Masarykova univerzita
Fakulta informatiky

RESTEasy integration with Apache Camel project

Master's thesis

Roman Jakubčo

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Roman Jakubčo

Advisor: Mgr. Marek Grác, PhD.

Acknowledgement

Abstract

Keywords

Contents

1 Introduction

2 Technologies

This chapter describes technologies that are used for implementation of Camel RESTEasy component. Each subsection introduces the technology and tries to explain what is its main functionality and common usage.

2.1 Apache Camel

Apache Camel is a open source rule-based routing and mediation framework implemented in Java. It is based on theory of Enterprise Integration Patterns or EIP, described in the book with same name written by Gregor Hohpe and Bobby Wolf[1].

Its main focus is on integration and interaction between various applications or systems for which Camel can provide standalone routing, transformation, monitoring and many other things. From this point of view Camel may seems like ESB[1], but this is not the case, because Camel doesn't provide a container support or reliable message bus, but it can be deployed into one and create full integration platforms(also know as ESB) like Apache ServiceMix[2], JBoss Fuse[3] and JBoss Fuse Service Works.

Camel also has extensible and modular architecture that allows implementation and seamlessly plug in support for new protocols and this architectural design makes Camel lightweight, fast and easy extendable for developers.[2]

2.1.1 The core principles/features of Camel

Camel is using convention over configuration approach to describe given task by domain-specific language (DSL) in declarative way. This way Camel minimize number of lines of the source code that is needed for implementation of integration scenarios. Another key feature that helps with this task is usage of theory of EIPs, which are already integrated in DSL and also getting the most from their potential.

1. ESB - Enterprise Service Bus
2. `http://servicemix.apache.org/`
3. `http://www.jboss.org/products/fuse/overview/`

Another fundamental principle of Camel is that it makes no assumptions about the data format. This feature is important because it makes possible for developers to integrate systems together without any need to convert data to some canonical format. This way there are no limitations for integration of any kind of systems together[2].

Routing and mediation engine

One of the core features of Camel is its routing and mediation engine. A routing engine selectively moves a data from one destination to another based on the route's configuration. Users also can define their own rules for routing, add processors to modify the data, filter them based on some predicate and at the end decide the final destination for the delivery.

Enterprise integration patterns

Like it is mentioned before Camel is based primarily on EIPs. EIPs describe integration problems and their solutions and also provide some basic vocabulary but the problem is this vocabulary isn't formalized. And into this comes Camel with its language which describes the integration solutions and tries to formalized the vocabulary. There's almost a one-to-one relationship between the patterns described in Enterprise Integration Patterns and the Camel DSL[4]. Almost all of the EIPs that are defined in the book are implemented as Processors or sets of Processors in the Camel. Processors are used for manipulation of messages between destinations specified in the Camel route.[3]

Domain-specific language

There are few other integration frameworks with DSL and also some have support for describing route rules in XML, but bonus that comes with Camel is its the support for specifying DSLs in regular programming languages as Java, Groovy, Ruby and even Scala. Of course there is also possibility to describe the route in XML document.

4. `http://camel.apache.org/enterprise-integration-patterns.html`

Example 2.1: Java DSL definition of route

```
from("undertow://localhost:8888/myapp")
    .to("file:log/access.log");
```

Example 2.2: XML definition of route

```
<route>
   <from uri='undertow://localhost:8888/myapp'/>
   <to uri='file:log/access.log'/>
</route>
```

Example 2.3: Scala definition of route

```
from "undertow://localhost:8888/" -> "file:log/access.log"
```

Modular and pluggable architecture

The next feature is the approach to the architecture, which is done in modular way. This means, that Camel can be easily extended to consume data from endpoint and produce data to endpoint. Camel is describing this as developing a new component, where each component is responsible for consuming or producing data for some specific endpoint and technology e.g. file, HTTP and many others. When developers want to develop a new component for some unique system and add its functionality to the Camel, they just need to follow structure specified by the framework and extend core classes.

By default Camel ships with the few most basic components called camel-core. This bundle includes 24 components including components like bean, file, log, seda and mock. Plus there are many more components developed by the Apache community and also third-parties[5]. There are already developed components that can be used for the most common integration scenarios that occur in systems. Some components worthy

5. `http://camel.apache.org/components.html`

of note are web services including SOAP[6] or REST[7], JMS[8], specialized JMS component for Apache ActiveMQ[9] or components for different database connections.

Configuration

As mentioned before Camel uses convention over configuration paradigm to minimize configuration requirements so that developers don't need to learn complicated configuration options and focus on more important things. This is reflected on configuration of endpoints in route definitions with URI options as can be seen on example 2.4.

Example 2.4: URI options configurations

```
//pattern to follow
"undertow://{host}:{port}/{path}?[{uriOptions}]"

"undertow://localhost:8080/foo?matchOnUriPrefix=true"
```

Type converters

Another feature of Camel which is one of the top features for Camel Community and that are build-in automatic converters. Out of the box Camel ships with more than hundred and fifty converters[2]. Plus if there is no converter for your types, there is possibility to create new custom converters for your specific types. Usage of the converter can be seen on example 2.5 and the example also demonstrates how is the converter used by Camel without user's knowledge in *getBody* method and its parameter.

Lightweight framework

From the start the whole framework was designed to be undemanding and lightweight as possible. The core library has only about 1.6 MB and

6. SOAP
7. REST
8. JMS
9. `http://camel.apache.org/activemq.html`

Example 2.5: TypeConverter invocation

```
//direct use of TypeConverter
TypeConverter tc = consumer.getEndpoint()
            .getCamelContext().getTypeConverter();
ByteBuffer bodyAsByteBuffer =
            tc.convertTo(ByteBuffer.class, body);

//automatic trigger under the hood
ByteBuffer bodyAsBuffer =
            message.getBody(ByteBuffer.class);
```

third parties dependencies are kept at minimum. This way Camel can be easily embedded into any platform, which can be e.g. OSGi bundle, Spring application, Java EE application or web application.

2.1.2 Message model

Until now we talk about sending data from one endpoint to another. That is exactly what is Camel doing but in reality it is sending and receiving messages which encapsulate the data. There two abstractions classes that are use for modelling messages in Camel and these are:

- `org.apache.camel.Message` – the basic entity containing data that is routed in the Camel
- `org.apache.camel.Exchange` – special abstraction used in Camel for exchange of messages, that has *in* message and *out* message as a reply

Message

Message object is representing data that is used by systems to communicate with each other. Messages are sent in one direction from a sender to a receiver. The message object consists of body, headers and optional attachments. All messages must be uniquely identified with an unique identifier(UID). The format of UID is not guaranteed and it is dependent on the used protocol. If the protocol doesn't have UID scheme, then generic generator from the framework is used.

Headers are name-value pairs associated with the message, similar to HTTP protocol. They provide additional information about the message such as sender identifiers, encoding, content type or authentication parameters. They are stored in a map within the message and each name of the header is unique case insensitive string and the value can be any Java object.

Body is representing content of the message and its type is generic Java Object, so message can store any kind or type of content. The sender should send body type acceptable by the receiver. If this is not the case then manual transformation inside the route is needed or more conveniently type converters are automatically used by the Camel.

Exchange

An *Exchange* is defined as message's container encapsulating the *Message* used during routing. There are various types of interactions between systems and they are supported by the *Exchange*. These interactions are called message exchange patterns (MEPs) and they are used to specify messaging styles in the property of the *Exchange*. This property has to two different messaging styles where one is one-way and the other one is request-response.

The request-response or *InOut* called in the Camel, is probably the more well-known style because it is used in HTTP-based transport, where client requests to retrieve a web page and it is waiting for the reply from the server. One-way is defined as *InOnly* and for example is primarily use in JMS, where message is sent to the queue and sender doesn't need any response from the queue. These two types are just the basic ones, but Camel provides few more special cases[10].[4]

The *Exchange* is little bit more complex than the *Message* and it is consisting from[2]:

- Exchange ID – is unique ID that identifies the exchange and it is also automatically generated by the Camel if ID is not explicitly set.
- MEP – defines type of messaging style.

10. Exchange patterns in Camel`http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/ExchangePattern.html`

- Exception – If an error occurred during routing, then *Exception* is set into this field.
- Properties – various properties of the *Exchange* used by Camel, similar to headers in *Message*, but with difference that the properties last for the duration of entire exchange and contain global information, whereas message header are specific to the *Message*. They can be also edited by developers.
- In message – mandatory input message containing request message.
- Out message – optional message containing reply message if the MEP is set to *InOut*.

2.1.3 Camel's architecture

2.2 Development of the new Camel component

2.2.1 Camel context

References

[1] HOHPE, Gregor and WOLF, Bobby. *Enterprise integration patterns.* Boston: Addison-Wesley, c2003, li, ISBN 978-0321200686.

[2] IBSEN, Claus and ANSTEY,Jonathan. *Camel in Action.* Greenwich, Conn.: Manning, c2011, xxxi, ISBN 19-351-8236-6.

[3] http://java.dzone.com/articles/open-source-integration-apache

[4] http://camel.apache.org/exchange-pattern.html

[5] APACHE. *Apache Camel* [online]. 2004- [cite 2014-12-12]. Available at: `http://camel.apache.org/`

[6] ORACLE. *Java* [online]. © 2004- [cite 2014-12-12]. Available at: `http://www.java.com/`

[7] CRANTON, Scott and KORAB, Jakub. *Apache Camel Developer's Cookbook.* Birmingham: Packt publishing, c2013, ISBN 9781782170303.

A Appendix