jamk.fi

INTEGRATION BETWEEN WEB STORE AND WEB SERVICE USING APACHE CAMEL INTEGRATION FRAMEWORK

Andrzej Kokoszka

Thesis

May 2014

Degree Programme in Software Engineering

Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU

JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s) KOKOSZKA, Andrzej	Type of publication Bachelor´s Thesis	Date 11-05-2014
	Pages 69	Language English
		Permission for web publication (X)

Title Integration between web store and web service using Apache Camel Integration Framework
Degree Programme Software Engineering
Tutor(s) Ari Rantala
Assigned by Descom Oy

Abstract

The main objective of this project was to provide integration for IBM WebSphere Commerce Server with Itella Media Bank (EMMi) Web Service. The integration concerns the product images available through web service to be transferred on demand to the WebSphere Commerce file system server. The communication channel was implemented using Apache Camel integration framework.

This thesis in order to solve the presented issue brings up the topics concerning WebSphere Commerce server, Java based web services and server application integration. The theoretical basic part introduces the concept of integration between two server applications and presents the theoretical solution in the form of enterprise integration patterns. The implementation part goes thoroughly over the process of implementing the theoretical approach utilizing the Apache Camel integration framework together with server implementation of the OSGi technology. The solution benefits from exploiting tools such as Apache Karaf runtime environment and Maven software project management. The final part of the thesis resolves the obstacles that occurred during thesis project and focuses on providing an explanation and comment on possibility of progress and further development.

The result of this thesis project is an application used by IBM WebSphere Commerce Server. The application works as a standalone product integrating the WebSphere Commerce Application Server with third party EMMi web service. This application was required by the developers creating products for customers using EMMi.

Keywords

Integration, web store, web service, Apache Camel, Java, server application

Miscellaneous

Contents

FIGURES

TABLES

1 Need for integration

1.1 Forward

Enterprises face a rapid transformation in modern world. Insatiable demand for improved customer service along with vast growing business environments inevitably seeks for comprehensive solution. Integration has emerged as a key to satisfy those demands. The business processes and applications need to cooperate flawlessly to achieve the goals imposed by the business industry. Integration has become a critical part of any enterprise application development effort. (Hohpe, 2002, 1)

The reality of entrusting an entire business entity into the hands of a single application is in most cases impossible to overcome. Of course, there are already systems that try to achieve it; however, for most enterprises this simply is not good enough. The main problem lays in theirs functionality which performs only a fraction of what a typical enterprise needs. Business functions that are required in typical situations are so complex and in so many numbers that the only way to fulfill all the needs is to let the multiple applications spread the responsibility among them. Such a solution allows to decide which application fits best the requirements for a specific business process and makes it the one that is entirely responsible for it. What is more, dividing the business application into small pieces - where each business process responds to separate application - offers a sufficient amount of flexibility. (Hohpe, 2002, 2)

In order to fully manage those applications, the integration is enforced. The enterprise applications must be able to exchange data in a secure, efficient and reliable manner and the integration process is obligated to enable it. Integration solutions are complex systems that span across many different technologies and levels of abstraction. (Hohpe, 2002, 2)

The thesis aims to point out the issue of integration between two business applications and presents a valid practical answer. The thesis project is assigned by Descom Oy as a part of creating a complex enterprise product for one of the company’s clients.

1.2 Project goal

The objective of the task to be completed is as follows:

To create an integration application connecting the IBM WebSphere Commerce application server and the Itella Media Bank (EMMi) Web Service.

The IBM WebSphere Commerce Server is a web store application which by default stores the images and the other media content on the web server file system. The client uses EMMi to catalog all of the images required by the web store to properly display the products. The web service does not provide a way to directly access images via link; instead, it hands over a web service interface to get the images. The integration application must be able to transfer those images from web server to the desired location on the file system server.

The solution is implemented using Apache Camel integration framework and it embraces the succeeding process:

- Acquiring product credentials for identifying the correct image
- Connecting to the web service
- Searching and generating downloadable link for image
- Transferring the image to server file system

The Apache Camel framework is to be responsible for creating and maintaining the communication channel between the web store and web service and for the downloading of the image. The custom web service client is to be created to conduct all the web service actions including connecting to the web service, searching and generating the link.

The goal is to develop an integration server application that creates and manages the connection and supports the data transfer between the web store and web service.

2 WebSphere Commerce

2.1 E-Commerce

E-commerce is one of the fastest growing businesses in today’s industry. It has emerged from being a commerce introducing and selling products online to the omni-channel experience that aims to deliver a smart and seamless shopping experience integrated across multiple platforms.

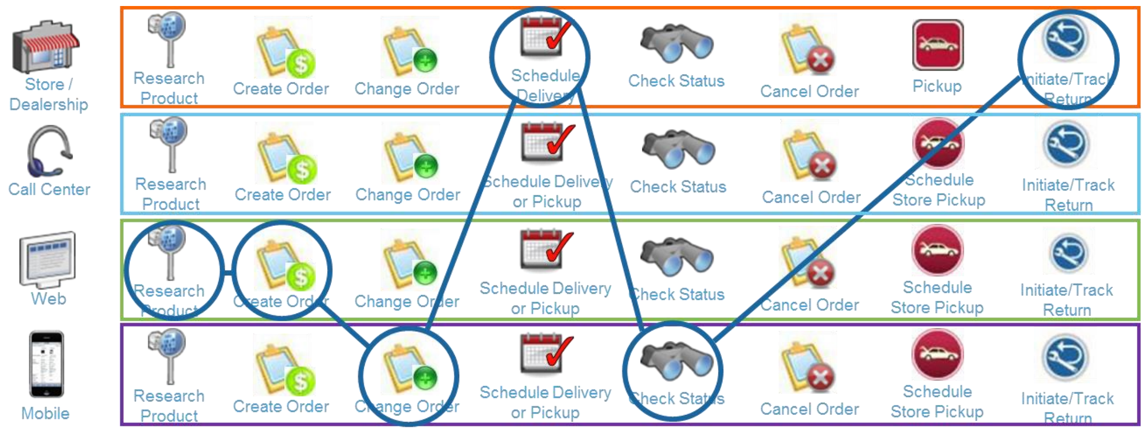


Figure 1. Omni-channel (Malmirae, 2014)

The shopping process as known is simply not sufficient. In a world driven by technology progress the whole trading group from small local merchant to global corporations is struggling to improve the marketing process efficiency to better meet the changing environment.

IBM WebSphere Commerce provides a powerful customer interaction platform for cross-channel commerce.(WebSphere Commerce product overview, 2014)

It is adaptable to fit large enterprises as well as minor businesses. It helps to deploy a cross-channel strategy by equipping business users with manageable tools. It is possible to build and conduct marketing campaigns, products, across all sales channels.

WebSphere Commerce allow to do business with consumers as well as other businesses all in one unified platform.(WebSphere Commerce product overview, 2014)

2.2 WebSphere Commerce architecture

The following diagram shows how the components of the WebSphere Commerce architecture relate to each other.

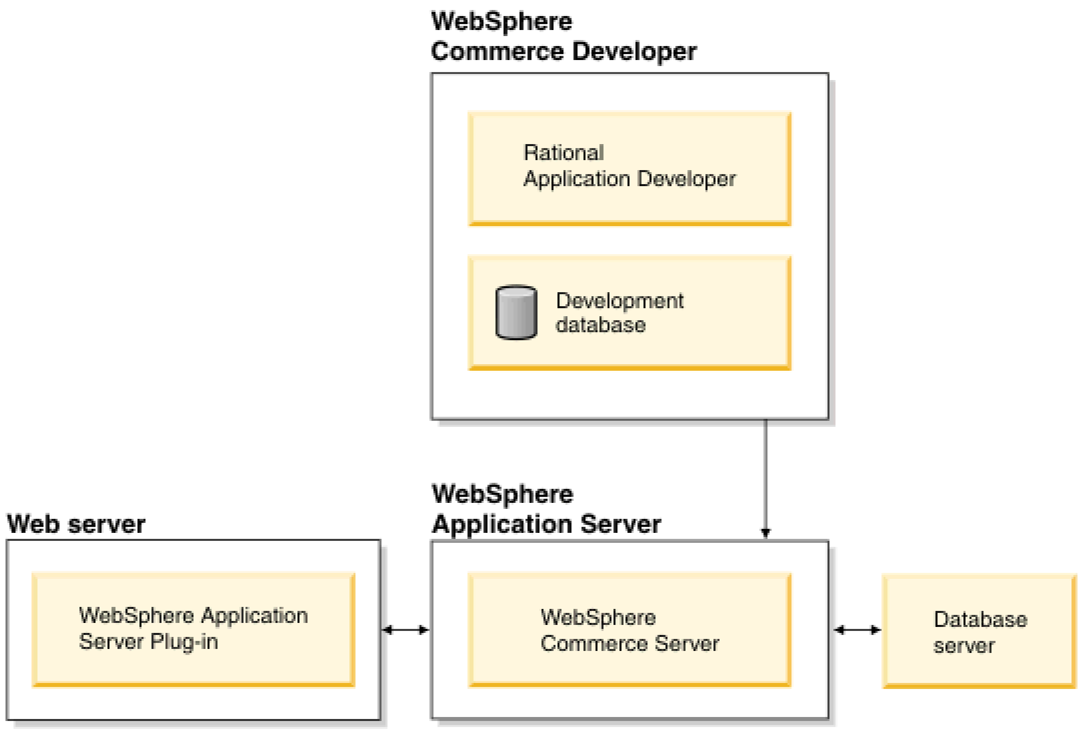

WebSphere Commerce Developer
Rational Application Developer
Development database
Web server
WebSphere Application Server Plug-in
WebSphere Application Server
WebSphere Commerce Server
Database server

Figure 2. WebSphere Commerce architecture (WebSphere Commerce common architecture, 2014)

The heart and core of the Web Sphere Commerce application is the WebSphere Commerce Server. This entity is deployed inside the WebSphere Application Server which gives all the benefits and features coming from server applications. The majority of the data is handled by the Database Server including merchandise and customer data. The link between the request coming from the outside world and WebSphere Commerce application is delegated to the Web server. Web server uses WebSphere Application Server Plug-in to interact smoothly and efficiently with the WebSphere Application Server. All the changes and development of the application features are implemented by customizing or adding code to the WebSphere Application Server. WebSphere Commerce Developer is the one to use when performing those actions. It provides an integrated development environment to modify or create business logic as well as the appearance of the web site. Rational Application Developer base on Eclipse environment and the Development datab ase for creating and testing new features. (WebSphere Commerce common architecture, 2014)

2.3 WebSphere Commerce application layers

WebSphere Commerce defines an application architecture in forms of layers. The application layers defines what parts of the system are free for developers to modify and what parts are not. (WebSphere Commerce application layers, 2014)

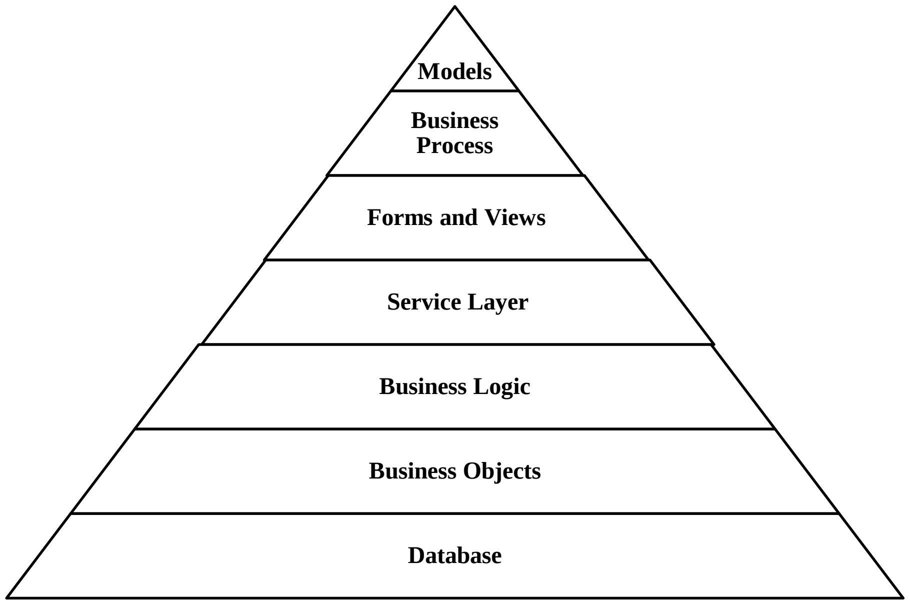


Figure 3. WebSphere Commerce application layers (WebSphere Commerce application layers, 2014)

Models represent the starter store that is available as a model along with the sample data. Business process is described as a particular flow. The catalog browsing flow or order processing flow can be an example of a business process. Forms and Views creates a presentation layer that displays the results such as shopping cart page, registration form or the product display page. Service Layer exposes the business logic to the outside world. Business Logic contains the actual functions of the WebSphere Commerce Server. Adding item to a shopping cart is one of them. Business objects are the Java representation of the data such as order or person. Database is the product persistence layer. It stores all the server data. (WebSphere Commerce application layers, 2014)

3 Web Service

3.1 Service Oriented Architecture (SOA)

In software development business there has always been a need for creating code that is extendable, easy to maintain and capable of integrating with other systems. The first theoretical solution was the *modularity*. Organizing code into modules fulfill both the need for low maintaining costs as well as the reusability. For quite some time there was a belief that object oriented programming could be the answer. Together with software components that was a natural step from object orientation it could have been the practical equivalent for modularity. (What is SOA, 2010)

What comes after those two is to consider the best fit for today's demands. The Service Oriented Architecture defines the software functionality as a collection of services.

Service Oriented Architecture is a style of architecting applications in such a way that they are composed of discrete software agents that have simple, well defined interfaces and are orchestrated through a loose coupling to perform a required function. (What is SOA)

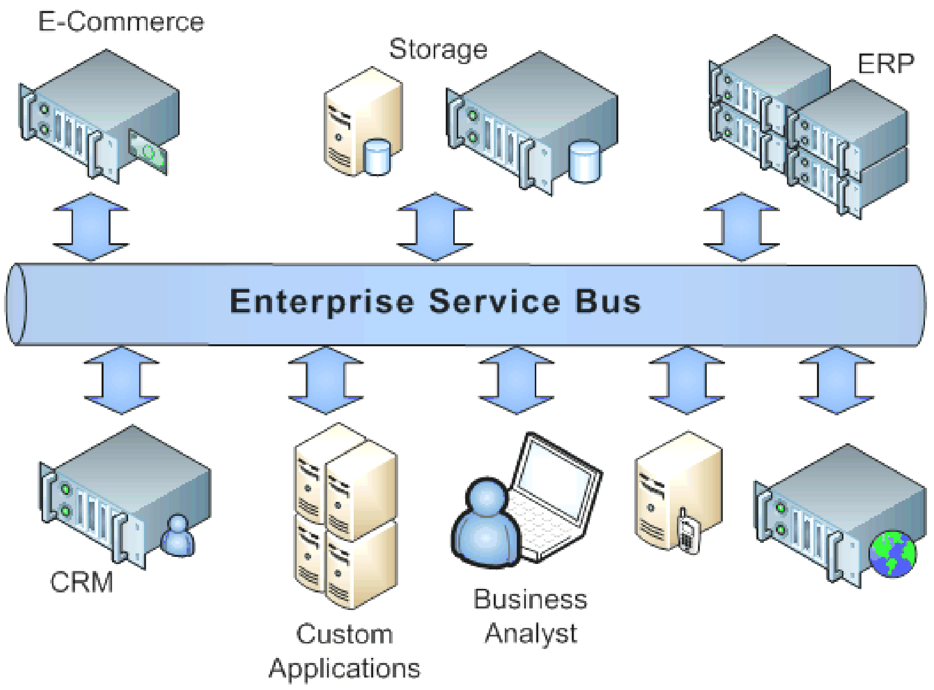


Figure 4. Service oriented Architecture communication using Service Bus (ESB)

The services communicate and can be invoked by messages. The messages from different services are transformed to uniform standard and sent by the Service Bus. (Kodali, 2005)

Those services are the implementation of the individual functions or the business logic of the application. The service model is based on the request/reply infrastructure. The loosely coupled structure allows to separate service implementation from the interface. Each of the services serves as different functionality and is independent from one another. The services do not require a specific knowledge about other services thus, the application built from different services can be easily created. (What is SOA, 2010)

All those specifications allow to build a system that is (What is SOA, 2010):

- Platform independent – the service implementation is separate and hidden in contrary to the interface that is public and available. The service communication is done by messaging system. This construction allows to use and develop services despite the underlying system.
- Available – services are platform independent and use message channel communication, which is why there is no restriction about the service location. Services do not need to possess the knowledge about other services location they work with.
- Scalable – because of the platform independent characteristic the services can be expanded accordingly to needs. There are no limitations of any kind to the service oriented system. Nevertheless considering the availability of the services there might be no need for creating systems that accumulate services whereas the services can exist on its own and still be used.
- Reliable – SOA communication channel bases on messaging. Systems based on messages provide certain functionality like “deliver only once”, “eliminate duplicates” “confirms delivery” that guarantee message delivery.
- Manageable – systems build from independent and self-efficient parts like services are easy to maintain due to the low dependency between components.

- Efficiently testable – small parts of the system like services are easy and fast to test. This helps to achieve more reliable code.

3.2 Web Service application

To fully benefit from service oriented architecture applications, a specific infrastructure is needed. Such an infrastructure must be built according to the software design principal which in this case is the SOA. (Kodali, 2005)

One of the technologies that implements the service oriented architecture is the Web Service.

Web Services are self-contained, modular, dynamic applications that can be described, located, or invoked over the network to create products and processes. These applications can be local, distributed, or Web-based. Web services application are typically delivered over Hyper Text Transport Protocol (HTTP). (What are Web Services, 2014)

Web-based services are published on web servers. A web service client executes the service functionality over HTTP.

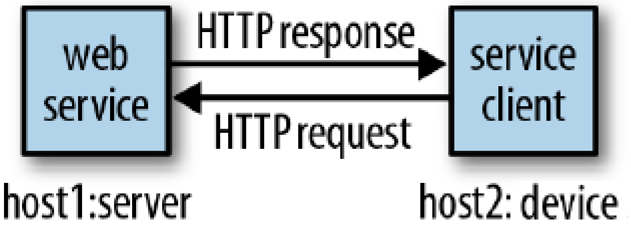


Figure 5. Communication between client and server (Kalin, 2009)

Communication between server and client is based on request/response model. Client requests the functionality from web service and server responses by invoking required service. All is done by the HTTP messages. (Kalin, 2009, 1)

HTTP according to its definition is an application-level protocol that is used to transfer data on the Web (Kristol, 1). When client and server establish a connection, the request and replies from one to another are sent by HTTP file.

There is more than one type of Web services. The one that will be discussed is also used in the further part of the thesis. The one in question is the Simple Object Access Protocol (SOAP)-based type. (Kalin, 2009, 2)

The SOAP-based type web service uses the Extensible Markup Language (XML) type documents as a message to communicate between client and server. The message exchange pattern between client and server that is described as request/response is carried out through SOAP messages. (Kalin, 2009, 2)

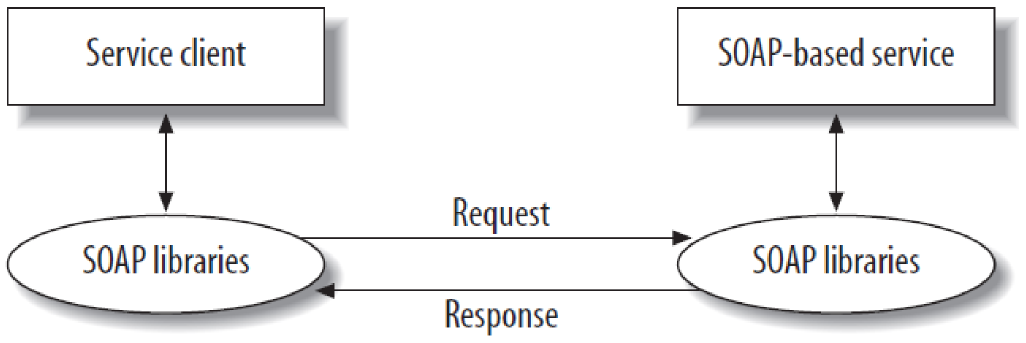


Figure 6. SOAP based server and client communication (Kalin, 2009)

In most cases web service client tends to be an application rather than a web browser. The key feature that puts SOAP type web service before any other is the interoperability. The interoperability allows clients and servers to operate seamlessly regardless of the programming languages they were written in or their platform. The server and client can be written in any language as long as it supports the required libraries.

This language transparency can be achieved by using a mediator that deals with the differences in data types between the client language and the language that service is written in. Such an intermediary is the XML technology.

Extensible Markup Language is a language that can be used to create markup languages for specific applications. It is used to define documents with a standard format that can be read by compatible applications. (XML, 2014)

XML allows to create sophisticated and service-specified documents that serves as a SOAP messages during server/client dialect. (Kalin, 2009, 1)

As mentioned before web services are mainly used by the dedicated applications. In order to develop the client application the service must provide a contract that describes its functionality.

The Web Service Description Language (WSDL) is an XML-based language that defines what operations, functions or methods are available on the web service. (Kalin, 2009, 31)

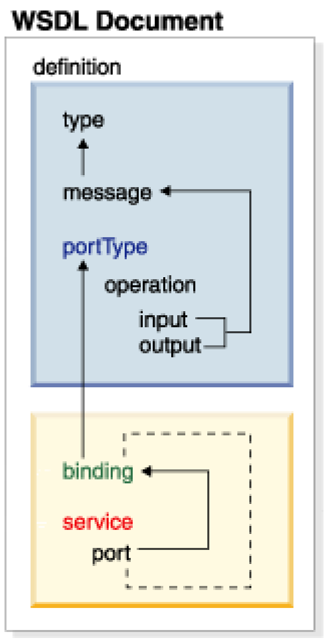


Figure 7. WSDL Document structure (WSDL, 2011)

The *portType* entity describes definition for abstract service interface. The service unit points to the location of the service. The *binding* define how the service is implemented and how the client application should interact with it. (Kalin, 2009, 37)

The WSDL file plays an important part in the Universal Description, Discovery and Integration (UDDI) registry. The UDDI is a platform independent registry where services are registered. UDDI allows to discover and invoke web services applications. It stores information about every web service that is signed in. Those information include a web service interface description in form of WSDL. (Rose, 2005)

All three technology presented in this chapter are essential parts of SOA infrastructure. The Web service technology based on those three pieces fulfill the requirements imposed by the service oriented architecture principals.

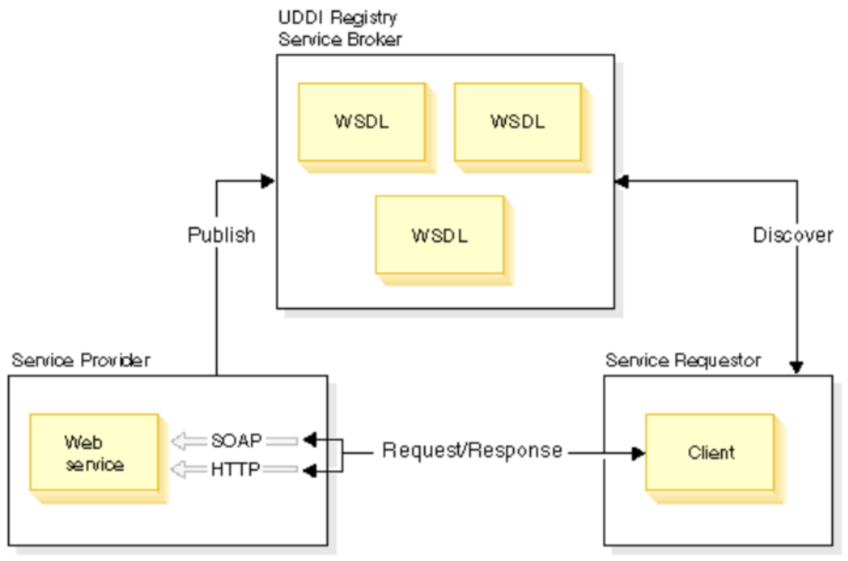


Figure 8. Implementations of Service Oriented Architecture by Web Service (Overview, 2014)

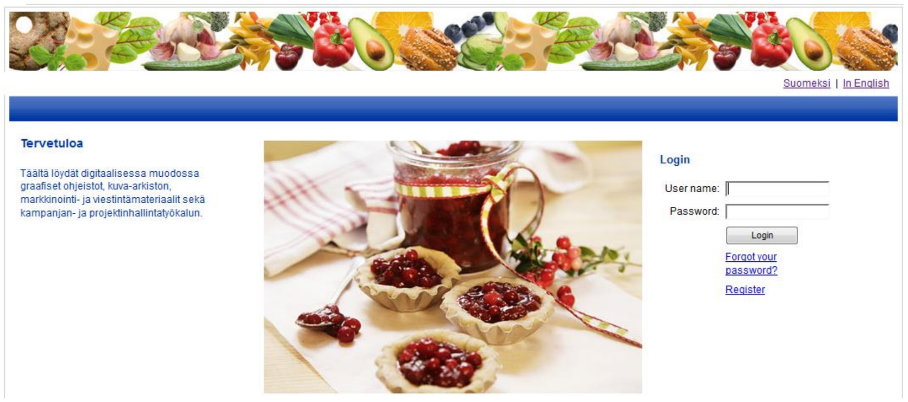
Modern software systems work on numerous platforms. Furthermore, the programming languages used to write those systems keep changing rapidly. This tendency is unlikely to change mostly because of the technological progress that improves every aspect of software development.

Web services can undertake those issues directly mainly because of their distinguish features such as (Kalin, 2009, 3):

- Language transparency – Clients and web services can operate despite their language difference.
- Modular design – One of the first rule that is arise from the SOA concept is the modularity. Web services were designed to be modular by means of every benefits that comes from it (scalability, management, reliability and much more).
- Open infrastructure – Web services use ubiquitous and standardized protocols among networking, which promotes the cooperation among them.

3.3 Itella Media Bank (EMMi) Web Service

The Itella Media Bank (EMMi) web service is a media service that as explained in project goal is the third party web service that is supplying the web store with the media content which specifically is the product images.



Tervetuloa

Täältä löydät digitaalisessa muodossa graafiset ohjeistot, kuva-arkiston, markkinointi- ja viestintämateriaalit sekä kampanjan- ja projektinhallintatyökalun.

Figure 9. Itella Media Bank (EMMi) Web Service graphical user interface

The images available through the web service are meant to be integrated with the web store located on a different server.

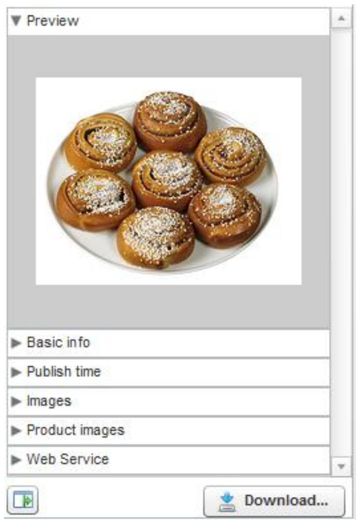


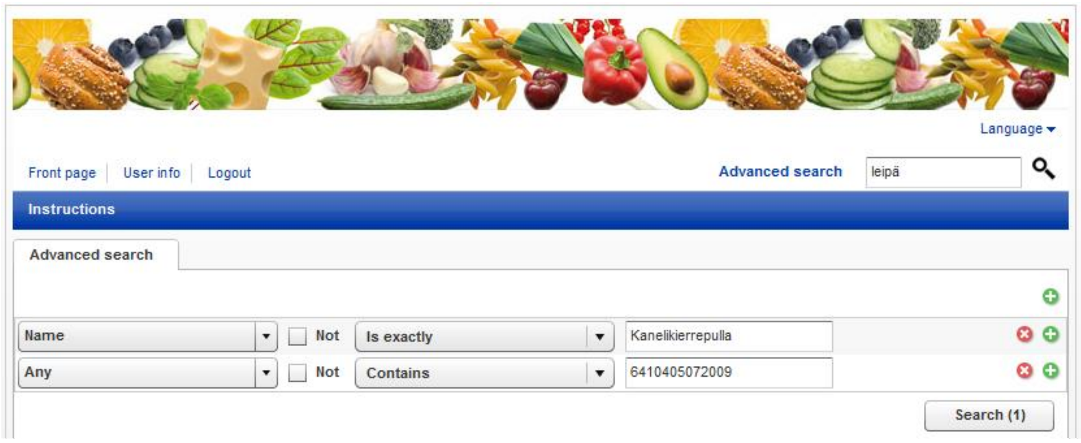
Figure 10. Example of product image in EMMi Web Service

Figure 9 and Figure 10 shown in this chapter illustrate the process of searching and downloading images using graphical user interface available through web browser. To access the *FileElement* object, the client application must use the methods of the interface described by the WSDL file.

Each image inside the system is described by the *FileElement* class. According to the specification: *File element has one or more links to file versions (class: FileVersion) which describes a physical file in the filesystem.* For downloading the image the file version and the unique identifier of the image are required. Besides those two values, the *conversion* type must also be specified for downloading different image sizes.

By obtaining those three variables together with the web address of the service it is possible to successfully download image by generating the web link.

In order to acquire those values the *FileElement* object must be found. The search method in EMMi web service allow to search using huge amount of criteria; however, to properly locate the image only the name of the image and EAN code are needed.



leipä

Name	Not	Is exactly	Kanelikierrepulla
Any	Not	Contains	6410405072009

Figure 11. Web Service graphical interface representation of search function.

The combination of name and EAN code returns the image object. While adopting concrete processing on the image, the required values can be extracted. After those actions the image can be downloaded and the service of the EMMi is no longer required.

4 Apache Camel

4.1 Overview of Apache Camel integration framework

Approaching enterprise projects it is considered common practice not to start working from scratch but use already finished components. Those components - whatever they might be - combined and joined determine the working product; however, in almost every case those components were not designed to be working together. Having that in mind it is decisive to determine the proper integration between components. This is where Apache Camel comes with a solution.

By definition Apache Camel is an open source integration framework that aims to make the integration of systems easier. (Ibsen & Anstey, 2011, 1).

The most important feature of the integration systems is the ability to communicate. That is why the message routing was chosen to be the most important Camel functionality. The key features that make a difference for Apache Camel are high-level abstraction and freedom of using any kind of data type. High-level abstraction in Apache Camel is fulfilled by components. Those components implement huge number of protocols and data types. What is more the modular architecture components can be created according to the needs, which eliminates any unnecessary conversions. The figure below picturing the Apache Camel architecture, shows the modular structure including components, processors and routes.

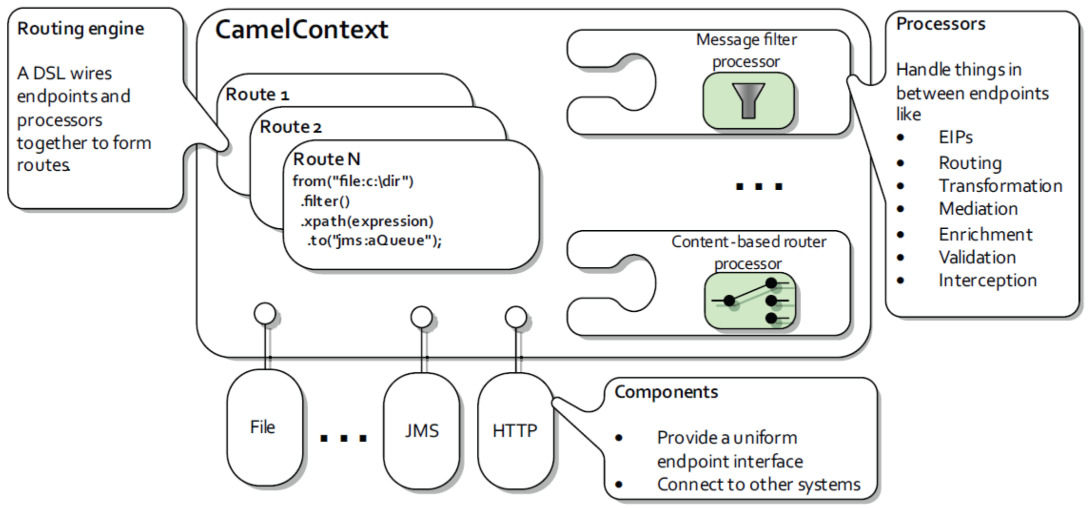


Figure 12. Apache Camel integration framework architecture diagram (Ibsen & Anstey, 2011, 15).

As mentioned before, the ability of creating routes for messages is the key. For creating those routes Camel delivers a domain-specific language. The processors are responsible for changing and managing the message itself. Components are the bridge between Camel core and any other systems that needs to be integrated. Those and more are the central features that will be explained next (Ibsen & Anstey, 2011):

- Routing and mediation engine
- Domain-specific language
- Enterprise integration patterns (EIPs)
- Test kit

4.1.1 Routing and mediation engine

The routing and mediation engine is the core of the Apache Camel. This engine delivers the basic and most important functionality which is consuming producing and processing the messages. Precisely this element relies on the developer because what Camel does is deliver a route engine builder. It is up to the user what routing rules does he/she define and how the path for message to follow will be implemented.

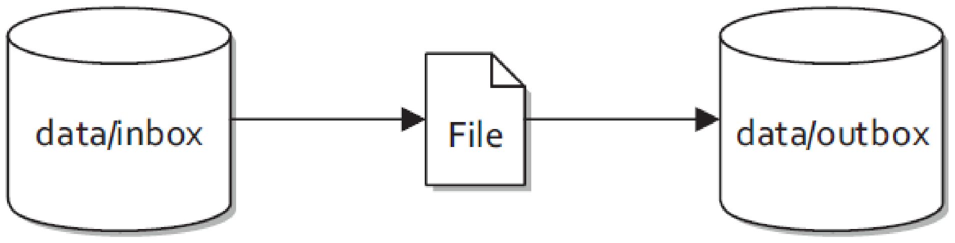


Figure 13. Example of file transfer by Apache Camel routing and mediation engine (Ibsen & Anstey, 2011, 9).

In Apache Camel terminology the consumer is the component that consumes the message meaning that it is in the start of the route. The producer on the contrary delivers the message to its destination point. (Ibsen & Anstey, 2011):

The source and destination for messages are called endpoints. An endpoint is virtual model at the both ends of the channel, as shown in the figure 14 below.

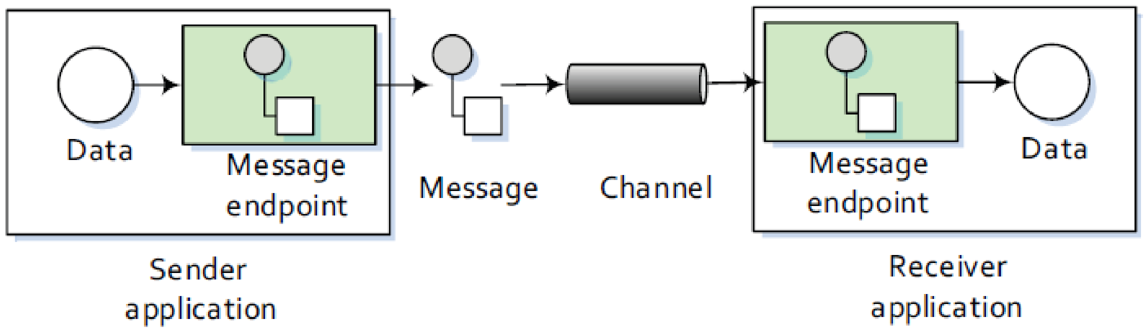


Figure 14. Sender and receiver Apache Camel's endpoints diagram (Ibsen & Anstey, 2011, 18).

The message endpoints are the branch for the producers and consumers described above. The producer and consumer entities associate particular endpoints with the needed data. Traveling message is able to arrive in particular endpoint due to the producer and data information. Analogically the process works the other way. When message is being pulled from the endpoint it is because of the consumer and data payload. (Ibsen & Anstey, 2011)

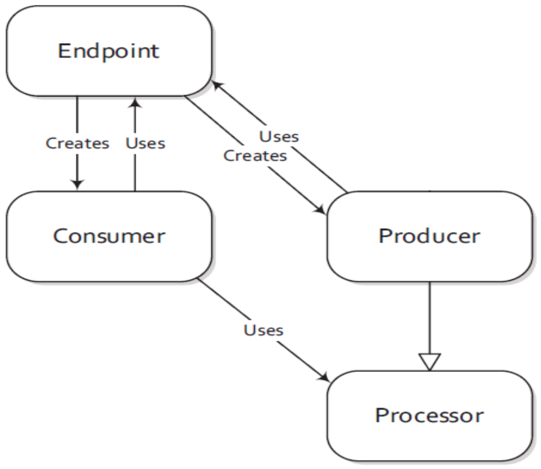


Figure 15. Dependency diagram between Producer, Consumer, Endpoint and Processor (Ibsen & Anstey, 2011, 19.

The processor is used to manipulate the messages. Technically speaking the producer and consumer are also processors programmed to work as endpoints. During routing, messages travel from one processor to another. This architectural solution simplifies internal message flow by using one type for every message. (Ibsen & Anstey, 2011):

Nonetheless, the processors are mainly used in their true way, by giving the programmer the access to the message during route flow. In every case there is a need for custom change of the traveling message somewhere between the nodes. The processor can work on the message any time during the route:

```
public class MyProcessor implements Processor {
  public void process(Exchange exchange) throws Exception {
    // do something...
  }
}
```

The Exchange type is the message type described in the previous paragraph. By making the message easy to handle through the Exchange type, it is possible to modify it without any restrictions. Camel does not make any rules and boundaries when it comes to message handling. Everything that is possible in Java language is welcome here. After all down beneath it is still the Java Object. After the changes the object is wrapped up by the exchange type and send further away. (Ibsen & Anstey, 2011):

The endpoints describe the start and finish. Together with processors they transform into routes. The route is configured using domain-specific language and enterprise integration patterns.

4.1.2 Domain specific-language

A domain-specific language is a programming language tailored specifically to an application domain: rather than being general purpose it captures precisely the domain's semantics. (Spinellis, 2001).

Camel defines a DSL to form a route. Those routes are nothing more than endpoints and processors connected together.

```
from("file:data/inbox")
       .process("myProcessor")
       .to("file:data/outbox ")
```

A fragment of the code presented above shows the simplest Camel route. The route contains two endpoints. The consumer starts pulling the messages in this case files from the specify destination. Those files converted to messages go to the processor that is defined, customized by developer. The changed message finally ends in the second endpoint which also points to the folder path. This example uses the Java DSL. Camel delivers also domain-specific language in other languages such as Spring and Scala.

The main advantage of using DSL is the help that user gets by focusing directly on the connection problem rather than on the tool. Usually routes that are being created are much more complex that the one introduced here. To help build those routes Camel gives the full implementation of enterprise integration patterns implanted into the DSL.

4.1.3 Enterprise Integration Patterns

A vast amount of enterprises today struggle with the integration of applications and business processes. A growing company, changing environment, and more demanding clients presents some of the many reasons. Enterprises are very often shaped of hundreds or thousands of applications each customized to fulfill a dedicated task. What is more, those applications usually came from different vendors in different formats with different purposes. This may easily advance to corporate mess and spider web of systems that are impossible to maintain. That is why it is crucial for an integration system to be lightweight, efficient and clear to those using it. Creating a business oriented application/system is hard. Many companies have been doing this for quite some time and have succeeded to the point where particular patterns have emerged. Those patterns have been cataloged by Gregor Hohpe and Bobby Woolf in their book *Enterprise Integration Patterns – the Book* (Hohpe & Woolf, 2004), which consists of 65 enterprise integration patterns, all of which are available and widely used in Camel applications. In this chapter some of them will be explained. Because of their number only those used in the project were chosen.

The basic enterprise design pattern which additionally describes the implemented architecture is the Message Channel. This pattern directly indicates that the system is based on message exchange. Such a system comes with many advantages, couple of them are:

- Data can be sent asynchronously
- Sender communicates with the receiver directly
- Each message can be processed and handled independently

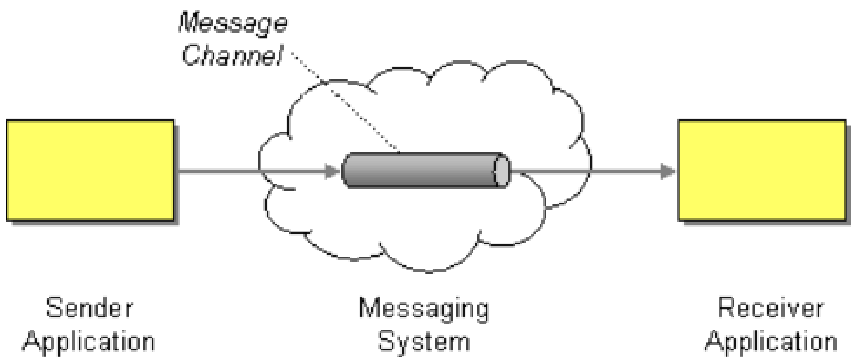


Figure 16. Diagram of the Message Channel enterprise integration pattern (EIP, 2004)

The pattern is by default implemented inside Camel. The user does not see it directly during developing the application. Consumer and producer along with the processor communicate using message system.

At the core of the message system is the message itself. It is the fundamental entity from which everything starts. There is no point in creating the message system without the proper design and implementation of the message.

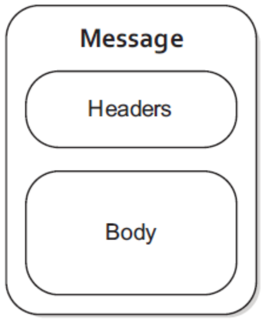


Figure 17. Vizualization of message inside message system (Ibsen & Anstey, 2011)

A message contains of *Body* and *Headers*. Headers point directly to the message properties such as information about content, sender identifiers and many more. There is no restriction about it. The Body is of Java Object type and it is able story any kind of content; however, various object data types inherited from Java Object are not the same. That is why Camel supports numbers of tools to translate the data into adequate format. The conversion is done automatically under the hood. (Ibsen & Anstey, 2011)

The next important pattern used by the project is the Message Translator. This pattern is the one to pick when it comes to message operations (Figure 18).

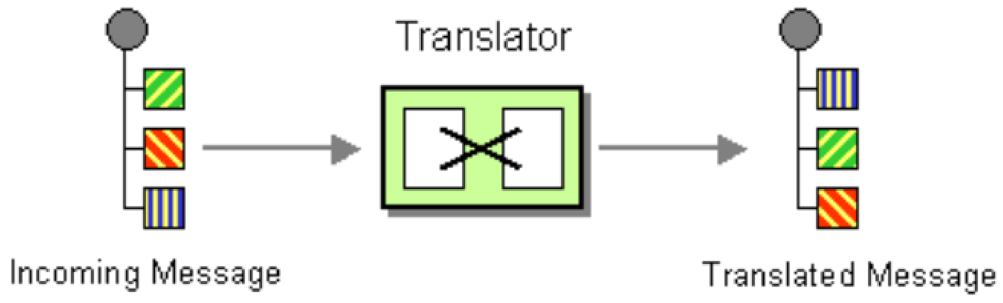


Figure 18. Diagram of Message Translator enterprise integration pattern (EIP, 2004)

The *Message Translator* is the pattern responsible for implementation of the Camel processor. This pattern is important not only for internal communication between nodes in Camel projects; but, moreover as a solution in connecting systems with different data formats. In many cases messages are routed between different systems and applications where each and every one of those has different understanding of corresponding entities. One application can see a data object through some of its properties whereas another application through other properties. Usually this is dictated by the underlying data schema. It is up to the Message Translator to deliver the message that is expected by each of the application. (EIP, 2004)

Plenty of messages traveling through different applications usually are built of very complex data types consisting of multiple elements. This introduces the *Splitter* pattern.

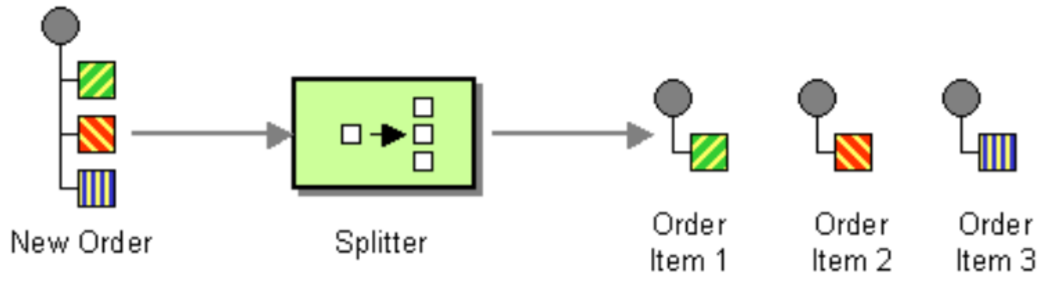


Figure 19. Diagram of Splitter enterprise integration pattern (EIP, 2004)

The Splitter pattern allows to split the message into pieces and to handle each of them separately. This solution may become needed when multiple applications talk to each other. A message originated from one system may contain numerous objects to be processed by different systems. The Splitter sends each of the components as a separate.

Following the Splitter pattern there might by cases where there is more than one receiving system. In this case the best practice is to adopt the *Recipient List* pattern.

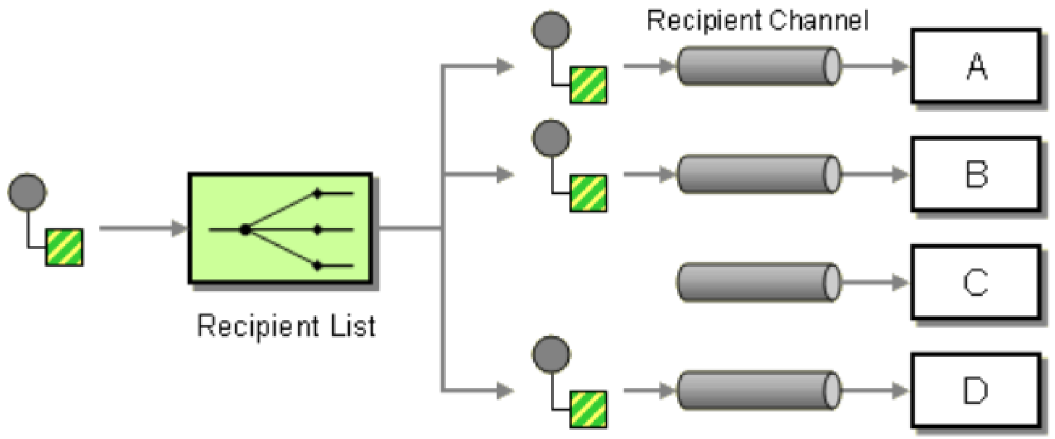


Figure 20. Diagram of Recipient List enterprise integration pattern (EIP, 2004)

The logic behind the *Recipient List* pattern indicates that there are multiple receiver systems that can evaluate the message. It might be used as comparison between systems with the same functionality. Also, this pattern may serve as notifier, using everyday life example such as mailing list. An author of the message can specify for each e-mail a list of recipients. Then it is up to the system to ensure that every message gets delivered properly to the right address.

Recipient List delivers one more very important functionality, namely, the ability to create recipients dynamically. It is the most used advantage of using this pattern. Let’s imagine that there are multiple web pages that the application needs to visit; however, the exact addresses of those web pages along with all the properties are created during the route.

The *Recipient List* can accomplish this issue without using any external tools. This example is real and was solved in the thesis application.

Yet another integration design pattern that finds its purposes in the project application is the *Message Router* pattern (Figure 21).

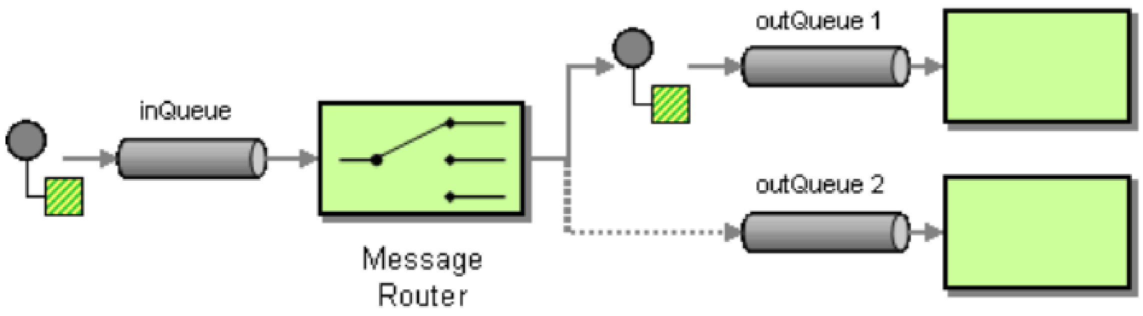


Figure 21. Diagram of Message Router enterprise integration pattern (EIP, 2004)

Message Router similarly to the *Recipient List* allows for multiple destinations. The main difference between those two patterns is the prospect of choice. In this pattern the output for the message can be chosen by evaluating a predicate. The condition statement inside this pattern decides where to send a message or which path of processors the message should followed.

```
from("direct:a")
            .choice()
                .when(header("foo").isEqualTo("bar"))
                    .to("direct:b")
                .when(header("foo").isEqualTo("cheese"))
                    .to("direct:c")
                .otherwise()
                    .to("direct:d");
```

The actual implementation of this pattern is very simple. The author is obligated to determine the conditions and output destinations. The condition inside the `.when()`

function decides which way to push the message. The condition statement is created using other DSL functions.

In every application one of the developer's tasks to do is to ensure correct error handling; especially when one is working with third party web services. Camel implements yet another integration design pattern that helps to secure this problem. *Dead Letter Channel* moves the message to the special channel when it determines that it cannot be delivered.

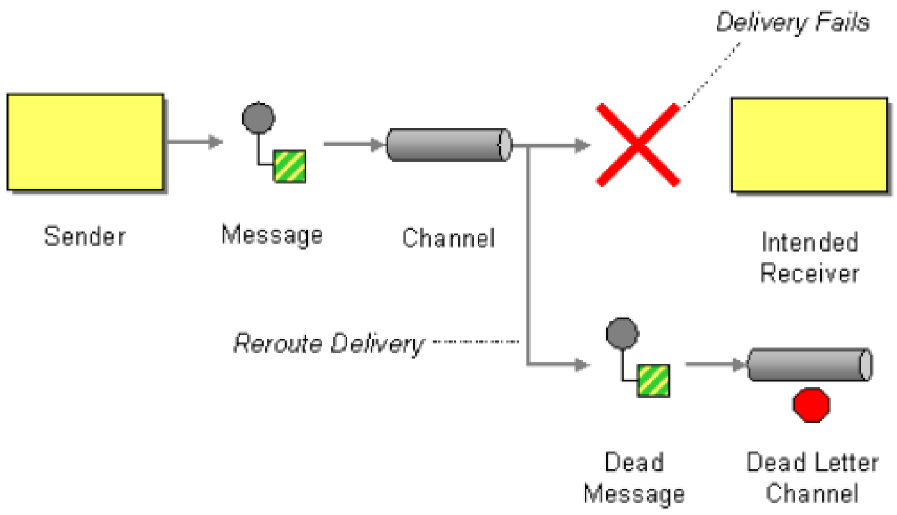


Figure 22. Diagram of Dead Letter Channel enterprise integration pattern (EIP, 2004)

It is a common issue - especially when it comes to work with third party web services or any application that communicates through the internet - for requests or replies to get lost during transition. In most cases a one or more tries of repeating the process fixes the problem. That is why *Dead Letter Channel* implements the *Redelivery Policy*. Redelivery Policy allows to specify how to redeliver the message:

- Amount of redelivery attempts before sending the message to dead letter queue
- Redelivery timeout
- Possibility of changing the time between redeliveries.

All three options create a delay pattern that can be customized for different receivers. The application can be suited to the environments and it keeps on trying to redeliver message or stop and consider the message a failure and send it to the dead letter queue from where it can be further processed. Dead letter queue can be set to accept all sorts of errors. The failed message in some cases might by desired, it can be used as a filter when the Camel route encounters heavy traffic. On the other hand, this pattern must be used with highest precaution. There is nothing more dangerous for the application than catching an unwanted error and letting it go without showing any warnings. (EIP, 2004)

4.1.4 Camel Test Kit

Testing is a fundamental part of the developing software process. Applications are tested all the time during their formation. The first test conducted by the developer is to run the application and see if the results are as expected.

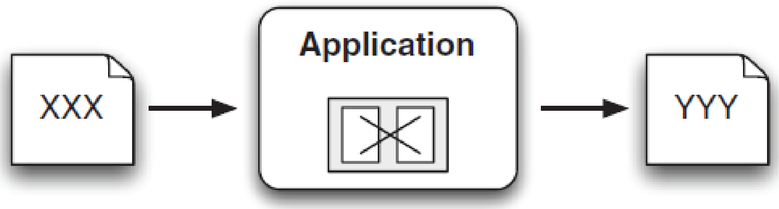


Figure 23. Diagram of processing the message (Ibsen & Anstey, 2011, 155)

Figure 23 shows a typical test case. The message is sent to the application, then it is being transformed according to the specifications and the result which is verified is returned. This scenario is repeated for every unit test:

1. The expectations are being set up.
2. Message is being sent and starts the test
3. The results are being verified.

Those are the three steps that define the unit test. Those are the principals on which Camel Test Kit was built. Every feature and function should be tested regardless if it is the integration module or the logic module of the application to ensure that different components work together. There is always a specification that needs to be followed which is why testing should be an integral part of every step in creating an application. especially the one that involves integration.

Camel Test Kit was designed to make unit tests with Camel much easier. It is built on top of the JUnit framework. JUnit was started by Kent Beck and Erich Gamma (Massol & Husted, 2004, 4). During the time it has emerged to be the standard for unit testing in Java applications.

A *unit test* examines the behavior of a distinct unit of work. Within a Java application, the "distinct unit of work" is often (but not always) a single method. (Massol & Husted, 2004, 6)

It was created due to the author's belief that unit testing is important enough that it should be standardized. Because of the large popularity of the JUnit most of developers that wrote tests before already are familiar with this framework. Camel Test Kit gives the tools to test Camel projects and embeds itself on top of JUnit so they use the same interface. Both of these features significantly ease the process of creating tests for developer.

The one component that is delivered by the Camel Test Kit and is needs to be pointed out is the Mock component. Mock component strictly speaking can simulate real components. It can be useful in many situations. For example, a Camel route is being built but the message itself does not exist yet. In this situation we can simulate the message by using the Mock component. It can be useful as well when the real component is very complex and requires couple some systems to initiate. Then for testing the Camel parts only the mock is used and do not engage other entities. The Mock component follows the three steps explained before. (Ibsen & Anstey, 2011)

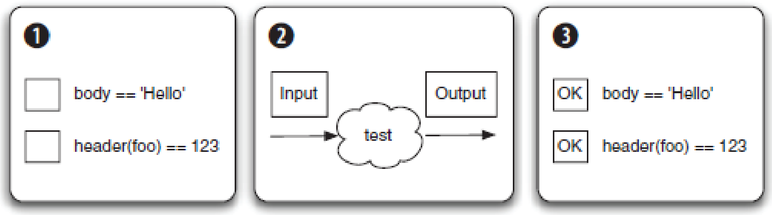


Figure 24. Diagram of test case in Apache Camel (Ibsen & Anstey, 2011, 167)

The first point is the set the expectations. The second point is the test started. What follows is the result being verified. The Mock component simplifies the process of implementing those steps when testing. It can verify numerous of expectations. The example below uses the simple route trying to copy files from one direction to the other.

```
	from("file:data/inbox").to("mock:outbox");
```

The Mock component is used to test this functionality:

```
@Test
public void testQuote() throws Exception {

	MockEndpoint quote = getMockEndpoint("mock:outbox");
	quote.expectedMessageCount(1);

	template.sendBody("file:data/inbox", "Camel rocks");

	quote.assertIsSatisfied();
}
```

In the above test the Mock component is set as an endpoint and is expected to receive one message. What follows is the file containing message “Camel rocks” being sent to the start endpoint. The last part checks if the test was passed and everything worked correctly or failed and the file was not moved. (Ibsen & Anstey, 2011)

4.2 Apache Camel runtime environment

4.2.1 Briefly about Open Service Gateway Initiative (OSGi)

OSGi technology is a set of specifications that defines a dynamic component system for Java. These specifications reduce software complexity by providing a modular architecture for large-scale distributed systems as well as small, embedded applications. (Technology, 2014)

Modularity in modern software development is an important aspect and it is expected as a standard for every enterprise environment. Systems built from independent modules greatly reduce the complexity of the application and save development and maintenance expenses. The OSGi technology started in 1998 and being a great success is developed until now by the consortium of the technology innovators called *OSGi Alliance*. The motives behind creating this technology were to allow the applications to rise from setting together different components, each of them without any knowledge about others. Furthermore, this process was expected to be dynamical. This way of creating applications significantly reduces complexity and operational costs. The OSGi modular and dynamic model brings numerous benefits:

- Reduced Complexity – Adopting OSGi technology involves using modules known as OSGi components. Components communicate over *services* without exposing itself.
- Easy deployment – The OSGi technology standardize the managing and installing the components. Because of the uniform interface OSGi is easy to integrate with already existing and future systems.
- Reuse – Many third party components can be used in an application. Also open source projects significantly expand the modules library.

The OSGi architecture is illustrated in Figure 25 as follows:

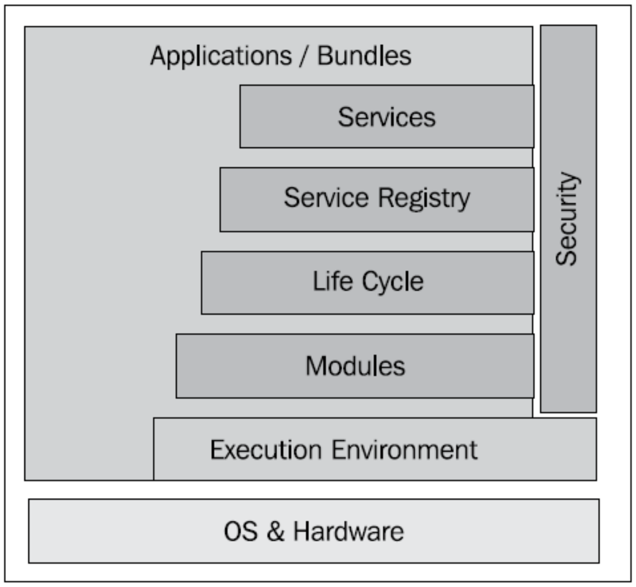


Figure 25. Vizualization of OSGi architecture (Technology, 2014)

Bundles are earlier mentioned components made by developers. The Service layer dynamically connects bundles. Life-Cycle is an OSGi interface used to manage and handle the bundles (update, start, stop uninstall). *Modules* is the layer responsible for importing and exporting code for bundle. Security layer is responsible for security aspects. Execution Environment defines functionality according to specific platform.

In the thesis from the project perspective the two most important layers are the *Bundles* and the *Services*. Bundles are the Java archive file that contains code responsible for some functionality. Bundles represent the fundamental concept of the OSGi which is modularity. Modularity by default is about keeping things private. Bundles hide their code and share only the object functionality. The less the bundles know about each other the lower the complexity of assembled applications/systems. The only sharing part of the bundles is to support the *Services*. The Services helps bundles to collaborate. Each bundle that creates entities is registered by the services and has access to all other bundles. Such a process is dynamic and basic for every modular application or system. (Technology, 2014)

4.2.2 Open Service Gateway Initiative implementation

Creating an OSGi compatible application means following the OSGi interfaces in building the application and deploying it into OSGi container. One of the freely available OSGi implementations serving as a container is the Apache Karaf.

Apache Karaf is a small OSGi based runtime environment which provides a lightweight container onto which various components and applications can be deployed. (Karaf, 2008)

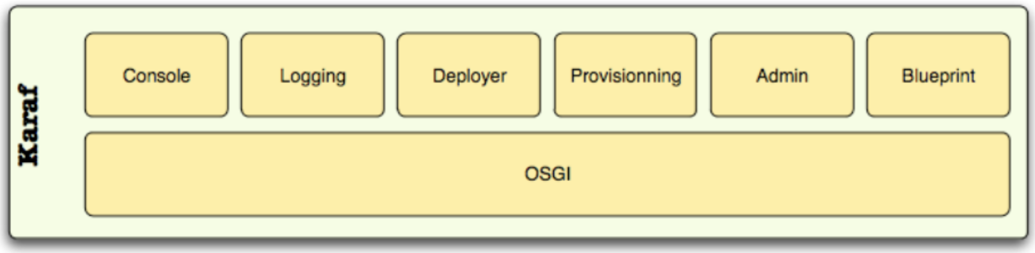


Figure 26. Ilustration of Apache Karaf structure (Karaf, 2008)

Apache Karaf supports list of features that are widely used and helps deploying and maintaining an OSGi type application (Karaf, 2008):

- Hot deployment – Karaf enables hot deployment allowing bundles to be installed by simply copying the Java archive file into specified directory. It is possible to perform changes on the file that will be handled automatically.
- Remote access – Karaf gives the ability to operate it by console using remote client.
- Logging System – The unified logging system supports various different interfaces

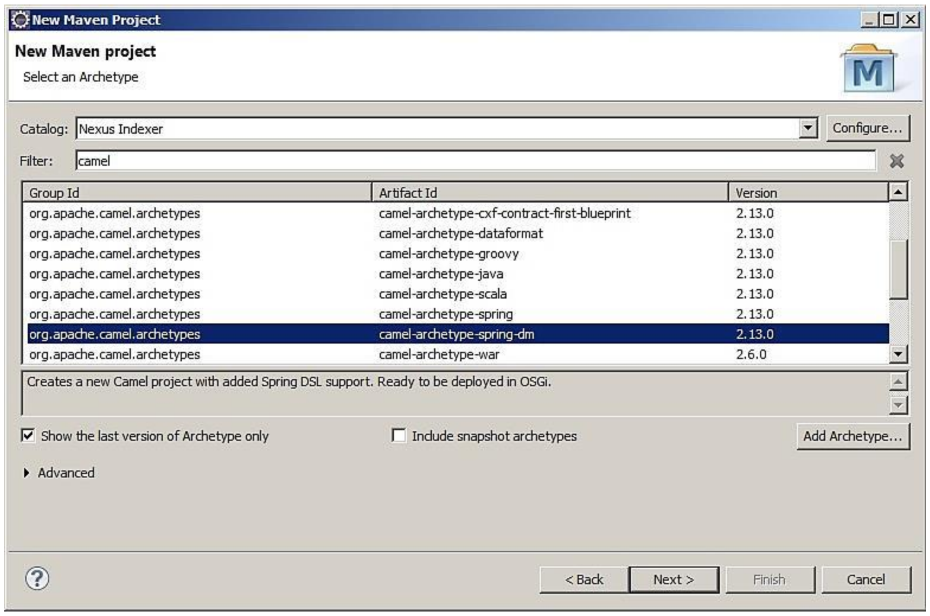
5 Implementation

The project application described in corresponding chapter reflects and contains all technical aspects including program code, scripts and print screens of the process and results; however, certain parts of the code have been concealed and replaced due to confidentiality concerns at Descom Oy.

5.1 Setting the environment

For creating Camel project as well as any other Java application it is inevitable to use one of the widely accessible IDEs. Eclipse was chosen not only because it is the most popular and supported development environment but mainly because the author has used it with every Java project he has worked on. Eclipse is an open-source integrated development environment created by IBM and supported by Eclipse Foundation used to develop application. Enough experience and knowledge were gained to use it freely and to be able to benefit from every of it features.

Eclipse used during the thesis project was preinstalled with Camel and equipped with Maven plugin. This configuration is powerful enough to enable to create a Camel project ready to be deployed in OSGi container.



New Maven project

Select an Archetype

Catalog: Nexus Indexer

Filter: camel

Group Id	Artifact Id	Version
org.apache.camel.archetypes	camel-archetype-cxf-contract-first-blueprint	2.13.0
org.apache.camel.archetypes	camel-archetype-dataformat	2.13.0
org.apache.camel.archetypes	camel-archetype-groovy	2.13.0
org.apache.camel.archetypes	camel-archetype-java	2.13.0
org.apache.camel.archetypes	camel-archetype-scala	2.13.0
org.apache.camel.archetypes	camel-archetype-spring	2.13.0
org.apache.camel.archetypes	camel-archetype-spring-dm	2.13.0
org.apache.camel.archetypes	camel-archetype-war	2.6.0

Creates a new Camel project with added Spring DSL support. Ready to be deployed in OSGi.

☑ Show the last version of Archetype only ☐ Include snapshot archetypes

Figure 27. Creating Camel project in Eclipse integrated development environment using Maven plugin

5.2 Dependencies and project structure

Developing projects that are big enough to include multiple platforms or the ones meant to work on servers always involve dealing with many externals modules. The dependency issue on many shared packages or libraries and their versions arises and needs to be addressed properly.

Maven is a software project management tool that allows to clarify a project's build. What follows and is of most importance is dependency management.

```
<!-- For OSGi bundle -->
<dependency>
	<groupId>org.apache.felix</groupId>
	<artifactId>org.osgi.core</artifactId>
	<version>1.0.0</version>
</dependency>

<!-- JUNIT -->
<dependency>
	<groupId>junit</groupId>
	<artifactId>junit</artifactId>
	<version>4.11</version>
	<scope>test</scope>
</dependency>
<!-- JAXBContext -->
<dependency>
	<groupId>com.sun.xml.bind</groupId>
	<artifactId>jaxb-impl</artifactId>
	<version>2.2.5</version>
</dependency>
```

The code above illustrates several of many dependencies included in the project. All this information came from the `pom.xml` file. This file contains the Project Object Model (POM) of the project and works as a basic unit of work in Maven. Essentially the POM file stores every important piece of information about the project. Each dependency figures as a module that is vital for project to be working. With Maven including, building and automatically updating those libraries is no longer the developer duty.

Moreover, Maven is much more than just a project building tool. There are areas of concern that Maven attempts to deal with:

- Delivery of a uniform build system
- Simplifying the build process
- Modeling based builds

Those are only several of many Maven features. These have been selected due to their direct influence on the thesis project.

Uniform build system allows to be indifferent on the IDE. More than one developer can work on code without being forced to use author's environment and preferences. This really is a helpful feature considering that properly configured IDE is the first step to fast and efficient working.

As mentioned before, making the build process easy is an invaluable advantage. It assures the programmer to focus on creating actual content rather than fixing the building process.

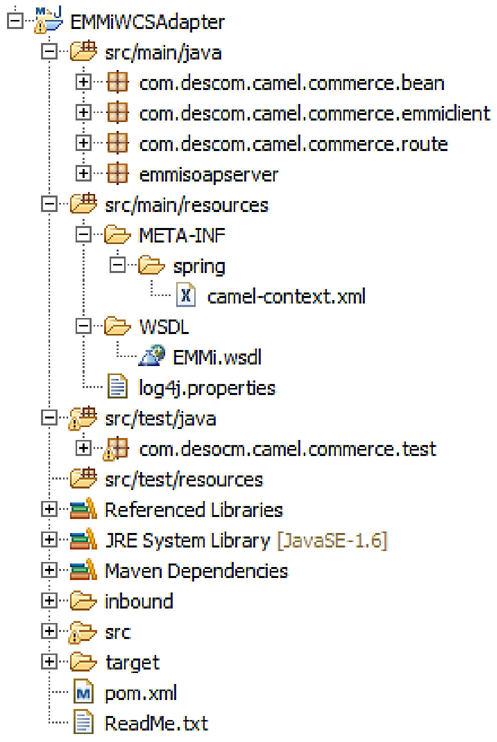
Model based build allows to build a project into a predefined output type. Maven can be customized to set the JAR file to be packed as an OSGi bundle. This feature is extremely useful because it complements with Karaf hot deployment functionality. The JAR file created from Maven's build is used by Karaf to install the application without any additional commands. The plugin responsible for this conversion is shown below.

```
<plugin>
	<groupId>org.apache.felix</groupId>
	<artifactId>maven-bundle-plugin</artifactId>
	<version>2.3.7</version>
	<extensions>true</extensions>
	<configuration>
		<instructions>
		<Bundle-SymbolicName>EMMiWCSAdapter</Bundle-SymbolicName>
		<Private-Package>com.descom.camel.commerce.*</Private-Package>
		<Import-Package>*</Import-Package>
		</instructions>
	</configuration>
</plugin>
```

The project structure generated by Eclipse divides the code into three main parts:

- Main
- Resources
- Test

The main part encloses the proper code of the application. All classes are allocated to the package. This is default for any Java application. The resources folder is the trigger for the program. The files inside that location are the application’s needed resources. The test directory has all tests files and classes able to work as a separate application allowing to run a test without deploying the application to the server. The last file that is worth mentioning is the Project Object Model file that already has been explained. This file’s location is directly in the project folder. The Figure 28 illustrates the project files tree.



```
EMMiWCSAdapter
  src/main/java
    com.descom.camel.commerce.bean
    com.descom.camel.commerce.emmiclient
    com.descom.camel.commerce.route
    emmisoapserver
  src/main/resources
    META-INF
      spring
        camel-context.xml
    WSDL
      EMMi.wsdl
    log4j.properties
  src/test/java
    com.desocm.camel.commerce.test
  src/test/resources
  Referenced Libraries
  JRE System Library [JavaSE-1.6]
  Maven Dependencies
  inbound
  src
  target
  pom.xml
  ReadMe.txt
```

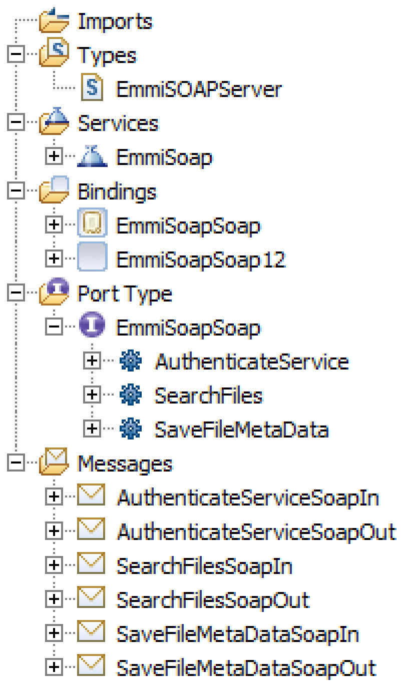
Figure 28. Project's files structure

5.3 Generating client-support code from WSDL

The first step in creating EMMi SOAP client that calls web service is to generate Java artifacts from WSDL file. The requirement of downloading an image from EMMi web service forces to create a web service client able to access the service and use it for the purpose of the goal.

5.3.1 WSDL structure

WSDL document is a contract between a service and a client. The contract provides information about service endpoint, service operations and data type required for those operations. The service contract also describes the message exchanged in the service. The following section represents definitions provided by WSDL (Kalin, 2009, 31):



- The types section provides data type definitions. If this section is empty then the service uses only simple data types such as string and int. However, in EMMi case there are several complex data types, for example:

```
<s:element name="AuthenticateService">
    <s:complexType>
     <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="serviceId" type="s:int"/>
      <s:element minOccurs="0" maxOccurs="1" name="userName" type="s:string"/>
      <s:element minOccurs="0" maxOccurs="1" name="password" type="s:string"/>
     </s:sequence>
    </s:complexType>
</s:element>
```

- The message section defines the message implementing the service. Messages are constructed from data types.

```
<wsdl:message name="AuthenticateServiceSoapIn">
    <wsdl:part name="parameters" element="tns:AuthenticateService"/>
  </wsdl:message>
```

For EMMi service, there are six messages:

```
  <wsdl:message name="AuthenticateServiceSoapIn">[..]
  <wsdl:message name="AuthenticateServiceSoapOut">[..]
  <wsdl:message name="SearchFilesSoapIn">[..]
  <wsdl:message name="SearchFilesSoapOut">[..]
  <wsdl:message name="SaveFileMetaDataSoapIn">[..]
  <wsdl:message name="SaveFileMetaDataSoapOut">[..]
```

The In/Out properties come from service perspective which means that an *in* message is to the service, whereas an *out* message is from the service. All those messages indicate request/response communication and specify the functionality of the service.

- The binding section is where the WSDL definitions go from the abstract to the concrete. A WSDL binding is akin to Java implementation of an interface. It also provides important concrete details about the service. It specifies the implementation details of a service defined abstractly as is shown below

```
<wsdl:binding name="EmmiSoapSoap" type="tns:EmmiSoapSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="AuthenticateService">[..]
      <wsdl:operation name="SearchFiles">[..]
      <wsdl:operation name="SaveFileMetaData">[..]
  </wsdl:binding>
```

The transport protocol for transporting the SOAP messages implementing the service is used to sending and receiving messages. The value of a transport element signals that the SOAP messages of the service will be sent and received over HTTP protocol.

- The service section specifies one or more endpoints at which the service's functionality is available. The service section lists ports elements where the port consists of a *portType* (interface) together with *binding* (implementation).

```
<wsdl:service name="EmmiSoap">
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      WebService provides basic functions for external use of EMMi service.
      </wsdl:documentation>
    <wsdl:port name="EmmiSoapSoap" binding="tns:EmmiSoapSoap">
      <soap:address location="http://localhost/webService"/>
    </wsdl:port>
    <wsdl:port name="EmmiSoapSoap12" binding="tns:EmmiSoapSoap12">
      <soap12:address location="http://localhost/webService"/>
    </wsdl:port>
  </wsdl:service>
```

As presented above, the WSDL file describes the service functionality in detail. It is important for further process to be able to read the WSDL file properly and see how and with what component the service is built. All this information will be later on converted into Java classes and the knowledge about their purpose is crucial for building a client upon them. (Kalin, 2009)

5.3.2 Maven plugin

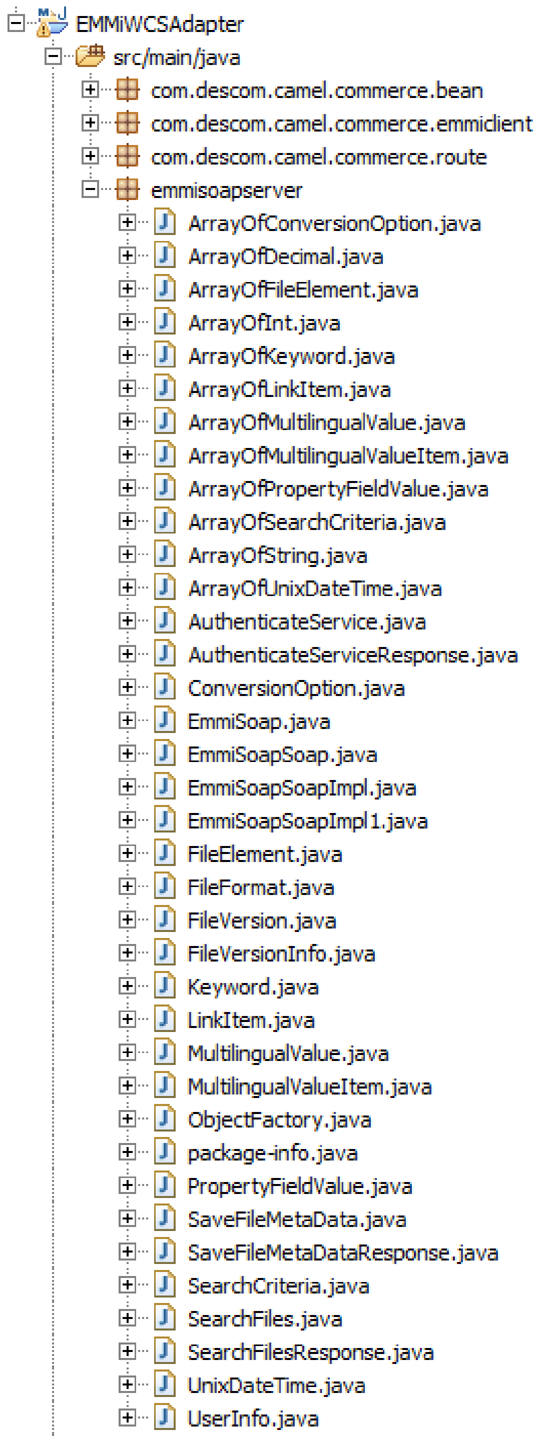
The actual process of creating client-support code is hidden from the programmer. It is all being handled by the project management tool Maven.

The Apache CXF service framework includes a Maven plugin which generates Java artifacts from WSDL.

```
<!-- generating Java code from WSDL -->
<plugin>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-codegen-plugin</artifactId>
   <version>${cxf.version}</version>
   <executions>
       <execution>
       <id>generate-sources</id>
       <phase>generate-sources</phase>
           <configuration>
           <sourceRoot>${basedir}/src/main/java</sourceRoot>
               <wsdlOptions>
                   <wsdlOption>
                   <wsdl>${basedir}/src/main/resources/WSDL/EMMi.wsdl</wsdl>
                       <extraargs>
                       <extraarg>-impl</extraarg>
                       </extraargs>
                   </wsdlOption>
               </wsdlOptions>
           </configuration>
           <goals>
           <goal>wsdl2java</goal>
           </goals>
       </execution>
   </executions>
</plugin>
```

After project build including the above code, Apache service plugin will generate artifacts in the *sourceRoot*. The whole procedure will be specified on the basis of the WSDL file the location of which needs to be put in *wsdlOption*. Those two settings along with the

CXF version are the basics allowing to proceed without any errors. The result of the above code that is the whole package of classes shown below:



```
EMMiWCSAdapter
  src/main/java
    com.descom.camel.commerce.bean
    com.descom.camel.commerce.emmiclient
    com.descom.camel.commerce.route
    emmisoapserver
      ArrayOfConversionOption.java
      ArrayOfDecimal.java
      ArrayOfFileElement.java
      ArrayOfInt.java
      ArrayOfKeyword.java
      ArrayOfLinkItem.java
      ArrayOfMultilingualValue.java
      ArrayOfMultilingualValueItem.java
      ArrayOfPropertyFieldValue.java
      ArrayOfSearchCriteria.java
      ArrayOfString.java
      ArrayOfUnixDateTime.java
      AuthenticateService.java
      AuthenticateServiceResponse.java
      ConversionOption.java
      EmmiSoap.java
      EmmiSoapSoap.java
      EmmiSoapSoapImpl.java
      EmmiSoapSoapImpl1.java
      FileElement.java
      FileFormat.java
      FileVersion.java
      FileVersionInfo.java
      Keyword.java
      LinkItem.java
      MultilingualValue.java
      MultilingualValueItem.java
      ObjectFactory.java
      package-info.java
      PropertyFieldValue.java
      SaveFileMetaData.java
      SaveFileMetaDataResponse.java
      SearchCriteria.java
      SearchFiles.java
      SearchFilesResponse.java
      UnixDateTime.java
      UserInfo.java
```

Figure 29. List of Web Service client supporting classes

5.4 Creating EMMi web service client

The support-code generated by Maven along with JAX-WS implementation delivers the basic functionality of the service, those classes will be used as a starting point in creating a web service consumer fully benefiting from EMMi inventory.

5.4.1 Analyzing EMMi web service interface specification

The next step is to understand the concepts of EMMi service. Given the WSDL file and service technical specification the up to bottom approach is required to find exactly how searching and downloading an image is executed in the service.

Following the given scenario looking from a client perspective, the direct path of actions can be distinguished in Figure 30:

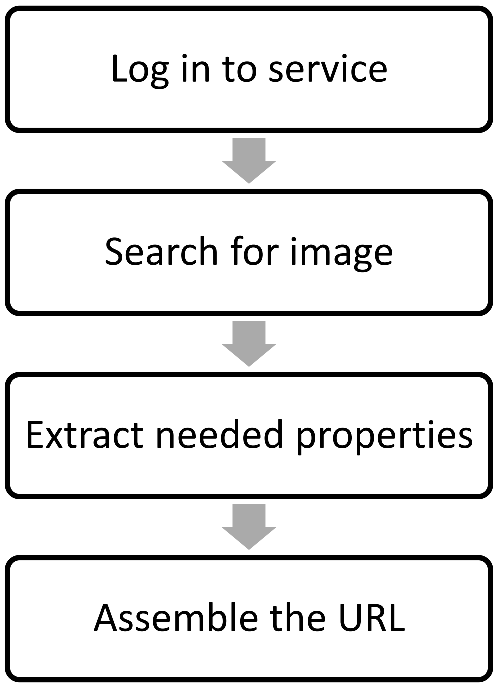


Figure 30. An algorithm for downloading the image from web service

The first step of the algorithm shown above is performed by the *AuthenticateService* method. Authentication verifies user credentials and creates a session after successful authentication. The return value is an identifier of the created session. If authentication fails the method throws Soap exception.

Table 8. Web Service method: AuthenticateService

Parameter	Description	Type
serviceId	Identifier of the service to login. Usually 1, provided by service provider	int
userName	User identifier	string
Password	Password	string

Searching for image is the most important functionality of EMMiClient class. It's the main purpose of the consumer to be able to locate the proper image and prepare it for further processing.

EMMi Service method responsible for image searching is *SearchFiles* method. This function searches for images with given search criteria resulting with array of *FileElement*.

Table 9. Web Service method: SearchFiles

Parameter	Description	Type
sessionId	Valid session identifier provided by AuthenticateService method	String
serviceId	Service identifier. Usually 1	Int
Query	Search criterias.	Array of SearchCriteria

SearchCriteria is an object used to specify a query. This object together with the text string as one of arguments creates the done query for *SearchFiles* method.

Table 10. Web Service class: SearchCriteria

Field	**Description**	**Type**
SearchableField	Information to search -1 = name -2 = description -3 = creation time -4 = publish time -5 = status -6 = keyword -7 = property field value -8 = anything -9 = folder id -13 = modification time -14 = filename	int
StringValues	Text criteria. Used when SearchableField is 1, 2, 6, 8, 14 or when SearchableField = 7 and PropertyFieldType is 1, 2, 3, 5, 7, 13	Array of string
StringSearchOption	Text criteria behavior. Must be used always together with StringValues -1 = contains -2 = contains words -3 = exactly -4 = begins with -5 = ends with	int

SearchableField and *StringSearchOption* fields are to specify the options for narrowing down the results of searching. They play a vital role in the searching process. Besides finding the correct image, it is a must for client searching method to return exactly one image object.

The image object to which references were made several times is the *FileElement* in EMMi service. The *FileElement* object describes the metadata and possesses a unique identifier. Each file element has the following properties:

- Name
- Description
- Publish time

All other are optional, however, included

Table 11. Web Service class: FileElement

Field	Description	Type
Id	Unique identifier	int
PublishStart	Publish start time	UnixDateTime
PublishEnd	Publish end time	UnixDateTime
StatusId	Status identifier	int
CreatorUser	Info of user which created the element	UserInfo
ModifierUser	Info of user which made the most recent change in element	UserInfo
Name	Human readable name	MultilingualValue
Description	Description	MultilingualValue
Created	Creation time	UnixDateTime
Modified	Most recent modification time	UnixDateTime
Write	Indicates if the current user has the privilege to modify element	boolean
PropertyValues	Additional metadata field values Read more	Array of PropertyFieldValue
Keywords	KeywordsRead more	Array of Keyword
Links	Links to folders	Array of LinkItem
ActiveVersion	Active version information	FileVersion
AllVersionIds	Identifiers for all file versions of element	Array of int

File element has one or more links to file versions which describe a physical file in the file system. Each file version has a unique identifier. The file version includes all information about the file.

Table 12. Web Service class: FileVersion

Field	Description	Type
Id	Unique identifier	int
ElementId	Identifier of related FileElement	int
Extension	Filename extension	string
Created	Creation time	UnixDateTime
Size	File size in bytes	long
PreviewSupported	Support for previews	boolean
Previews	Keys of supported preview settings	Array of int
Filename	Filename without extension	string
CreatorUser	Info of user which created the version	UserInfo
Description	Description	MultilingualValue

Conversions	Available conversions	Array of ConversionOption
Info	Additional info about file if detected	FileVersionInfo
Format	Additional info about file format	FileFormat

The file version might include the defined conversions. *ConversionOption* is an object describing a possible conversion of the file version, in other words, an image can be downloaded in more than one resolution.

Table 13. Web Service class: ConversionOption

Field	**Description**	**Type**
Id	Unique identifier	int
Original	Identifies if this option represents the original file of FileVersion	boolean
Name	Human readable name	MultilingualValue
LinkedVersion	If NULL, option is automatic conversion, else option is manually linked to the described FileVersion	FileVersion

When the searching process is complete all needed information for image download are available:

- Session identifier
- File version
- Conversion identifier

Session identifier value is the result of *AuthenticatService* method and it is collected during the login process. The file version can be extracted from File element object which is the result of *SearchFiles* method. Conversion identifier is the Id filed from *ConversionOption* class and also originates from File element. The last step involves putting all this data into one URL string as specified below:

Table 14. Web Service parameters for image download

GET parameter	Description	GET parameter
a	Download action = CONVERSION	a
s	Service identifier. Usually the same that was used during the authentication.	s
fv	Identifier of file version to be downloaded(FileElement.ActiveVersion.Id)	fv
coid	Conversion identifier (ConversionOption.Id) Available conversion can be found at: FileElement.ActiveVersion.Conversions	coid
sid	Valid session identifier provided by AuthenticateService method.	sid

5.4.2 Logging into EMMi Web Service

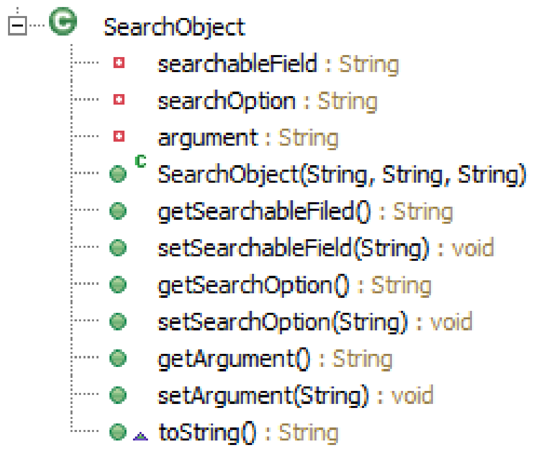
The connecting process is essential for all other operations. Without a valid session identifier the service cannot be accessed and all following actions are pointless.

This phase was completed using the *EmmiSoap* and *AuthenticateService* classes. *EmmiSoap* class provides an object that represents endpoint for service. Such object inherits from *javax.xml.ws.Service* class and is the client view of a Web service. The code below illustrates creating the *EmmiSoap* object and using it to call for *authenticateService* method.

```
public class EMMiClient{

       private EmmiSoap emmiSoap;
       private String sessionId;

       public EMMiClient(){
              emmiSoap = new EmmiSoap();
       }

       public String connect(String userName, String password) {

              sessionId = emmiSoap.getEmmiSoapSoap().authenticateService(1,
              userName, password);
       return sessionId;
       }
```

5.4.3 Search Object

In order to process and extract incoming data from WCS a supporting class needed to be built. The *SearchObject* class was designed especially for this purpose. It is a small and simply class that serves as container for options and arguments of an image that need to be passed on further in the code.



```
SearchObject
    searchableField : String
    searchOption : String
    argument : String
    SearchObject(String, String, String)
    getSearchableFiled() : String
    setSearchableField(String) : void
    getSearchOption() : String
    setSearchOption(String) : void
    getArgument() : String
    setArgument(String) : void
    toString() : String
```

Figure 31. SearchObject class outline

SearchObject class features three string variables and getter and setter methods to those variables. Those strings represent the basic search unit in *EMMi* Web Service. For the basic search there must be defined:

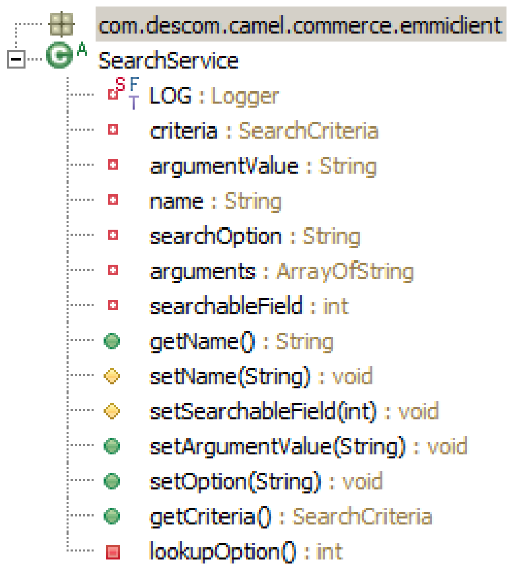
- SearchableField
- SearchOption
- Argument

The searchable field defines what kind of field to search, e.g. “Name”, “Description”. *SearchOption* defines how to search the string argument value, e.g. “Contains”, “Exactly”. Argument represents the string value to be searched. . List of *SearchObjects* will be set into array of *SearchCriteria* class.

5.4.4 Search Service

Search Service class is an abstract class that serves as base for every possible combination of search option and searchable field. While Search Object class is dedicated to be used as a container for data, the Search Service class is an actual functionality designed for creating every possible search query. Search query in this case is a mix of argument, search option and search field.

Search Service allows EMMiClient to build arrays of *SearchCriteria* that are directly applied into *EMMi* searchFiles method. Search Service class objects and methods tree is shown below.



```
com.descom.camel.commerce.emmiclient
SearchService
    LOG : Logger
    criteria : SearchCriteria
    argumentValue : String
    name : String
    searchOption : String
    arguments : ArrayOfString
    searchableField : int
    getName() : String
    setName(String) : void
    setSearchableField(int) : void
    setArgumentValue(String) : void
    setOption(String) : void
    getCriteria() : SearchCriteria
    lookupOption() : int
```

Figure 32. SearchService class outline

Search Service class was designed to be a frame for classes that represent values from search field. Simplifying, each and every option available in search field can be represent by class in EMMiClient. This approach was achievable by using simplified Service Locator Pattern.

The service locator design pattern is used when there is a need to locate various services. The original service locator pattern makes use of a caching technique, which in this case is unnecessary due to the fact that search field services do not carry any heavy abilities.

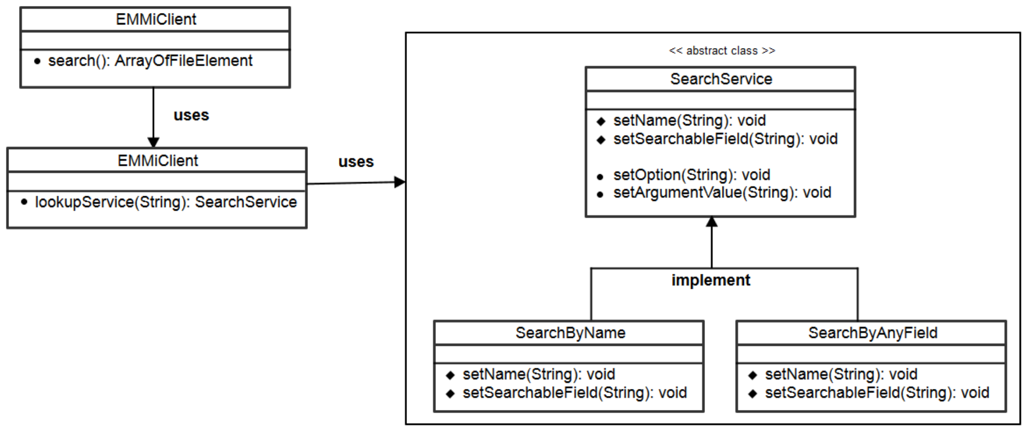

EMMiClient
search(): ArrayOfFileElement
uses
EMMiClient
lookupService(String): SearchService
uses
<< abstract class >>
SearchService
setName(String): void
setSearchableField(String): void
setOption(String): void
setArgumentValue(String): void
implement
SearchByName
setName(String): void
setSearchableField(String): void
SearchByAnyField
setName(String): void
setSearchableField(String): void

Figure 33. Diagram of Service Locator Pattern

Not only such a solution allows to move from different search field classes dynamically but also allows to use these classes outside the main module.

For the time being only two *SearchService* classes have been implemented. In the future when the search method will need to be even more precise, more searchable field classes can be implemented very easily because all processing work was moved to *SearchService* abstract class.

The *SearchOption* preference is implemented inside *SearchService* abstract class. *SearchOption* is single type selection and it does not carry any correlation with *SearchableField* , which is why it has been decided to treat it as part of it. Together with argument value *SearchableFiled* method completes the search unit functionality.

5.4.5 EMMiClient search method

The search unit action appears in many-to-one relationship with the search method in EMMiClient. This search method is parent for all unit actions. The amount of unit actions depends on inbound data if there are more than one combination of argument, *SearchOption* and *SearchableField*, then there will be more units. For image purposes two units are sufficient enough to meet the demands.

EMMiClient search method works as an adapter between the *SearchObject* data, *SearchService* process and actual search service in *EMMi*.

```
public ArrayOfFileElement search(){

        criterias = new ArrayOfSearchCriteria();

        for(SearchObject o : searchObjects){

                service = lookupService(o.getSearchableFiled());
                service.setArgumentValue(o.getArgument());
                service.setOption(o.getSearchOption());

                criterias.getSearchCriteria().add(service.getCriteria());
                }
        return elements = emmiSoap.getEmmiSoapSoap().searchFiles(sessionId, 1,
                                criterias);
}
```

Search method extracts data from *SearchObject* objects and creates and calls for proper search unit actions. The reason why it is feasible is the service locator pattern; because of this approach right units can be created quickly and what is most important dynamically:

```
public SearchService lookupService(String jndiName){

        if(jndiName.equalsIgnoreCase("NAME")){
            LOG.info("Looking up and creating a new NAME search class");
           return new SearchByName();
        }else if (jndiName.equalsIgnoreCase("ANYFIELD")){
           LOG.info("Looking up and creating a new ANYFIELD search class");
           return new SearchByAnyField();
        }else{
           LOG.error(jndiName + " search class not found");
               }
        return null;
}
```

The unit with loaded arguments from *SearchObject* is being turned into *SearchCriteria* object. Each unit adds search criteria to the *ArrayOfSearchCriteria* object which is passed as an argument to the actual search function in *EMMi* service. The result from *EMMi* in *ArrayOfFileElement* format is being saved and ends the search process.

5.4.6 Image Object

Image class like the *SearchObject* class was designed to serve as container, yet for image data. There is an exact amount of variables needed for generating URL and those are the content of Image class. Search method returns the the *FileFormat* type that is too complex to use deftly. That is why Image class was brought to life.

```
public class Image {

        private int idOfFileVersion;
        private int conversionIdentifier;
        private String sessionID;
        private String base = "http://localhost/file/Download";
        private String and = "&";
        private String sid;
        private String fv;
        private String coid;

        public int getIdOfFileVersion() {[..]
        public void setIdOfFileVersion(int idOfFileVersion) {[..]

        public void setConversionIdentifier(int conversionIdentifier) {[..]
        public int getConversionIdentifier() {[..]
        public void setConversionIdentifier(int conversionIdentifier) {[..]
        public String getSessionID() {[..]
        public String URL(){
                return base+and+sid+and+fv+and+coid;
        }
}
```

Image class poses numerous String values all of which needed to properly create a downloadable link. All methods are setters and getters including *URL*() method that returns fixed link. It is simply enough for a class to know which data is required for download and sufficient enough to be used inside EMMiClient.

5.4.7 Generating URL

The output data that comes from *generateURL*() method successfully ends the logic process of acquiring the image.

One of the Image class variables that must be mentioned before explaining how this method executes is the *conversionIdentifier* integer. Besides the *SearchObject* object as input data there is also the size of the image. Because all methods in EMMiClient work on internal private variables, different methods can be called independently and in different parts of the application which is why the size of the image did not have to be proceed along with other inbound data.

```
public String generateURL(String size){

                if(size.equalsIgnoreCase("SMALL")){
                   LOG.info("Looking up and creating URL for SMALL image");
                   image.setConversionIdentifier(2);
               }else if (size.equalsIgnoreCase("MEDIUM")){
                  LOG.info("Looking up and creating URL for MEDIUM image");
                   image.setConversionIdentifier(8);
               }else if (size.equalsIgnoreCase("LARGE")){
                  LOG.info("Looking up and creating URL for LARGE image");
                          image.setConversionIdentifier(1);
                }else if(size.equalsIgnoreCase("THUMBNAIL")){
                LOG.info("Looking up and creating URL for THUMBNAIL image");
                             image.setConversionIdentifier(12);
                  }else{
                       LOG.error(size + " Image size string not found");
                  }
image.setIdOfFileVersion(elements.getFileElement().get(0).getActiveVersion().g
etId());
                image.setSessionID(sessionId);
                return image.URL();
        }
```

The above method checks the required size of the image through incoming argument. According to the value of this argument, a different conversion identifier is signed in to the Image object. The next value is the id of the file version which essentially is the unique code that determines the right image. It is being pulled from the “elements” variable that is the outcome of search method. Session id String text is the last one. The complete URL fulfills the task set to the EMMiClient client. The next parts of the application do not concern *EMMi* service any more.

5.5 Camel route

One of the most significant parts of the application is routing. Routing is also considered as one of the most important features of Camel. Without routing the program will essentially be a stack of loosely coupled functions. Routing provides the fundamental ability of moving data across the process pipeline.

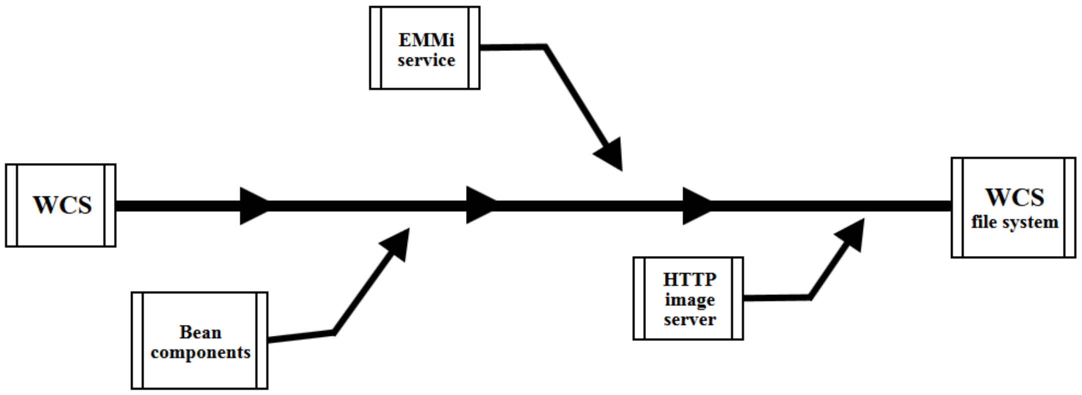


Figure 34. Diagram of message flow

The route shown in figure 34 was designed using more than one process. The first part was to create the main route from first endpoint to the last one on which more details will be added.

Camel allows to use various methods for creating routes. The one used in the project adopts Java DSL. The class responsible for building routes in Camel is the abstract *org.apache.camel.builder.RouteBuilder* class. For building a custom route the particular class needs to extend the camel abstract class mentioned previously and implement the *configure* method. (Ibsen & Anstey, 2011)

The code shown below has been modified due to confidentiality reasons. The changed parts have been replaced with a sample code defining how the actual data should be inserted. Any other inconsistence presented below will be explained.

```
public class AdapterRoute extends RouteBuilder{

	/*
	 *
	 * (non-Javadoc)
	 * @see org.apache.camel.builder.RouteBuilder#configure()
	 */

	@Override
	public void configure() throws Exception {

from("file://./inbound/start/?noop=true")

   .choice()
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .otherwise()
          .to("sftp://username@hostName/directoryName/?options
```

The main concept of complete route is simple. The source files are consumed by Camel because of the *from* method. The *to* method sends the file away. In Camel terminology the *from* method is called the consumer and the *to* method is called the producer. The naming conventions may be misleading. To fully understand this concept one has to look outside the route. The consumer starts the route because it consumes files from source location. The producer ends the route because it produces the outcome files and sends them to their final destination. (Ibsen & Anstey, 2011)

The *file* as well as *sftp* components implementations allow to be called by URIs. Uniform resource identifiers are strings of characters used to identify a name of the resource. The *file* component can pool a file from directory. What is important the file stays in the same directory unchanged in consequence of *noop* option. The *sftp* component is more complex. It gives the ability to receive, in this case sending files over Secure File Transfer Protocol. The only obligation using the *sftp* component is in providing a correct address and credentials. All trouble of creating and maintaining the connection lies within the component duty. (Ibsen & Anstey, 2011)

One of the requirements of the application was the ability to download different image sizes. The Message Router, one of the Enterprise Integration Patterns gives the perfect solution. According to the predicate expressions that are inside simple function (replaced by three dots in the code snippet above) the Camel can deliver the message to different receivers which in this case are different locations for different image sizes on the WCS file system.

5.2 Image download

The second part of building the route is the implementation of image download process. After EMMiClient returns the website link the Camel has to take care of actual download action.

Working on the main route concept there are two patterns that give a suitable solution for image download.

The first one is the *Splitter*. *Splitter* allows to split a message into a number of pieces and process them individually. This pattern is required for the same reason as the Message Router. The possibility of downloading more than one image imposes the message to be split so more than one web link can be executed. (EIP, 2004)

The second solution comes from Recipient List pattern. The recipient List allows to route a message to a number of dynamically specified recipients. One of the recipients is the HTTP image server that the image was downloaded from. Recipient List pattern meets the project needs because the link is created after the routes start. (EIP, 2004)

```
public class AdapterRoute extends RouteBuilder{
	@Override
	public void configure() throws Exception {
```

```
from("file://./inbound/start/?noop=true")
   .split(body())
   .recipientList(body())
   .choice()
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .otherwise()
          .to("sftp://username@hostName/directoryName/?options
```

5.3 Exception handling

Another detail that needs to be considered and added to the main route is the download delay. As specified, one of the problems with downloading image from EMMi service is the delay caused by the conversion. For some sizes the conversion needs to be done before the image can be downloaded. Conversion itself is not fast enough to be completed before the server response is sent and therefore the server returns error, which is why more than one request to the image server needs to be done.

To overcome this issue the Dead Letter Channel pattern was used. This pattern allows to perform many redeliveries with custom delay between each of them. The redelivery policy defines how the message is to be redelivered. Customizable conditions are (EIP, 2004):

- Amount of times the message is attempted to be redelivered before it is considered a failure

This option allows to set maximum redelivery tries.

- The initial redelivery timeout

The time between every redelivery is set here.

- Whether or not exponential back off is used

The exponential back off is a function that can increase the time between retires.

Once all attempts at redelivering the message fail, then the message is forwarded to the dead letter queue. Dead letter queue is nothing more than a specified location. In this case it is the local folder.

Before Dead Letter Channel can work, the error response from server needs to be handled. The Exception Clause can be used to specify error handling. Thus, if certain exceptions are raised, the specific piece of processing can be performed instead or reporting error. For Exception Clause to be working *onException()* method must be applied; however, this function carries the danger of catching other errors and letting them pass through unnoticeably. Hence, the conditional statement has been added to secure the application.

```
public class AdapterRoute extends RouteBuilder{
	@Override
	public void configure() throws Exception {


errorHandler(deadLetterChannel("file://./inbound/error/?autoCreate=true"));

onException(HttpOperationFailedException.class)
   .onWhen(simple("${exception.statusCode} == '591' ||  ${exception.statusCode}
                  == '592' || ${exception.statusCode} == '593'"))
   .handled(true)
   .redeliveryDelay(1000)
   .useExponentialBackOff()
   .backOffMultiplier(2)
   .maximumRedeliveries(13)
   .end();

from("file://./inbound/start/?noop=true")
   .split(body())
   .recipientList(body())
   .choice()
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .otherwise()
          .to("sftp://username@hostName/directoryName/?options
```

The delay pattern created by the combination of all three Dead Letter Channel features starts redelivery after one second. The next attempt is performed after two seconds, next one after four and so on. It stops when it has encountered thirteen trials.

5.4 Bean Component

The last part of the route to be included is the logic part. Beside the parts of the task for which Camel route was directly responsible the processing logic must also take place on the pipeline. The messages in some point must be handed to the components that manage the EMMiClient. Those components are Beans.

Beans are not the only components that allow to transform the data. The reason behind choosing beans is to reduce coupling. Using beans is a great practice because it allows to use any Java code and library. Java DSL comes with a special treatment as far as beans are concerned. Instead of calling methods directly from the route and specifying it explicitly as the endpoint, it is possible to call it to be reference by inserting the class name and method name as shown below (EIP, 2004):

```
public class AdapterRoute extends RouteBuilder{
@Override
public void configure() throws Exception {

errorHandler(deadLetterChannel("file://./inbound/error/?autoCreate=true"));

onException(HttpOperationFailedException.class)
   .onWhen(simple("${exception.statusCode} == '591' ||  ${exception.statusCode}
                   == '592' || ${exception.statusCode} == '593'"))
   .handled(true)
   .redeliveryDelay(1000)
   .useExponentialBackOff()
   .backOffMultiplier(2)
   .maximumRedeliveries(13)
   .end();

from("file://./inbound/start/?noop=true")

   .beanRef("processData", "extractImageProperties")
   .beanRef("processData", "connectEMMi")
   .beanRef("processData", "searchImage")
   .beanRef("processData", "generateDownloadURL")
   .split(body())
   .recipientList(body())
   .beanRef("processData","setImageFileName")
   .choice()
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .when(simple("${...}"))
          .to("sftp://username@hostName/directoryName/?options
      .otherwise()
          .to("sftp://username@hostName/directoryName/?options
       }
}
```

5.6 Camel logic

The fundamental unit in Camel transmission is the message. These entities are used by the systems to communicate with each other when using messaging channel. Messages have a body and headers where the body is of type *java.lang.Object* and headers are values associated with the message. The *java.lang.Object* class guarantees to store any kind of content. During pushing the message through different beans the *exchange* abstractions for modeling message are exploited. This message container provides two fundamental elements used during routing. *In message* and *Out message*. These methods allow to control input and output messages by modifying the body and headers of the message. As shown in the route, the *processData* class handles the Camel logic. Each method in this class takes control of image accusation using EMMiClient:

```
public class ProcessDataFile {

private static final transient Logger LOG =
LoggerFactory.getLogger(ProcessDataFile.class);

public void extractImageProperties(Exchange exchange) throws Exception {
      LOG.info("Extracting image properties");
      String [] size = {"large"};
      List<SearchObject> list = new ArrayList<SearchObject>();
      list.add(new SearchObject("Name","Exactly","BROILERIHAMPURILAINEN"));
      exchange.getIn().setHeader("imageSizesToDownload", size);
      exchange.getIn().setBody(list);
}
```

The *extractImageProperties* method imitates the starting message by setting the list of sizes and the list of *SeachObjects*. The code above has been created as an example because the original function cannot be shown.

```
public void connectEMMi(Exchange exchange) throws Exception {
        LOG.info("Connecting to EMMi");
        String login = "descom-mhelin";
        String password = "passw0Rd";
        EMMiClient em = new EMMiClient();
        em.connect(login, password);
        exchange.getIn().setHeader("emmiclient", em);
}
```

The *connectEMMi* method creates authentications in EMMi service using EMMiClient. The EMMiClient object is then sent in the message header.

```
public void searchImage(Exchange exchange) throws Exception {
        LOG.info("Searching for image");
        EMMiClient em = (EMMiClient) exchange.getIn().getHeader("emmiclient");
        em.setSearchObjects((List<SearchObject>)exchange.getIn().getBody());
        exchange.getIn().setBody(em.search());
}
```

The *searchImage* method looks for image, packs the results in the message body and pushes it forward.

```
public void generateDownloadURL(Exchange exchange) throws Exception {
        LOG.info("Generating URL for download");
        EMMiClient emmi = (EMMiClient) exchange.getIn().getHeader("emmiclient");
        String[] sizes =
        (String[])exchange.getIn().getHeader("imageSizesToDownload");
        List<String> lst = new ArrayList<String>();
                for(String st : sizes){
                        lst.add(emmi.generateURL(st));
                }
        exchange.getIn().setBody(lst);
}
```

The *generateDonwloadURL* method creates an URL for every size of the image specified at the *extractImageProerties* method. The list of URLs is put in the message body. The body is then examined and processed by *.spliy*() and *.recipentList*() functions.

```
public void setImageFileName(Exchange exchange) throws Exception {
        String []parts = exchange.getIn().getHeader("Content-
        Disposition").toString().split("\"");
        exchange.getIn().setHeader(Exchange.FILE_NAME, parts[1]);
        }
}
```

The *setImageFile* extracts data from the *Content-Disposition* header. This header was set by the HTTP response from the image server. Based on content the file name is set.

5.7 Camel runtime

All the components and routes are contained within the *CamelContext* which is the Camel's runtime. Because the Camel context is configured by the spring framework it starts automatically along with any reference routes defines inside. (Ibsen & Anstey, 2011)

```
<bean id="route" class="com.descom.camel.commerce.route.AdapterRoute" />
  <bean id="processData"
class="com.descom.camel.commerce.bean.ProcessDataFile"/>

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

5.8 Unit Tests

The application tests were performed using Camel test kit. Camel Test Kit is built on top of the JUnit. JUnit is a Java based framework for unit testing. Being familiar with JUnit automatically makes one competent enough to write tests for Camel. The Test Kit delivers special endpoints call Mocks. Mocks can simulate a real component so the actual source files from WCS are not needed. For mocks to be implanted the special test route needed to be created (Ibsen & Anstey, 2011):

```
public class TestRoute extends RouteBuilder{
            @Override
            public void configure() throws Exception {
                from("direct:start")
                .beanRef("processData", "extractImageProperties")
                .to("mock:Properties")
                .beanRef("processData", "connectEMMi")
                .to("mock:EMMi")
                .beanRef("processData", "searchImage")
                .to("mock:Search")
                .beanRef("processData", "generateDownloadURL")
                .to("mock:URL")
                .split(body())
               .recipientList(body())
                .beanRef("processData","setImageFileName")
                .choice()

                .when(simple("${"))
                .to("mock:output")
                .when(simple("${...}"))
                        .to("mock:output")
                        .when(simple("${...}"))
                        .to("mock:output")
                        .otherwise()
                        .to("mock:output");

                        from("direct:startWCSConnection")
                        .to("sftp://username@hostName/directoryName/?options")
                        .to("mock:WCS");

                }
```

The location of Mock endpoints indicates that every route step has its test. Presented below *ImageServerResponseTest* method tests if the response from the image server has the same type of image it should have:

```
@Test
public void ImageServerResponseTest() throws Exception{
        LOG.info("TEST: start EMMiResponseTest ");
        MockEndpoint mock1 = (MockEndpoint) context.getEndpoint("mock:output");
        mock1.expectedMessageCount(1);
        mock1.message(0).body().isInstanceOf(java.io.InputStream.class);
        mock1.assertIsSatisfied();
        LOG.info("TEST: EMMiResponseTest:complete");
    }
```

Another test case analyze if the connection between thesis application and server file system is establish properly. Without the access to the file system server the application cannot work because the destination endpoint is invalid.

```
@Test
public void connectEMMiTest() throws Exception{

  LOG.info("TEST: start connectEMMiTest");

  MockEndpoint mock1 = (MockEndpoint) context.getEndpoint("mock:EMMi");

  mock1.expectedMessageCount(1);
  mock1.message(0).header("emmiclient").isInstanceOf(EMMiClient.class);

  mock1.assertIsSatisfied();

   LOG.info("TEST: connectEMMiTest:complete");
}
```

Beside the Mock's advantage when it comes to simulating real components, Camel test can also be run locally and thus relies the application from the necessity of being installed on a server. The special set up is being conducted to run the test locally:

```
public class TestCamel extends CamelTestSupport {
@Override
public void setUp() throws Exception {

  SimpleRegistry registry = new SimpleRegistry();
   registry.put("processData", new ProcessDataFile());

   list = new ArrayList<SearchObject>();
   list.add(new SearchObject("Name","Exactly","BROILERIHAMPURILAINEN"));
   size = new String[]{"large"};

   context = new DefaultCamelContext(registry);

   template = context.createProducerTemplate();

   context.addRoutes(new TestRoute());

   context.start();
}
```

In tests the new Camel context is being created. The context is filled with route and registry with *processData* class. It is started and the results can be seen inside Eclipse without any server interpose.

6 Results and Discussion

The main objective of the thesis project was to deliver a reliable communication channel between a web store and a web service. The integration application needed to face various conversions and breakpoints along the route. Most of the burdensome work with communications was resolved by Apache Camel and its components:

- File Transfer Protocol Component – This component imported into Apache Camel framework significantly simplifies the process of transferring images into web store file server system.
- Hypertext Transfer Protocol Component – The image download part of the process was not be able to perform without this component. Downloading image was performed by Camel simply by providing this component with the correct web link. The process of sending a request to the server and fetching the image was all done “under the hood” without the developer’s contribution.
- Apache Web Service Framework Component - This component took a huge part in creating and managing the web service client that directly delivered the web service functionalities. It allowed to create an interface of the web service functions from the Web Services Description Language which later on turned into the implementation of those functions. Furthermore, it manages to connect to the web service each time there is an image request, even right in this moment.

One of the biggest obstacles that was encountered during the process of developing the application was the web service image conversion. The particular image on web service had more than one resolution and there were cases where more than one resolution needed to be downloaded. The problem laid in the conversion that must be done before downloading, which caused the application to pause and wait.

The solution comes with the Enterprise Integration Patterns. Those design patterns were used widely during each step of the application working process. Their functionality was exploited to control the flow of every data object that was moving along the channel. The conversion problem in particular was solved by creating a combination of delay and redeliver patterns that successfully united the control of the waiting action.

The thesis project was designed to follow the Service Oriented Architecture. It was built in consequence of abiding the modularity pattern. That is why the web service client is able to work separately, outside the Apache Camel route. This feature was sought by the other Descom employees who are working with this client.

The web service client was built inter alia to search for a particular image; however, it was designed to do much more. The modular and service oriented interface of the client allows for further development. More functions with more search criteria are to be implemented in an easy and simple way.

Without any questions, the integration development takes a notable role when it comes to e-commerce. The number of different software and vendors still grows, which even more assures the need for integration. The combination of Apache Camel integration framework together with WebSphere Commerce Server application may bring the correct answer for the industry’s growing needs.

REFERENCES

EIP, 2004. Enterprise Integration Patterns. Article on Apache Camel website. Accessed 18.2.2014. Retrieved from https://camel.apache.org/enterprise-integration-patterns.html

ESB, n.d. Article about Enterprise Service Bus posted on centeractive portal. Accessed on 10.3.2014. Retrieved from http://www.centeractive.com/content/enterprise-service-bus

Hohpe, G. 2002. *Enterprise Integration Patterns*. Accessed on 6.3.2014. Retrieved from http://hillside.net/plop/plop2002/final/Enterprise%20Integration%20Patterns%20-%20PLoP%20Final%20Draft%203.pdf

Ibsen, C., Anstey, J. 2011. *Camel in Action*. Manning Publications Co.

Kalin, M. 2009. *Java Web Services: Up an Running*. Published by O'Reilly Media Inc.

Karaf, 2008. Article about Apache Karaf. Accessed on 17.2.2014. Retrieved from http://karaf.apache.org/index.html

Kodali, R. 2005. *What is service-oriented architecture*. Accesed on 10.3.2014. Retrieved from http://www.javaworld.com/article/2071889/soa/what-is-service-oriented-architecture.html

Kristol, D. n.d *HTTP*. Silicon Press. Accessed on 13.3.2014. Retrieved from http://www.silicon-press.com/briefs/brief.http/brief.pdf

Malmirae, P. 2014 *Future of Commerce*. Internal presentation in Descom Oy. Accessed on 12.3.2014

Massol, V., Husted, T. 2004. *JUnit in Action*. Manning Publications Co.

Overview, 2014. Article posted on the IBM Info Center. Accessed on 12.3.2014. Retrieved from http://pic.dhe.ibm.com/infocenter/adiehelp/v5r1m1/index.jsp?topic=%2Fcom.ibm.etools.scenario.hospital.doc%2Fhtml%2Fzxmlovr.htm

Rose, M. 2005. *UDDI (Universal Description, Discovery and Integration).*Accesed on 12.3.2014. Retrieved from http://searchsoa.techtarget.com/definition/UDDI

Spinnelis, 2011. *Notable Design Patterns for Domain-Specific Languages*. Accesed on 7.3.2014. Retrieved from http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html

Technology, 2014. Article on OSGi Alliance web site. Accesed 19.2.2014. Retrieved from http://www.osgi.org/Technology/HomePage

WebSphere Commerce application layers, 2014. Accessed on 12.3.2014. Retrieved from http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/topic/com.ibm.commerce.developer.doc/concepts/csdapplication.htm

WebSphere Commerce common architecture, 2014. Article posted on the IBM Info Center page. Accessed on 12.3.2014. Retrieved from http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/topic/com.ibm.commerce.developer.doc/concepts/csdsoftwarecomp.htm

WebSphere Commerce product overview, 2014. Accessed on 12.3.2014. Retrieved from http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/topic/com.ibm.commerce.admin.doc/concepts/covoverall.htm

What are Web Services, 2014. One of the tutorialspoint's portal articles. Accesed on 11.3.2014. Retrieved from http://www.tutorialspoint.com/webservices/what_are_web_services.htm

What is SOA, 2010. Article about Service Oriented Architecture by IWD Services. Accesed on 10.3.2014. Retrieved from http://www.indiawebdevelopers.com/resource_center/articles/soa.html

WSDL, 2011. Example tutorial on teqlog portal. Accessed on 12.3.2014. Retrieved from http://www.teqlog.com/wsdl-example-explained-step-by-step.html

XML, 2014. Article on TechTerms portal. Accesed on 14.3.2014 Retrieved from http://www.techterms.com/definition/xml