com.example.Test1Interceptor" })
@org.apache.cxf.interceptor.InFaultInterceptors (interceptors =
{"com.example.Test2Interceptor","com.example.Test3Intercepotor" })

@WebService(endpointInterface =
"org.apache.cxf.javascript.fortest.SimpleDocLitBare",
targetNamespace = "uri:org.apache.cxf.javascript.fortest")

public class SayHiImplementation implements SayHi {
    public long sayHi(long arg) {
        return arg;
    }
    ...
}
```

MyInterceptor to the bus:

```
<beans>  
    <bean id="MyInterceptor"  
          class="demo.interceptor.MyInterceptor"/>  
  
    <cxf:bus>  
        <cxf:inInterceptors>  
            <ref bean="MyInterceptor"/>  
        </cxf:inInterceptors>  
        <cxf:outInterceptors>  
            <ref bean="MyInterceptor"/>  
        </cxf:outInterceptors>  
    </cxf:bus>  
</beans>
```

## **Asynchronous invocation with a callback**

The Synchronization callback interface is defined as follows:

```
package org.apache.camel.spi;

import org.apache.camel.Exchange;

public interface Synchronization {
    void onComplete(Exchange exchange);
    void onFailure(Exchange exchange);
}
```

```
Exchange exchange = context.getEndpoint("direct:start").createExchange();

exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncCallback("direct:start", exchange, new
SynchronizationAdapter() {
    @Override
    public void onComplete(Exchange exchange) {
        assertEquals("Hello World", exchange.getIn().getBody());
    }
});
```

You still have the option of accessing the reply from the main thread, because the `asyncCallback()` method also returns a `Future` object—for example:

```
// Retrieve the reply from the main thread, specifying a timeout
Exchange reply = future.get(10, TimeUnit.SECONDS);
```

# JPA Endpoint

Using a consumer with a named query:

For consuming only selected entities, we can use the consumer.namedQuery URI query option.

First, we have to define the named query in the JPA Entity class:

```
@Entity  
@NamedQuery(name = "step1", query = "select x from MultiSteps x where  
x.step = 1")  
public class MultiSteps {  
    ...  
}
```

After that you can define a consumer uri like this one:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.namedQuery=step1")
.to("bean:myBusinessLogic");
```

## Using XSLT endpoints

For example you could use something like

```
from("activemq:My.Queue").  
to("xslt:com/acme/mytransform.xsl");
```

To use an XSLT template to formulate a response for a message for InOut message exchanges (where there is a JMSReplyTo header).

If you want to use InOnly and consume the message and send it to another destination you could use the following route:

```
from("activemq:My.Queue").  
to("xslt:com/acme/mytransform.xsl").  
to("activemq:Another.Queue");
```

Jdbc component :

```
from("direct:projects")
.setHeader("lic", constant("ASF"))
.setHeader("min", constant(123))
.setBody("select * from projects where license = :?lic and id > :?min order by
id")
.to("jdbc:myDataSource?useHeadersAsParameters=true")
```

\* This component can only be used to define producer endpoints

## **Controlling Start-Up and Shutdown of Routes**

By default, routes are automatically started when your Apache Camel application (as represented by the CamelContext instance) starts up and routes are automatically shut down when your Apache Camel application shuts down.

For non-critical deployments, the details of the shutdown sequence are usually not very important. But in a production environment, it is often crucial that existing tasks should run to completion during shutdown, in order to avoid data loss.

We typically also want to control the order in which routes shut down, so that dependencies are not violated (which would prevent existing tasks from running to completion).

Apache Camel provides a set of features to support graceful shutdown of applications.

## Setting the route ID

It is good practice to assign a route ID to each of your routes. As well as making logging messages and management features more informative, the use of route IDs enables you to apply greater control over the stopping and starting of routes.

Java DSL:

```
from("SourceURI").routeId("myCustomRouteId").process(...).to(TargetURI);
```

Spring DSL:

```
<camelContext id="CamelContextID">  
  <route id="myCustomRouteId" >
```

Disabling automatic start-up of routes:

We can disable automatic start-up of a route in the Java DSL by invoking noAutoStartup()

Java DSL:

```
from("SourceURI").routeId("nonAuto").noAutoStartup().to(TargetURI);
```

Spring DSL:

```
<camelContext id="CamelContextID" >  
  <route id="nonAuto" autoStartup="false">
```

## **Manually starting and stopping routes :**

We can manually start or stop a route at any time in Java by invoking the startRoute() and stopRoute() methods on the CamelContext instance.

```
context.startRoute("nonAuto");
```

```
context.stopRoute("nonAuto");
```

# Startup order of routes

By default, Apache Camel starts up routes in a non-deterministic order. In some applications, however, it can be important to control the startup order.

```
from("jetty:http://fooserver:8080")
    .routeld("first")
    .startupOrder(2)
    .to("seda:buffer");
```

```
from("seda:buffer")
    .routeld("second")
    .startupOrder(1)
    .to("mock:result");
```

The route with the lowest integer value starts first, followed by the routes with successively higher startup order values.

# Shutdown sequence

When a CamelContext instance is shutting down, Apache Camel controls the shutdown sequence using a pluggable shutdown strategy. The default shutdown strategy implements the following shutdown sequence:

- > Routes are shut down in the reverse of the start-up order.
- > Normally, the shutdown strategy waits until the currently active exchanges have finished processing. The treatment of running tasks is configurable, however.
- > Overall, the shutdown sequence is bound by a timeout (default, 300 seconds). If the shutdown sequence exceeds this timeout, the shutdown strategy will force shutdown to occur, even if some tasks are still running.

## **Shutting down running tasks in a route:**

`ShutdownRunningTask.CompleteCurrentTaskOnly`

(Default) Usually, a route operates on just a single message at a time, so you can safely shut down the route after the current task has completed.

`ShutdownRunningTask.CompleteAllTasks`

Specify this option in order to shut down batch consumers gracefully. Some consumer endpoints (for example, File, FTP, Mail, iBATIS, and JPA) operate on a batch of messages at a time. For these endpoints, it is more appropriate to wait until all of the messages in the current batch have completed.

java DSL:

```
from("file:target/pending")
    .routeId("first").startupOrder(2)
    .shutdownRunningTask(ShutdownRunningTask.CompleteAllTasks)
```

Spring DSL:

```
<route id="first"
    startupOrder="2"
    shutdownRunningTask="CompleteAllTasks">
```

## Shutdown timeout

The shutdown timeout has a default value of 300 seconds. You can change the value of the timeout by invoking the `setTimeout()` method on the shutdown strategy.

```
context.getShutdownStrategy().setTimeout(600);
```

# **Camel type converters**

## **Camel type converters**

Camel provides a built-in type-converter system that automatically converts between well-known types

```
String custom = exchange.getIn().getBody(String.class);  
The getBody method is passed the type you want to have returned.
```

## TypeConverterRegistry

The TypeConverterRegistry is where all the type converters are registered when Camel is started. At runtime, Camel uses the TypeConverterRegistry's lookup method to look up a suitable TypeConverter:

```
TypeConverter lookup(Class<?> toType, Class<?> fromType);
```



The **TypeConverterRegistry** contains many **TypeConverters**

## **convertBodyTo:**

Convert the message body to the given class type. To do so camel uses a hierarchy of TypeConverters

Java

```
convertBodyTo(Class type [, String charset])
```

Spring XML

```
<convertBodyTo type=<String> [charset=<String>] >
```

EX:

```
<convertBodyTo type="java.lang.byte[]" charset="utf-16"/>
```

## **Dozer Type Conversion:**

Dozer is a fast and flexible framework for mapping back and forth between Java Beans. Coupled with Camel's automatic type conversion.

example : a simple Customer Support Service. The initial version of the Service defined a 'Customer' object used with a very flat structure.

```
public class Customer {  
    private String firstName;  
    private String lastName;  
    private String street;  
    private String zip;
```

```
    public Customer() {}
```

... getters and setters for each field

```
}
```

Exercises : Lab1

Compare the output for the below routes:

1. with to

```
from("direct:a").to("direct:x", "direct:y", "direct:z");
```

2. with Multicast sequentially processing

```
from("direct:a").multicast().to("direct:x", "direct:y", "direct:z");
```

3. with Multicast parallel processing

```
from("direct:a").multicast().parallelProcessing().to("direct:x", "direct:y", "direct:z");
```

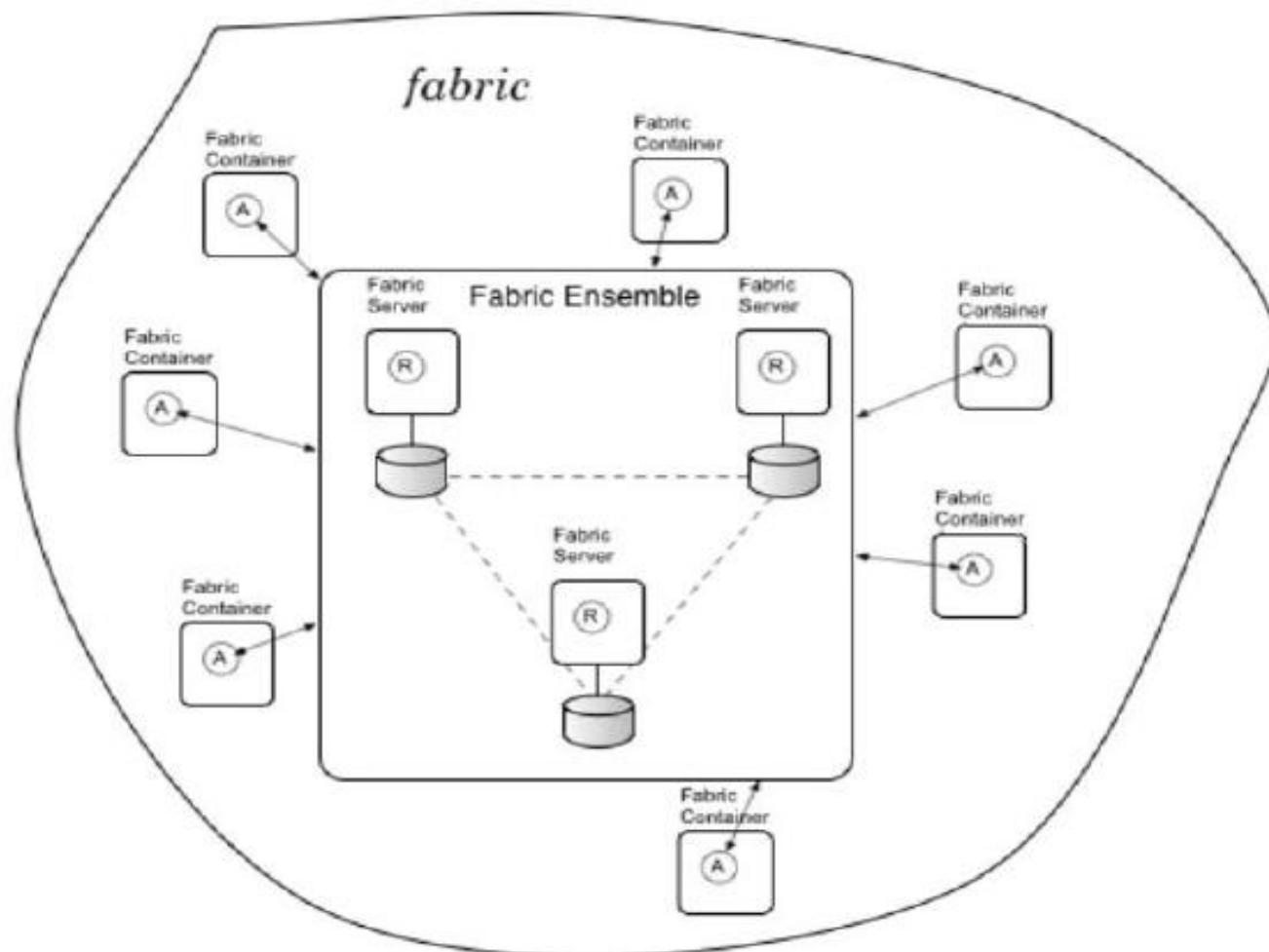
```
from("direct:x").transform().simple("ok");
```

```
from("direct:y").log("${body}");
```

```
from("direct:z").log("${body}");
```

# Fuse Fabric

# Fabric Architecture



# Fabric

A **fabric** is a collection of containers that share a fabric registry, where the **fabric registry** is a replicated database that stores all information related to provisioning and managing the containers.

A fabric is intended to manage a distributed network of containers, where the containers are deployed across multiple hosts.

## **Fabric Repository (Apache ZooKeeper)**

Fuse Fabric uses Apache ZooKeeper, which is highly reliable distributed coordination service, as its registry to store the cluster configuration and node registration.

ZooKeeper is designed with consistency and high availability across data centers in mind while also protecting against network splits by using a quorum of ZooKeeper servers

## **Conceptually Fabric has 2 registries:**

Configuration Registry which is the logical configuration of your fabric and typically contains no physical machine information; its your logical configuration.

Runtime Registry which contains details of how many machines are actually running, their physical location details and what services they are implementing.

## **Fabric Ensemble**

A Fabric Ensemble is a collection of Fabric Servers and Fabric Containers that collectively maintain the state of the fabric registry.

The Fabric Ensemble implements a replicated database and uses a quorum-based voting system to ensure that data in the fabric registry remains consistent across all of the fabric's containers.

## Zookeeper ensemble:

Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an Zookeeper ensemble.

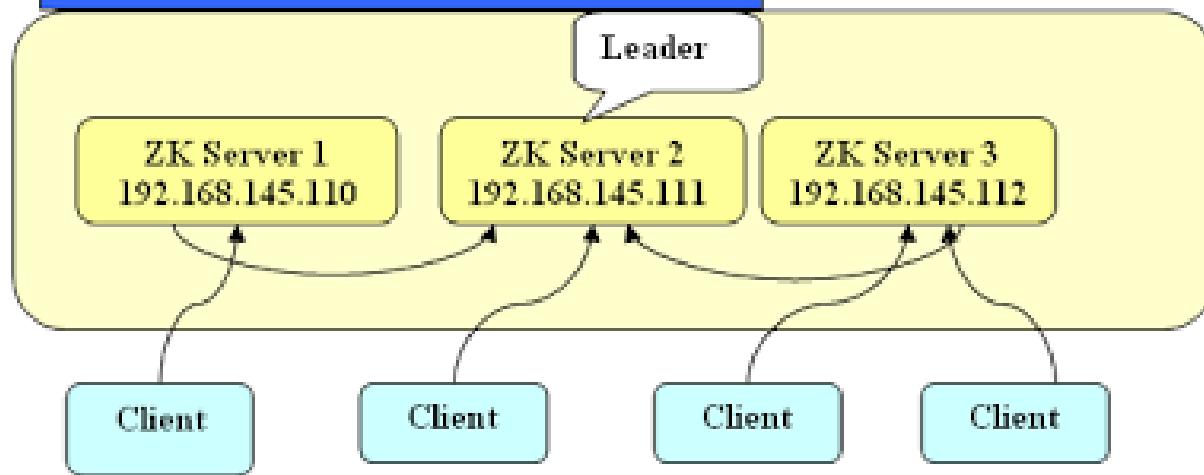
In zookeeper ensemble , all zookeeper server must all know about each other zookeeper server . They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats.

If the TCP connection to the server breaks, the client will connect to a different server.

ZooKeeper stamps each update with a number that reflects the order of all ZooKeeper transactions.

Subsequent operations can use the order to implement higher-level abstractions, such as synchronization primitives, deadlocks etc.

## Zookeeper Ensemble on across Network



server.1 : /usr/local/zookeeper1/conf/zoo.cfg

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/zookeeper1
clientPort=2184
server.1=localhost:2888:3888
server.2= localhost:2889:3889
server.3= localhost:2890:3890
```

server.2 : /usr/local/zookeeper2/conf/zoo.cfg

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/zookeeper2
clientPort=2185
server.1=localhost:2888:3888
server.2= localhost:2889:3889
server.3= localhost:2890:3890
```

## MQ-Fabric Client

Jndi.properties:

```
java.naming.factory.initial =  
org.apache.activemq.jndi.ActiveMQInitialContextFactory
```

```
java.naming.provider.url = discovery:(fabric:amq-east)
```

## **Fabric Server**

A Fabric Server has a special status in the fabric, because it is responsible for maintaining a replica of the fabric registry.

In each Fabric Server, a registry service is installed. The registry service (based on Apache ZooKeeper) maintains a replica of the registry database and provides a ZooKeeper server, which ordinary agents can connect to in order to retrieve registry data.

## **Fabric container (managed container)**

A Fabric container (or managed container) is aware of the locations of all of the Fabric Servers, and it can retrieve registry data from any Fabric Server in the Fabric Ensemble.

A Fabric agent is installed in each Fabric container.

The Fabric Agent actively monitors the fabric registry, and whenever a relevant modification is made to the registry, it immediately updates its container to keep the container consistent with the registry settings.

## **Fabric Agent**

Fabric defines a provisioning agent or Fabric agent, which relies on profiles.

The Fabric agent runs on each managed container and its role is to provision the container according to the profiles assigned to it.

The Fabric agent retrieves the configuration, bundles and features (as defined in the profile overlay), calculates what needs to be installed (or uninstalled) and, finally, performs the required actions.

## **Profile**

A Fabric profile is an abstract unit of deployment, capable of holding all of the data required for deploying an application into a Fabric Container.

Profiles are used exclusively in the context of fabrics.

A profile consists of a collection of OSGi bundles and Karaf features to be provisioned, and a list of configurations for the OSGi Configuration Administration service.

Multiple profiles can be associated with a given container, allowing the container to serve multiple purposes.

## Camel Drools with Guvnor :

```
<drools:resource-change-scanner id="s1" interval="10" enabled="true" />
<drools:resource id="cs" type="CHANGE_SET"
source="http://localhost:8080/drools-
guvnor/rest/packages/com.test/assets/CHANGE_SET/source" basic-
authentication="enabled" username="admin" password="admin" />

<drools:kagent kbase="kbase1" id="kagent" new-instance="false">
  <drools:resources>
    <drools:resource ref="cs" />
  </drools:resources>
</drools:kagent>

<drools:grid-node id="node1"/>
<drools:ksession id="ksession1" type="stateless" kbase="kbase1" node="node1"/>

<drools:kbase id="kbase1" node="node1">
  <drools:resources>
    <drools:resource ref="cs" />
  </drools:resources>
</drools:kbase>
```

# MAVEN

## What is Maven?

*“Maven is a software management and comprehension tool based on the concept of Project Object Model (POM) which can manage project build, reporting, and documentation from a central piece of information”*

## What is POM?

*“As a fundamental unit of work in Maven, POM is an XML file that contains information about project and configuration details used by Maven to build the project”*

## **What is a Maven Repository?**

In Maven terminology, a repository is a place i.e. directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily.

Maven repository are of three types

local

central

remote

## Local Repository

Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time.

To override the default location, mention another path in Maven settings.xml file available at %M2\_HOME%\conf directory.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>C:/MyLocalRepository</localRepository>
</settings>
```

## **Central Repository**

Maven central repository is repository provided by Maven community. It contains a large number of commonly used libraries.

When Maven does not find any dependency in local repository, it starts searching in central repository using following URL:

<http://repo1.maven.org/maven2/>

### **Key concepts of Central repository:**

- >> This repository is managed by Maven community.
- >> It is not required to be configured.
- >> It requires internet access to be searched.

To browse the content of central maven repository, maven community has provided a URL: <http://search.maven.org/#browse>. Using this library, a developer can search all the available libraries in central repository.

## **Remote Repository**

**Remote Repository** is developer's own custom repository containing required libraries or other project jars.

Pom.xml

```
<repositories>
  <repository> <id>companynname.lib1</id>
  <url>http://download.companynname.org/maven2/lib1</url>
  </repository>

  <repository>
    <id>companynname.lib2</id>
    <url>http://download.companynname.org/maven2/lib2</url>
    </repository>

</repositories>
```

## **Maven Dependency Search Sequence**

When we execute Maven build commands, Maven starts looking for dependency libraries in the following sequence:

**Step 1** - Search dependency in local repository, if not found, move to step 2 else if found then do the further processing.

**Step 2** - Search dependency in central repository, if not found and remote repository/repositories is/are mentioned then move to step 4 else if found, then it is downloaded to local repository for future reference.

**Step 3** - If a remote repository has not been mentioned, Maven simply stops the processing and throws error (Unable to find dependency).

**Step 4** - Search dependency in remote repository or repositories, if found then it is downloaded to local repository for future reference otherwise Maven as expected stop processing and throws error (Unable to find dependency).

## **What are Maven Plugins?**

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to :

- create jar file
- create war file
- compile code files
- unit testing of code
- create project documentation
- create project reports

## Plugin Types:

Maven provided following two types of Plugins:

Type	Description
Build plugins	They execute during the build and should be configured in the <build/> element of pom.xml
Reporting plugins	They execute during the site generation and they should be configured in the <reporting/> element of the pom.xml

Following is the list of few common plugins:

Plugin	Description
clean	Clean up target after the build. Deletes the target directory.
compiler	Compiles Java source files.
surefile	Run the JUnit unit tests. Creates test reports.
jar	Builds a JAR file from the current project.
war	Builds a WAR file from the current project.
javadoc	Generates Javadoc for the project.
antrun	Runs a set of ant tasks from any phase mentioned of the build.

## archetype plugins

Maven uses archetype plugins to create projects. To create a simple java application, we'll use **maven-archetype-quickstart plugin**.

In example below, We'll create a maven based java application project in C:\MVN folder.

Let's open command console, go the C:\MVN directory and execute the following mvn command.

```
C:\MVN>mvn archetype:generate  
-DgroupId=com.companyname.bank  
-DartifactId=consumerBanking  
-DarchetypeArtifactId=maven-archetype-quickstart  
-DinteractiveMode=false
```

# Camel Maven Archetypes:

camel-archetype-activemq  
camel-archetype-blueprint  
camel-archetype-component  
camel-archetype-scala  
camel-archetype-spring  
etc.,

Ex :

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-scala
-DarchetypeVersion=2.9.0
-DarchetypeRepository=https://repository.apache.org/content/
groups/snapshots-group
```

## **External Dependency. :**

Maven does the dependency management using concept of Maven Repositories. But what happens if dependency is not available in any of remote repositories and central repository? Maven provides answer for such scenario using concept of External Dependency.

For an example, let us do the following changes to project created in Maven Creating Project section.

>> Add lib folder to src folder

>> Copy any jar into the lib folder. We've used ldapjdk.jar, which is a helper library for LDAP operations

### Pom.xml

```
<dependency> <groupId>ldapjdk</groupId> <artifactId>ldapjdk</artifactId>
<scope>system</scope> <version>1.0</version>
<b><systemPath>${basedir}\src\lib\ldapjdk.jar</systemPath>
</dependency>
```

## **What is SNAPSHOT?**

SNAPSHOT is a special version that indicates a current development copy. Unlike regular versions, Maven checks for a new SNAPSHOT version in a remote repository for every build.

Now data-service team will release SNAPSHOT of its updated code everytime to repository say data-service:1.0-SNAPSHOT replacing a older SNAPSHOT jar.

### **Snapshot vs Version**

In case of Version, if Maven once downloaded the mentioned version say data-service:1.0, it will never try to download a newer 1.0 available in repository. To download the updated code, data-service version is be upgraded to 1.1.

In case of SNAPSHOT, Maven will automatically fetch the latest SNAPSHOT (data-service:1.0-SNAPSHOT) everytime app-ui team build their project.

A "release" is the final build for a version which does not change.

A "snapshot" is a build which can be replaced by another build which has the same name. It is implies the build could change at any time and is still under active development.

## Dependency Scope

Dependency scope is used to limit the transitivity of a dependency, and also to affect the classpath used for various build tasks.

There are 6 scopes available:

### compile

This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.

### provided

This is much like compile, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope provided because the web container provides those classes. This scope is only available on the compilation and test classpath, and is not transitive.

### runtime

This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.

## test

This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.

## system

This scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

## import (only available in Maven 2.0.9 or later)

This scope is only used on a dependency of type pom in the <dependencyManagement> section. It indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

# HawtIO

hawtio has lots of plugins such as: a git-based Dashboard and Wiki, logs, health, JMX, OSGi, Apache ActiveMQ, Apache Camel, Apache OpenEJB, Apache Tomcat, Jetty, JBoss and Fuse Fabric

We can dynamically extend hawtio with your own plugins or automatically discover plugins inside the JVM.

<http://localhost:8181/hawtio/index.html#/login>

# OSGI Class loading

```
bootstrap classloader (includes Java standard libraries from jre/lib/rt.jar etc)
^
extension classloader
^
system classloader (ie stuff on $CLASSPATH, including OSGi core code)
  (** limited access to types from parent classloader
common OSGi classloader
 \
  \-- OSGi classloader for bundle1    -> (map of imported-package->classloader)
    \-- OSGi classloader for bundle2    -> (map of imported-package->classloader)
      \-- OSGi classloader for bundle3    -> (map of imported-package->classloader)
          /
          /
=====
| shared bundle registry, holding info about all bundles and their exported-packages
\=====
```

Bootstrap classes: the runtime classes in rt.jar, internationalization classes in i18n.jar, and others.

Installed extensions: classes in JAR files in the lib/ext directory of the JRE, and in the system-wide, platform-specific extension directory (such as /usr/jdk/packages/lib/ext on the Solaris™ Operating System, but note that use of this directory applies only to Java™ 6 and later).

The class path: classes, including classes in JAR files, on paths specified by the system property java.class.path. If a JAR file on the class path has a manifest with the Class-Path attribute, JAR files specified by the Class-Path attribute will be searched also. By default, the java.class.path property's value is ., the current directory. You can change the value by using the -classpath or -cp command-line options, or setting the CLASSPATH environment variable. The command-line options override the setting of the CLASSPATH environment variable.

## Pax - OPS4J

OPS4J stands for Open Participation Software for Java.

OPS4J is a community that is trying to build a new, more open model for Open Source development, where not only the usage is Open and Free, but the Participation is Open as well.

# Pax Web

OSGi R4 Http Service and Web Applications implementation using Jetty 8.

extends OSGi Http Service with better servlet support, filters, listeners, error pages and JSPs and some others in order to meet the latest versions of Servlet specs.

Pax Web facilitates an easy installation of WAR bundles as well as discovery of web elements published as OSGi services. All of this beside the, standard, programmatic registration as detailed in the HTTP Service specs.

Pax Web 4.x supports the below technologies :

Servlet 3.0

JSP 2.0

JSF 2.1

Jetty 9.0.x

Tomcat 7.x (still experimental, but better supported)

support of CDI (through Pax-CDI)

support of only Servlet 3.0 annotated Servlets in JAR

Web-Fragments

## OSGi Services

The OSGi core framework defines the OSGi Service Layer, which provides a simple mechanism for bundles to interact by registering Java objects as services in the OSGi service registry.

One of the strengths of the OSGi service model is that any Java object can be offered as a service.

## OSGi Service Layer

In Red Hat JBoss Fuse, the natural way to communicate between deployed bundles is to use OSGi services. An OSGi service exposes Java methods that can be invoked by other bundles in the container.

# **Java Fluent API**

```
public class Customer{  
  
    private String name;  
    private Integer age;  
    private String email;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public Integer getAge() {  
        return age;  
    }  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String email) {  
        this.email = email;  
    }  
}
```

```
public class CustomerBuilder{  
    private Customer Customer;  
  
    public CustomerBuilder(){  
        Customer = new Customer();  
    }  
  
    public CustomerBuilder name(String name){  
        Customer.setName(name);  
        return this;  
    }  
  
    public CustomerBuilder age(Integer age){  
        Customer.setAge(age);  
        return this;  
    }  
  
    public CustomerBuilder email(String email){  
        Customer.setEmail(email);  
        return this;  
    }  
  
    public Customer build(){  
        return Customer;  
    }  
}
```

```
public class Main{
    public static void main(String[] args) {
        CustomerBuilder builder = new CustomerBuilder();
        builder
            .name("Gabriel")
            .age(21)
            .email("imnotgivingyoumyemail@gmail.com");
        Customer customer = builder.build();
        System.out.println(customer.getName());
        System.out.println(customer.getAge());
        System.out.println(customer.getEmail());
    }
}
```

## Controlling the mapping strategy selected

You can use the `jmsMessageType` option on the endpoint URL to force a specific message type for all messages.

In the route below, we poll files from a folder and send them as `javax.jms.TextMessage` as we have forced the JMS producer endpoint to use text messages:

```
from("file://inbox/order").to("jms:queue:order?jmsMessageType=Text");
```

We can also specify the message type to use for each message by setting the header with the key `CamelJmsMessageType`. For example:

```
from("file://inbox/order").setHeader("CamelJmsMessageType",  
JmsMessageType.Text).to("jms:queue:order");
```

The possible values are defined in the enum class,  
`org.apache.camel.jms.JmsMessageType`.