APACHE®

Camel

A routing-engine builder

What is Camel?

At the core of the Camel framework is a routing engine —or more precisely, a routing-engine builder. It allows you to define your own routing rules, decide from which sources to accept messages, and determine how to process and send those messages to other destinations. Camel forms the glue between disparate systems.

What is not Camel?

- Camel isn't an enterprise service bus (ESB), although some call Camel a lightweight ESB because of its support for routing, transformation, orchestration, monitoring, and so forth.
- Camel doesn't have a container or a reliable message bus, but it can be deployed in one, such as the previously mentioned Apache ServiceMix. For that reason, we prefer to call Camel an integration framework rather than an ESB.

Why use Camel?

These are the main ideas behind Camel:

- Routing and mediation engine
- Extensive component library
- Enterprise integration patterns (EIPs)
- Domain-specific language
- Payload-agnostic router
- POJO model
- Automatic type converters
- Lightweight core ideal for microservices
- Test kit

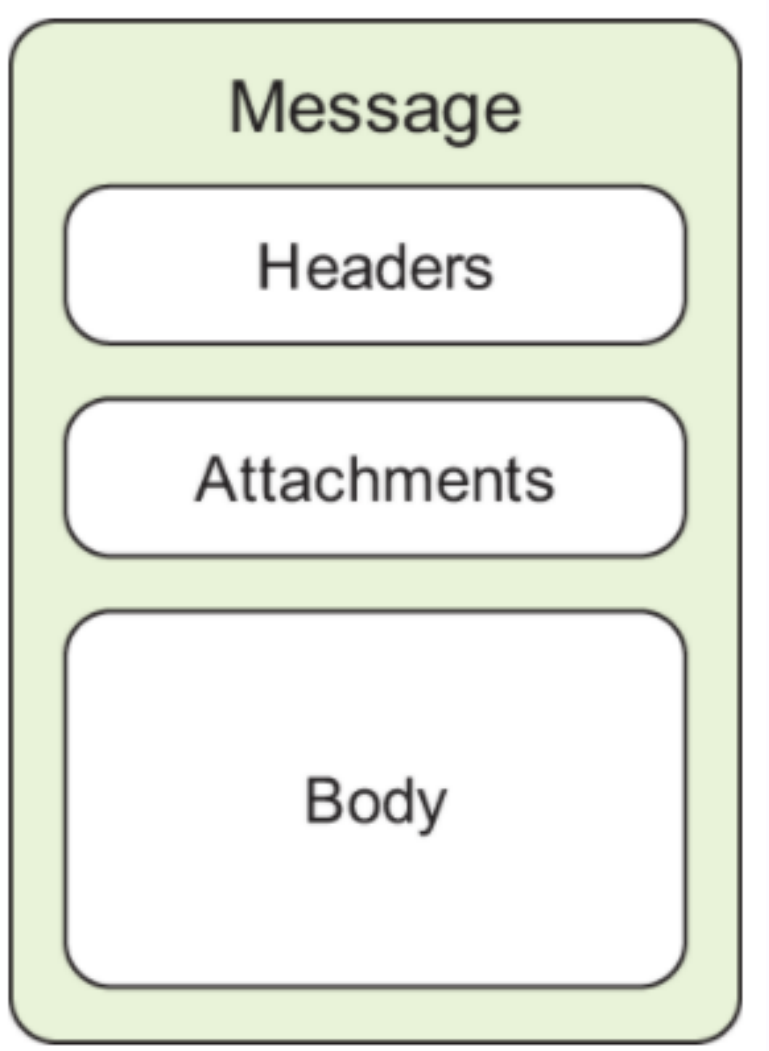
Camel's Message Model

Camel uses two abstractions for modeling messages:

- **org.apache.camel.Message**: The fundamental entity containing the data being carried and routed in Camel.
- **org.apache.camel.Exchange**: The Camel abstraction for an exchange of messages. This exchange of messages has an in message, and as a reply, an out message.

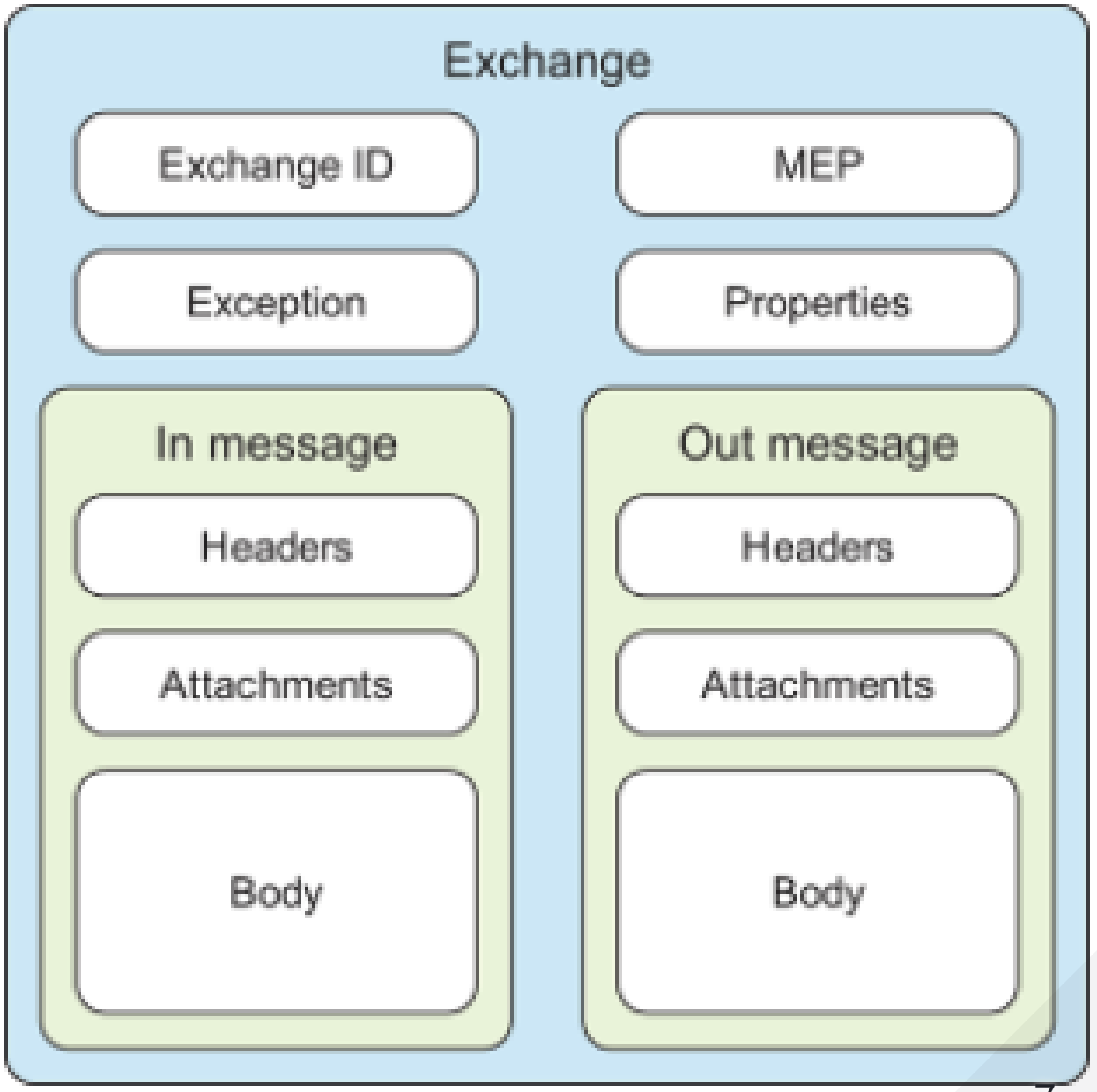
Message

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.

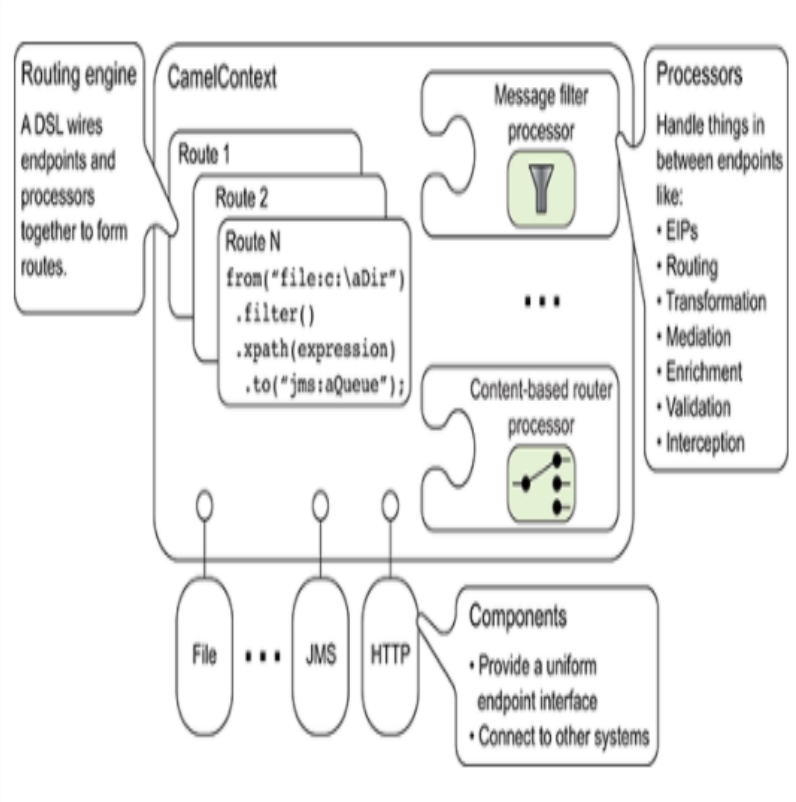


Exchange

An exchange in Camel is the message's container during routing. The exchange is the same for the entire lifecycle of routing, but the messages can change- e.g., if messages are transformed from one format to another.

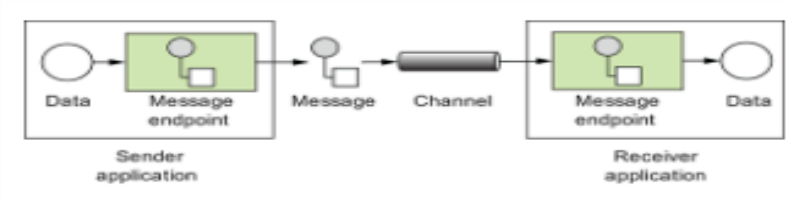


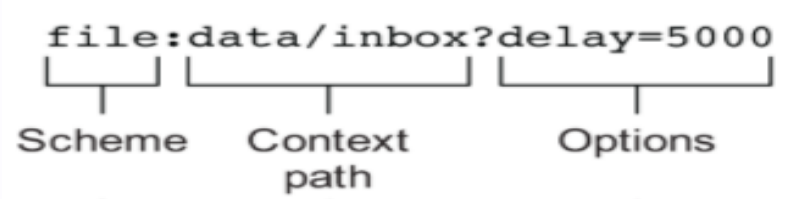
Camel's architecture



Endpoint

An endpoint is the Camel abstraction that models the end of a channel through which a system can send or receive messages.

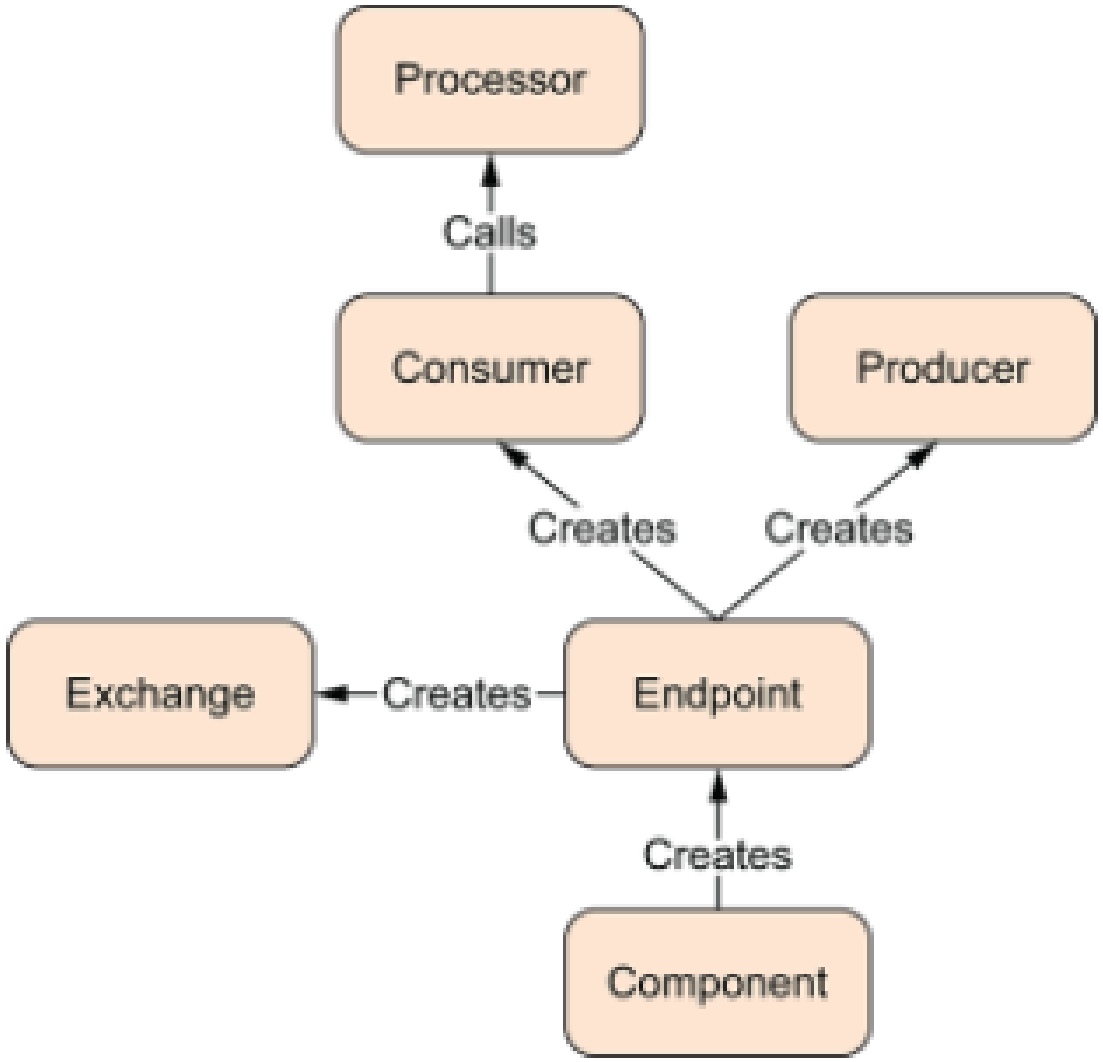




Scheme denotes a component that works like a factory,creating Endpoint based on the remaining parts of the URI.

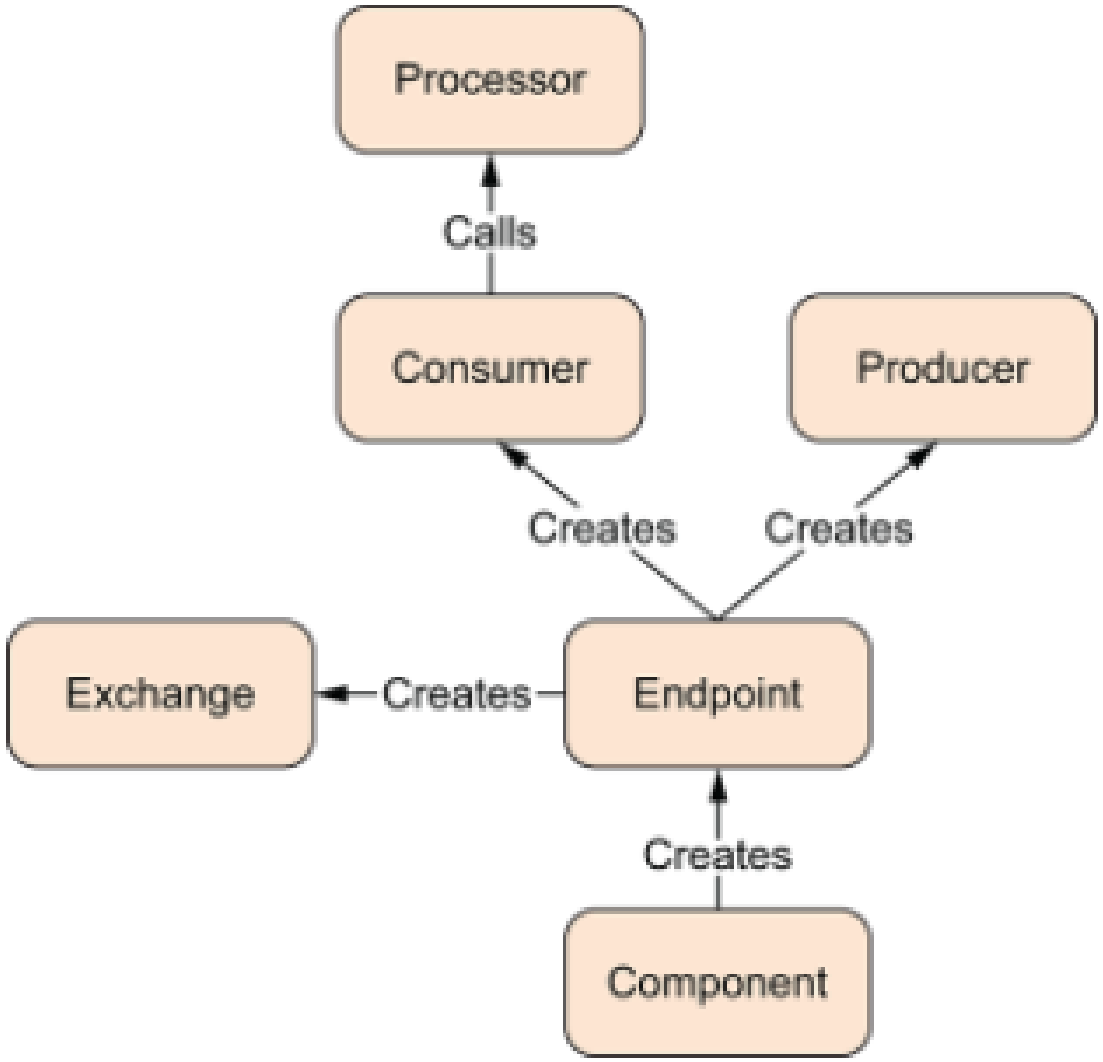
Producer

A producer is the Camel abstraction that refers to an entity capable of sending a message to an endpoint. The producer handles the details of getting the message data compatible with that particular endpoint.



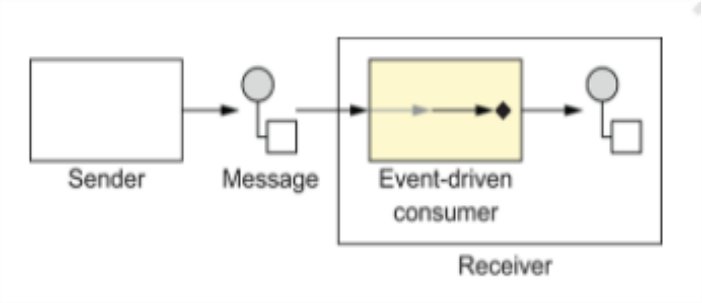
Consumer

A consumer is the service that receives messages produced by some external system, wraps them in an exchange, and sends them to be processed. Consumers are the source of the exchanges being routed in Camel.

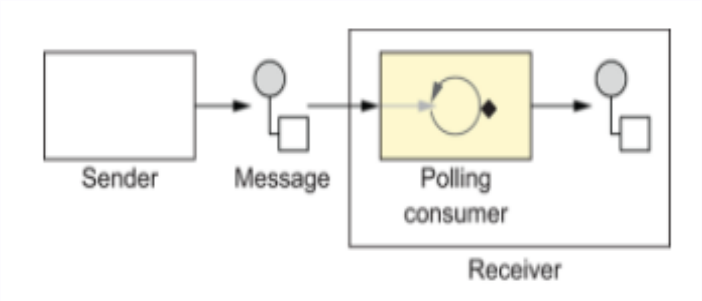


Kinds of consumers

Event-driven consumer



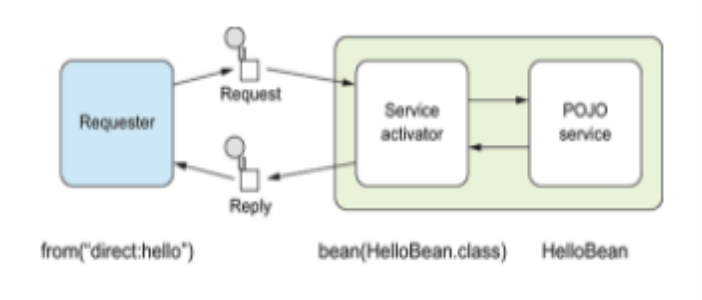
Polling consumer



Bean

Camel recognizes the power of the POJO programming model and goes to great lengths to work with your beans.

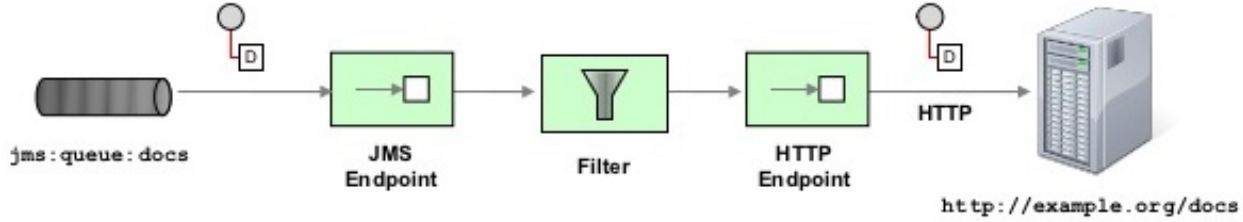
Service Activator pattern



Camel uses Bean EIP to do the work.

Routing with Camel

Routing is the process by which a message is taken from an input and, based on a set of conditions, sent to one of several outputs.

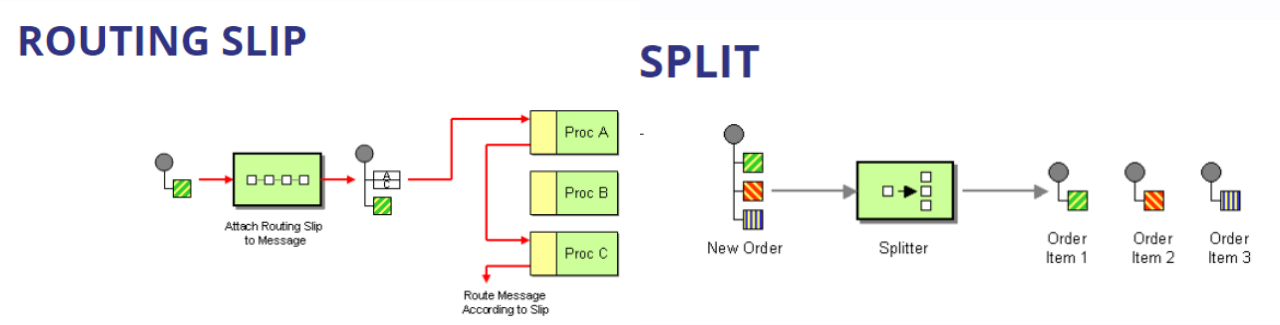


Java DSL:

```
from("jms:queue:docs")
  .filter().xpath("/person[@name='Jon']")
  .to("http:example.org/docs");
```

Enterprise Integration Patterns

Camel supports most of the Enterprise Integration Patterns from the excellent book by Gregor Hohpe and Bobby Woolf.



Routing with EIP I - Routing Slip

```
from("direct:start")
  // compute the routing slip at runtime using a bean
  .setHeader("mySlip").method(ComputeSlip.class)
  // use the routing slip EIP
  .routingSlip(header("mySlip"));
```

```
public class ComputeSlip {
    public String compute(String body) {
        String answer = "mock:a";
        if (body.contains("Cool")) {
            answer += ",mock:b";
        }
        return answer;
    }
}
```

Routing with EIP II - Split

```
    from("direct:start")
       .split().method(CustomerService.class, "splitDepartments")
         .to("mock:split");
```

```
public class CustomerService {
    public List<Department> splitDepartments(Customer customer) {
        return customer.getDepartments();
    }
    public static Customer createCustomer() {
        List<Department> departments = new ArrayList<Department>();
        departments.add(new Department(222, "Oceanview 66", "89210", "USA"));
        departments.add(new Department(333, "Lakeside 41", "22020", "USA"));
        Customer customer = new Customer(123, "Honda", departments);
        return customer;
    }
}
```

Other issues

- Clustering
- Testing
- Error handling
- Transactions and idempotency
- Parallel processing
- Securing
- Management and monitoring