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

Apache Camel, RESTEasy, JAX-RS, REST, RESTful, integration framework, component, web service

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("resteasy:/say/hello?servletName=restServlet")
    .to("file:log/response.log");
```

Example 2.2: XML definition of route

```
<route>
   <from uri='resteasy:/say/hello?servletName=restServlet'/>
   <to uri='file:log/response.log'/>
</route>
```

Example 2.3: Scala definition of route

```
from "resteasy:/say/hello?servletName=restServlet"
        -> "file:log/response.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[10] options as can be seen on example 2.4.

Example 2.4: URI options configurations

```
// consumer pattern to follow
"resteasy:/{path}?[{uriOptions}]"

"resteasy:/say/foo?servletName=restServlet&proxy=true"

// producer pattern to follow
"resteasy:{protocol}://{host}:{port}/{path}?[{uriOptions}]"

"resteasy:http://localhost:8080/foo?resteasyMethod=POST"
```

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

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

converter used by Camel without user's knowledge in *getBody* method and its parameter.

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);
```

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 third parties dependencies are kept at minimum. This way Camel can be easily embedded into any platform, which can be e.g. OSGi[11] 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

11. OSGi

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[12].[4]

12. http://tinyurl.com/exchangePattern

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.
- 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

This chapter will describe architecture of the Camel from high-level and then take closer look on some specific concepts. The basic architecture of the Camel is shown in the figure 2.1 and should help with understanding of the runtime of *CamelContext*.

Camel context

From the figure 2.1 it is clear that CamelContext is somewhat similar to container but more precisely it is Camel's runtime system. This system keeps everything together and provides services during the runtime that are:

- Routes – routes that have been added to the context
- Endpoints –endpoints that have been created in context

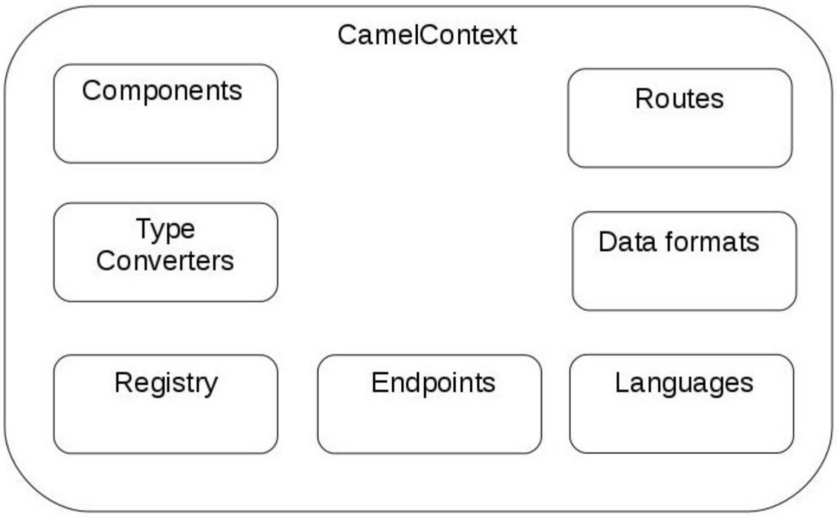

Figure 2.1: Overview of CamelContext

- Components – components used by the application and they can be added on the fly
- Type converters – loaded type converters by the context
- Data formats – data formats loaded into to the context
- Languages – Camel supports different languages used in the expressions and these are loaded into the context
- Registry – registry is used for the look up of beans. JNDI[13] registry is used by default, but if Camel is deployed into Spring or OSGi container then it uses native registry mechanism specific to these technologies.

Routing engine and routes

The routing engine is the thing that actually moves messages, but is not visible to users. It guarantees correct routing from the sender to the receiver.

13. JNDI

Routing engine is using routes for its routing. Routes are the essence of the Camel and are the main and only approach how to specify what should be moved where for the routing engine. This means that the route must hold definition of input source to output target. There many possibilities for definition of the route, but the simplest way is to define route as a chain of processors[2]. Similar to Message or Exchange, each route has unique identifier that is used for different operation inside of Camel like monitoring, starting or stopping the route. One of the constraint for the route is restriction for the number of input sources, where each route must have exactly one input source that is tied to input endpoint and there can be one or several output targets.

Processor

As mentioned before the simplest route consists from the chain of processors. The processor represents a node responsible for using, creating or modifying content of incoming *Exchange* and also its headers. In the chain of processors, exchanges moves during routing from one processor to a another, in the their order defined in the route. This way a route can be seen as graph with nodes where output from the one node is input of another.

As stated previously, almost all built-in processors or their combinations are implementation of EIPs, but Camel also supports implementation of own custom processors and their easy addition to the route.

Component

A components help with modular approach and they are main extension point in Camel. Their main purpose and task is to be a factory of endpoints. As mentioned before developers can create their new components, more information and detail on this topic will be given in subchapter 2.2.

Endpoint

An endpoint is a abstraction in the Camel that models the the end of message channel. In other words it represents sender or receiver. Endpoints are configured and referred in route using URIs and Camel

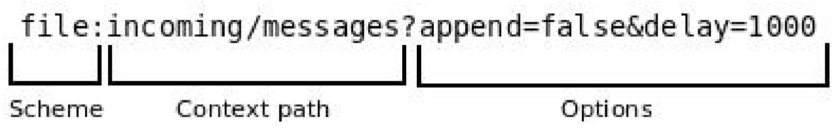

Figure 2.2: Overview of endpoint URI

also looks up a endpoint during runtime by its defined URI. Format of the URI is show on figure 2.2 and it is consisting from three parts: scheme, context path and options.

The scheme specifies which component should be used. In the figure 2.2 is used scheme textitfile that represents Camel *FileComponent* that creates *FileEndpoint*. The component developed in this thesis uses scheme called *resteasy*. The context path describes the location of specific resource for the endpoint, similar to a web page on the server. The last part of the URI is options that is used for specific configuration of the endpoint.

The last task of endpoint is to be factory for creating producers(sender) and consumers(receiver) that are capable of receiving and sending messages in routes.

Producer

A producer is a entity capable of creating and sending a message to an endpoint. Its main task is creating a exchange and populating it with content compatible with the endpoint specification. For example, *JmsProducer* will map Camel message to JMS message before sending it to JMS destination. The producer developed in this thesis acts as HTTP client sending HTTP requests using client API from RESTEasy.

Consumer

As stated before, every route has just one starting point and that is a consumer. The consumer is something like receiver of messages, where the message is sent and its task is to wrap the message into *exchange*, add headers. Exchanges created by consumer are then send and routed in defined chain of processors.

Camel defines two types of consumers: event-driven and polling consumers. The event-driven consumer is probably more famous and known because it is associated with client-server architecture and its communication. In EIPs is this consumer referred also as *asynchronous* receiver and its job is to listen on messaging channel, waiting for the incoming messages.

A polling consumer is working little bit different than event-driven consumer. It is active consumer that goes to defined address in endpoint and fetches messages from it. Similar to event-driven consumer, polling consumer has different name in EIP world and it is commonly called *synchronous* receiver. This means that all received message has to be processed before the consumer polls for another one. Common usage of polling consumer in Camel is scheduled polling consumer that checks and polls messages in defined time interval.

2.2 Development of the new component

As already stated, developers can exploit Camel modular architecture and easily extend Camel and add new protocols without necessity to change core of the framework. This is feature is achieved by Camel components and by developing new custom component for new protocol. This section describes some of the fundamental principles associated with creation of the custom Camel component.

Camel is built using Apache Maven[14] so the easiest and fastest way to create a custom component from a scratch is to use Maven archetype[15]. Camel offers several archetypes[16] for different tasks and projects. Archetype *camel-archetype-component* is used for creating a maven project, that is a base for developing a new custom component[5].

The created project is fully functional *HelloWorld* demo component containing consumer that generates messages and producer for printing them to the console. Modifying this example is great for creating a custom component. The first thing to do is to decide what name will be use in endpoints for referencing the component. This name must be unique so it doesn't create conflict with other components. List of the

14. `https://maven.apache.org/`
15. `https://maven.apache.org/guides/introduction/introduction-to-archetypes.html`
16. `http://camel.apache.org/camel-maven-archetypes.html`

existing Camel components can be found on official web pages[17]

2.2.1 Hierarchy of classes

Camel component consists from four main classes that together create component and that is Component, Endpoint, Producer and Consumer. Of course component can have many more classes used in the component and its correct functionality, but only these 4 are important for creation of correct component in Camel. As stated before their relationship is that everything starts with Component class, which creates an Endpoint. An endpoint then creates Producers and Consumers.

Once again Camel offers some default classes for each of these classes, that can be extended for easier development and not everything needs to be created from the scratch.

Component class

Main job of this class is to be a factory of endpoints. This class needs to implement *Component* interface and primarily implement its *createEndpoint()* method. The easiest way to achieve this is to extend *DefaultComponent* class. Of course there is also possibility to extend other component classes from other components and leverage theirs functionality.

Endpoint class

Main task of endpoints is to be factory for Consumers and Producers. Endpoint class needs to implement Endpoint interface that has creation methods for both of them. The simplest way is to extend DefaultEndpoint. Also not all components has to have both Producer and Consumer. It is common that in some components one of them is not needed or doesn't make sense in regards to used technology. This class can also contains all parameters that can be set as URI options on the endpoint. Parameters are usually annotated with @UriParams annotation and set though reflection by Camel.

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

Consumer and Producer class

As already stated, Consumer is starting point though which messages enter the route. Camel of course offers some default implementation for event-driven and polling consumers. This class has no major restrictions how should be implemented, it just needs to implement *Consumer* interface with its method *getEndpoint()*. In the component developed in this thesis, *ResteasyConsumer* connects to running web servlet and it is consuming requests and responses from the RESTEasy web service.

Producer class is responsible for sending messages outside. Similar to Consumer class there are no given restrictions on the implementation, it just needs to implement *Producer* interface, which extends from the *Processor* interface that has only one method *process()*. But it is recommended to extend *DefaultProducer* class to keep it simple. *ResteasyProducer* acts as HTTP client that sends HTTP requests to specified target.

2.3 REST

For most of people in modern world, World Wide Web is almost fundamental part of their life and they take it for granted. It is a given fact that Web has been very successful and it has grown from simple network for researchers and academics to interconnected worldwide community.

Roy Fielding tried to understand this phenomenon and find out what was the reason or factor for this incredible change in his PhD thesis, Architectural Styles and the Design of Network-based Software Architectures[7]. In it, he asks three important questions connected with the Web:

- Why is the Web so prevalent and ubiquitous?
- What makes the Web scale?
- How can I apply the architecture of the Web to my own applications?

From answers to these questions, he identifies five specific architecture principles called Representation State Transfer (REST) and these principles are:

- Addressable resources – resource is REST is abstraction of information and data and it must be addressable via URI.
- A uniform, constrained interface – in your application use only small set of well-defined methods that manipulate with resources.
- Representation-oriented – a resource referenced by one URI can have many different formats, similar to different platforms that need different formats, e.g. HTML[18] for web browsers or JavaScript needs JSON[19]. REST application should interact with services using representations of that service.
- Communicate statelessly – stateless applications can be scale more easily.
- HATEOAS[20] – data formats should drive state transitions in the application

These principles are the reasons, why Web became so successful, enormous and pervasive.[6]

2.3.1 REST over HTTP

REST architecture isn't protocol-specific, but it is usually associated with HTTP protocol. The HTTP protocol is primarily used in browser-based web applications, but these application don't fully leverage all features from it. There are also others web technologies like SOAP[21] that uses HTTP protocol only for transmission and use only small fraction of its capabilities. Because of this, it may seems like HTTP protocol is not very useful and it has only small set of features.

In reality, HTTP is very rich and powerful synchronous request/response-based application network protocol with many useful and interesting capabilities for developers. It is used for distributed, collaborative, document-based systems. The protocols works in simple way, the client

18. HyperText Markup Language
19. JavaScript Object Notation
20. Hypermedia as The Engine of Application State
21. Simple Object Access protocol

sends request message with defined HTTP method to be invoked, headers, location of resource for invocation and it can also contain message body that can almost anything.

The server that handled the requests message, will send response message with response code, message explaining the code, headers and optional message body. HTTP defines several response codes for different scenarios[22].[6]

2.3.2 RESTful webservices

As stated previously, REST was really just explanation for Web's success and growth, but after few years after publication of Fielding's PhD thesis, developers realized real potential of REST. They realized that concepts described in REST architecture can be used for building distributed services and modelling SOAs[23].

SOA is a design pattern that is used for a long time. The simplest description of main concept and idea of SOA is that systems should be designed as set of small reusable, decoupled and distributed services. By combining these services and publishing them on the network, should create larger and more complex systems. There were several technologies used for building SOAs in the past like CORBA or Java RMI. Nowadays, most associated technology with SOA are SOAP-based web services[24].

2.3.3 RESTful architectural principles

Web service build upon principles of REST architecture are called RESTful web services. These services are used for building system based on SOA principles. This section will describe each of the architectural principles of REST in more detail and explain why these principles are important for writing a web service.

Addressability

Addressability in the systems means that every resource is reachable by unique identifier. For this standardized object identity in environment is

22. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`
23. Service-Oriented Architectures
24. Simple Object Access Protocol

needed, which is not so common in environments. In RESTful services, addressability is achieved by use of URIs and each HTTP request must contain URI of the object that is requested. The format of URI must be of course standardized and it shown on example 2.6.

Example 2.6: URI format

```
// URI format
"{scheme}://{host}:{port}/{path}?{query}#{fragment}"

// example of actual URI
"http://restexample.com/customers?firstName=Roman&age=25"
```

The format URI is break down to six parts, where first part is *scheme* that specifies protocol used for communication, in RESTful web is usually *http* or *https*. Next ones are *host* that contains DNS[25] name or IP address and optional *port*. These two parts represent location of resource on the network. After them is *path* part that specifies path to desired resource, similar to directory list of a file on file system. Last two parts are optional where *query* is list of parameters represented as name-value pairs delimited by "&" character and *fragment* is usually used to point in certain place in queried document.

A uniform, constrained interface

To fulfil this principle of REST, application must only use finite set of operations of the application protocol on which are its services distributed. When developing RESTful web service this means only use methods of HTTP protocol. There are only few operational methods defined on HTTP where each method has specific purpose and these methods are:

- *GET* – read-only operation used for querying server for specific information. This operation is idempotent, which means that applying this operation many times, the result is always the

25. Domain Name System

same. It also safe in meaning that it doesn't change state of the server.

- *PUT* – operation used for storing message body on the server, usually modelled as insert or update. It is also idempotent, because sending the same message more than once has no effect on server.
- *DELETE* – the name is self explanatory, idempotent operation for removing resources.
- *POST* – operation for modifying (updating) service. It is only nonidempotent and unsafe operation of HTTP. This means that request may or may not contain information and also response may or may not contain information.
- *HEAD* – similar to *GET* except no body is returned, only headers and response code.
- *OPTIONS* – operation for getting information about communication options. Mainly used for getting information on capabilities of the server and a resource without triggering any action.
- *TRACE* and *CONNECT* – unimportant operations without any use in RESTful web service.

Plus constraining interface of web service has few more advantages. One them is *interoperability*. HTTP is ubiquitous protocol and almost all modern programming languages have HTTP client library. So, it makes sense to expose web service over HTTP, because people will able to use the exposed web service without any additional requirements. Older technologies have vendor specific client libraries and generated stub codes like *WSDL* files, that creates problem with vendor interoperability in the applications. RESTful web services don't have this problems and developers can focus on more important things in the application.

A constrained interface with its well-defined methods have predictable behaviour that can be leverage for for better performance. This means RESTful web services are better in scaling, which is another advantage and that is *scalability*. RESTful web services are taking advantage of caching mechanism of HTTP protocol. HTTP has very rich configuration for caching semantics that can be used for better performance. So, it is possible to configure caching semantics on defined method and make the same calls on same clients load much faster, because most of the information were cached on the first call. This is use of the same principle as with web pages and browsers.

Representation-oriented

Complexity of interaction between client and server in RESTful application is in representations being sent back and forth. Representations can be for almost in any format used in systems, for example XML, JSON, stream, YAML[26]. Because RESTful web services are communicating via HTTP, messages bodies of requests and responses are acting as representations.

HTTP provides *Content-Type* header for specification of data formats use between client and server. Value of this header is string in MIME[27] format, which is very simple: *type/subtype;name=value;name=value*. Where *type* is the main format family and *subtype* is category. Plus there is possibility to define name/value properties. Some of the common examples are :

- *text/plain*
- *application/xml*
- *text/html;charset=iso-8859-1*

Another feature of HTTP is possibility for negotiation of the message formats sent between client and server, which very useful in web services. There is possibility to define *Accept* header on client that describes preferred formats of responses. This can be used for definition of services on the same URIs and same methods but with different return MIME type. Using this, application can have similar methods but each for different client with its preferred data format.

26. YAML Ain't Markup Language
27. Multipurpose Internet Mail Extension

Stateless communication and HATEOAS

Next mentioned principle of RESTful web service is its statelessness, but that mean the application cannot have a state. Stateless in RESTful means that server doesn't store any client session data and instead only manages states of resources it exposes. Session specific data, if they are needed, must be maintained by client and can be send with request if needed. This feature can be leverage for easier scaling in clustered environments because only machines need to be added for scale up.

The last principle of RESTful web service is HATEOS where the main idea is using Hypermedia As The Engine of Application State (HATEOAS). This approach is very useful because hypermedia have added support for embedding links to other services or information within document format. One of the uses is for example aggregating complex sets of information from different sources and using hyperlinks in document to reference additional information without bloating responses. But the "engine" part is much more useful because it is different approach from traditional and older distributed applications that have list of services they know exist. These applications then call central server for location of these services. Instead RESTful web services with HATEOAS with each response returned from a server define new possible interactions that can be done next, as well transition state of application.

2.4 JBoss RESTEasy

JBoss RESTEasy is project providing frameworks for building RESTful Web Services and RESTful Java applications. This means that it is a fully certified and portable implementation of the JAX-RS specification which can run in any servlet container. It is also no surprise that it is fully integrated with JBoss Application Server to make better experience in that environment. To understand better what exactly is JBoss RESTEasy, it is required to understand what exactly is JAX-RS.[10]

2.4.1 JAX-RS 1.0 and 1.1

RESTful services could be developed in Java for a long time using servlet API, but this approach is really hard and requires lot of code for simple operations. To simplify implementation of RESTful service, a new specification was defined in 2008 called JAX-RS. JAX-RS is a new JCP specification that provides a Java API for RESTful Web Services over the HTTP protocol. This section will just provide some basic information about the this specification. The more detailed description and documentation can be found on official web page of JAX-RS[28] or on RESTEasy web page[29].

JAX-RS is a framework that focus on applying Java annotations introduced in Java SE 5 on plain Java objects. JAX-RS API is part of JSR-311[30]. To simplify development of RESTful web services, this framework has annotations to bind specific URI patterns and HTTP methods to individual method in basic Java class. It has also parameter injection annotations for easier parsing of information from HTTP requests. Another feature is message body readers and writers for decoupling data format marshalling and unmarshalling from custom Java objects. Also it has exception mappers for mapping application-thrown exceptions to HTTP response code and message. The last useful feature that it provides are facilities for HTTP content negotiation.[6][11]

JAX-RS 1.1 is upgrade version of the same framework with few changes. The biggest one is that is became official part of Java EE 6, which means that no configuration is necessary to start using JAX-RS. Some of the enhancements to framework include adding examples to clarifying and correcting Javadoc comments related to use of JAX-RS. There was also added new annotation *@ApplicationPath*, which can be use to specify base URI for all *@Path* annotations. Complete list of all changes can be found in official changelog[31]. [8][9]

It is important to note that both JAX-RS 1.0 and 1.1 are only server-side specifications without any support for client. Because of that, RESTEasy innovated JAX-RS and added RESTEasy JAX-RS Client framework and more about this technology in subsection **??**. But this drawback was removed in version 2.0 and will be described to more detail in next subsection.

28. `https://jax-rs-spec.java.net/`
29. `http://resteasy.jboss.org/`
30. `https://jcp.org/en/jsr/detail?id=311`
31. `https://jcp.org/aboutJava/communityprocess/maintenance/jsr311/311changelog.1.1.html`

2.4.2 JAX-RS 2.0

With Java EE 7 release also came new version of JAX-RS 2.0 that upgraded the framework and added new features from which the key features are:

- Client API
- Server-side asynchronous HTTP
- Filters and interceptors

This subsection provides basic overview of mentioned features. Of course there are other minor features and upgrades to make the framework more useful and stable.

Client API

Both previous versions of JAX-RS were missing client API, as it was stated previously. Because of this missing feature, each implementation of JAX-RS created their own client API, which is not so great for standardized framework. So version 2.0 contains fluent, low-level, request building API that can be seen on example 2.7.

Example 2.7: Client API

```
Client client = ClientFactory.newClient();

WebTarget target = client.target("http://resteasy.com/books");

Form form = new Form().param("author", "Claus Ibsen")
.param("name","Camel in Action");

Response response = target.request().post(Entity.form(form));
Order order = response.readEntity(Order.class);

// direct get of Java object
Car car = client.target("http://resteasy.com/cars").
queryParam("brand", "Ferrari").request().get(Car.class);
```

The base of the API is *Client* interface that manages HTTP connections and acts also as factory for *WebTargets*. *WebTarget* represents specific URI for the request and whole request is build and executed on it. In most times the return object is *Response* from which *readEntity()* method is use for getting the entity. API also provides way to get specific Java object directly without working with *Response* object as shown on example 2.7 with *Car* object.

Client API also provides support for asynchronous requests which can be use for execution of HTTP requests in the background. There are two possibilities to get the response and that is either polling or receiving a callback. For polling is used *Future* interface which is part of JDK from version 5.0 and client example with *Future* is shown on A.1. For callbacks is used *InvocationCallback* interface shown on example 2.8, where request is registered with callback instance and invoked in the background. The interface depending on the status of response executes a defined code[14][15]. Client example with callback is shown on example A.2.

This subsection just gave a quick look on Client API. For more detail about Client API check specification and Javadoc.

Example 2.8: InvocationCallback interface

```
InvocationCallback<Response> callback = new InvocationCallback {
  public void completed(Response res) {
    //do something
  }

  public void failed(ClientException e) {
     // do something
  }
};
```

Asynchronous server-side

Typical HTTP server works in a way that when requests comes in, one thread is responsible for the processing and generating response to the client. This is no problem because requests are short-lived, so few hundred threads can handle few thousand concurrent users with good response times. This was fine until evolution of services and HTTP traffic with JavaScript clients. One scenario started to became a problem and that is scenario where server needs to push events to the client. In this scenario, clients need to know actual information from the server, for example stock price, so they usually send GET request and just block indefinitely until server was ready to send back response. With many clients, this creates large amount of open, long-running requests that are just idling and with them threads too. This scenario is very consuming on operating systems resources and it is really hard to scale up these server-push applications because JAX-RS had one thread per connection model.[6]

To conquer this problem, JAX-RS 2.0 provides a new feature and that is support for asynchronous HTTP. With it, it is possible to suspend current server-side request and have different thread handle sending back the response to the client. This feature can be used to implement long-polling interfaces or server-side push as mentioned. The server-side push problem can be resolved with small set of threads delegated just for sending the responses back to polling clients.

This feature is very analogous to Servlet 3.0 specification and similar to other features of JAX-RS is also annotation driven. To use this feature, application must interact with *AsyncResponse* interface. This is done by injecting this interface into JAX-RS method with *@Suspend* annotation.[6][15] Example is provided in A.3 and more detail information can be found in specification and Javadoc.

Filters and entity interceptors

Last notable new feature of JAX-RS 2.0 are filters and entity interceptors. They are use for intercepting requests and response processing. Notable use cases are authentication, caching or encoding. Similar to Client API, most JAX-RS provides implemented their own interceptor prior to version 2.0. The framework defines two concepts for interceptions and they are:

- filters
- entity interceptors

Filters are used for modification or processing of incoming and outgoing requests or responses. They are executed before and after request and response processing. Main task for entity interceptors is marshalling and unmarshalling of message bodies.

In filters, there are two main groups: server side filters and client side filters. Both groups have 2 different filter for request and response. Server side filters are *ContainerRequestFilter* that runs before JAX-RS resource method is invoked and *ContainerResponseFilter* that runs after invocation of resource method. Added feature for ContainerRequestFilter is possibility to specify when the filter should be invoked, before resource method is matched or after it is matched. This is defined by *@PreMatching* and *@PostMatching* annotations. Client side filters have also two types: ClientRequestFilter and ClientResponseFilter where request filters run before sending the HTTP request to the server and response filters run after receiving response from the server, but before the response body is unmarshalled. [15][6]

Interceptors deal with message bodies and are executed in the same call stack as their corresponding reader and writer. Again there are two different types. One type is *ReaderInterceptor* that wraps around *MessageBodyReaders* and the second type is *WriterInterceptor* that wraps around *MessageBodyWriters*. They are many uses like implementation of specific encoding, generating digital signatures or posting and preprocessing Java object before or after it is marshalled.[15]

2.5 Existing RESTful components

The official distribution of Camel already provides several components that are used for integration with RESTful web services. There are mainly similar project to RESTEasy, that are also certified implementations of JAX-RS 2.0 specification. Notable examples are:

- Camel CXF
- Camel Restlet
- Camel Spark-rest
- Camel Rest

Each component and technology has specific pros and cons. Spark-rest is component integrating Spark Rest Java library running only on Java 8 and supporting only consumer endpoint. Restlet and CXF components are integrating Restlet and CXF frameworks, both have producer and consumer endpoints along with few bonus features.

Last component is Camel Rest that was added in Camel 2.14 and it is used for defining REST endpoints using Rest DSL[32] right in the camel route. There is also possibility to configure this component to use some other RESTful component as base. Any component can be integrated with Rest DSL if they have Rest consumer in Camel. For the integration, new component must implemented *RestConsumerFactory* and the component itself then must implement logic to create a Camel consumer that exposes the REST services based on the given parameters, such as path, verb, and other options.

2.5.1 Camel Resteasy – motivation

As for now Camel already provides lot of RESTful components, so why exactly do we need a new RESTful component if there are components capable to satisfy user's needs. Best answer is probably, because we can. If there are more implementations of JAX-RS specification, then there should be more Camel components for each. This way, users are not limited and can choose component fro their preferred implementation.

Another thing to consider is tight connection between RESTEasy and products like JBoss EAP, JBoss AS or WildFly. All these servers have RESTEasy integrated and are using out for box. Adding possibility to also create Camel routes integrated with RESTEasy on these servers can possibly help lot of users which are already familiar with RESTEasy framework. Of course they is no problem to deploy application with CXF implementation and user Camel CXF component on these servers, but complicate things if we can just create Camel RESTEasy component.

32. `http://camel.apache.org/rest-dsl.html`

3 Analysis and Design

References

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

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

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

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

[5] http://camel.apache.org/creating-a-new-camel-component.html

[6] BURKE, Bill. *RESTful Java with JAX-RS 2.0.* Sebastopol: O'Reilly Media, c2013, ISBN 978-1-449-36134-1

[7] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine, 2000.

[8] http://www.javaworld.com/article/2073264/jax-rs-1-1–what-s-new-.html

[9] http://www.infoq.com/news/2010/02/javaee6-rest

[10] `http://docs.jboss.org/resteasy/docs/3.0.9.Final/userguide/html_single/index.html`

[11] `http://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services`

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

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

[14] http://www.infoq.com/news/2013/06/Whats-New-in-JAX-RS-2.0

[15] http://java.dzone.com/articles/whats-new-jax-rs-20

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

A Appendix

Example A.1: Future example

```
Future<Customer>future = client.target("http://e.com/customers")
.queryParam("name", "Bill Burke").request().async()
.get(Customer.class);

try {
  Customer cust = future.get(1, TimeUnit.MINUTES);
} catch (TimeoutException ex) {
  System.err.println("timeout");
}
```

Example A.2: Callback example

```
InvocationCallback<Response> callback = new InvocationCallback {
  public void completed(Response res) {
    System.out.println("Request success!");
  }

  public void failed(ClientException e) {
    System.out.println("Request failed!");
  }
};

client.target("http://example.com/customers").queryParam("name", "Bill Burke
.request().async().get(callback);
```

Example A.3: Asynchronous server-side

```
TODO
```