

Welcome

Apache Camel

Venkata Ramana



❑ Specification

❑ Framework

❑ Pattern

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 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

JSR 311,370 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)

A **framework** is a body of pre-written code that acts as a template or skeleton, which a developer can then use to create an application by filling in their own code as needed to get the app to work as they intend it to.

A framework is created to be used over and over so that developers can program their application without the manual overhead of creating every line of code from scratch.

Java frameworks are bodies of prewritten code used by developers to create apps using the Java programming language.

A Java framework is a type of framework specific to the Java programming language, used as a platform for developing software applications and Java programs.

Ex: Spring, Hibernate, Spring Boot, Apache Camel etc.,

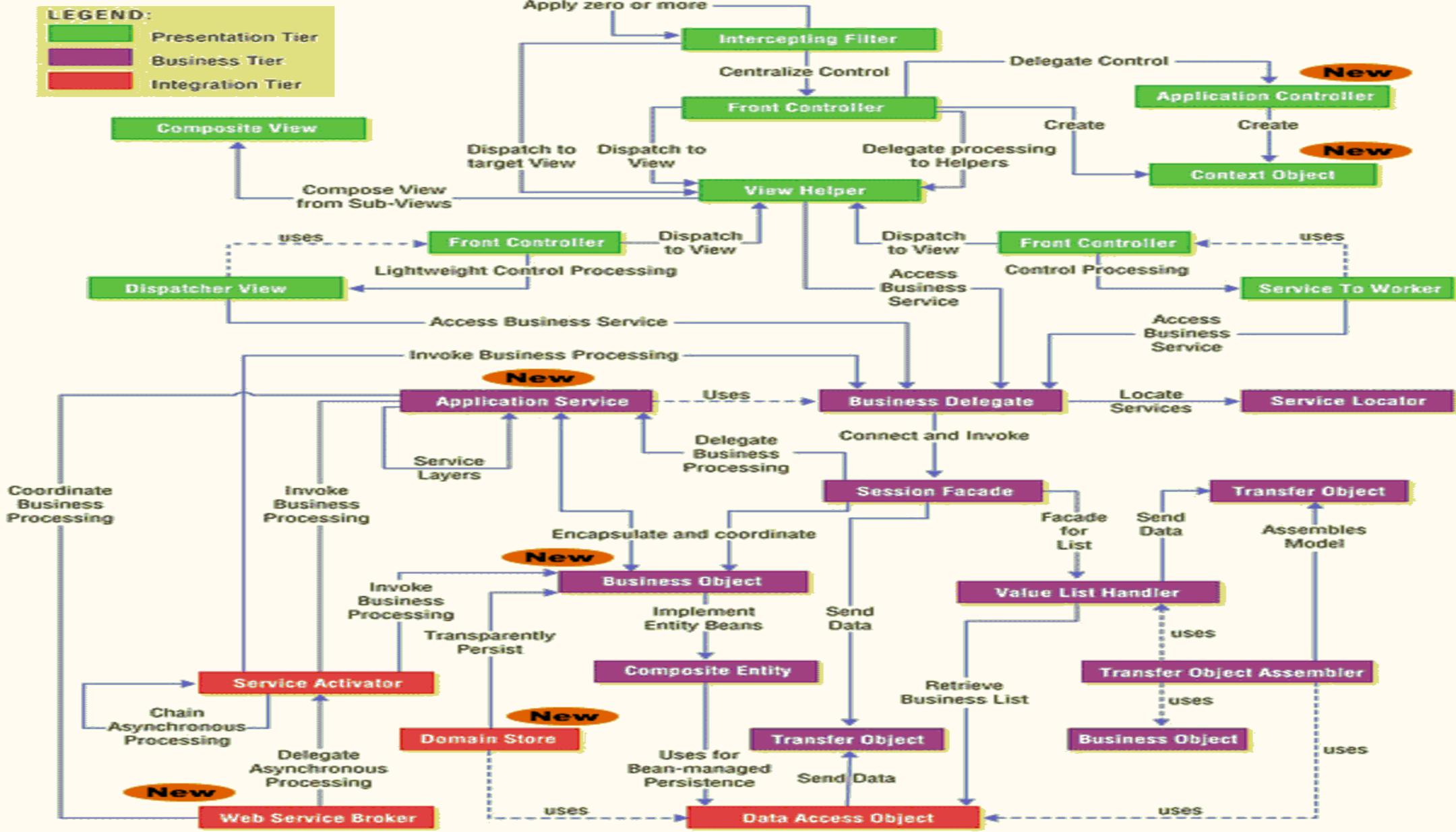
Design Pattern

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design

GOF Design Patterns

		Purpose		
		Creational	Structural	Behavioral
S C O P E	Class	Factory Method	Class Adapter	Interpreter Template Method
	O	Abstract Factory	Bridge	Chain of Responsibility
	B	Builder	Composite	Command
	J	Prototype	Decorator	Iterator
	E	Singleton	Façade	Mediator
	C		Flyweight	Memento
	T		Object Adapter	Observer
			Proxy	State
				Strategy
				Visitor

Core J2EE Patterns, 2nd Edition



Architectural Patterns:

MVC, SOA, Restful, MOM, Microservice

Enterprise Integration Pattern

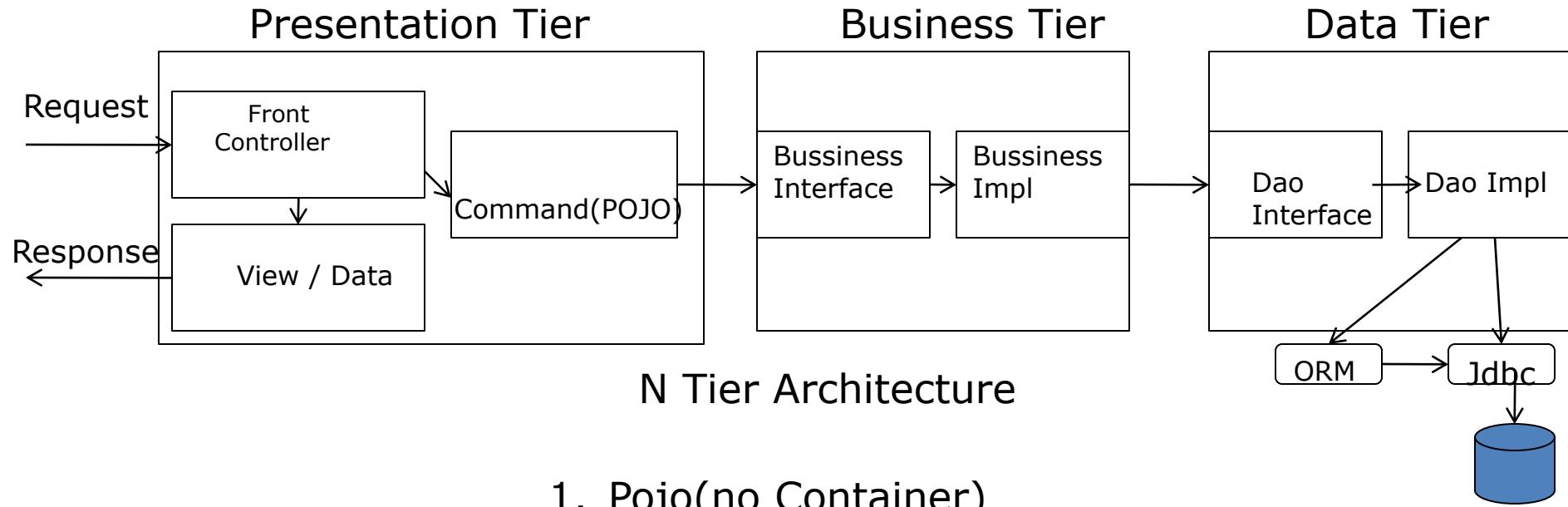
Splitter, Aggregator, Content-based Router

RESTful API Patterns

- Statelessness
- Content Negotiation
- URI Templates
- Pagination
- Versioning
- Authorization
- API facade
- Discoverability
- Idempotent
- Circuit breaker

Microservice Patterns:

- API gateway
- Service registry
- Circuit breaker
- Messaging
- Database per Service
- Access Token
- Saga
- Event Sourcing & CQRS



- | | | |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> 1. Servlet/jsp 2. MVC Struts JSF Flex Gwt Spring MVC ... | <ol style="list-style-type: none"> 1. Pojo(no Container) 2. Ejb 2.x(HW Container) <ul style="list-style-type: none"> -Session Bean -Mdb 3. Pojo + LW Container <ul style="list-style-type: none"> - Spring - Microcontainer - Xwork 4. Ejb3.0 | <ol style="list-style-type: none"> 1. Jdbc(pojo) 2. Ejb 2.x – Entity Bean 3. Jdo 4. ORM <ul style="list-style-type: none"> - Hibernate - Kodo - Toplink - MyBatis 5. JPA <p>+ Spring Templates</p> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Spring Framework Features:

1. Core Container - IOC, DI & AOP
Application Context (Object creation – Singleton/Prototype)
Dependency Injection (Object Graph Creation)
2. Data Access Layer (Template pattern)
3. Spring MVC & REST
4. Other features like Security, Messaging, Spring Integration,
Spring Batch, Spring Cloud etc.,

IOC is used to decouple common task from implementation.

Six basic techniques to implement Inversion of Control.

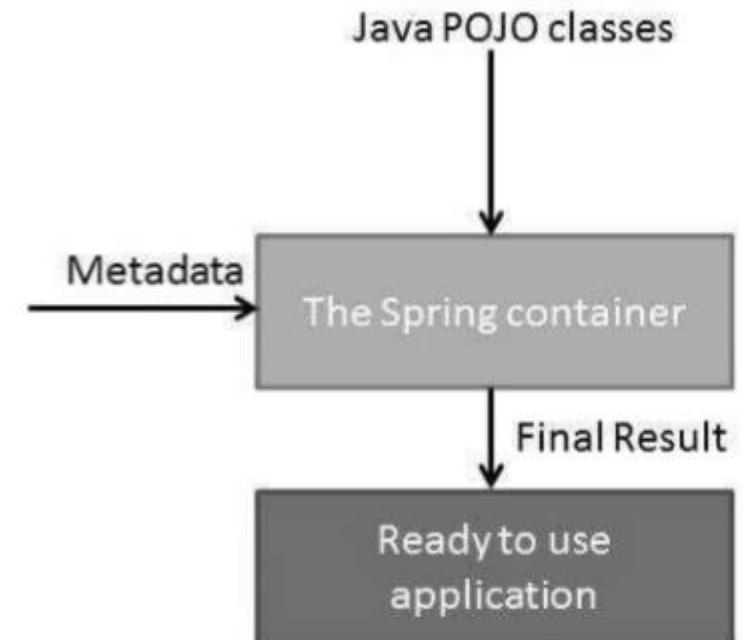
These are:

- 1.using a factory pattern
- 2.using a service locator pattern
- 3.using a constructor injection
- 4.using a setter injection
- 5.using an interface injection
- 6.using a contextualized lookup

Constructor, setter, and interface injection are all aspects of Dependency injection.

What is a container?

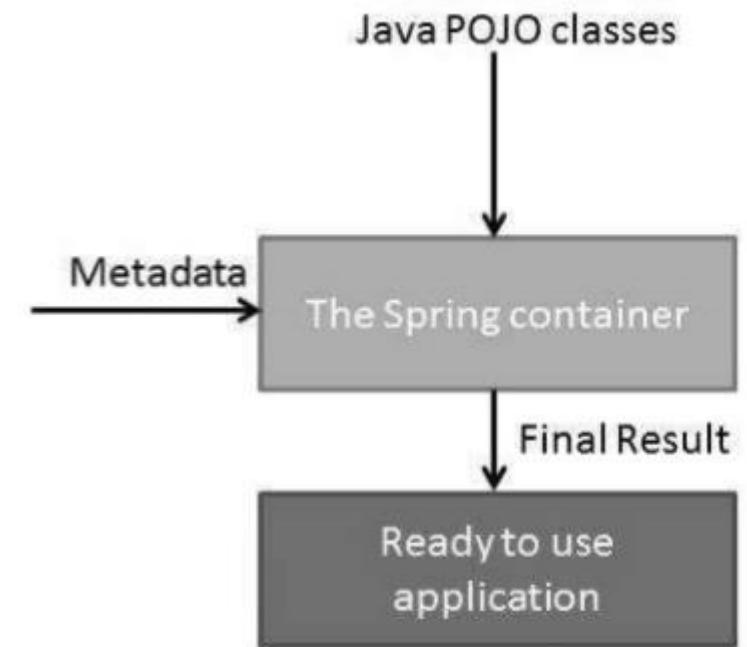
- ✓ The container will create the objects,
- ✓ wire them together,
- ✓ configure them,
- ✓ and manage their complete life cycle from creation till destruction.



The container gets its instructions on

- ✓ what objects to instantiate,
- ✓ configure,
- ✓ and assemble by reading the configuration metadata provided.
- ✓ The configuration metadata can be represented either by XML or Annotation.

- Apache Tomcat is a Servlet Container.
- Weblogic/Websphere/JBoss provides EJB Container
- Spring is a POJO container.



Spring XML Configuration

```
package demo.web;

Class AccountDaoJdbc implements AccountDao
{

DataSource dataSource;

Public voic setDatasource(DataSource datasource)
{
dataSource = datasource;
}
..}
                                <bean id = "accountDao" class="demo.web.AccountDaoJdbc">
                                <property name="datasource">
                                <ref bean="dataSource"/>
                                </property>
                                </bean>
```

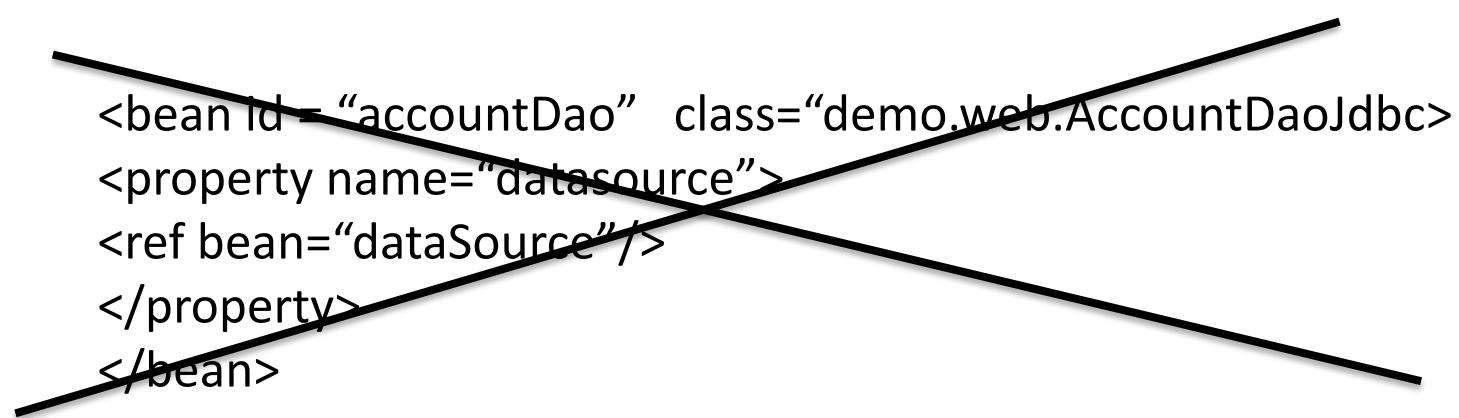
Spring Annotation Configuration

```
package demo.web;

@Repository("accountDao")
Class AccountDaoJdbc implements AccountDao
{

    @Autowired
    DataSource dataSource;

    Public void setDatasource(DataSource datasource)
    {
        dataSource = datasource;
    }
    ..
}
```



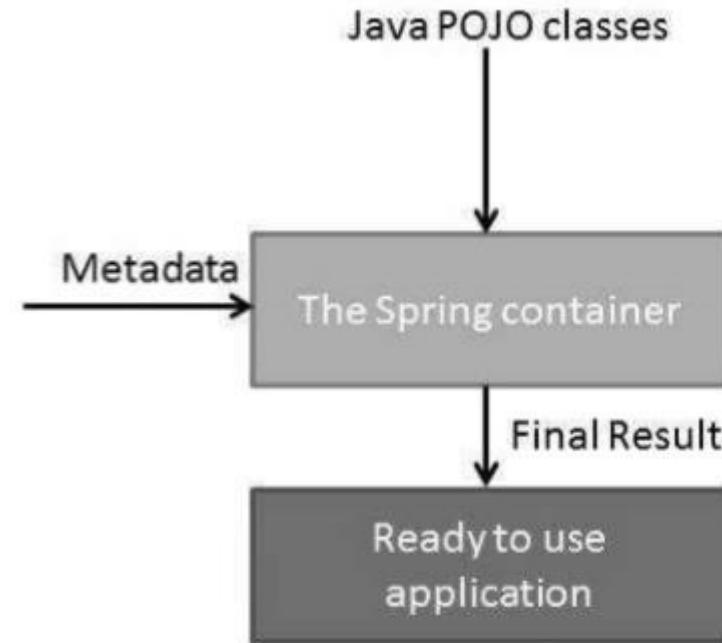
```
<bean id="accountDao" class="demo.web.AccountDaoJdbc">
    <property name="datasource">
        <ref bean="dataSource"/>
    </property>
</bean>
```

Lifecycle Management

Bean Managed Life cycle

```
AccountBean acc = new AccountBean();
acc.setAmt(5000);
....
acc=null;
```

Container Managed Life cycle

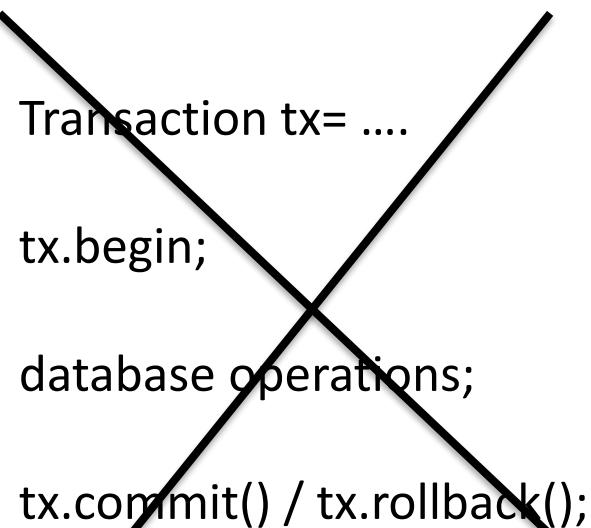


```
ApplicationContext ctx = ....;
AccountBean acc = ctx.getBean("account");
```

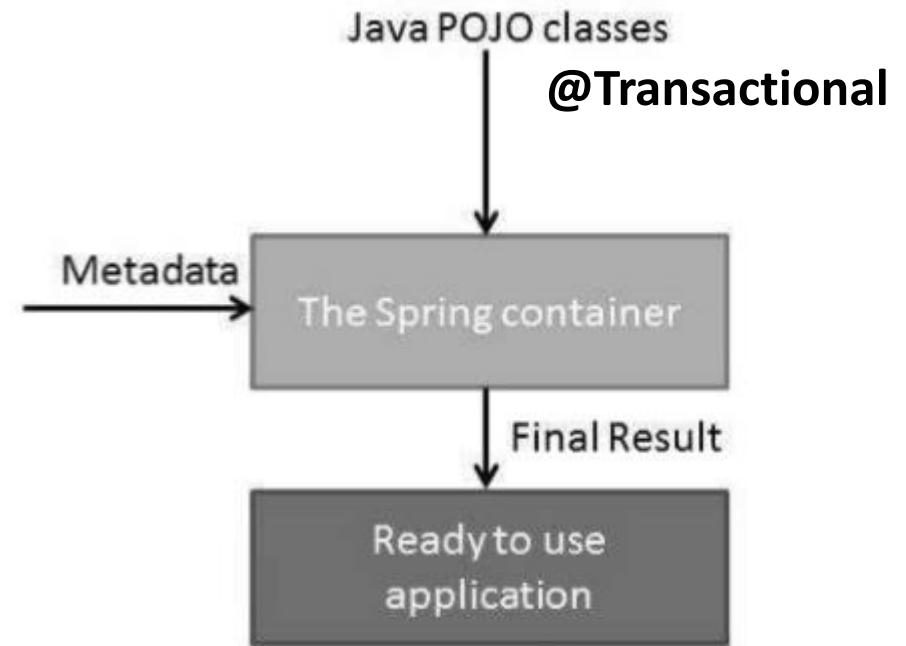
Container Managed Services

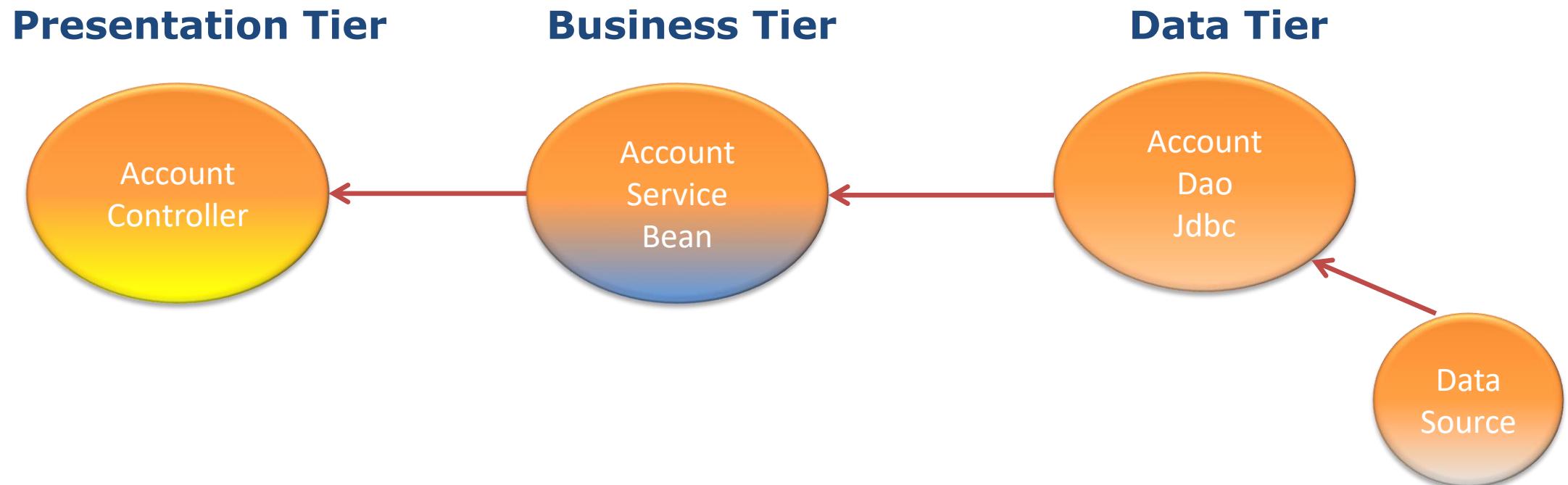
Bean managed transactions

Transaction tx=
tx.begin;
database operations;
tx.commit() / tx.rollback();



Container managed transactions





Presentation Tier



```
@RestController  
Class AccountController  
{  
    @Autowired  
    AccountService accService;  
    ....  
}
```

Business Tier

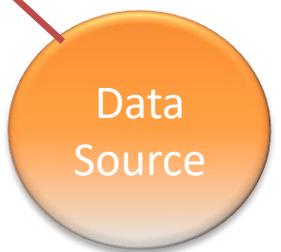


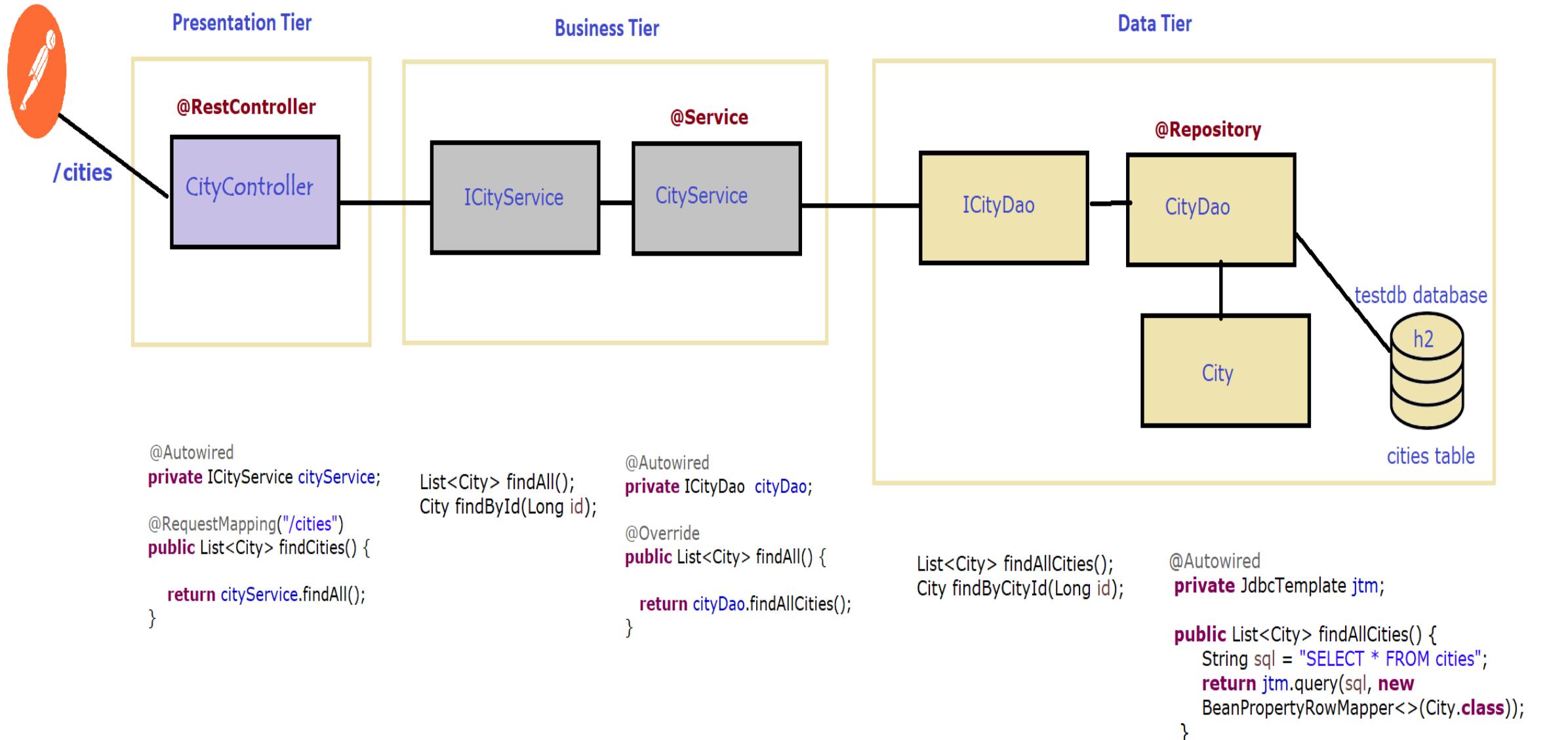
```
@Service  
Class AccountServiceBean  
    implements AccountService  
{  
    @Autowired  
    AccountDao accountDao;  
    ....  
}
```

Data Tier



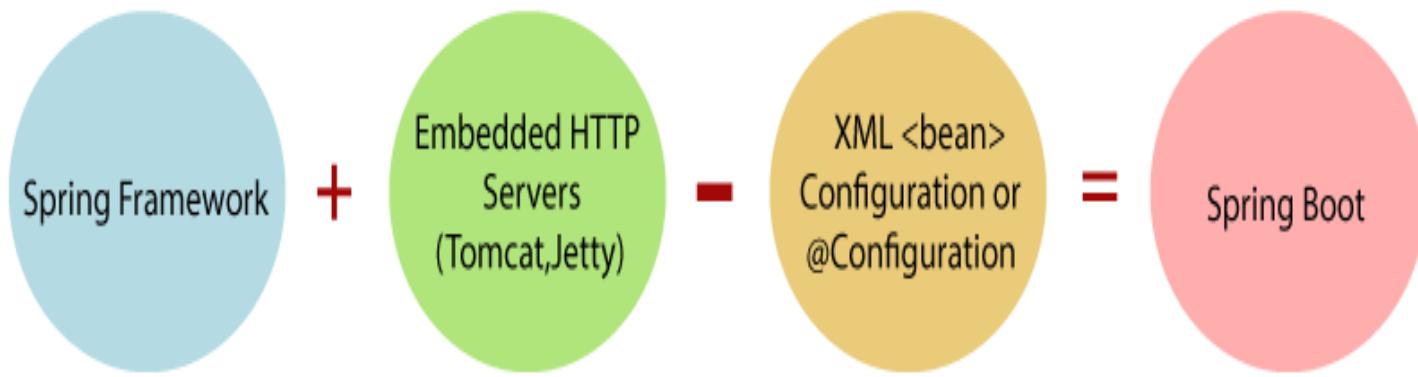
```
@Repository  
Class AccountDaoJdbc  
    implements AccountDao  
{  
    @Autowired  
    DataSource dataSource;  
    ....  
}
```





Spring Boot is a project that is built on the top of the **Spring** Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that we can “just run”.

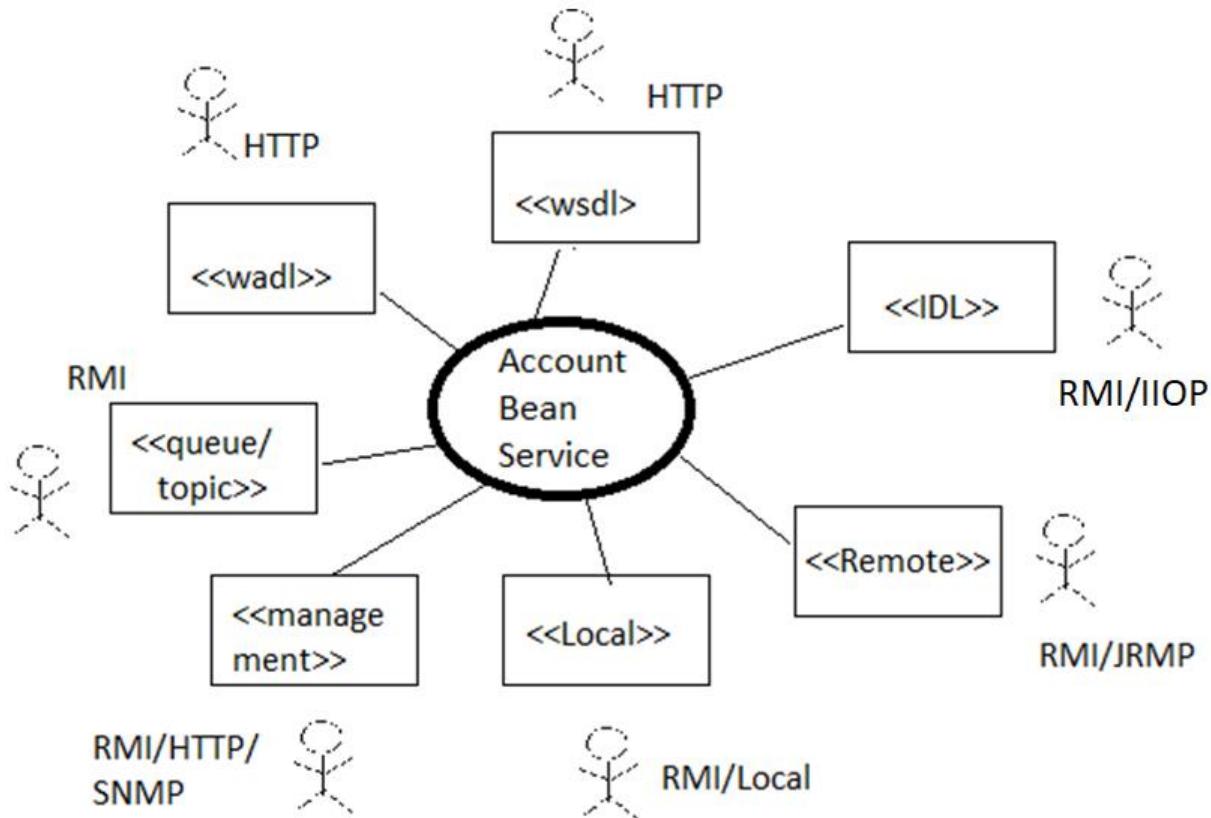


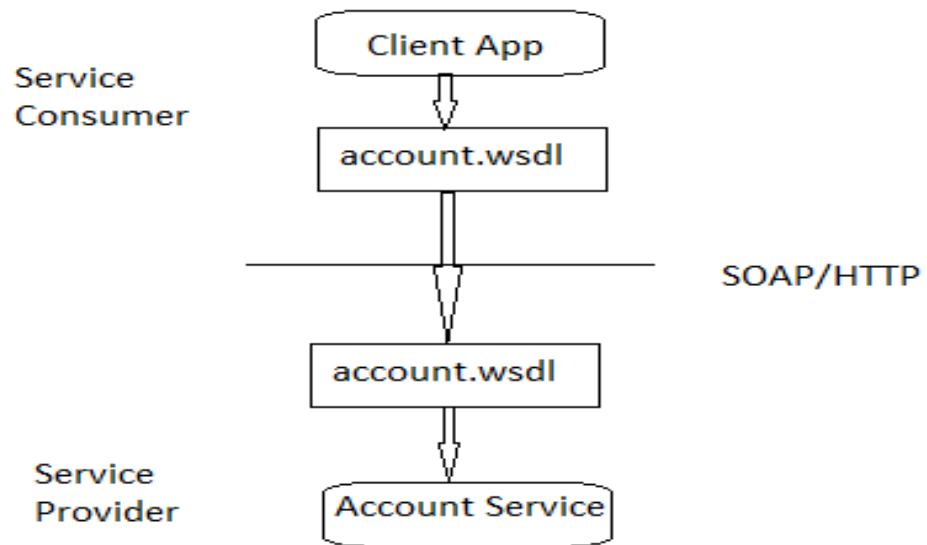
Spring Boot is an

- Opinionated,
- Convention over configuration,
- Get started with minimum effort,
- Create stand-alone,
- Production-grade applications.

```
localhost:8080/actuator/      ⓘ localhost:8080/actuator/
{
    "loggers": {
        "href": "http://localhost:8080/actuator/loggers",
        "templated": false
    },
    "loggers-name": {
        "href": "http://localhost:8080/actuator/loggers/{name}",
        "templated": true
    },
    "heapdump": {
        "href": "http://localhost:8080/actuator/heapdump",
        "templated": false
    },
    "threaddump": {
        "href": "http://localhost:8080/actuator/threaddump",
        "templated": false
    },
    "metrics-requiredMetricName": {
        "href": "http://localhost:8080/actuator/metrics/{requiredMetricName}",
        "templated": true
    },
    "metrics": {
        "href": "http://localhost:8080/actuator/metrics",
        "templated": false
    },
    "scheduledtasks": {
        "href": "http://localhost:8080/actuator/scheduledtasks",
        "templated": false
    },
    "httptrace": {
        "href": "http://localhost:8080/actuator/httptrace",
        "templated": false
    },
    "mappings": {
        "href": "http://localhost:8080/actuator/mappings",
        "templated": false
    }
}
```

Distributed computing and web services

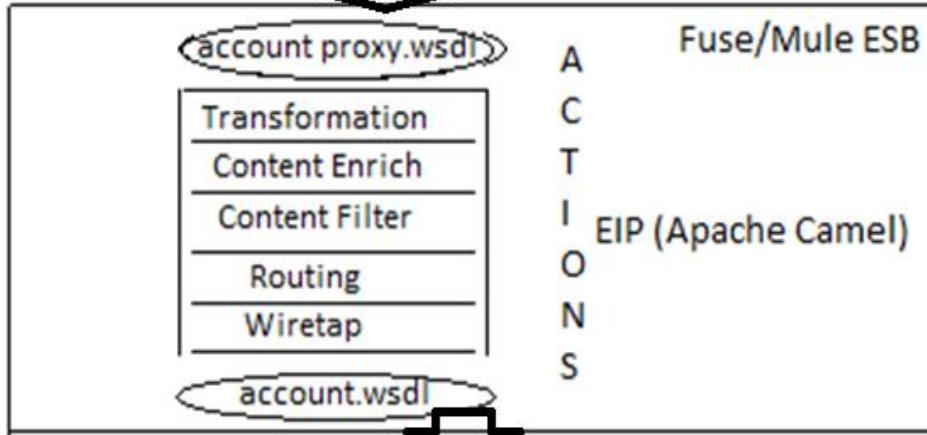
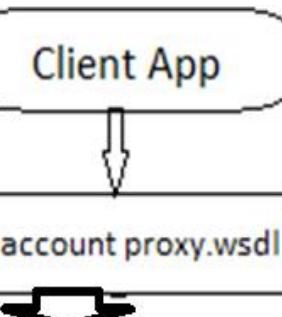




Webserivce OFF the BUS

W
E
B
S
E
R
V
I
C
E

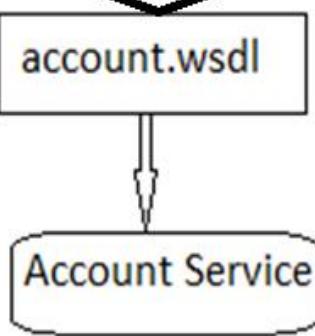
Service
Consumer



A
C
T
I
O
N
S

EIP (Apache Camel)

Service
Provider



O
N
T
H
E
B
U
S

SOA

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

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

The fluent builder pattern is one of the most useful patterns, especially when you want to build complex objects. For example, say we want to build a complex object by initiating the builder, calling the respective setters, and finally, calling the build method. Once the build method is executed, we will get the desired model/entity/pojo object back.

```
Email email = Email.EmailBuilder()  
    .setFrom("Test@gmail.com")  
    .setTo("mail@gmail.com")  
    .setSubject("Test with only required Fields")  
    .setContent(" Required Field Test")  
    .build();
```

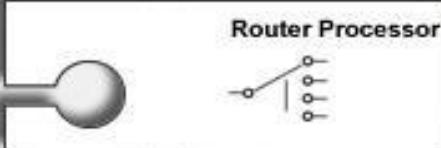
Camel

Integration Engine And Router

Camel Endpoints

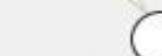
- Camel can send messages to them
- Or Receive Messages from them

Filter Processor



Camel Processors

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



JMS Component

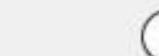
JMS API



HTTP Component

Servlet API

Web Container
Jetty | Tomcat | ...



File Component

File System

Camel Components

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



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



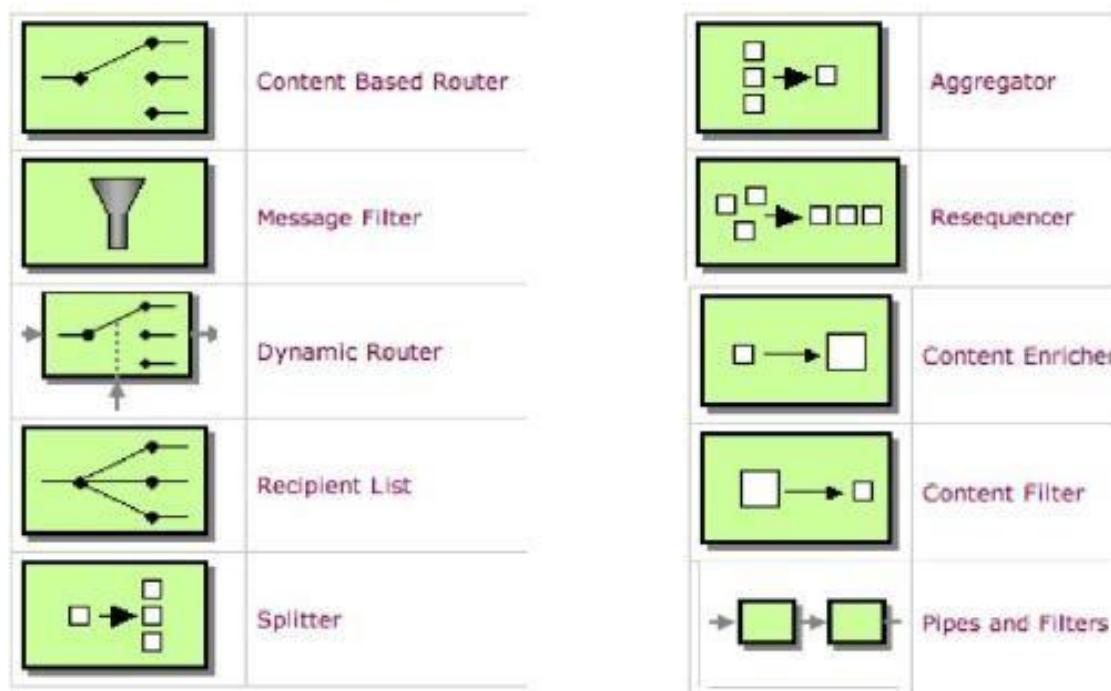
HTTP Client



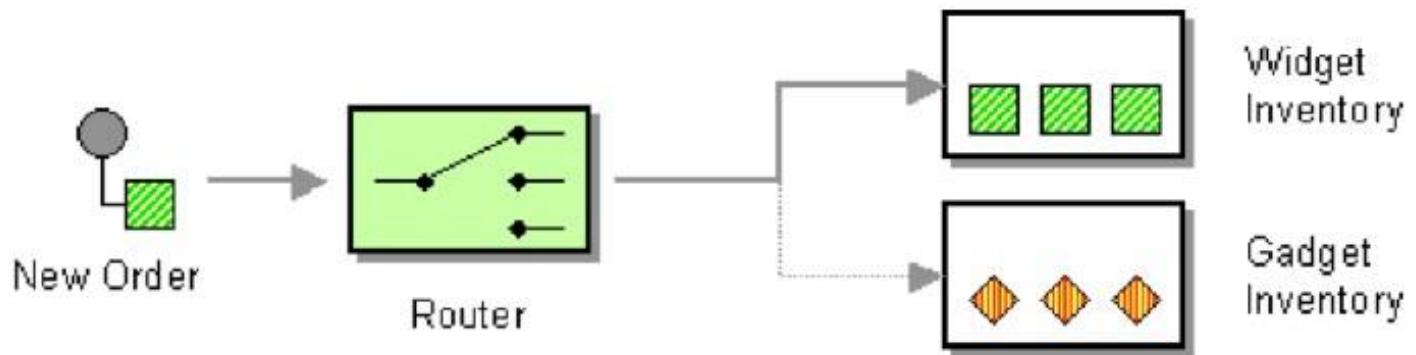
Local File System

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>
```

YAML DSL

Since Camel 3.9

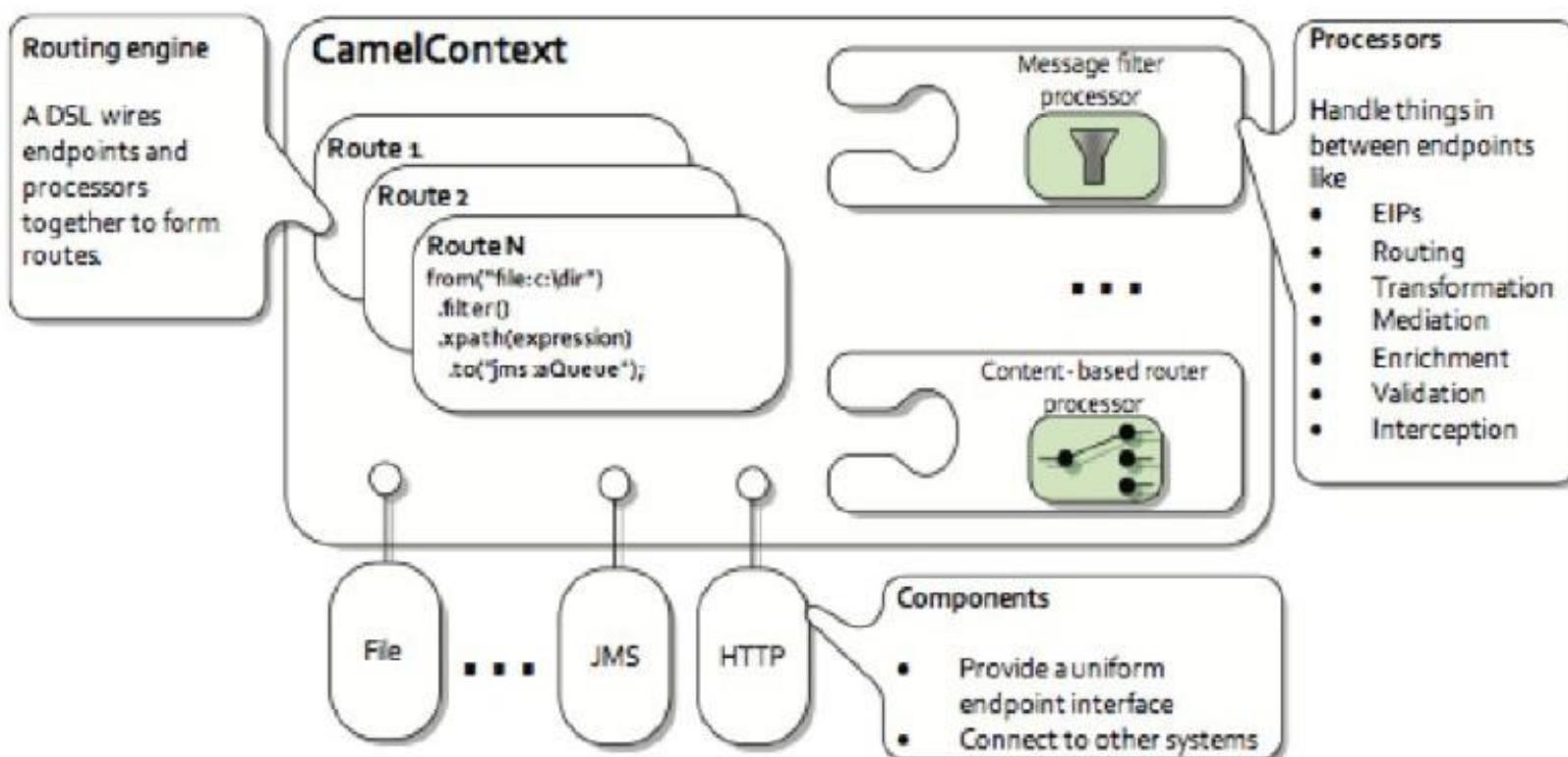
A route is collection of elements defined as follows:

```
- from: ①
  uri: "direct:start"
  steps: ②
    - filter:
        expression:
          simple: "${in.header.continue} == true"
        steps: ②
          - to:
              uri: "log:filtered"
    - to:
        uri: "log:original"
```

- ① route entry point, by default `from` and `rest` are supported
- ② processing steps

What is Apache Camel?

- Camel's Architecture



What is Apache Camel?

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

Camel Core Concepts:

Message contains data which is being transferred to a route. Each message has a unique identifier and it's constructed out of a body, headers, and attachments.

Exchange is the container of a message and it is created when a message is received by a consumer during the routing process. Exchange allows different types of interaction between systems – it can define a one-way message or a request-response message.

Endpoint is a channel through which system can receive or send a message. It can refer to a web service URI, queue URI, file, email address, etc.,

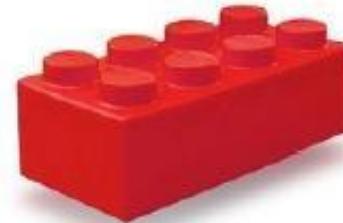
Component acts as an endpoint factory. To put it simply, components offer an interface to different technologies using the same approach and syntax. Camel already supports a lot of components in its DSLs for almost every possible technology, but it also gives the ability for writing custom components.

Processor is a simple Java interface which is used to add custom integration logic to a route. It contains a single process method used to perform custom business logic on a message received by a consumer.

At a high level, the architecture of Camel is simple. CamelContext represents the Camel runtime system and it wires different concepts such as routes, components or endpoints.

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)

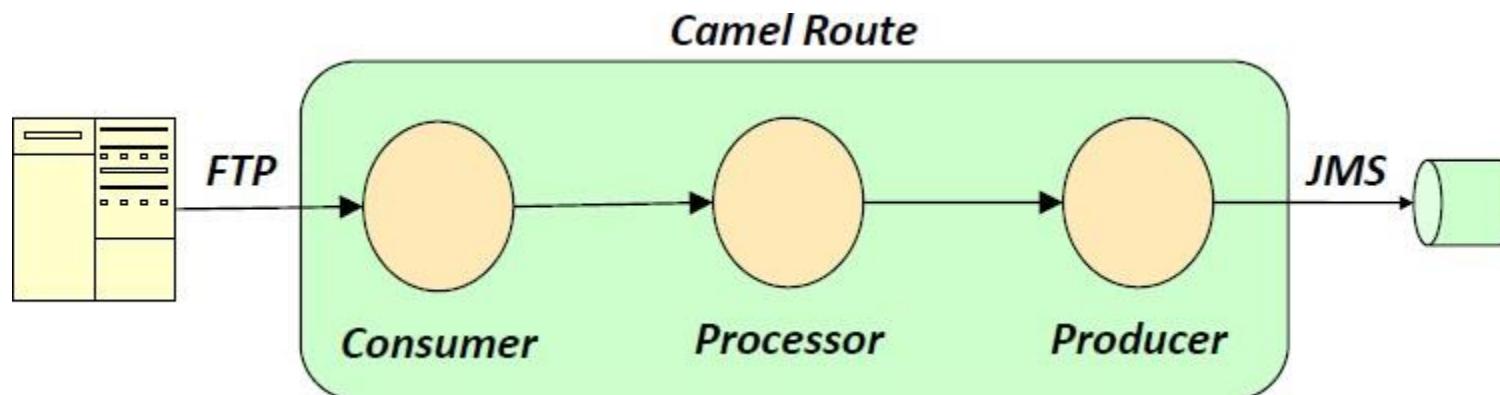
A Camel Route (2 of 2)

- > 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

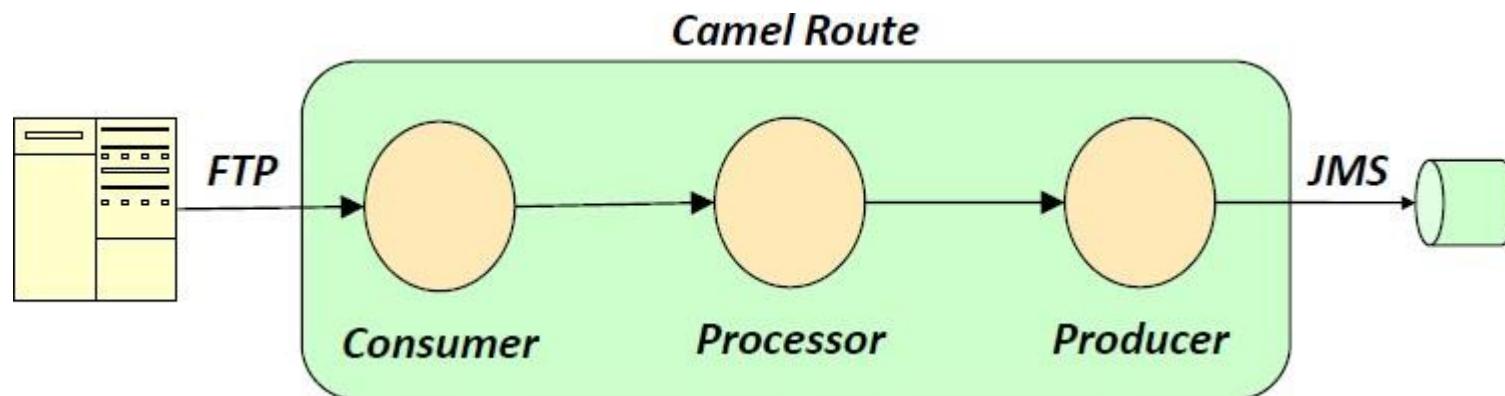
- Examples: an FTP server, a Web Service, or a JMS broker

Camel defines endpoints using URI syntax

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

ftp://surya@localhost?password=surya123

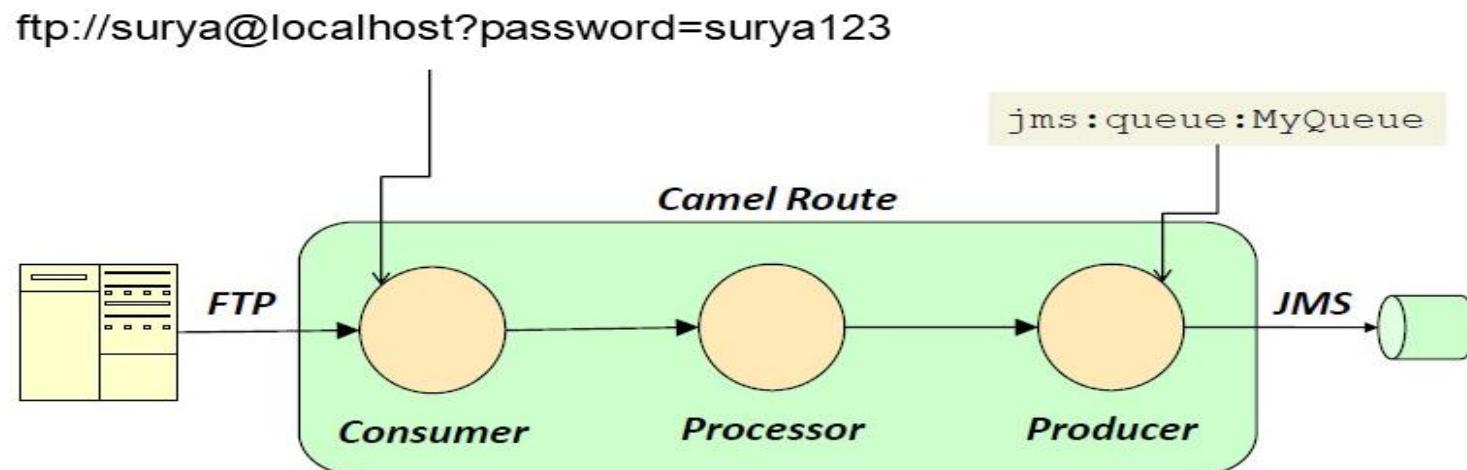
jms:queue:MyQueue



Consumers and Producers

Created from endpoints

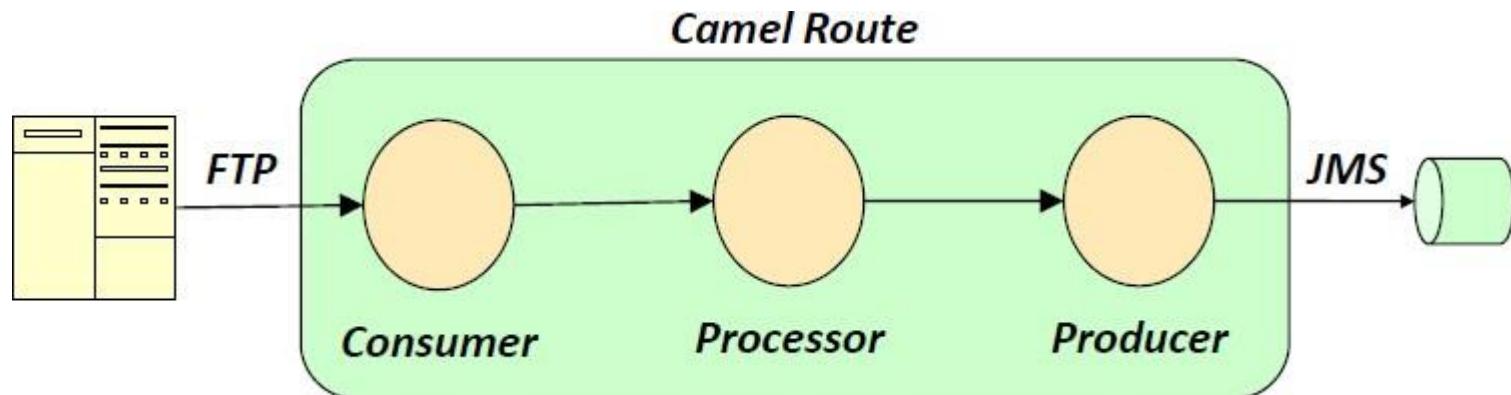
- Some create producers and consumers
- Some support the creation of either a producer or a consumer



Camel DSL

Camel provides a DSL to allow you to easily define routes

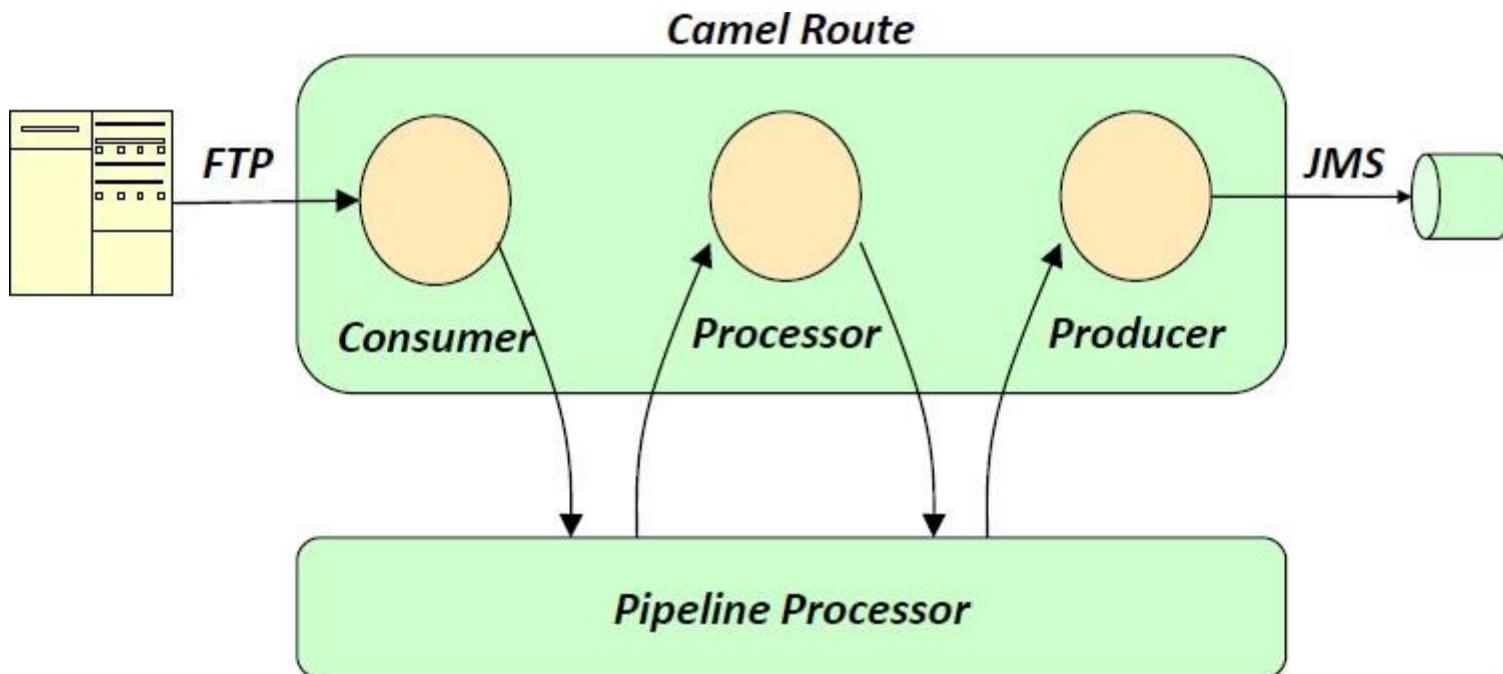
```
from("ftp://surya@localhost?password=surya123")
    .process("xslt:MyTransform.xslt")
    .to("jms:queue:MyQueue")
```



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

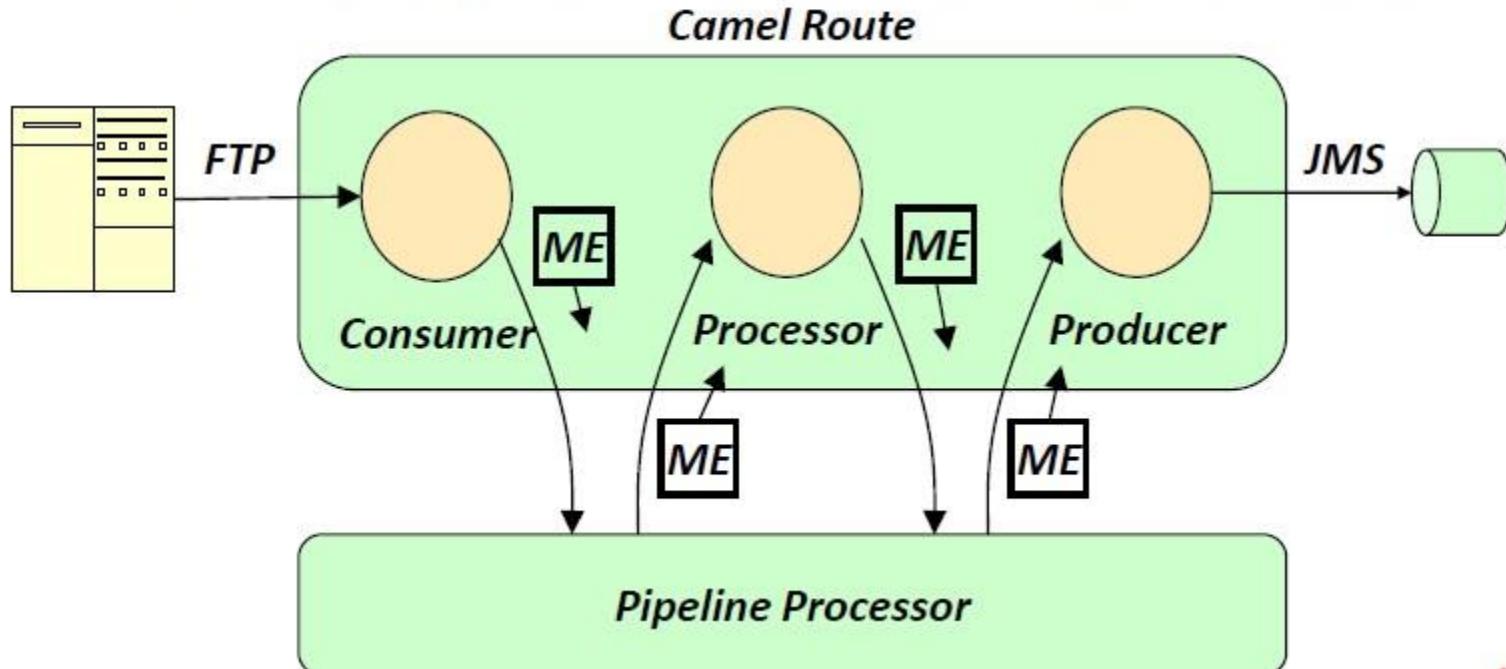


Sample Processors

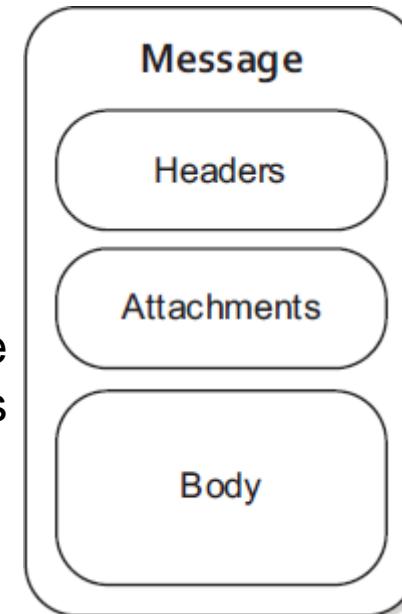
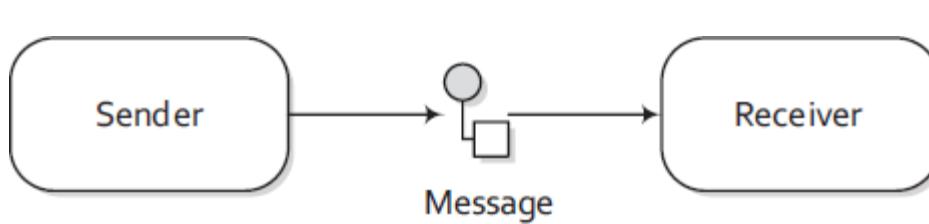
- **Choice** `choice()`
 - Used to route incoming messages to alternative producer endpoints
 - Each alternative producer endpoint is preceded by a `when()` method, which takes a predicate argument
- **Filter** `filter()`
 - Used to prevent uninteresting messages from reaching the producer endpoint
 - 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.

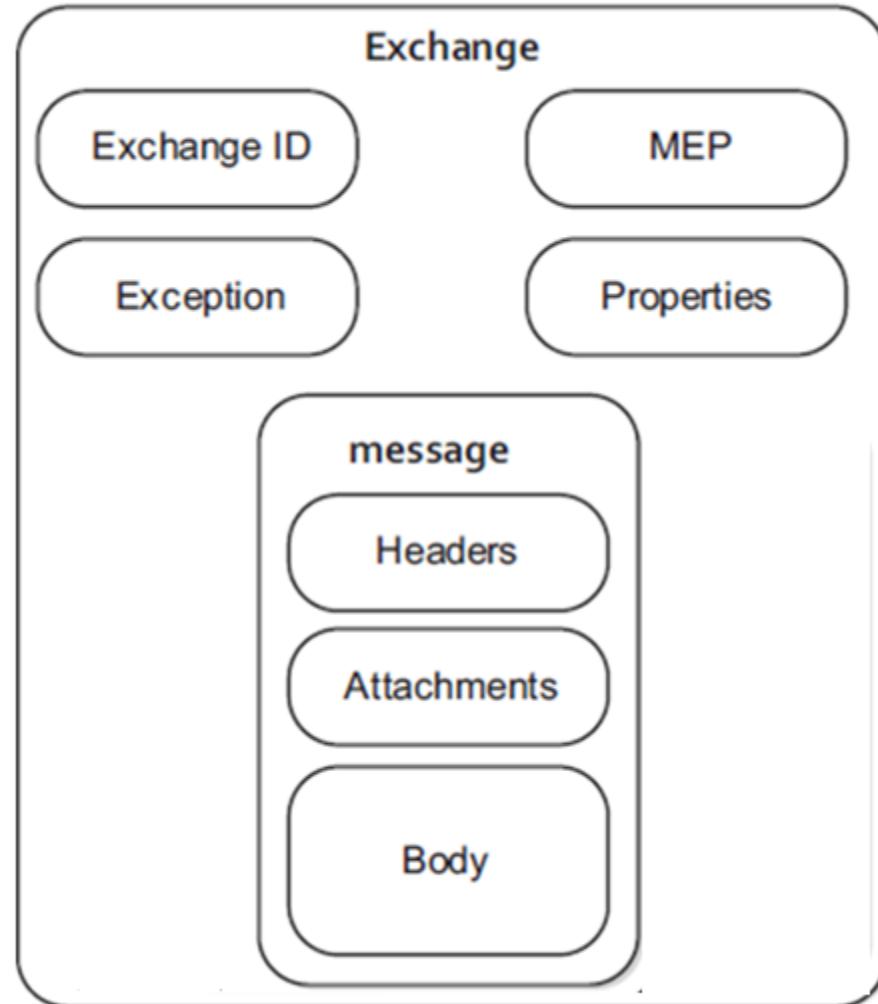
An exchange also provides support for the various types of interactions between systems, also known as message exchange patterns (MEPs).

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.



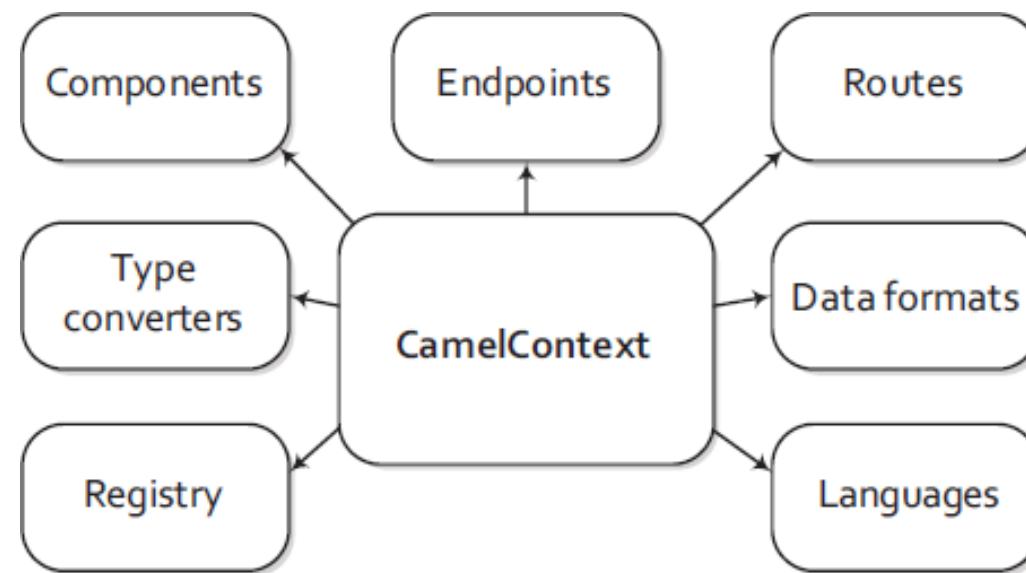
Deprecated: `setOut(Message out)`
use `setMessage(Message)`

- 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.

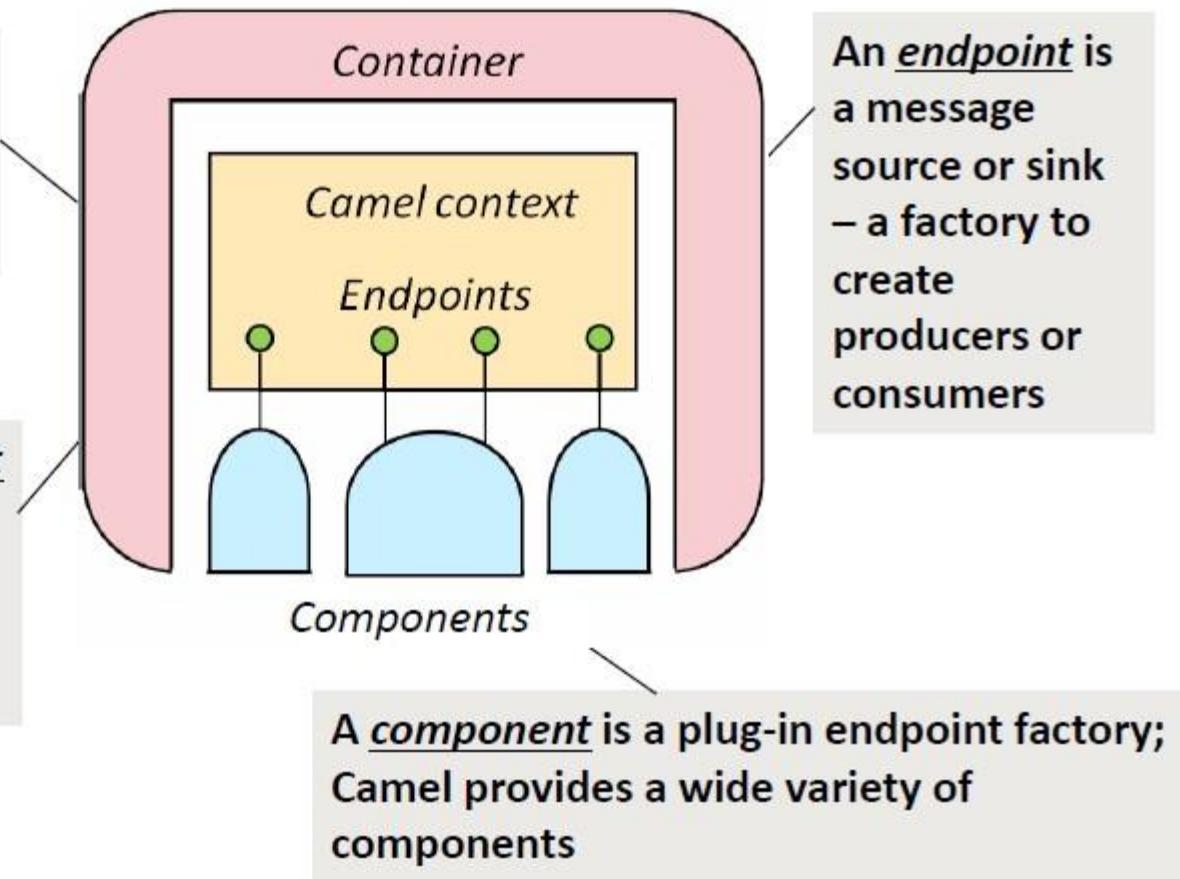


Service	Description
Components	Contains the components used. Camel is capable of loading components on the fly either by autodiscovery on the classpath or when a new bundle is activated in an OSGi container.
Endpoints	Contains the endpoints that have been created.
Routes	Contains the routes that have 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

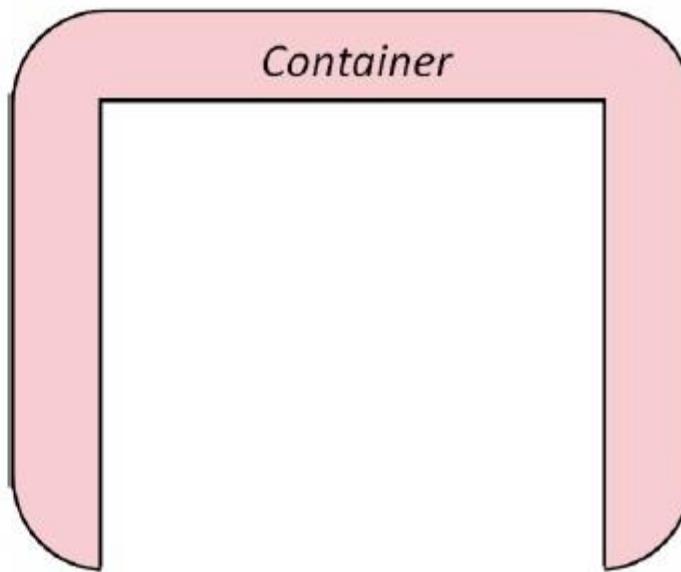
Camel can be deployed in a wide variety of container types

The Camel context is an instance of the Camel runtime environment



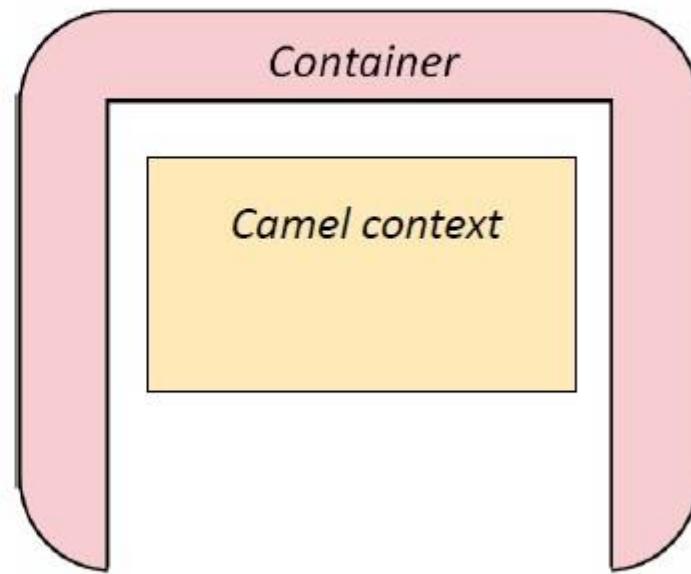
Container

- **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

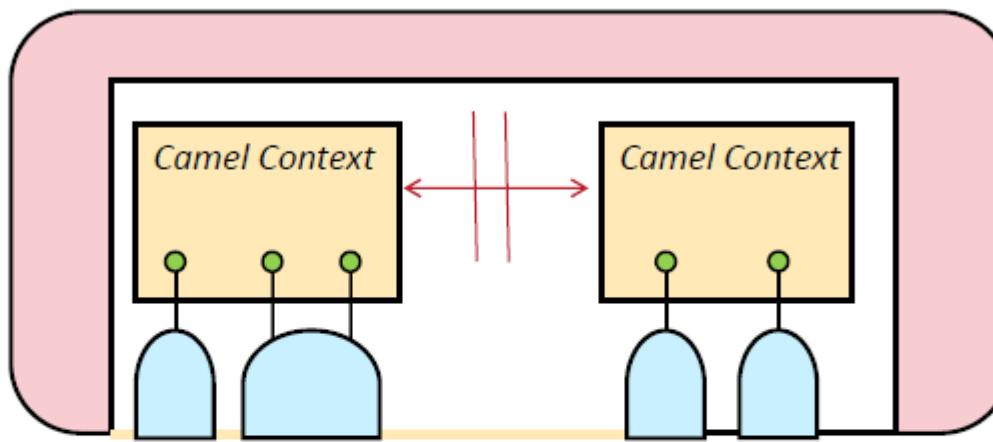


Camel Context (1 of 2)

- **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**



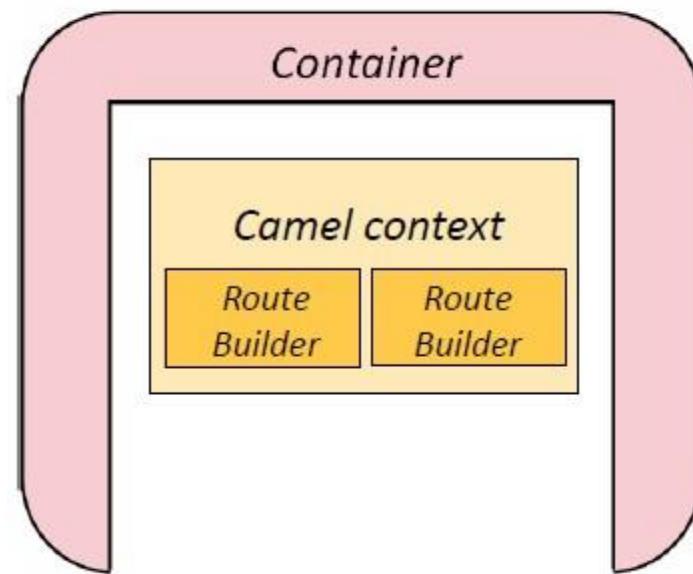
Camel Context (2 of 2)



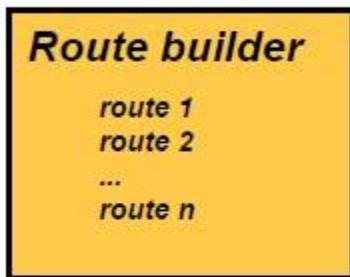
- Camel Contexts are isolated from each other
- Endpoints cannot communicate between CamelContexts
 - A transport layer will be required in this case (JMS, NMR, VM)
- CamelContext has a lifecycle. It can be started or stopped.

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



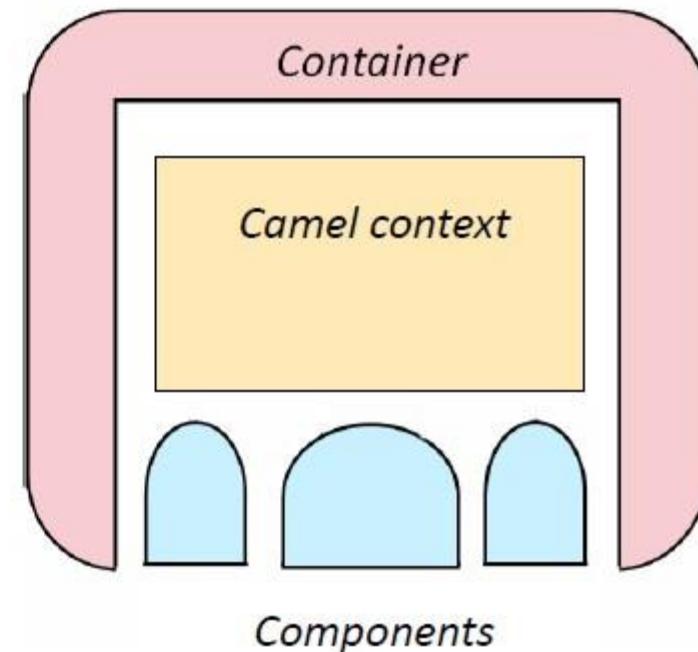
Route Builders (2 of 2)



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

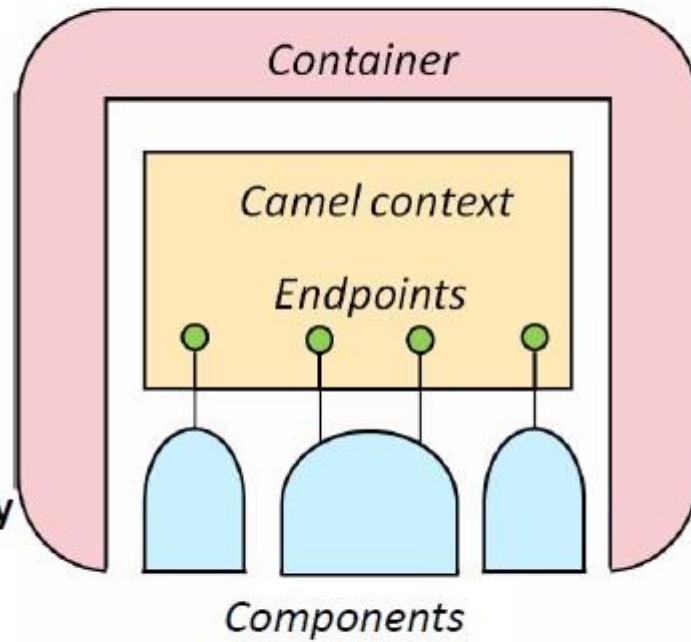
Components

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

Add the Camel schema location

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
        disposal
    }
}
```

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

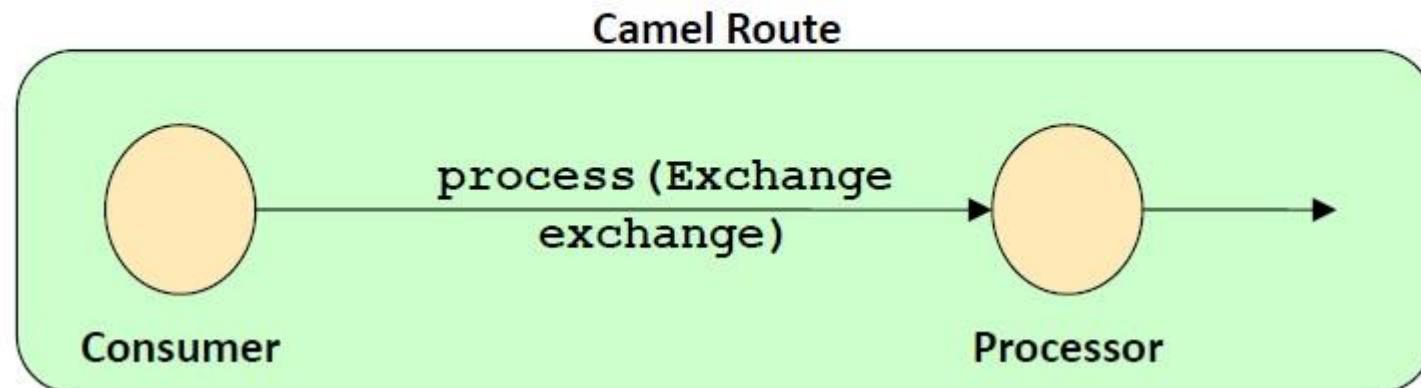
- **Spring alternative**
- **Create an endpoint with the** CamelContext
 - Can (or not) use propertiesPlaceHolder
 - Pass the reference to the route class

```
<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="demoImpl" class="com.example.webservice.DemoImpl"/>  
  
<camelContext xmlns="http://camel.apache.org/schema/spring">  
  <route id="DemoRoute">  
    <from uri="cxf:bean:webConsumer"/>  
    <to uri="log:com.example?showAll=true&multiline=true"/>  
    <bean ref="demoImpl" method="doWork"/>  
  </route>  
</camelContext>  
  
</beans>
```

Instantiate the JAX-WS implementation class as a Spring bean

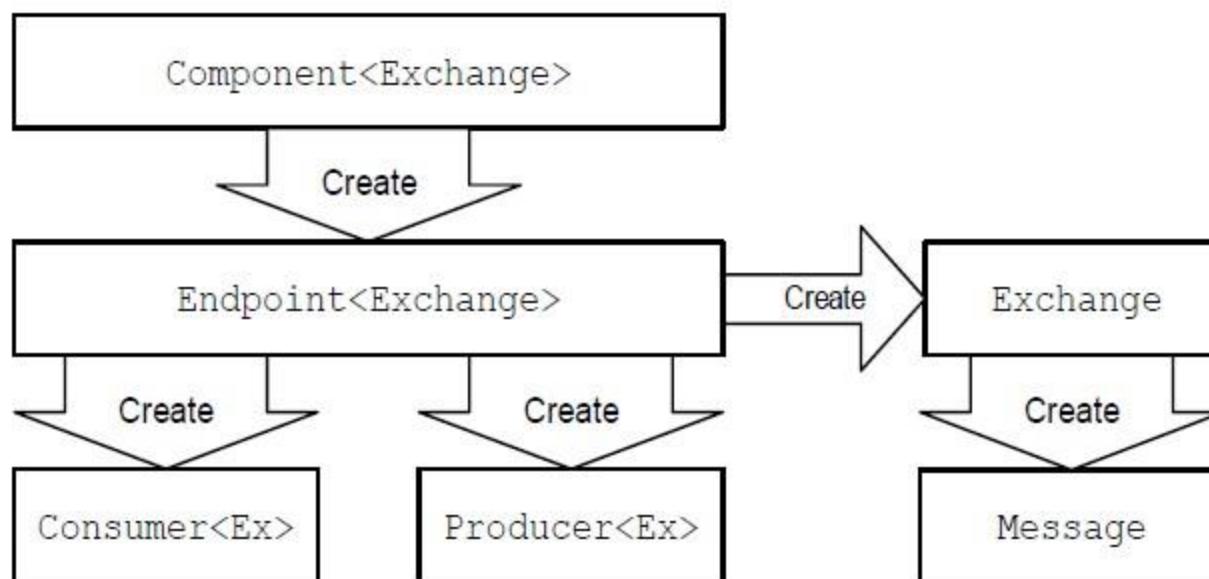
Define a Camel route to accept web service requests, log their contents, invoke the JAX-WS bean, and return a response to the original client

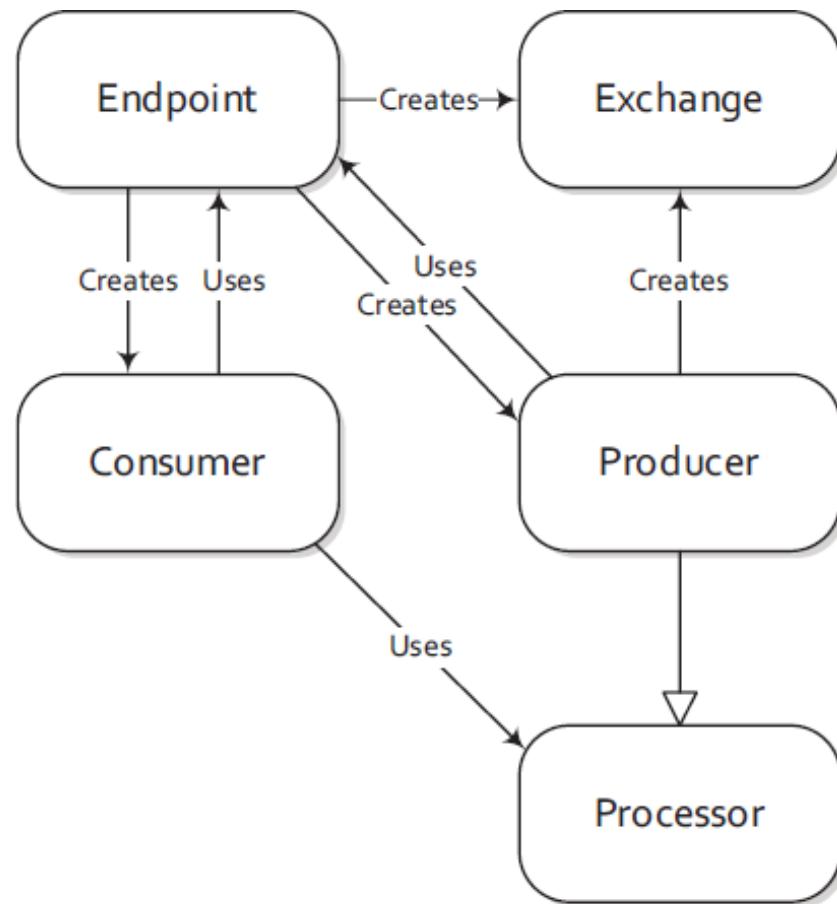
Done!

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 URLs:

```
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

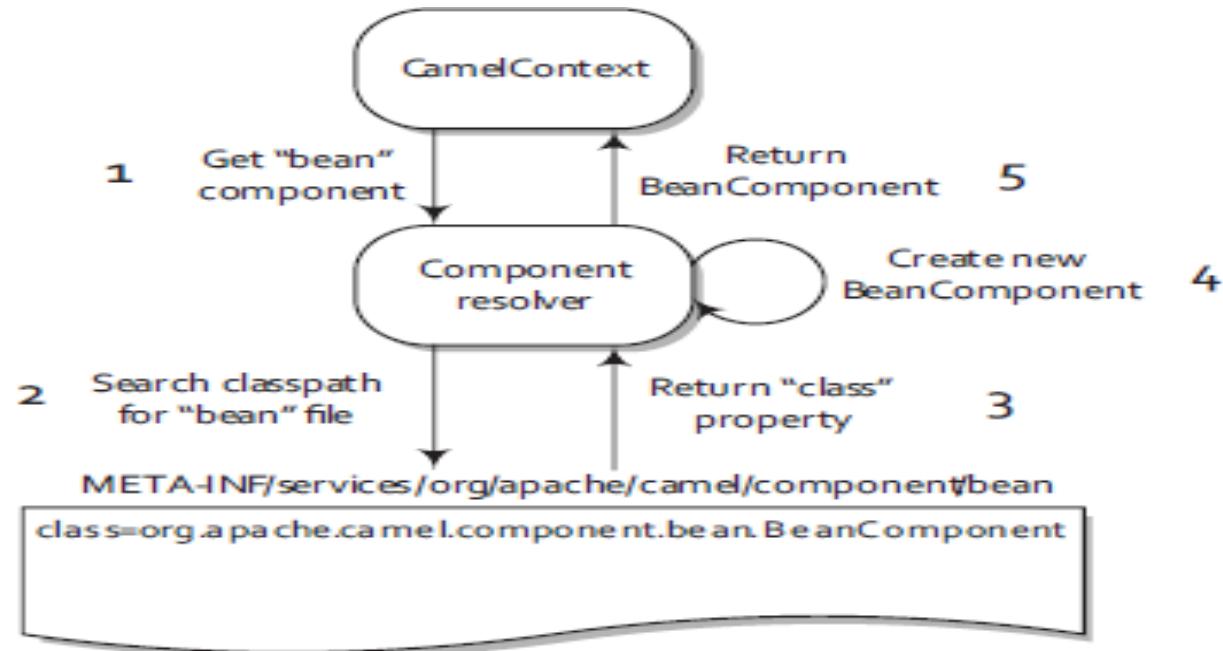
```
public class MyFileProducer<E extends Exchange>
    extends DefaultProducer<E> {
    public void process(Exchange exchange) throws
Exception {
    File target = createFileName();
    InputStream in =
        ExchangeHelper.getMandatoryInBody(
            exchange, InputStream.class);
    writeToFile(in,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

	Within Context	Within JVM
Synchronous	Direct	Direct-VM
Asynchronous	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:

- > 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 JavaEE 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 :

camellId
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

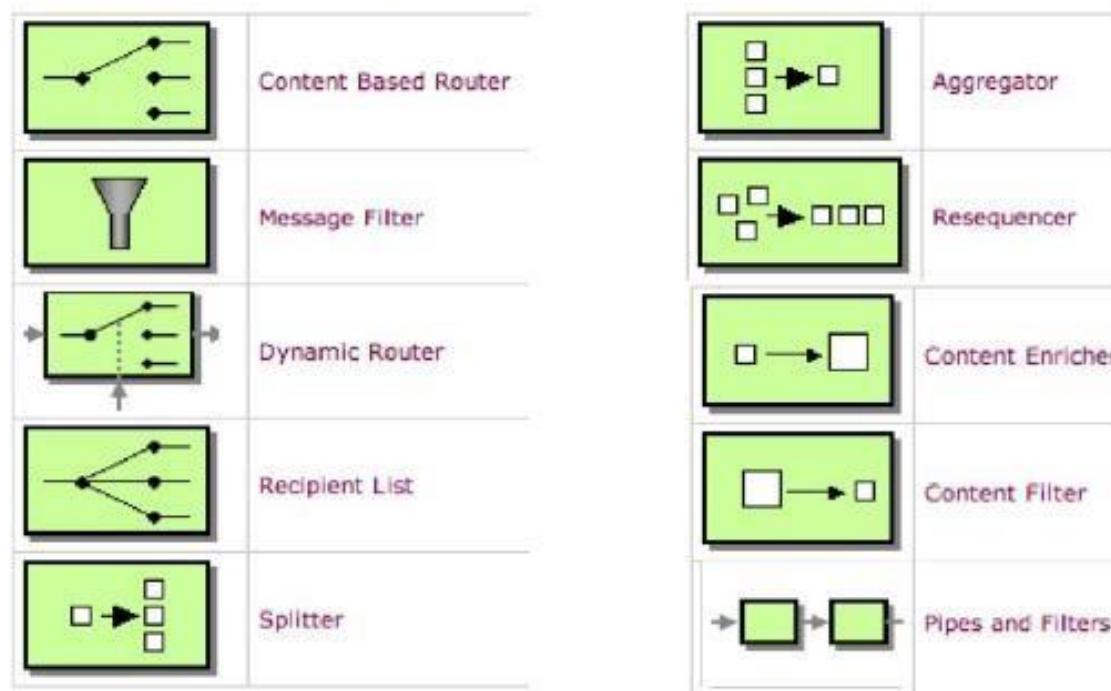
file:length to access the file length as a Long type

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

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

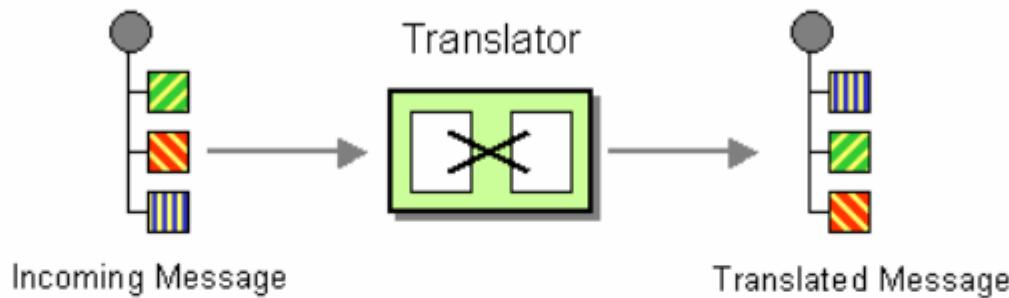
What is Apache Camel?

- Enterprise Integration Patterns



MARSHAL EIP

The [Marshal](#) and [Unmarshal](#) EIPs are used for [Message Transformation](#).

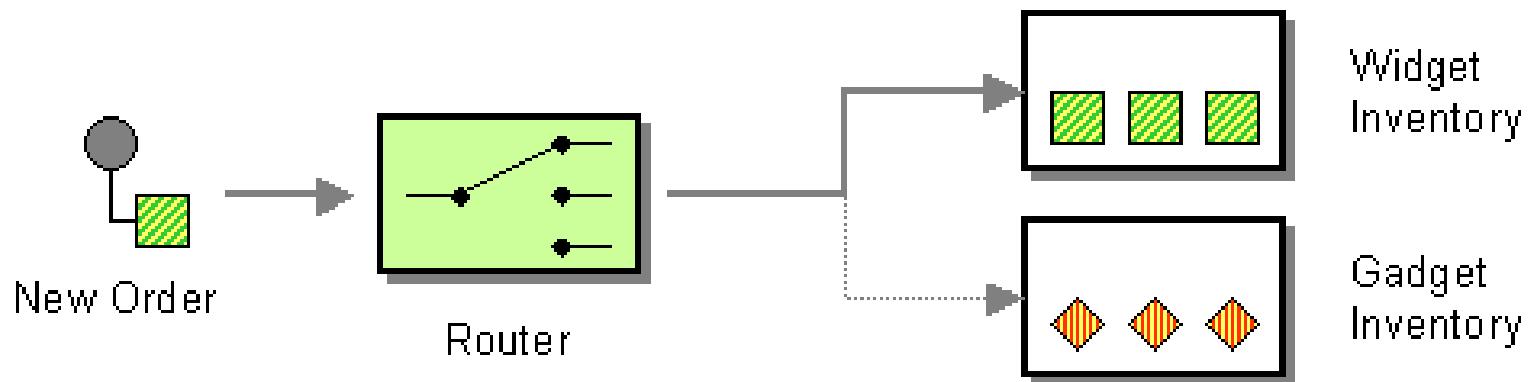


Camel has support for message transformation using several techniques. One such technique is [Data Formats](#), where marshal and unmarshal comes from.

So in other words the [Marshal](#) and [Unmarshal](#) EIPs are used with [Data Formats](#).

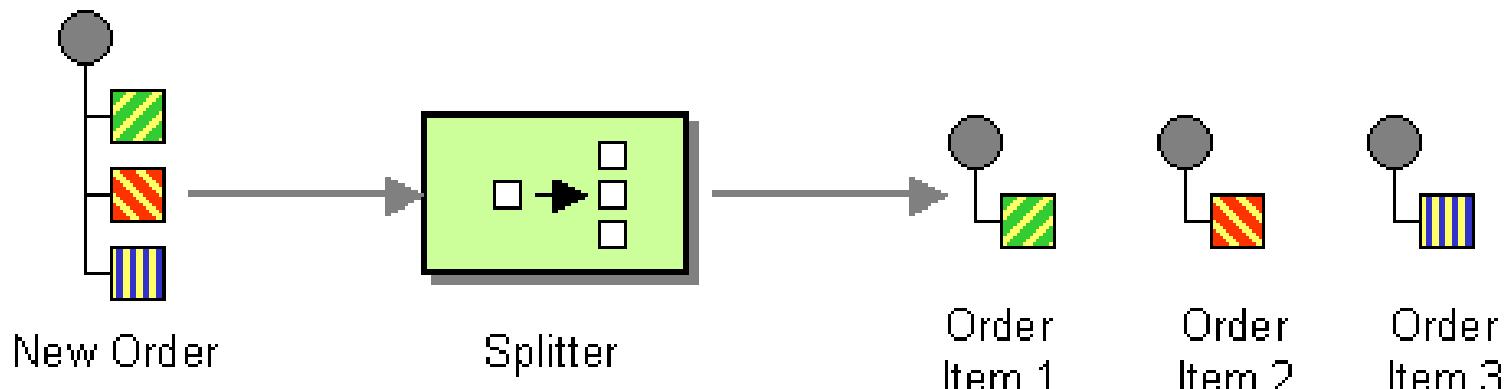
- *Marshal* - Transforms the message body (such as Java object) into a binary or textual format, ready to be wired over the network.
- *Unmarshal* - Transforms data in some binary or textual format (such as received over the network) into a Java object; or some other representation according to the data format being used.

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:

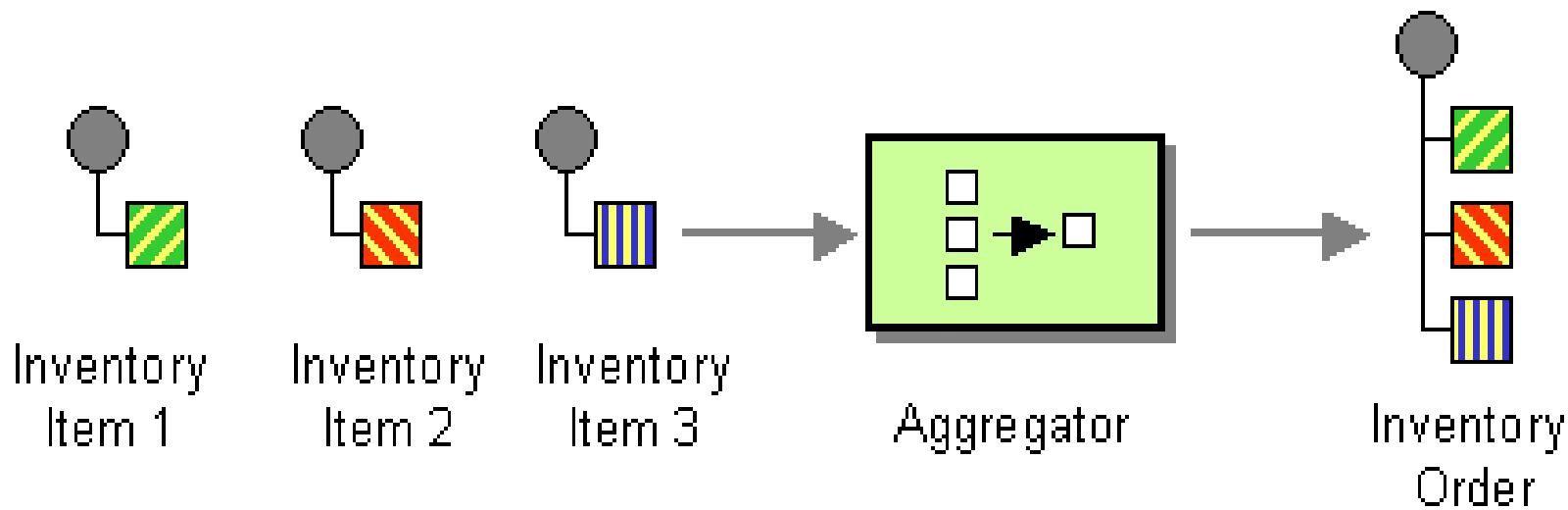
The Tokenizer language has a new option group that allows you to group N parts together, for example to split big files into chunks of 1000 lines.

```
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

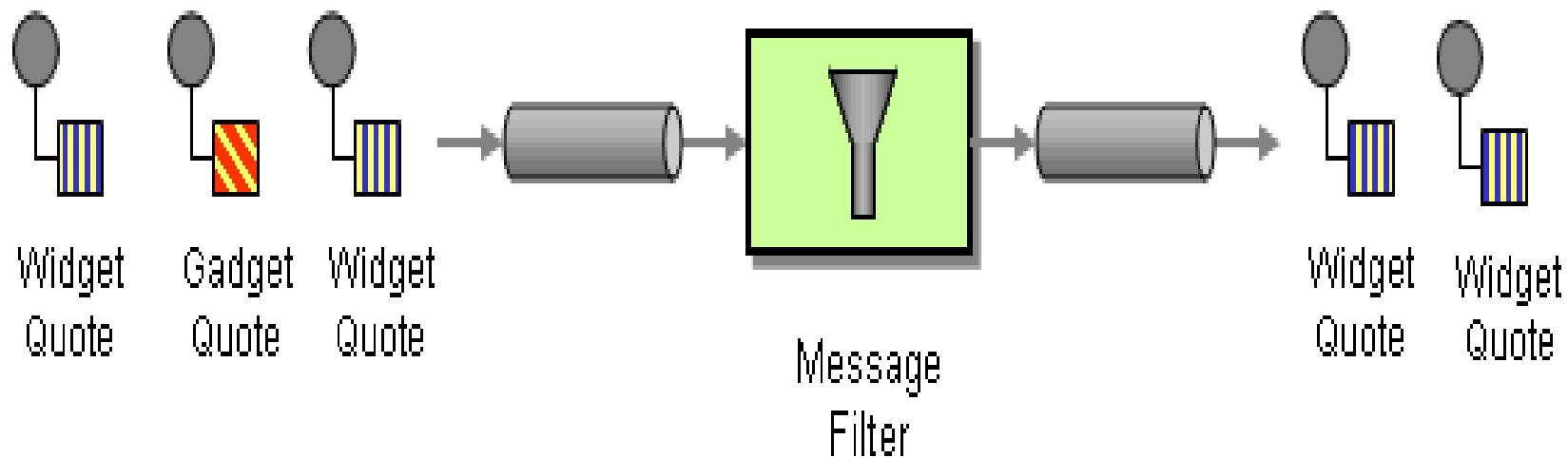
```
<route>
    <from uri="direct:aggregator" />
    <aggregate completionSize="500" completionInterval="60000"
eagerCheckCompletion="true">
        <correlationExpression>
            <xpath>/fizz/buzz</xpath>
        </correlationExpression>

        <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");  
    }  
};
```

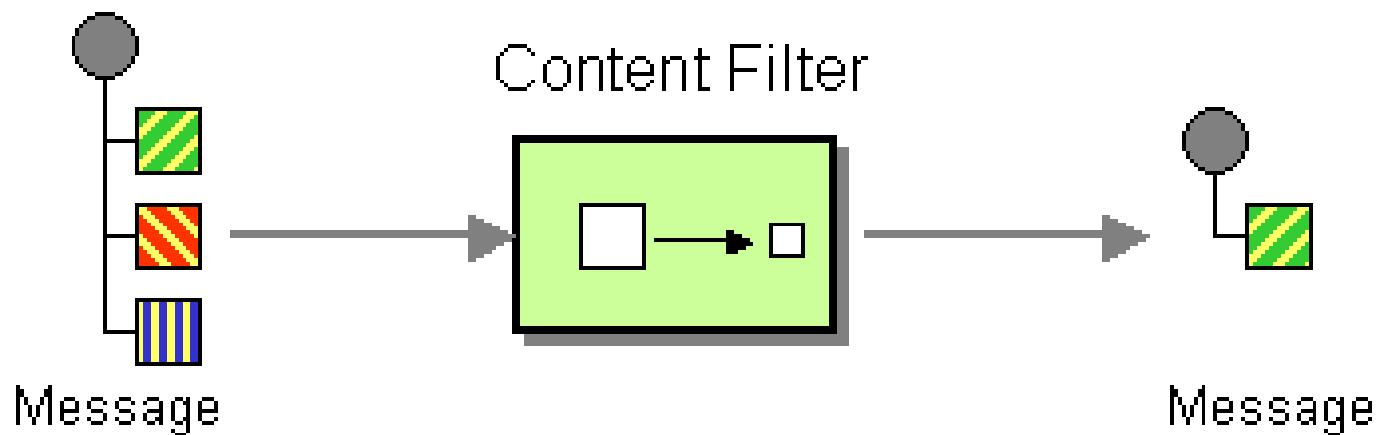
Using the Spring XML Extensions

```
<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

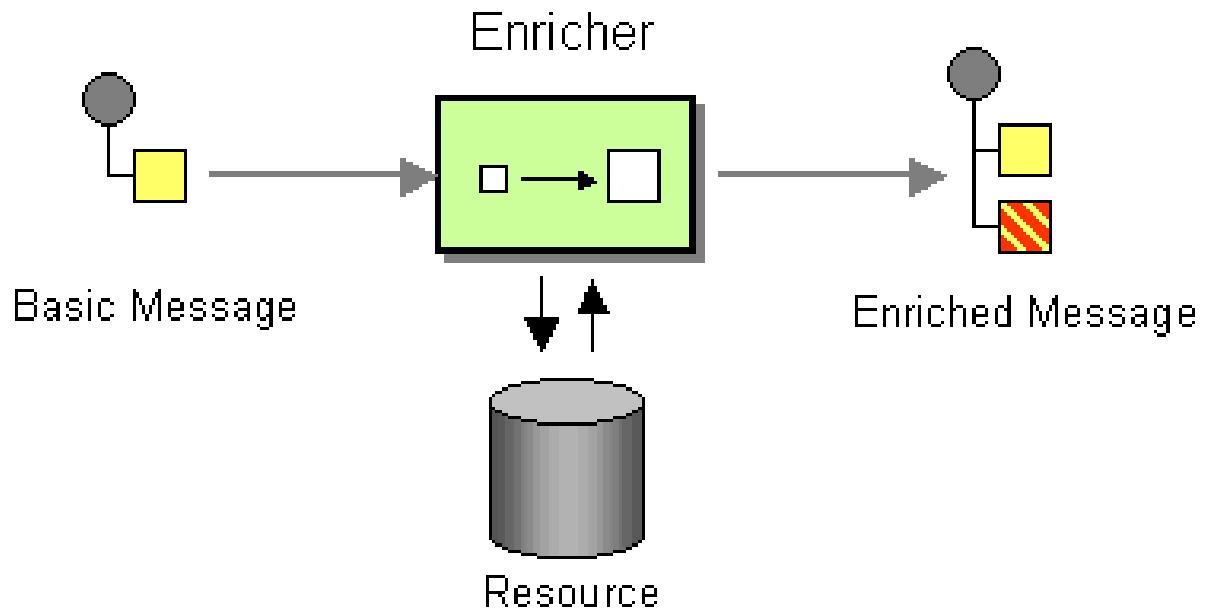
```
from("direct:start").setBody(body()).append(" World!").to("mock:result");
```

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")
```

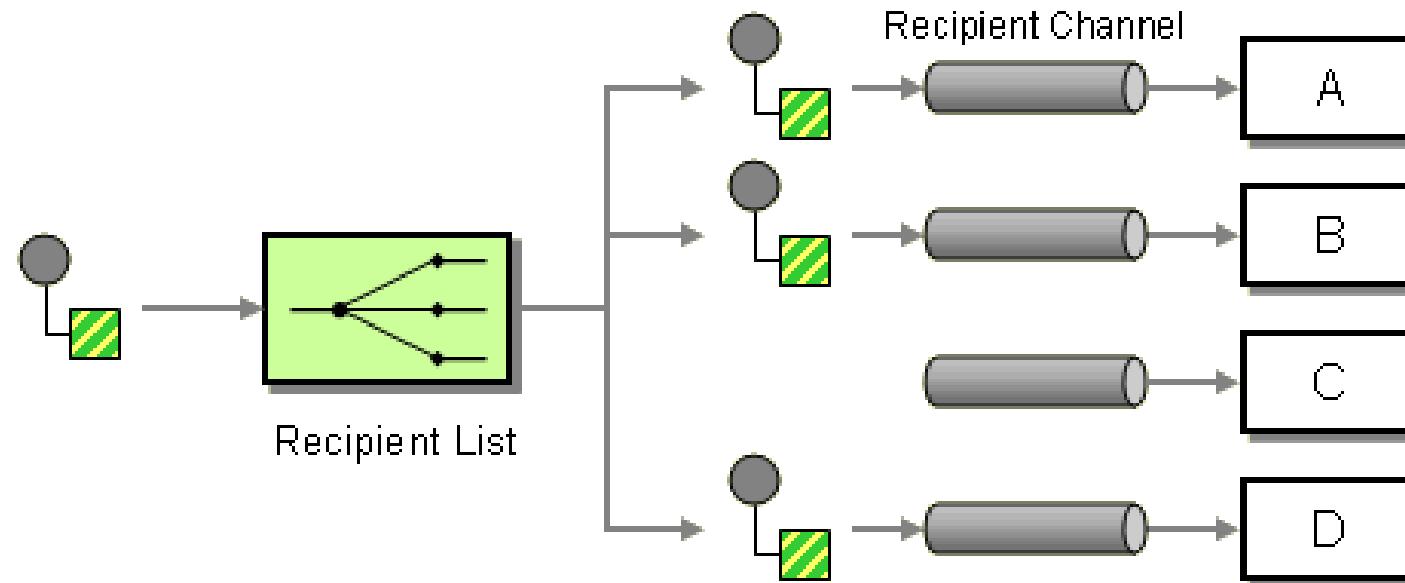
```
...
```

```
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 URLs:

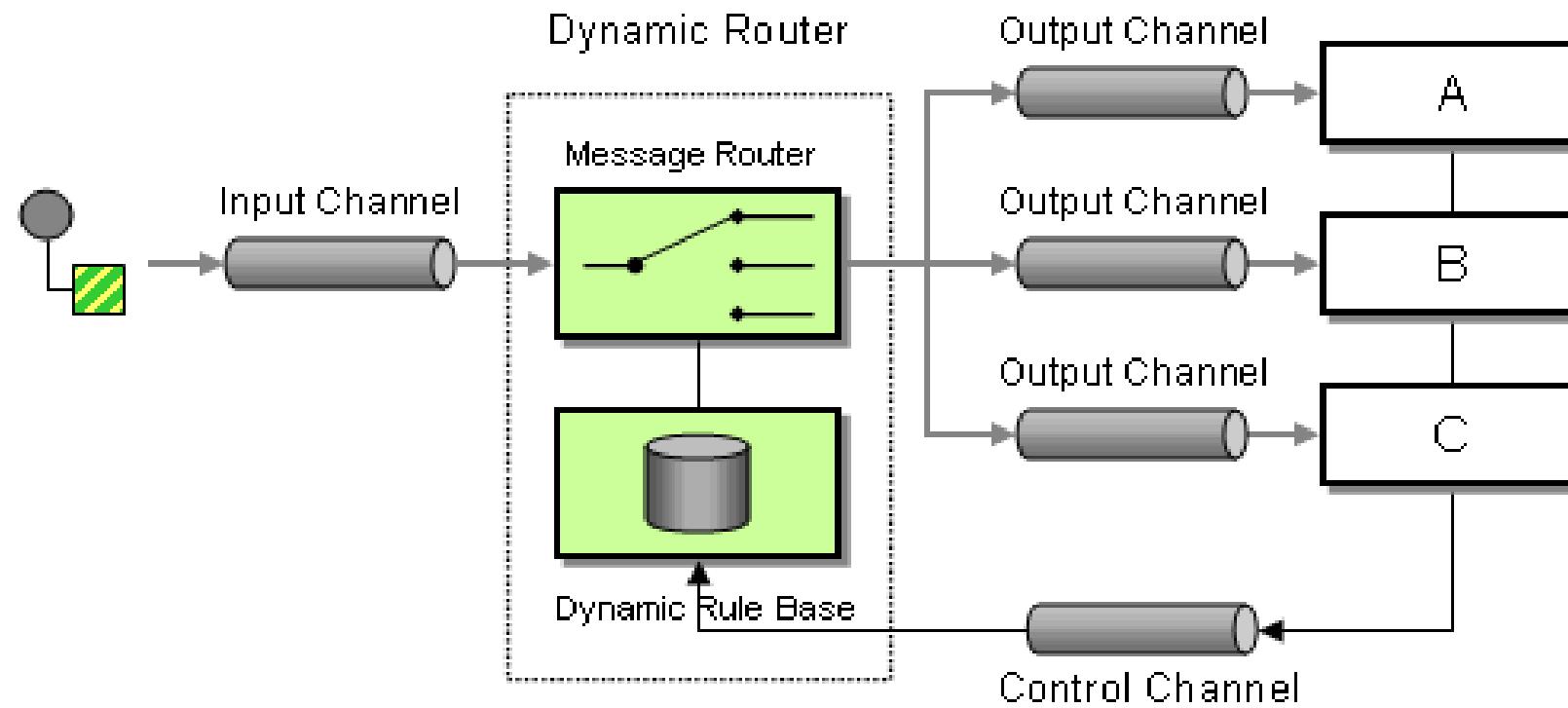
```
from("direct:a")
    .recipientList(header("recipientListHeader")
        .tokenize(","));
```

Dynamic Router

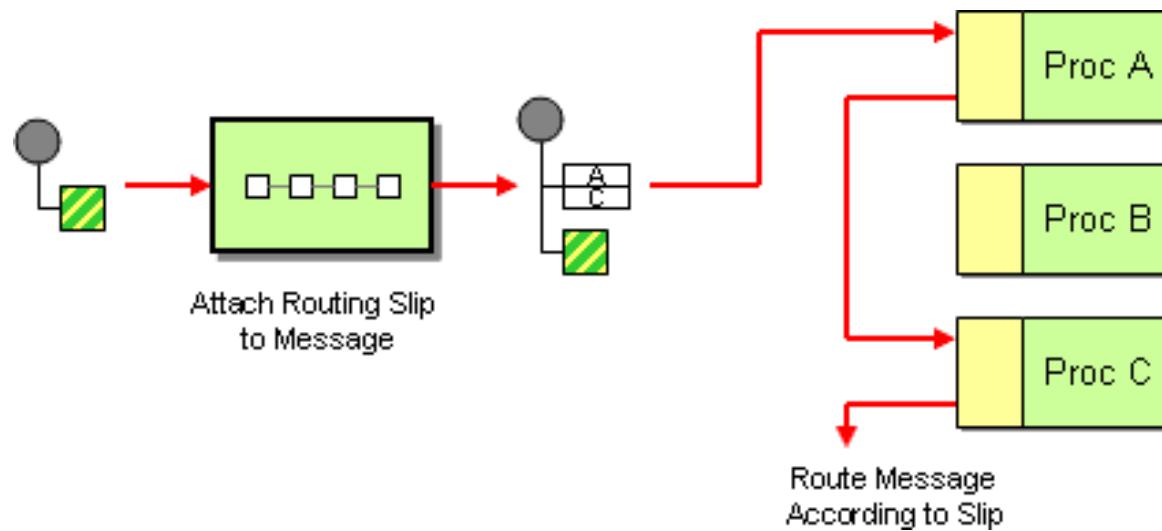
This pattern enables you to route a message consecutively through a series of processing steps, where the sequence of steps is not known at design time.

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

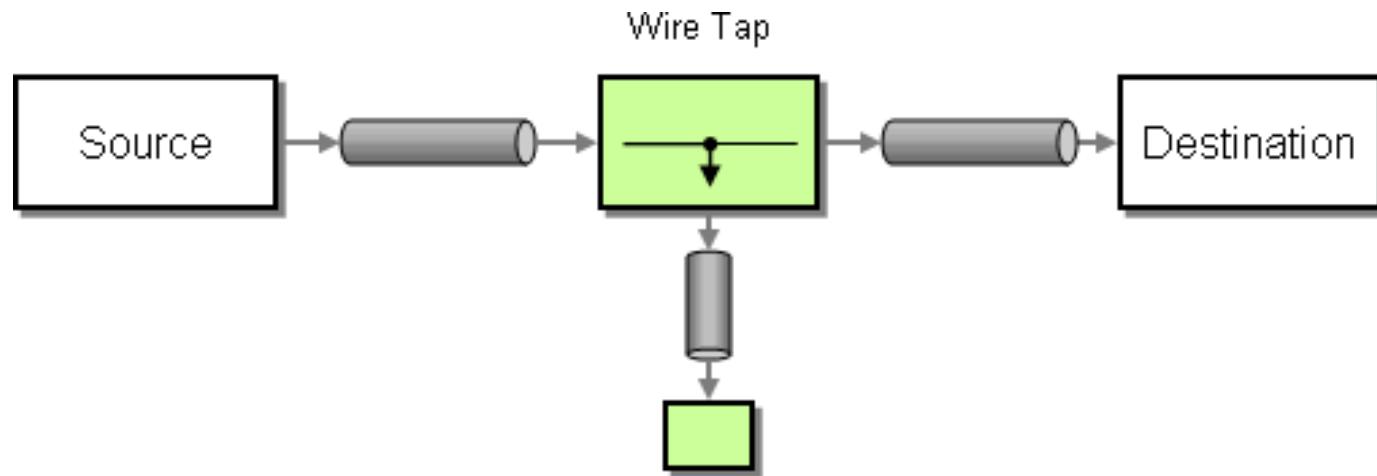
Each time the message returns from an endpoint, the dynamic router calls back on a bean to discover the next endpoint in the route.



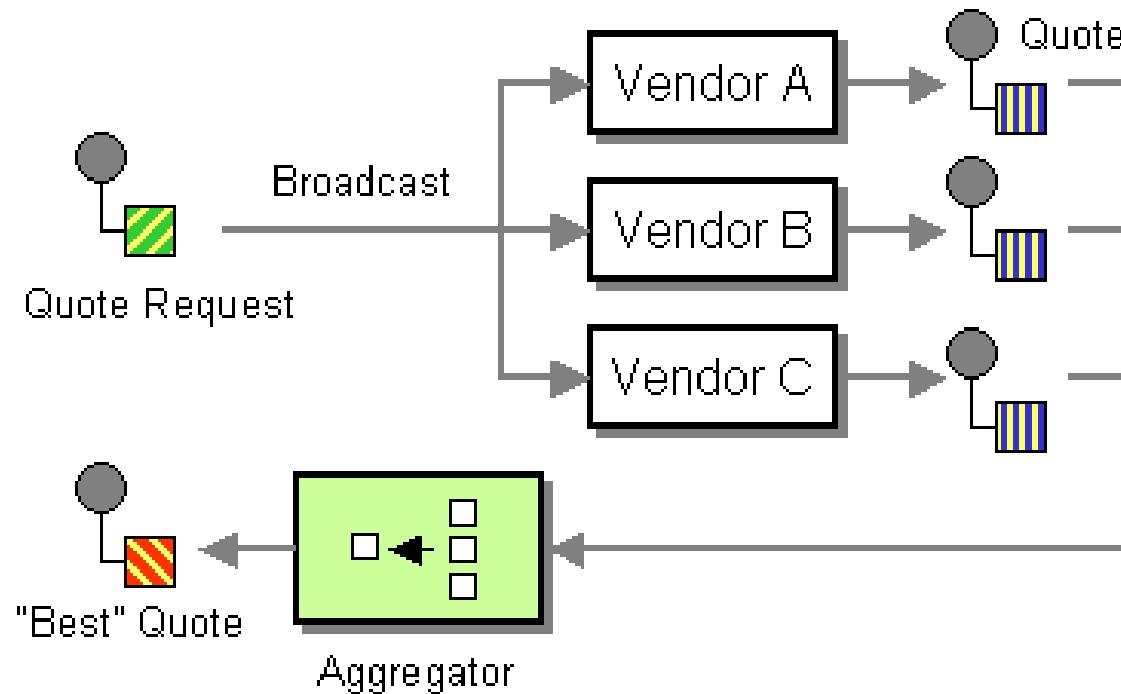
The Routing Slip from the EIP patterns allows you to route a message consecutively through a series of processing steps where the sequence of steps is not known at design time and can vary for each message.



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

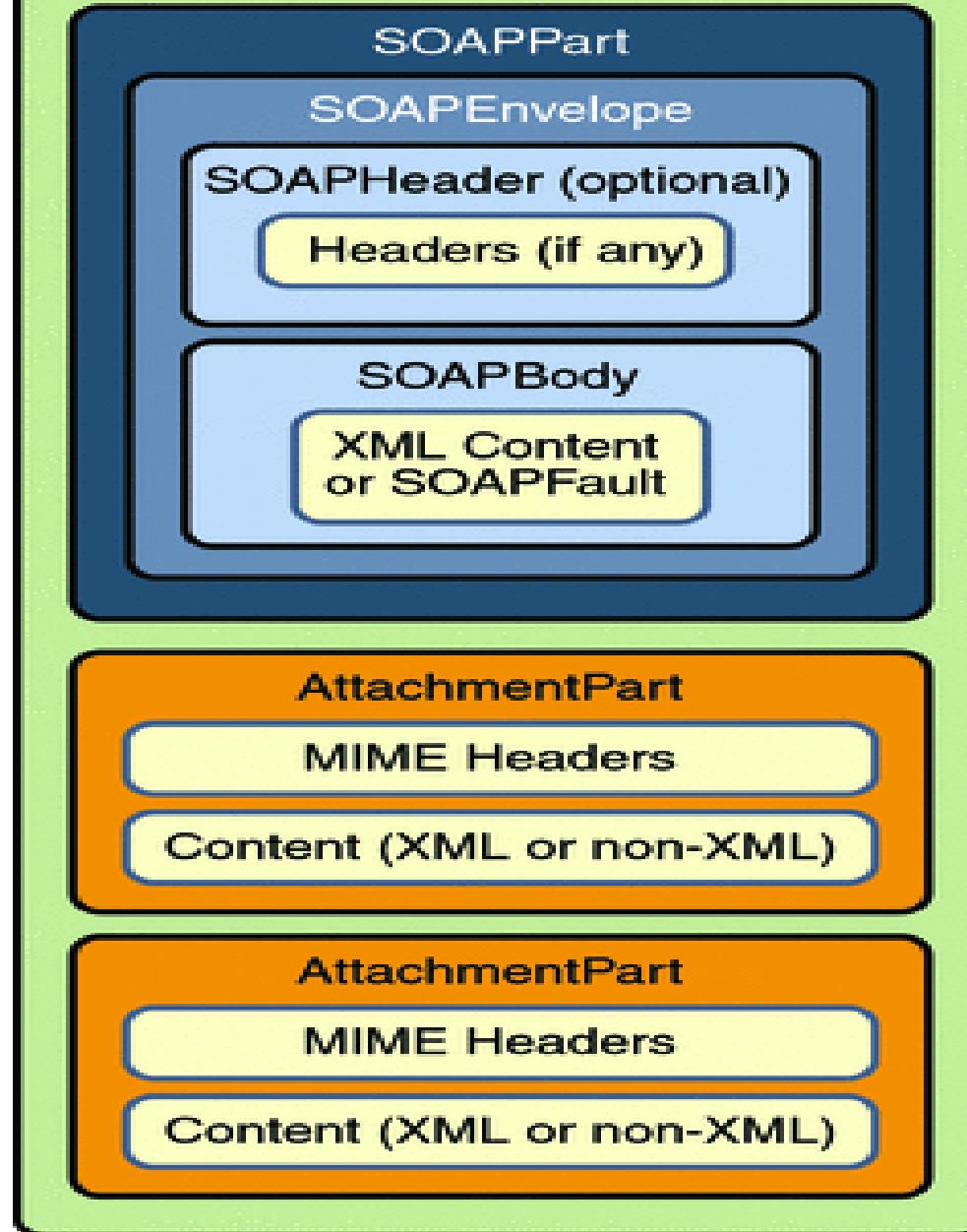
Apache CXF webservices

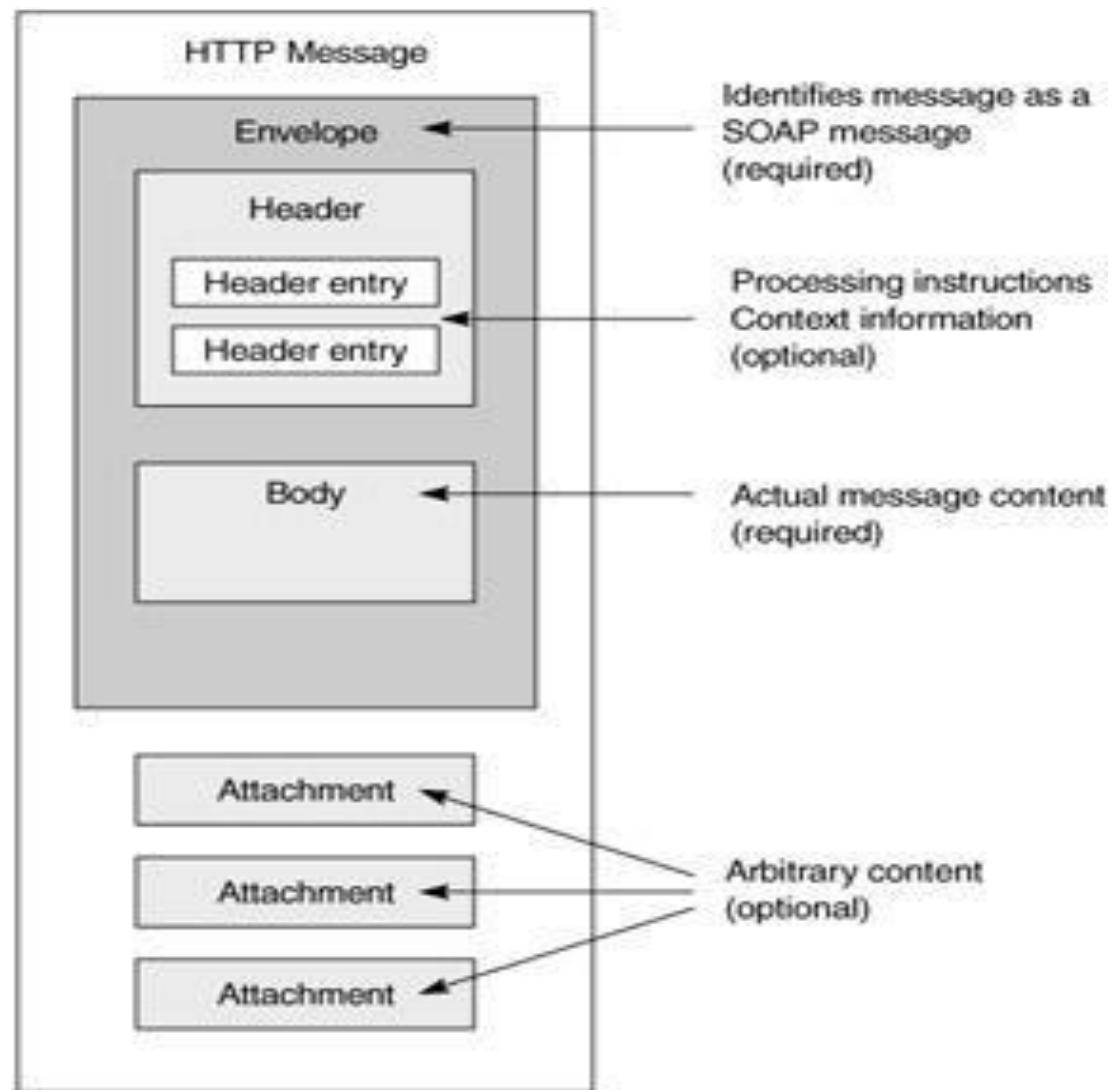
Wsdl document consists of :

- 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)





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" /><!-- The web consumer -->
            <to uri="cxft:bean:webClient" /><!-- The web producer -->
        </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:

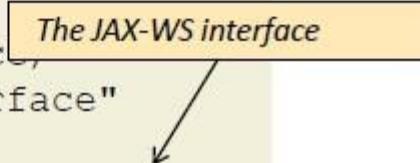
```
<beans xmlns:cxf="http://camel.apache.org/schema/cxf">

    <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*

```
<cxf:cxfEndpoint 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">
</cxfEndpoint>
```

...

*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` }
 - **PAYOUT**: 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:

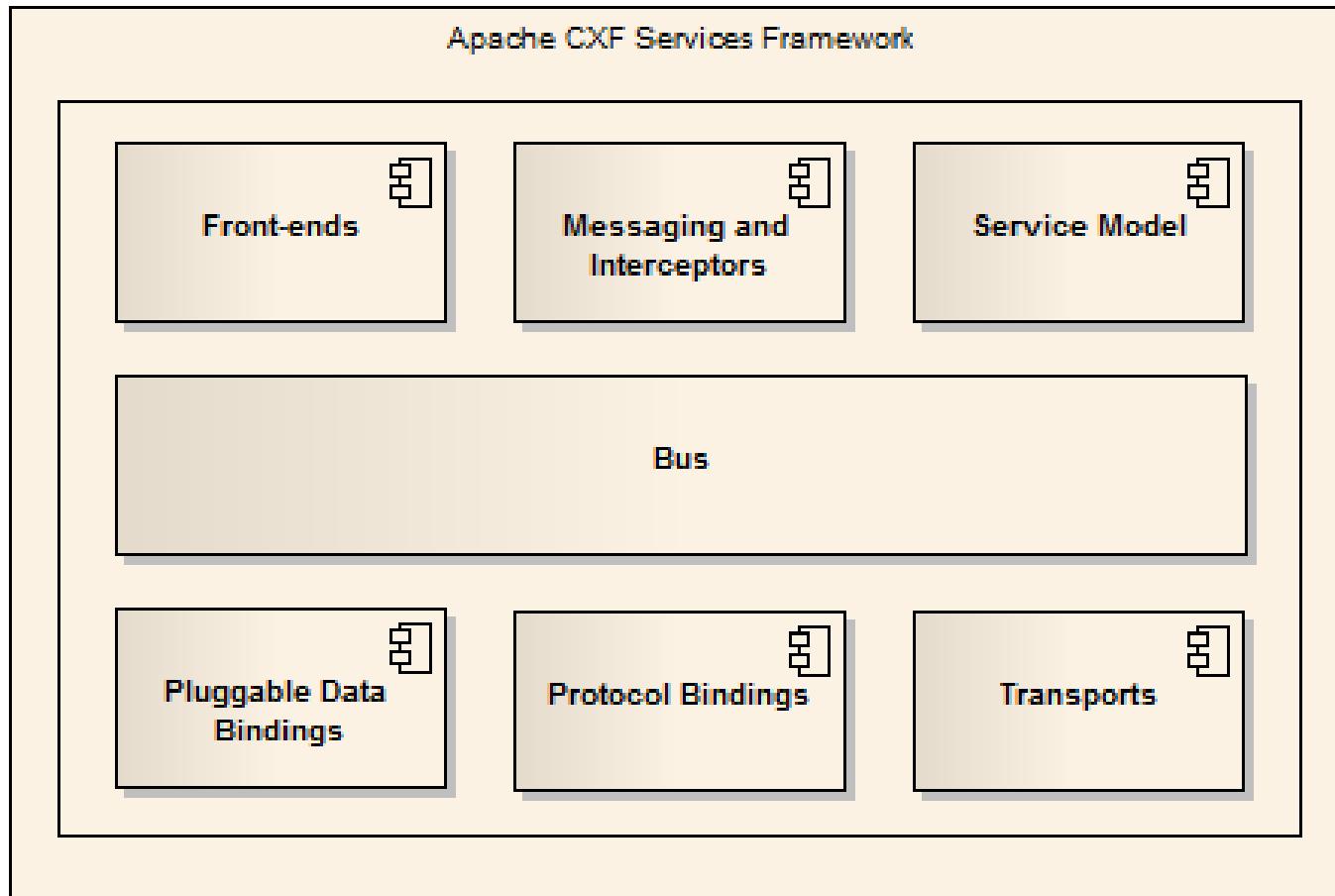
The message body is effectively an XML payload of org.w3c.dom.Node type (wrapped in a CxfPayload object).

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

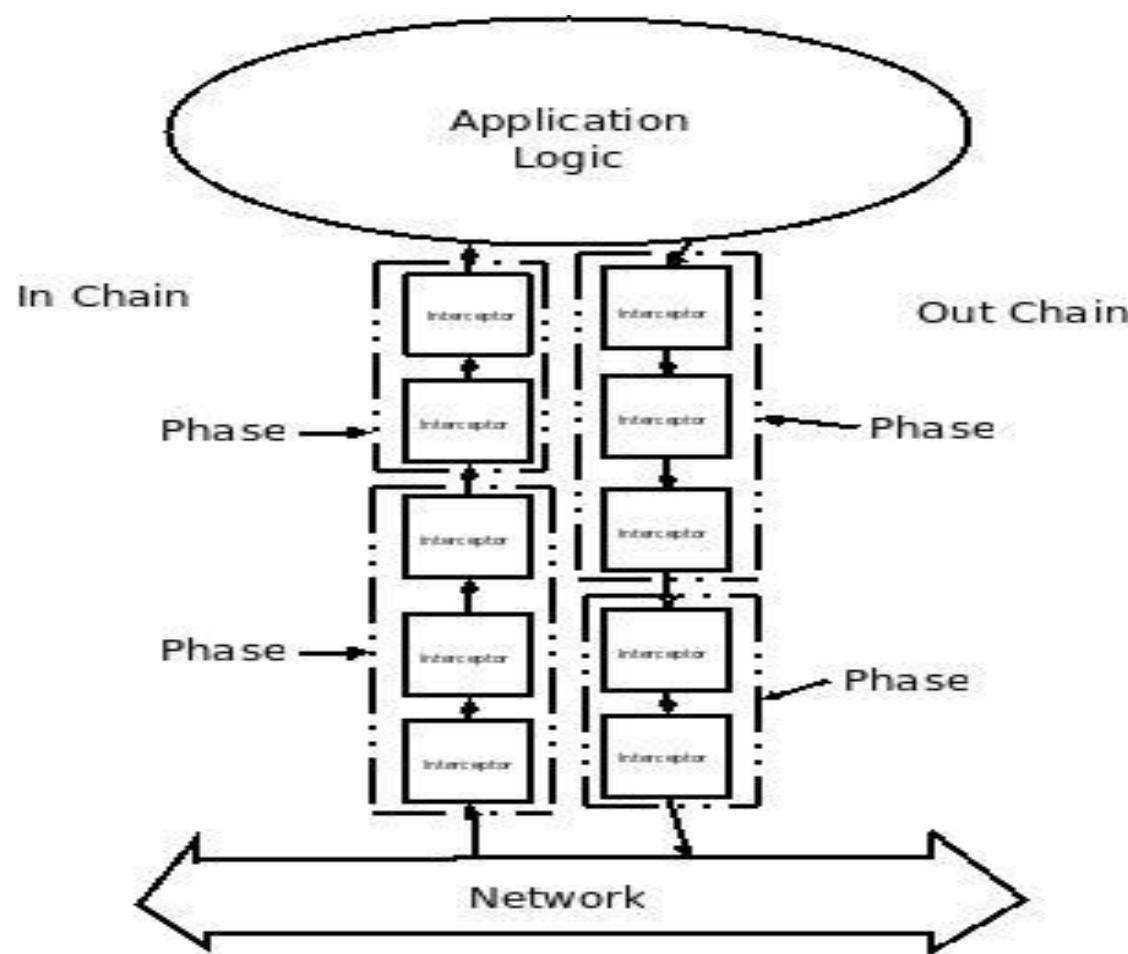
The overall CXF architecture is primarily made up of the following parts:

- 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

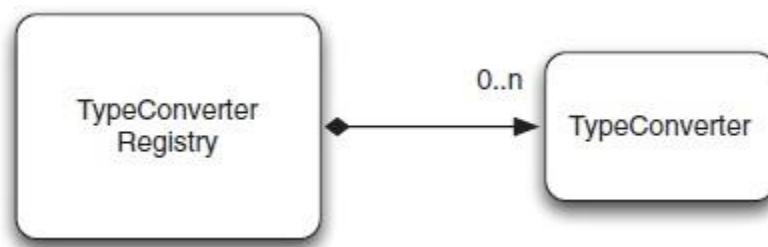
Camel provides a built-in type-converter system that automatically converts between well-known types

```
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  
}
```

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}");
```

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");
```

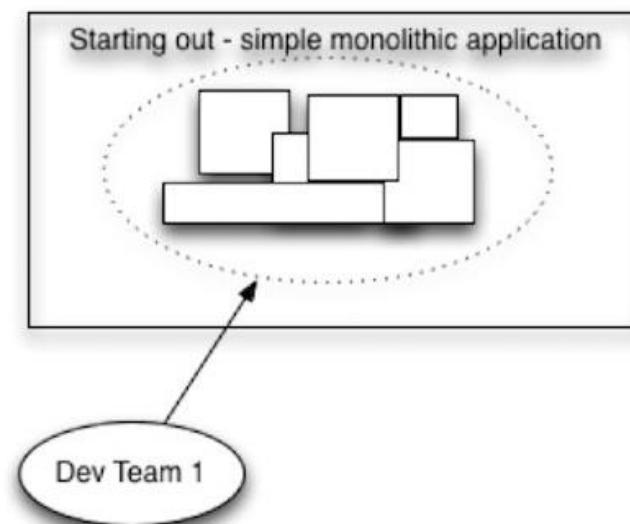
The possible values are defined in the enum class, `org.apache.camel.jms.JmsMessageType`.

Introduction To Microservice

Monolith Application:

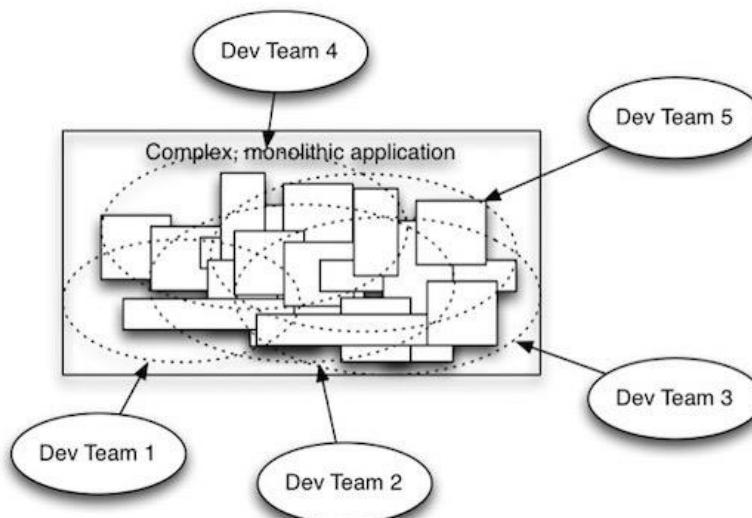
A single monolith would contain all the code for all the business activities an application performed.

We write a simple application, which is developed and managed by a single team.



But what happens if your application turns out to be successful?
Users like it and begin to depend on it.

Traffic increases dramatically. And almost inevitably, users request improvements and additional features, so more developers are roped in to work on the growing application. Before too long, your application looks more like this:



What is a Microservices Architecture in a Nutshell?

“single responsibility principle” which states “gather together those things that change for the same reason, and separate those things that change for different reasons.”

A microservices architecture takes this same approach and extends it to the loosely coupled services which can be developed, deployed, and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.

Example

A class or module should have one, and only one, reason to be changed (i.e. rewritten).

As an example, consider a module that compiles and prints a report.

Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change.

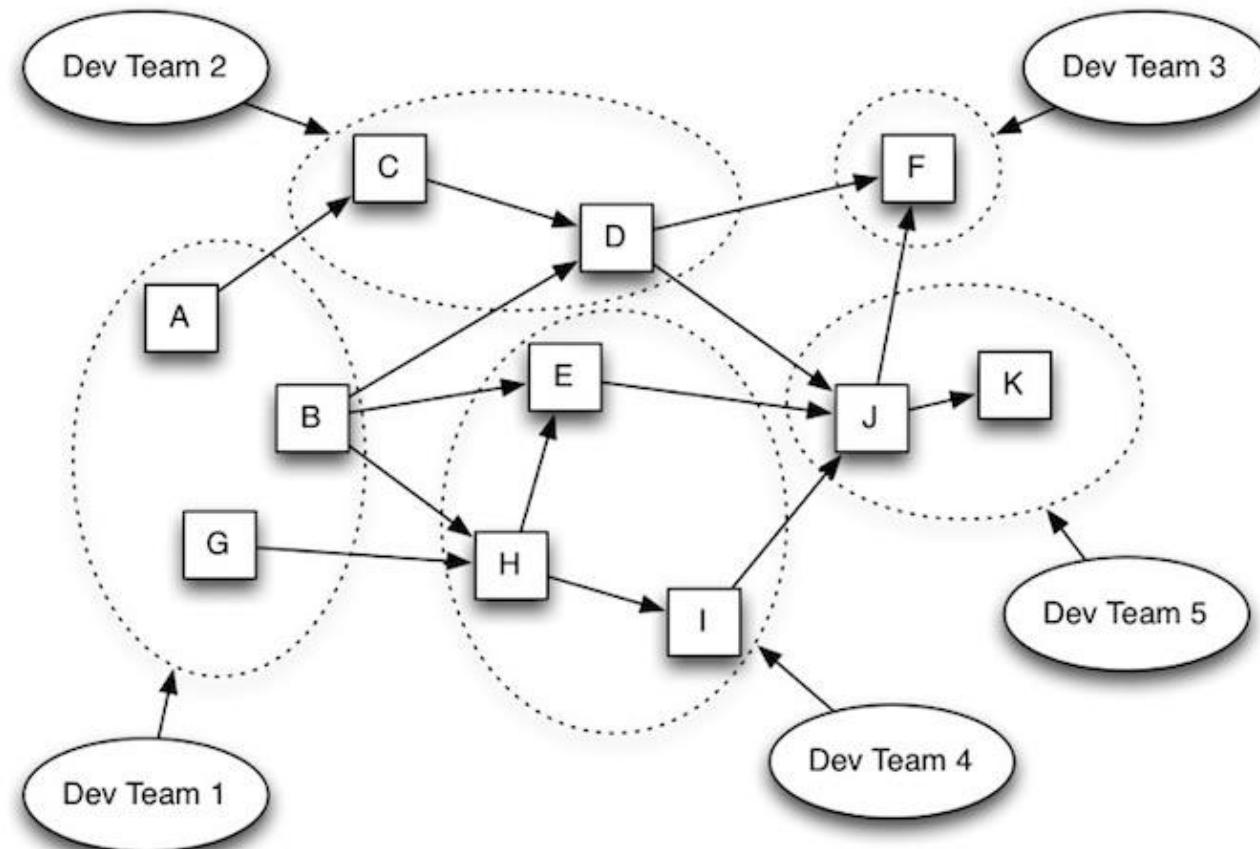
The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

Applications built using microservices possess certain characteristics.

In particular, they:

- ❑ Are fragmented into multiple modular, loosely coupled components, each of which performs a discrete function
- ❑ Have those individual functions built to align to business capabilities
- ❑ Can be distributed across clouds and data centres
- ❑ Treat each function as an independent service that can be changed, updated, or deleted without disrupting the rest of the application



Microservices

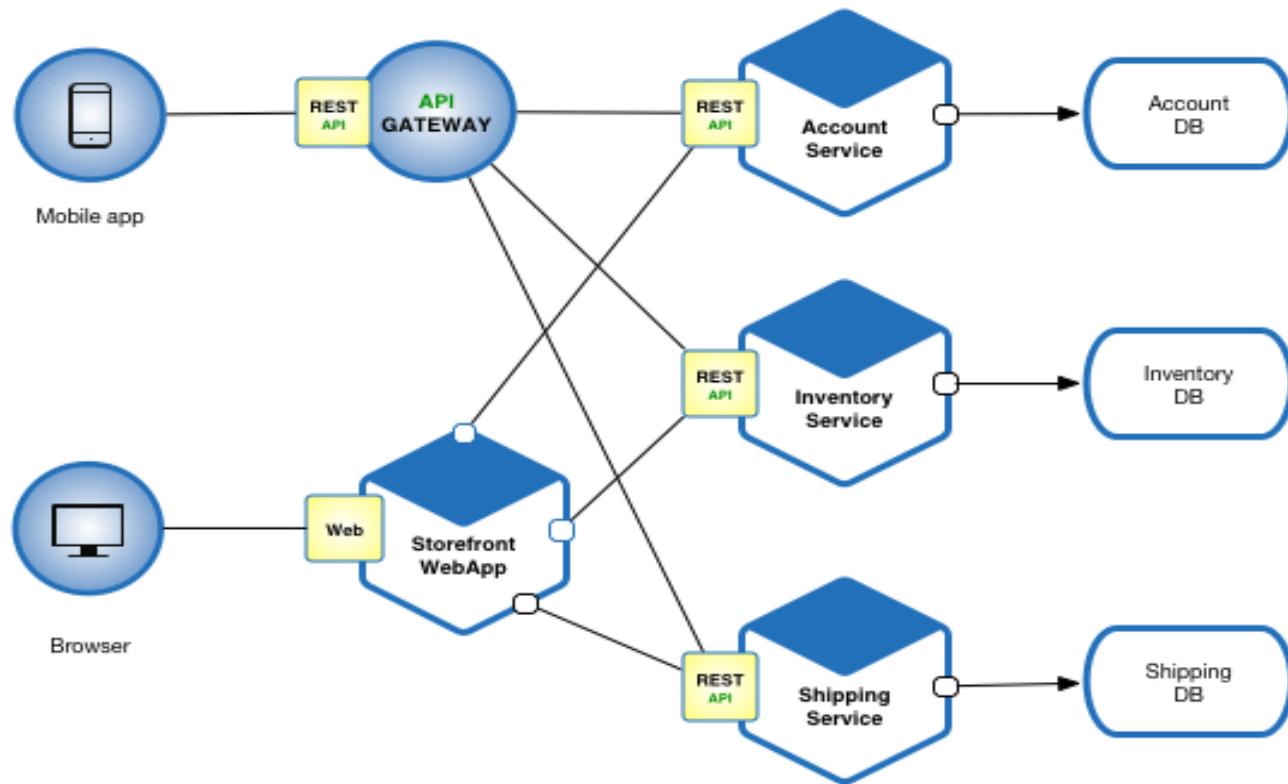
So what exactly is a microservice architecture?

According to Martin Fowler:

A microservice architecture consists of “set of independently deployable services” organized “around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.”

Microservice architecture is a method of developing software applications as a set of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

Microservice Architecture



Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

*The microservices style is usually organized around
business capabilities and priorities.*

Unlike a traditional monolithic development approach—where different teams each have a specific focus on, say, UIs, databases, technology layers, or server-side logic—microservice architecture utilizes cross-functional teams.

The responsibilities of each team are to make specific products based on one or more individual services communicating via message bus. That means that when changes are required, there won't necessarily be any reason for the project, as a whole, to take more time or for developers to have to wait for budgetary approval before individual services can be improved.

Domain Driven Design Patterns



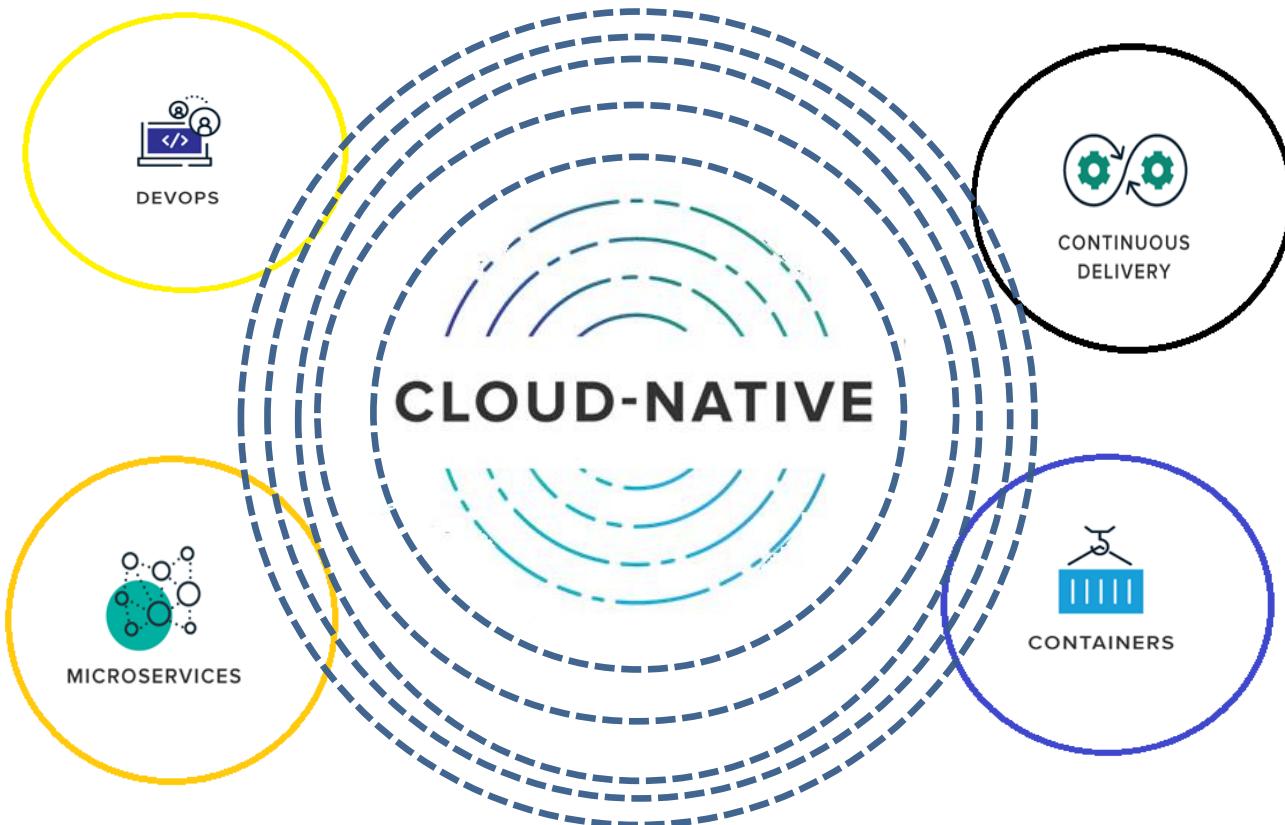
Cloud Native Platform



- ❑ **Cloud-native** is an approach to building and running applications that exploits the **advantages** of the **cloud computing delivery** model.

- ❑ Cloud-native applications conform to a framework or "**contract**" designed to maximize resilience through predictable behaviours.

- ❑ Organizations require a platform for building and operating cloud-native applications and services that automates and integrates on **FOUR characteristics**.

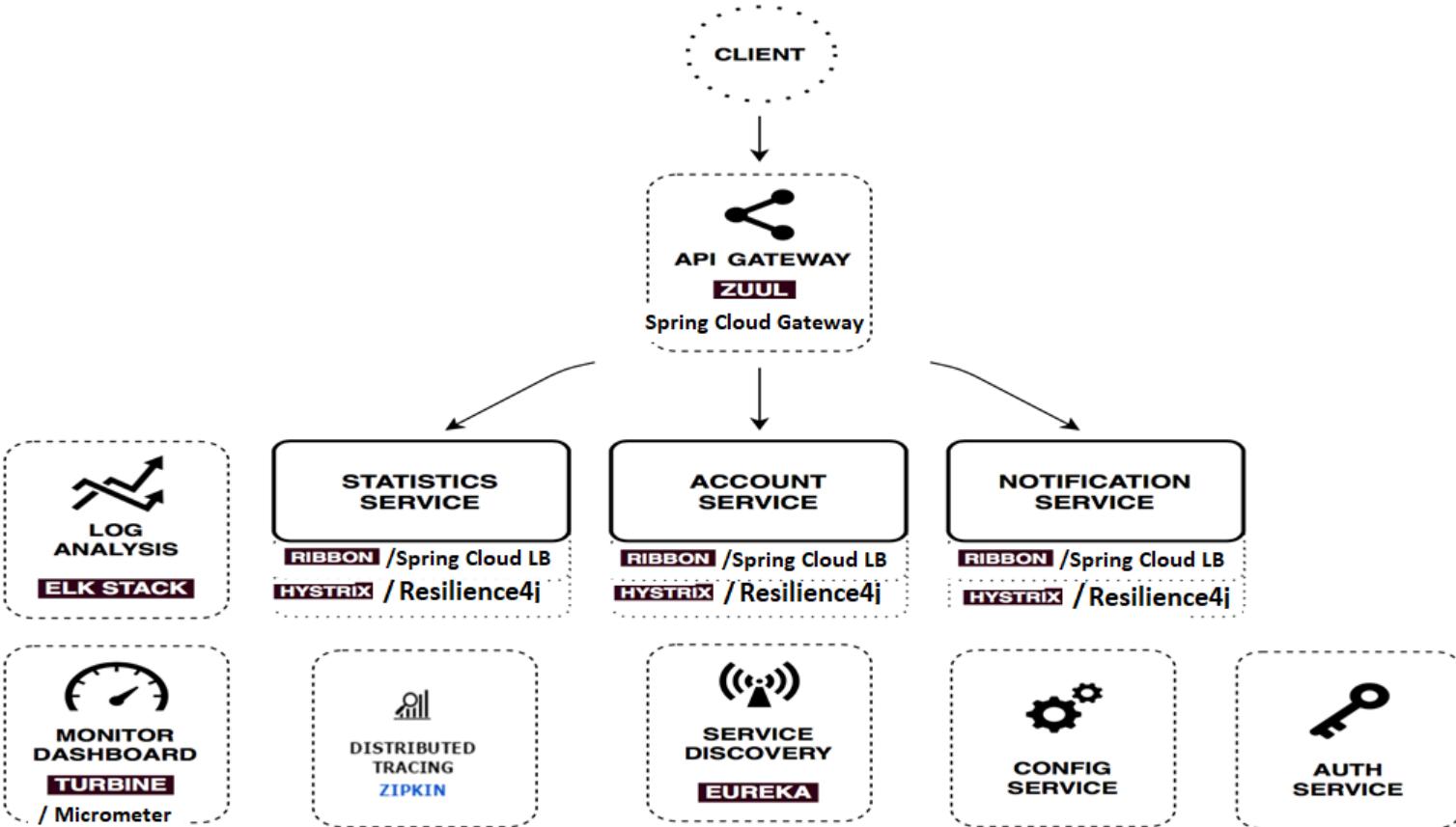


Twelve-Factor Applications

1. CODEBASE One codebase tracked in SCM, many deploy	2. DEPENDENCIES Explicitly declare isolate dependencies	3. CONFIGURATION Store config in the environment
4. BACKING SERVICES Treat backing services as attached resources	5. BUILD, RELEASE, RUN Strictly separate build and run stages	6. PROCESSES Execute app as stateless processes
7. PORT BINDING Export services via port binding	8. CONCURRENCY Scale out via the process model	9. DISPOSABILITY Maximize robustness & graceful shutdown
10. DEV/ PROD PARITY Keep dev, staging, prod as similar as possible	11. LOGS Treat logs as event stream	12. ADMIN PROCESSES Run admin / mgmt tasks as one-off processes

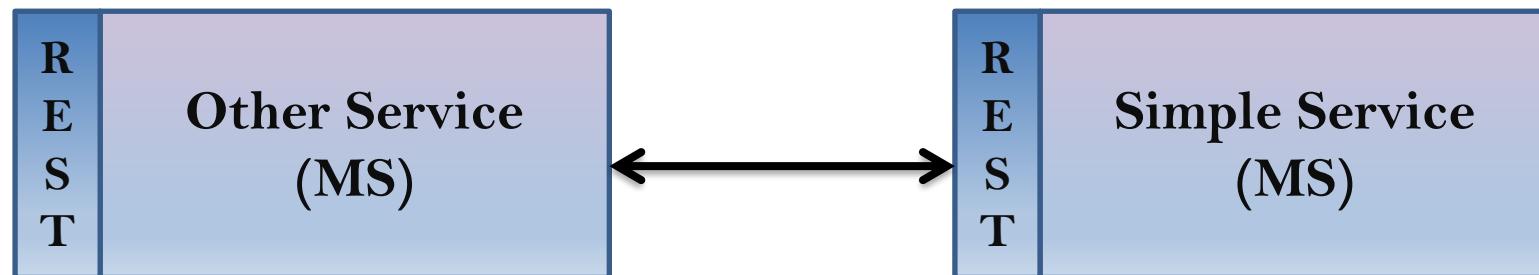
Microservice Patterns:

- API gateway
- Service registry
- Circuit breaker
- Messaging
- Database per Service
- Access Token
- Saga
- Event Sourcing & CQRS



Problem

How do clients of a service (in the case of Client-side discovery) and/or routers (in the case of Server-side discovery) know about the available instances of a service?



Solution

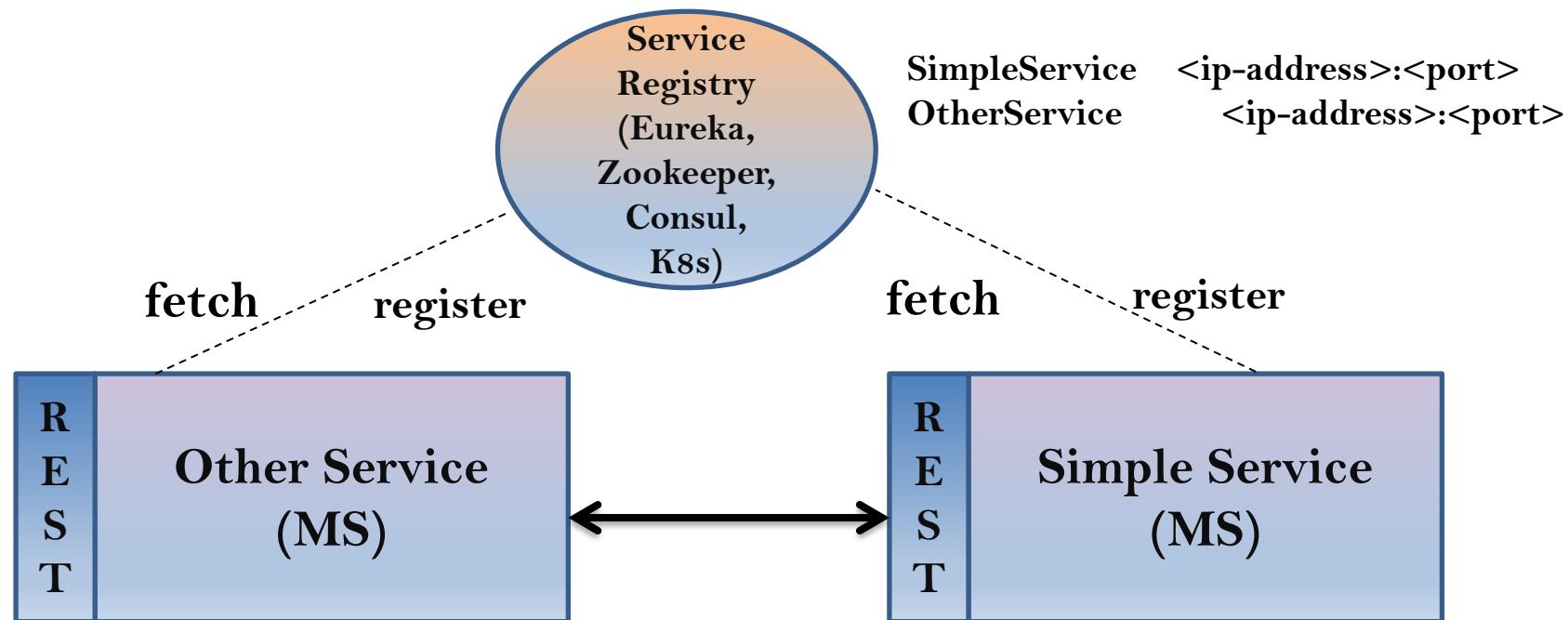
Implement a service registry, which is a database of services, their instances and their locations.

Service instances are registered with the service registry on startup and deregistered on shutdown.

Client of the service and/or routers query the service registry to find the available instances of a service.

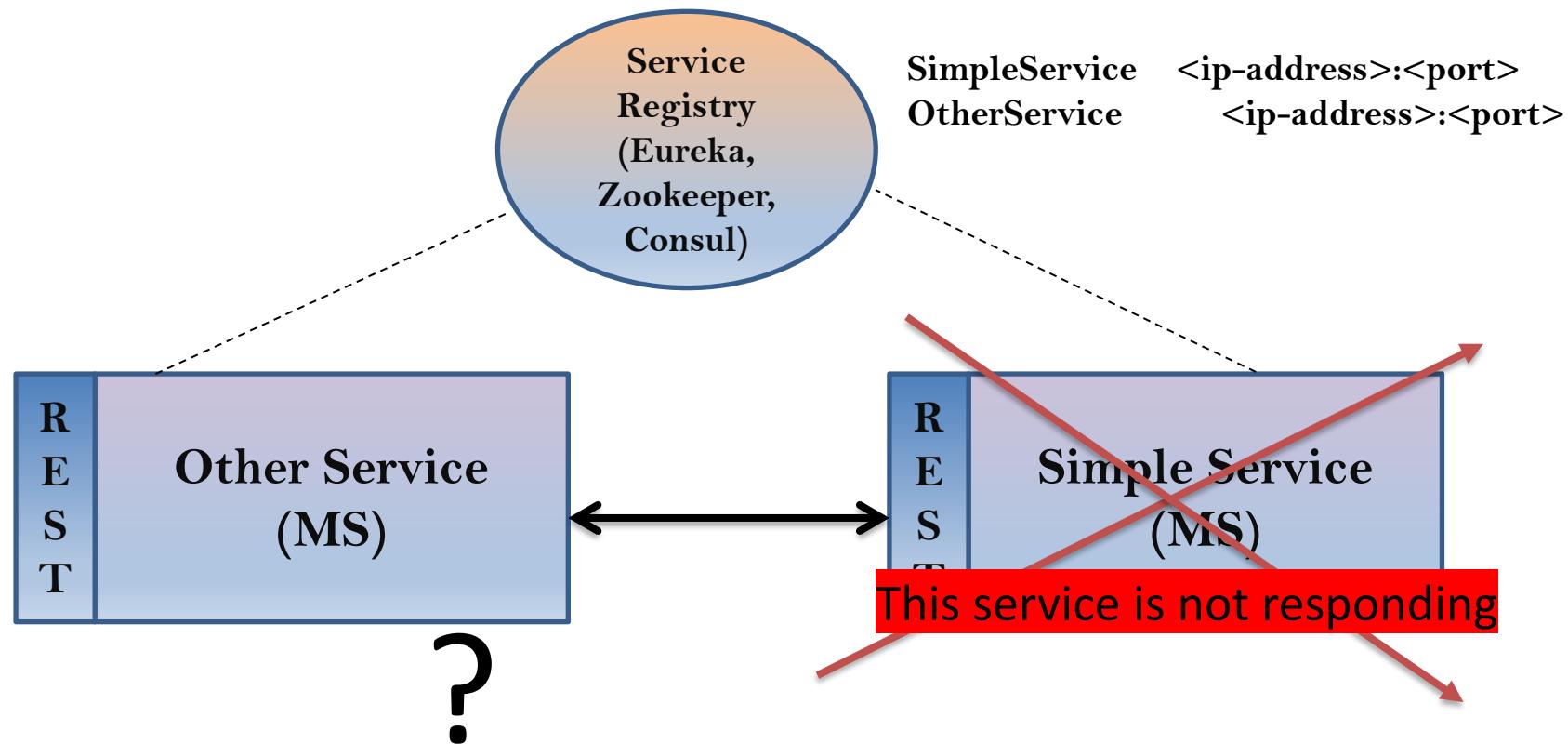
A service registry might invoke a service instance's health check API to verify that it is able to handle requests

Service registry Pattern



Problem

How to prevent a network or service failure from cascading to other services?



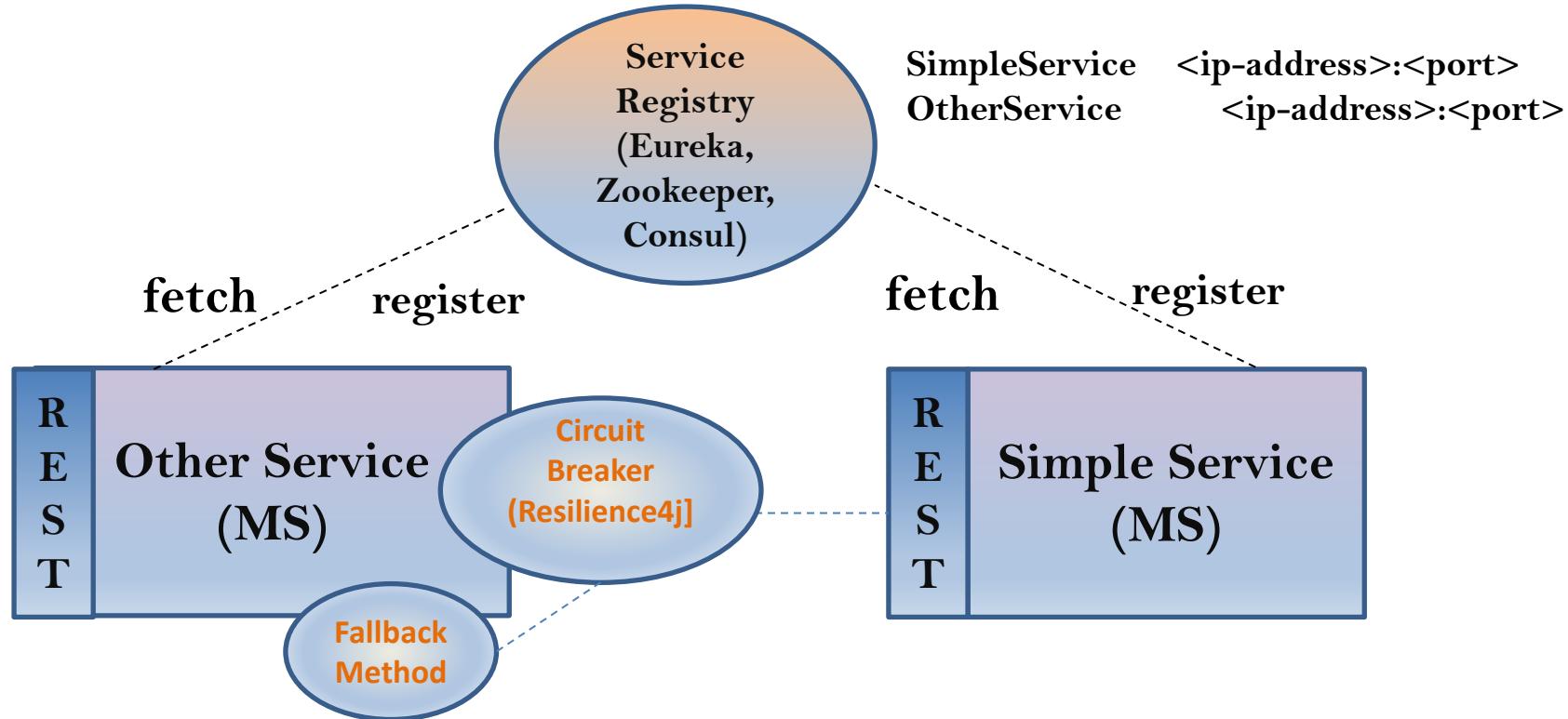
Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

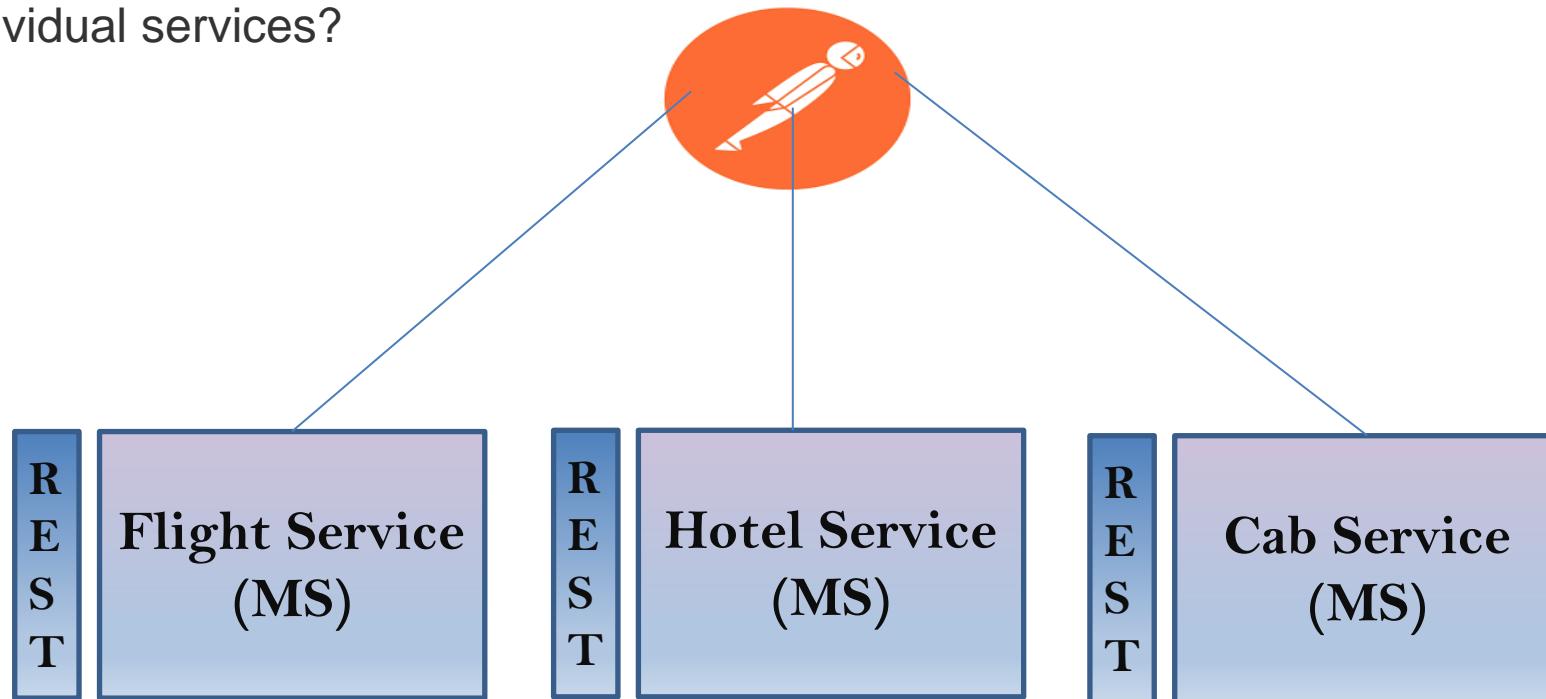
After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

Circuit Breaker Pattern



Problem

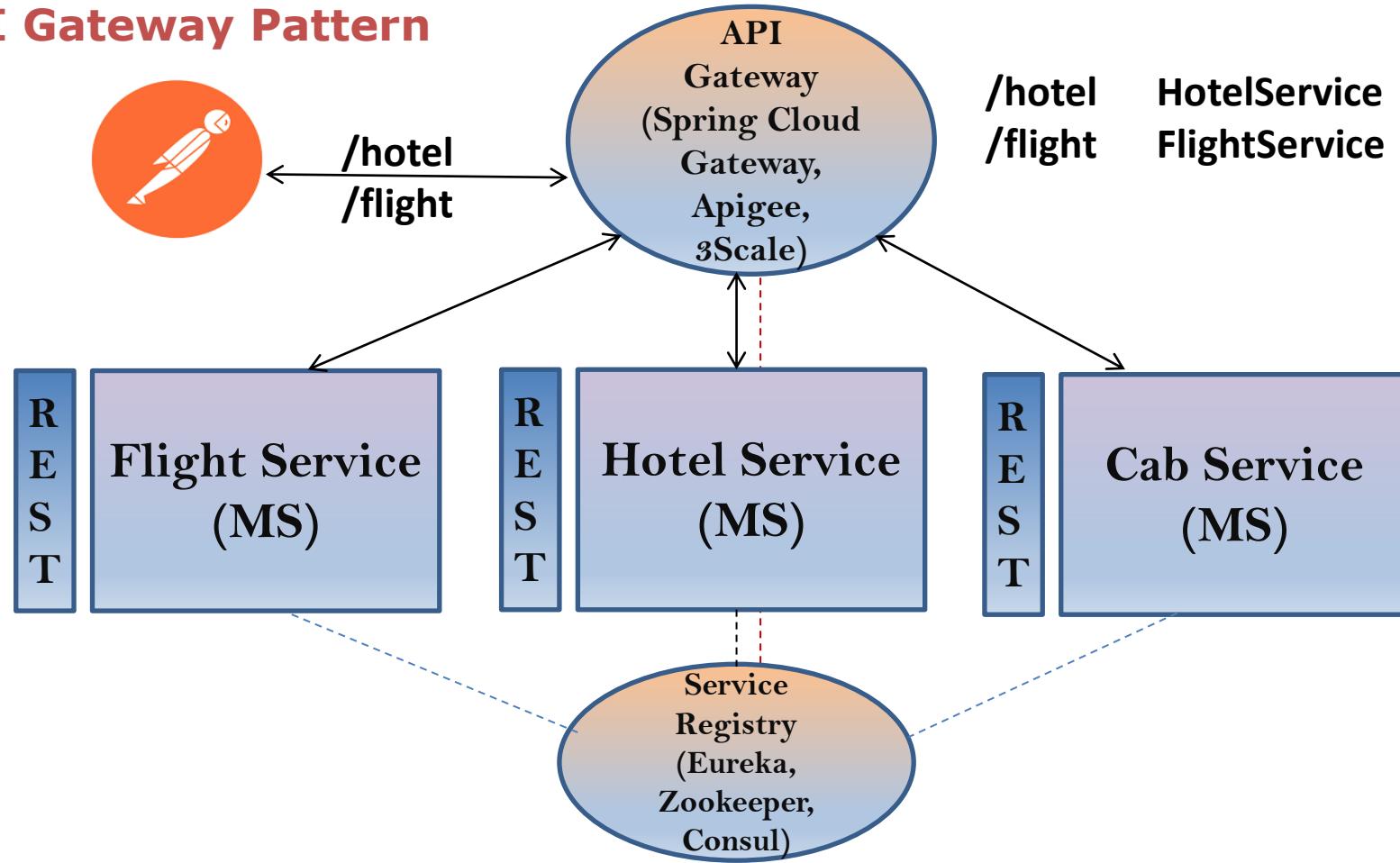
How do the clients of a Microservices-based application access the individual services?



Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles

API Gateway Pattern



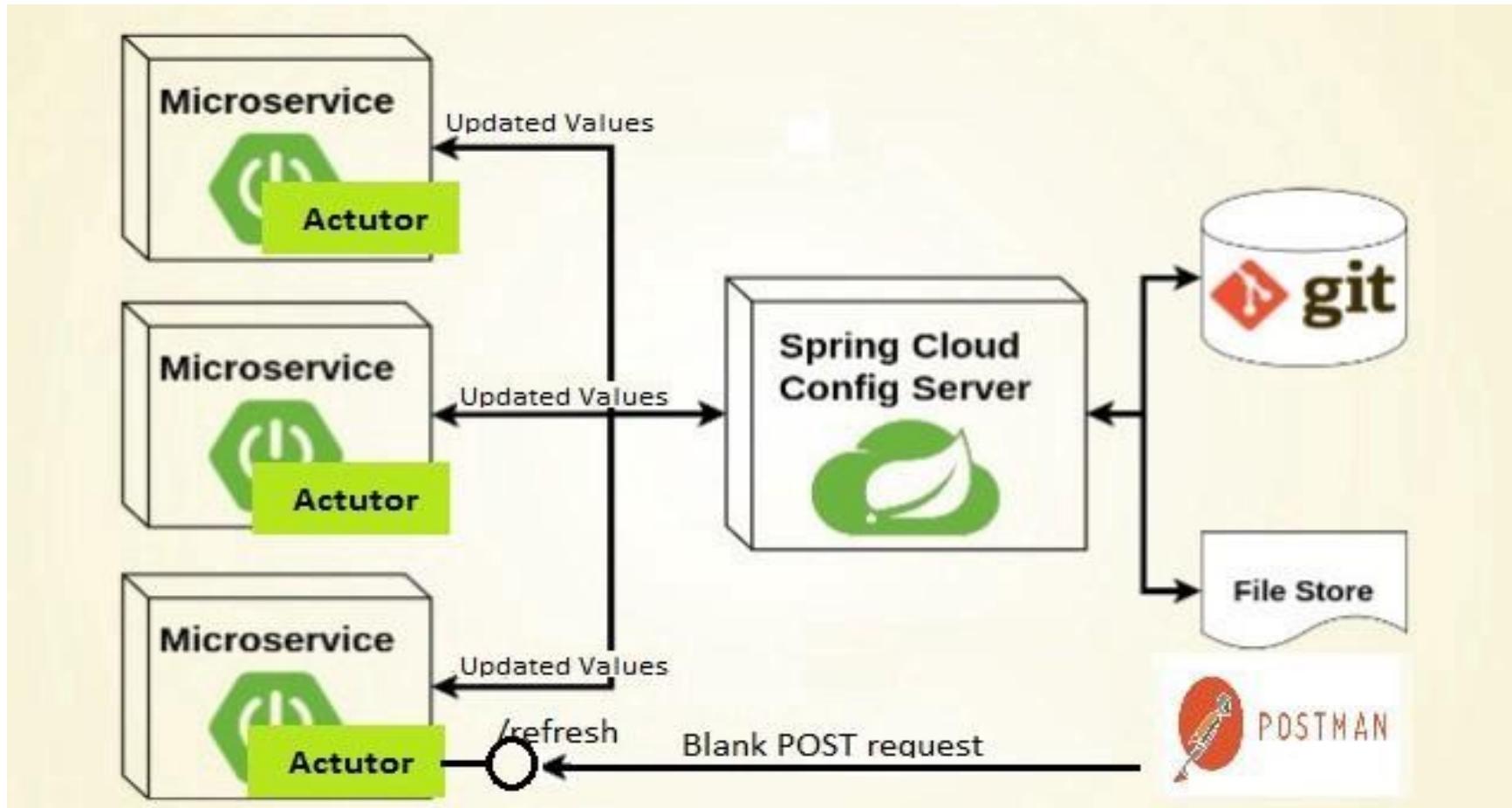
Pattern: Externalized configuration

Problem

How to enable a service to run in multiple environments without modification?

Solution

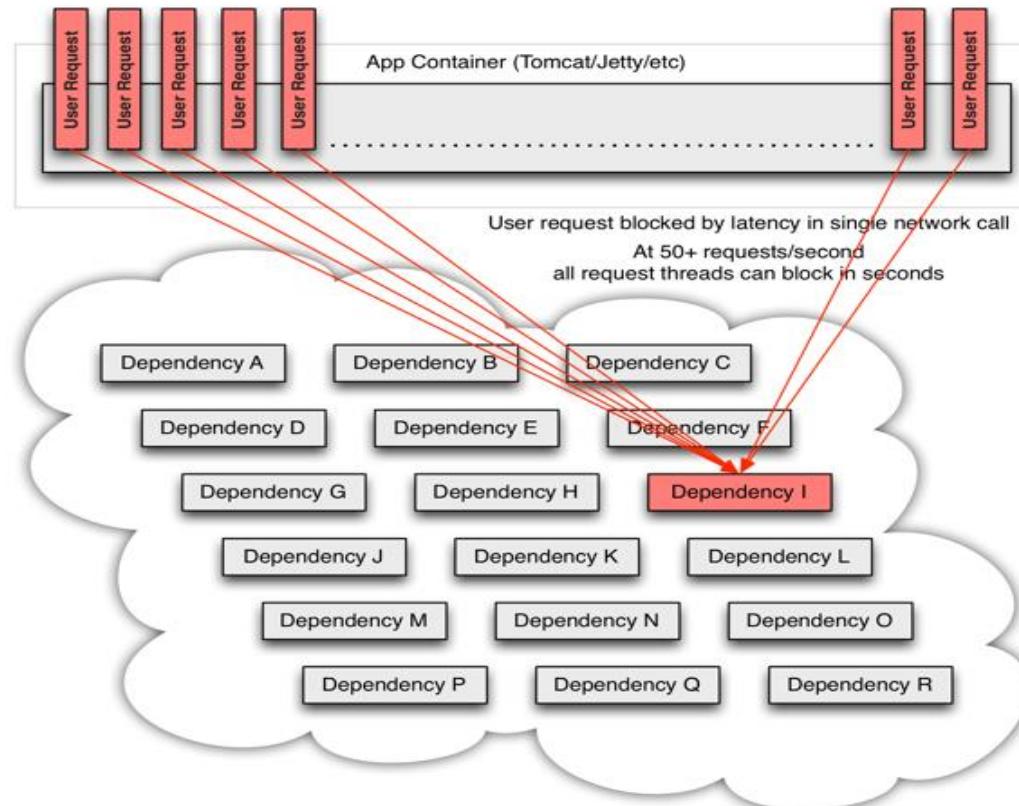
Externalize all application configuration including the database credentials and network location. On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.



Latency and Fault Tolerance for Distributed Systems

(Circuit Breaker Pattern – Hystrix / Resilience4j)

With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.



These issues are even worse when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.

All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

Pattern: Circuit Breaker

Problem

How to prevent a network or service failure from cascading to other services?

Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

CURRENT	REPLACEMENT
Hystrix	Resilience4j
Hystrix Dashboard / Turbine	Micrometer + Monitoring System
Ribbon	Spring Cloud Loadbalancer
Zuul 1	Spring Cloud Gateway
Archaius 1	Spring Boot external config + Spring Cloud Config

Resilience4j

Resilience4j has been inspired by Netflix Hystrix but is designed for Java 8 and functional programming.

It is lightweight compared to Hystrix as it has the Vavr library as its only dependency.

Vavr (Javaslang) core is a functional library for Java. It helps to reduce the amount of code and to increase the robustness.

A first step towards functional programming is to start thinking in immutable values.

Vavr provides immutable collections and the necessary functions and control structures to operate on these values.

Resilience4j

```
@Bulkhead(name = "bulkHeadPostToES")
public String postCallProxyES (String endpoint, String payload) {
    return this.initiateCallEsProxy (endpoint, payload);
}
```

```
bulkhead:
  backends:
    bulkHeadPostToES:
      maxWaitDuration: 20ms
      maxConcurrentCalls: 20

thread-pool-bulkhead:
  backends:
    bulkHeadPostToES:
      maxThreadPoolSize: 1
      coreThreadPoolSize: 1
      queueCapacity: 1
```

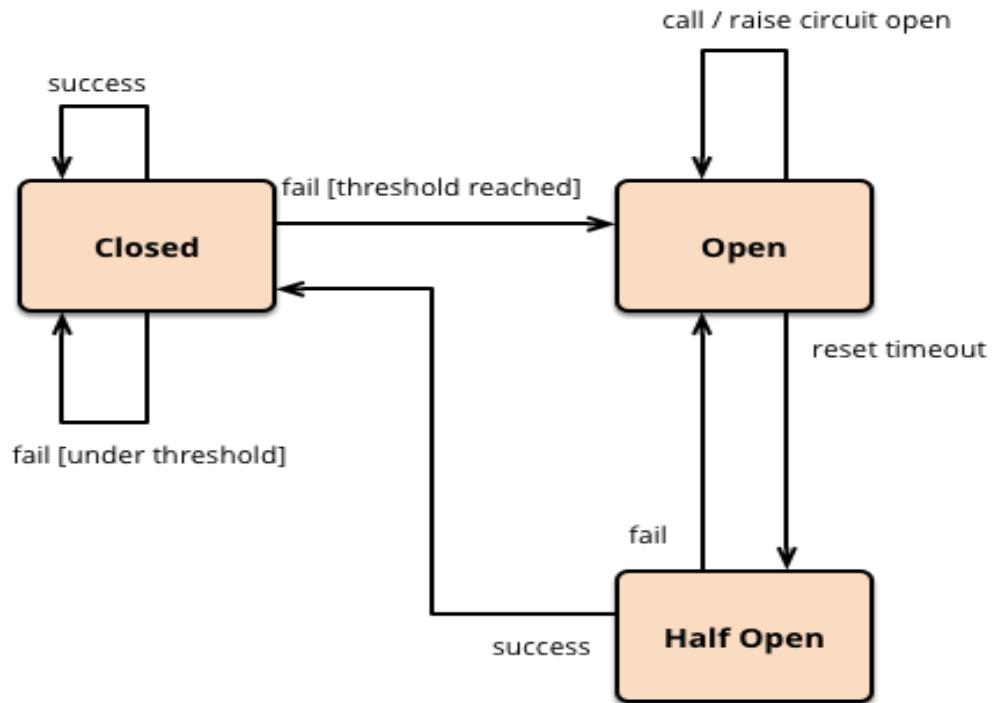
CircuitBreaker

When a service invokes another service, there is always a possibility that it may be down or having high latency.

This may lead to exhaustion of the threads as they might be waiting for other requests to complete.

This pattern functions in a similar fashion to an electrical Circuit Breaker:

- When a number of consecutive failures cross the defined threshold, the Circuit Breaker trips.
- For the duration of the timeout period, all requests invoking the remote service will fail immediately.
- After the timeout expires the Circuit Breaker allows a limited number of test requests to pass through.
- If those requests succeed the Circuit Breaker resumes normal operation.
- Otherwise, if there is a failure the timeout period begins again.



RateLimiter

Rate Limiting pattern ensures that a service accepts only a defined maximum number of requests during a window.

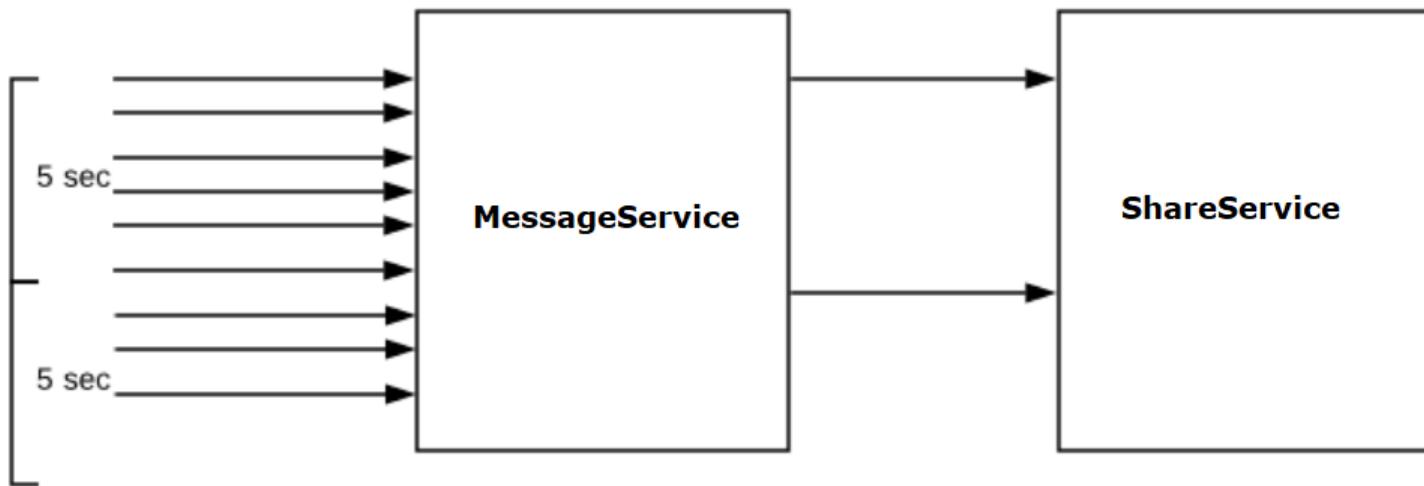
This ensures that underline resources are used as per their limits and don't exhaust.

Retry

Retry pattern enables an application to handle transient failures while calling to external services.

It ensures retrying operations on external resources a set number of times. If it doesn't succeed after all the retry attempts, it should fail and response should be handled gracefully by the application.

RateLimiter



Bulkhead

Bulkhead ensures the failure in one part of the system doesn't cause the whole system down. It controls the number of concurrent calls a component can take. This way, the number of resources waiting for the response from that component is limited.

There are two types of bulkhead implementation:

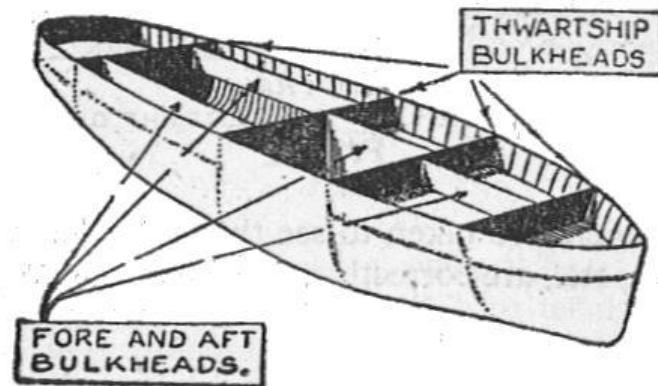
The semaphore isolation approach limits the number of concurrent requests to the service. It rejects requests immediately once the limit is hit.

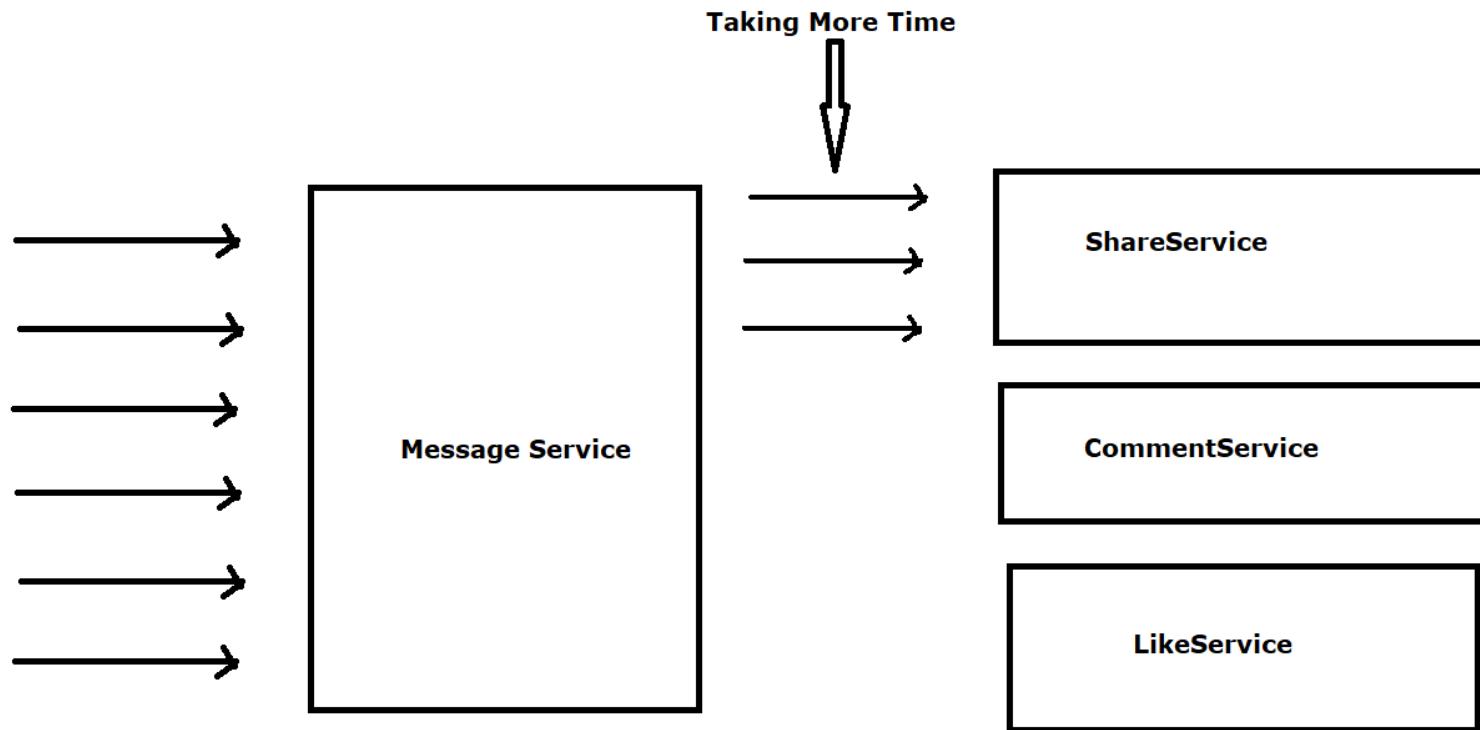
The thread pool isolation approach uses a thread pool to separate the service from the caller and contain it to a subset of system resources.

The thread pool approach also provides a waiting queue, rejecting requests only when both the pool and queue are full. Thread pool management adds some overhead, which slightly reduces performance compared to using a semaphore, but allows hanging threads to time out.

A ship is split into small multiple compartments using Bulkheads. Bulkheads are used to seal parts of the ship to prevent entire ship from sinking in case of flood.

Similarly failures should be expected when we design software. The application should be split into multiple components and resources should be isolated in such a way that failure of one component is not affecting the other.





CircuitBreaker

Config property	Default Value	Description
failureRateThreshold	50	Configures the failure rate threshold in percentage. When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallRateThreshold	100	Configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than <code>slowCallDurationThreshold</code> . When the percentage of slow calls is equal or greater the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallDurationThreshold	60000 [ms]	Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.
permittedNumberOfCallsInHalfOpenState	10	Configures the number of permitted calls when the CircuitBreaker is half open.

slidingWindowType	COUNT_BASED	<p>Configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.</p> <p>Sliding window can either be count-based or time-based.</p> <p>If the sliding window is COUNT_BASED, the last <code>slidingWindowSize</code> calls are recorded and aggregated.</p> <p>If the sliding window is TIME_BASED, the calls of the last <code>slidingWindowSize</code> seconds recorded and aggregated.</p>
slidingWindowSize	100	<p>Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.</p>

minimumNumberOfCalls	10	<p>Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate.</p> <p>For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated.</p> <p>If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed.</p>
waitDurationInOpenState	60000 [ms]	The time that the CircuitBreaker should wait before transitioning from open to half-open.

Create and configure a Bulkhead

You can provide a custom global BulkheadConfig. In order to create a custom global BulkheadConfig, you can use the BulkheadConfig builder. You can use the builder to configure the following properties.

Config property	Default value	Description
maxConcurrentCalls	25	Max amount of parallel executions allowed by the bulkhead
maxWaitDuration	0	Max amount of time a thread should be blocked for when attempting to enter a saturated bulkhead.

RateLimiter

Config property	Default value	Description
timeoutDuration	5 [s]	The default wait time a thread waits for a permission
limitRefreshPeriod	500 [ns]	The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
limitForPeriod	50	The number of permissions available during one limit refresh period

Retry

Config property	Default value	Description
maxAttempts	3	The maximum number of retry attempts
waitDuration	500 [ms]	A fixed wait duration between retry attempts
intervalFunction	numOfAttempts -> waitDuration	A function to modify the waiting interval after a failure. By default the wait duration remains constant.
retryOnResultPredicate	result -> false	Configures a Predicate which evaluates if a result should be retried. The Predicate must return true, if the result should be retried, otherwise it must return false.
retryOnExceptionPredicate	throwable -> true	Configures a Predicate which evaluates if an exception should be retried. The Predicate must return true, if the exception should be retried, otherwise it must return false.
retryExceptions	empty	Configures a list of error classes that are recorded as a failure and thus are retried.
ignoreExceptions	empty	Configures a list of error classes that are ignored and thus are not retried.

```
RetryConfig config = RetryConfig.custom()
    .maxAttempts(2)
    .waitDuration(Duration.ofMillis(1000))
    .retryOnResult(response -> response.getStatus() == 500)
    .retryOnException(e -> e instanceof WebServiceException)
    .retryExceptions(IOException.class, TimeoutException.class)
    .ignoreExceptions(BusinessException.class,
                      OtherBusinessException.class)
    .build();
```

resilience4j has the ability to add multiple fault tolerance features into one call. It is more configurable and amount of code needs to be written is less with right amount of abstractions.

```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
@RateLimiter(name = BACKEND)
@Bulkhead(name = BACKEND)
@Retry(name = BACKEND, fallbackMethod = "fallback")
@TimeLimiter(name = BACKEND)
Public String postCallProxyES(...) {
    ...
}
```

The default Resilience4j Aspects order is the following:

```
Retry ( CircuitBreaker ( RateLimiter ( TimeLimiter ( Bulkhead ( Function ) ) ) ) )
```

Bulkhead	Rate Limiter
Limit number of concurrent calls at a time.	Limit number total calls in given period of time
Ex: Allow 5 concurrent calls at a time	Ex: Allow 5 calls every 2 second.
In above example, first 5 calls will start processing in parallel while any further calls will keep waiting. As soon as one of those 5 in-process calls is finished, next waiting calls will be immediately eligible to be executed.	In above example, 2 second window starts & first 5 calls will start processing (may be parallel or not parallel). Those 5 calls might finish before 2 seconds, but next waiting calls will NOT be immediately eligible to be executed. After 2 seconds window is over, next 2 seconds window will start & then only next waiting calls will be eligible to be executed.

ONCOMPLETION

Camel has this concept of a Unit of Work that encompass the Exchange. The unit of work among others supports synchronization callbacks that are invoked when the Exchange is complete.

ONCOMPLETION WITH ROUTE SCOPE

```
from("direct:start")
    .onCompletion()
        // this route is only invoked when the original route is
        complete as a kind
        // of completion callback
        .to("log:sync")
        .to("mock:sync")
    // must use end to denote the end of the onCompletion route
    .end()
    // here the original route continues
    .process(new MyProcessor())
    .to("mock:result");

```

In xml

```
<onCompletion>
    <!-- so this is a kinda like an after
    completion callback -->
    <to uri="log:sync"/>
    <to uri="mock:sync"/>
</onCompletion>
```

ONCOMPLETION WITH GLOBAL SCOPE

This works just like the route scope except from the fact that they are defined globally. An example below:

```
// define a global on completion that is invoked when the
exchange is complete
onCompletion().to("log:global").to("mock:sync");

from("direct:start")
    .process(new MyProcessor())
    .to("mock:result");
```

USING ONCOMPLETION WITH ONWHEN PREDICATE

```
from("direct:start")
    .onCompletion().onWhen(body().contains("Hello"))
        // this route is only invoked when the original route is
        complete as a kind
            // of completion callback. And also only if the onWhen
            predicate is true
                .to("log:sync")
                .to("mock:sync")
        // must use end to denote the end of the onCompletion route
    .end()
```

USING ONCOMPLETION WITH OR WITHOUT THREAD POOL

```
onCompletion().parallelProcessing()  
    .to("mock:before")  
    .delay(1000)  
    .setBody(simple("OnComplete:${body}"));
```

We can also refer to a specific thread pool to be used, using the executorServiceRef option

```
<onCompletion executorServiceRef="myThreadPool">  
    <to uri="before"/>  
    <delay><constant>1000</constant></delay>  
    <setBody><simple>OnComplete:${body}</simple></setBody>  
</onCompletion>
```

USING ONCOMPLETION TO RUN BEFORE ROUTE CONSUMER SENDS BACK RESPONSE TO CALLEE

OnCompletion supports two modes

AfterConsumer - Default mode which runs after the consumer is done

BeforeConsumer - Runs before the consumer is done, and before the consumer writes back response to the callee

```
.onCompletion().modeBeforeConsumer()  
    .setHeader("createdBy", constant("Someone"))  
.end()
```