WebLab

Demo & Tutorial

WebLab development guidelines
Date: 2011–01–07

Cover and control page of document	
Project	WebLab
Version	1.1
Date	2011–01–07
Authors (Organisation)	Émilien Bondu (EADS) Jérémie Doucy (EADS) Gérard Dupont (EADS) Bruno Grilhères (EADS) Patrick Giroux (EADS) Khaled Khelif (EADS) Yann Mombrun (EADS) Arnaud Saval (EADS)

Document type PUB = public

Abstract: WebLab demonstration and tutorial for development of new WebLab services using the WebLab Core as a baseline.

Keword List: Tutorial, technical baseline, web services, WebLab, orchestration, ESB, portlets

Modification control		
Version	*Date*	*Modification*
1.0	2009–09–28	First version of the Demo README
1.1	2011–01–07	Adaptation to model 1.1. Integration of a new WeblLab service Modification of the chain and the visualisation portlet

Contents

CHAPTER 1

Introduction

1.1 Overall presentation

The WebLab platform is the generic name of the development and execution environment platform provided by CASSIDIAN in several research projects (Vitalas, WebContent, e–Wok Hub, Citrine, SAIMSI, VIRTUOSO, AXES. . .) involving the process of several types of media.
This document is a tutorial for programmers who want to use the WebLab platform to develop their own SOA-based applications. It contains the description of (i) a functional demo of the Weblab platform and (ii) the different steps needed to integrate a new service.

This document is composed of the 4 following chapters:

- Introduction (current chapter) ;
- WebLab Demo 1.2 ;
- Tutorial for the development of a new service ;
- Tutorial for the integration of a service in the WebLab demo.

1.2 Prerequisites

To launch only the demo :

1. You should have a jdk1.6.20 or greater installed in order to run the WebLab demo ;
2. JAVA_HOME should be declared and Java should be available in your path ;
3. Your computer should have at least 2Go of memory (4Go recommended) to run the WebLab demo efficiently.

To integrate a new service :

1. You should have Maven[1] 2.0.9 (or later)
2. You should have soapUI[2] if you want to test your service before integrating it in the chain

To retrieve the sources on the OW2 forge, wou will also need a subversion[3] client (see for instance Tortoise SVN[4]).

1.3 Vocabulary and definitions

In the following chapters some ambiguous terms will appear. Here is the reference for their definition in the context of this document:

[1]http://maven.apache.org/download.html
[2]http://www.soapui.org/
[3]http://subversion.apache.org/
[4]http://tortoisesvn.tigris.org/

API	Application programming interface.
BPEL	Business Process Execution Language which is an XML language using to orchestrate services (W3C standard).
Component	Piece of software developed by a partner and which should be integrated into the platform.
ESB	Enterprise Service Bus which is a software infrastructure to connect a set of heterogeneous software components.
Orchestrator	Component dedicated to the orchestration of services and thus the definition of processing chain.
POM	Project Object Model, the fundamental unit of work in Maven.
Service	Piece of software which embeds a component and provides the component functionality as a service (i.e. as much independent from other components as possible). It can a be a part of a processing chain and requires some pre-processing.
Service engine	Software component embedded in the ESB which realises technical process such as distribution or orchestration.
Service interface	Definition of the input and output data for each method proposed by a service (see service interface definitions).
SOAP	Simple Object Access Protocol (W3C standard).
Technical service	Service which could be embedded or not in the ESB and providing a common technical functionality used by most of the services.
WebLab	Name of the platform composed among other by the ESB, the data exchange model and the portal.
Web server	A server which can handle HTTP request/response and thus host web services.
Web service	W3C standard defined to allow client and server components to communicate using XML messages. It is based on the Web service description language (WSDL) and use the SOAP protocol.
WSDL	WSDL (Web Services Description Language) document describes the contract between the web service endpoint and the client. A WSDL document may include and/or import XML schema files used to describe the data types used by the web service (W3C standard).
XQuery	W3C standard defined to Query XML data.
XSD	XML Schema Definition (W3C standard).

Note that most of them are software related concepts and that a basic understanding is needed on all of them. For some specific part of the platform, strong competences on some of them are mandatory and this glossary provide you some resources online for this sake.

CHAPTER 2

WebLab Demo 1.2

2.1 Goal and description

This simple demo just aims to show the WebLab platform basics in term of service integration and orchestration for unstructured document processing and retrieval.
In a nutshell, this demo crawl a local folder, analyze text based documents, index them and finally offer access to them through a web portal. This portal allows to search by text in the crawled documents and to visualize their annotated text content.
The processing capabilities are very limited (only gazetteer based on static dictionary) but it allows to have a complete processing chains and aims to ease integration and test of new components either on processing chain or on user interface.

It is composed of several WebLab services :

- A simple folder crawler able to crawl the content of a given folder ;
- A normaliser that will extract the text content of various files (ms-office, pdf, rtf, etc.) based on Apache Tika[1] ;
- A simple text formatter that removes extra newlines and offer a better (yet still raw) presentation ;
- A simple information extraction service that detects words from gazetteers (technical gazetteers) in the document and annotate it ;
- An indexer that will index the text content and make it searchable based on Apache SOLR[2] ;

[1]http://lucene.apache.org/tika/
[2]http://lucene.apache.org/solr/

- A WebLab BPEL chain that chains the 5 previously mentioned services ;
- 4 WebLab portlets :
  - A launchCrawl portlet that allows users to launch the crawling of documents in a fixed directory ;
  - A search portlet that will launch query on the SOLR searcher ;
  - A result portlet that display the results of the query ;
  - A annotated document portlet that display the document annotated with the annotation added by the gazetteer service.

2.2 Download and installation

Waiting for update

CHAPTER 3

Development of a new service

3.1 Goal of the tutorial

In this chapter we will describe the development of a new web service w.r.t the weblab model. The development of web services is possible in several languages, in this tutorial we chose to use JAVA. Moreover in each language, and especially in JAVA, multiple solutions exists to implement a web service. In this tutorial we will rely on JAXWS[1] which is one of the reference implementation of the implementation of the JAX-WS APIs (Java API for XML Web Services). Finally we will use maven[2] as project configuration management and building automation tool.
The objective of the new service will be to detect the language of text embedded in WebLab document. To do this we will use an existing API called 'the cngram library'[3].

Next sections will detail :

1. Creation of a Java application scheme using maven
2. Implementation of the web service class
3. Packaging and test of the web service

This service will be added into the demo chain in the next chapter.

[1]http://jax-ws.java.net/
[2]http://maven.apache.org/
[3]http://ngramj.sourceforge.net/

3.2 Maven configuration

First of all, if it's not done, you have to install maven (see maven documentation[4]). Then, you can download the WebLAb repository for service development available here[5].

Unzip the `<weblabdir>/repository.zip` in a new folder. This folder will be named `<mavenRepository>`. By default, the repository is placed in `<home_folder>/.m2/repository`, but you can change it by modifying the file `<maven directory>/conf/settings.xml`and change the `localRepository` markup (see embedded comments) to your `<mavenRepository>`, as follows :

```
[...]
  <!-- localRepository
   | The path to the local repository maven will use to store artifacts.
   | Default: ~/.m2/repository
  -->
<localRepository>/path/to/your/mavenRepository</localRepository>
[...]
```

Since the Maven repository is provided, you can set Maven to work offline in order to avoid network connections. For this sake, just set the offline markup to true in the `settings.xml`, such as: `<offline>true</offline>`.

3.3 Creation of a Java web application

Thanks to Maven this task is extremely easy and is reduce to a simple command:

```
mvn archetype:generate
-DgroupId=org.tutorial.ws
-DartifactId=langDetectorService
-DarchetypeArtifactId=maven-archetype-webapp
-Dversion=1.0-SNAPSHOT
```

The interactive creation plugin will ask you to confirme the configuration you defined (ie. groupID, artifactID, archetype and version). Just type "y" and valid. The plugin will tehn generate the following fodlers and files:

```
langDetectorService/
|-- pom.xml
`-- src
    `-- main
        |-- resources
        `-- webapp
            |-- WEB-INF
            |   `-- web.xml
            `-- index.jsp
```

[4] http://maven.apache.org/
[5] http://download.forge.objectweb.org/weblab/repository.zip

Now we will modify the XML file called `pom.xml` that contains information about the project and configuration details used by maven to build the project.

Open the file `langDetectorService/pom.xml`, which should look like this:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.tutorial.ws</groupId>
  <artifactId>langDetectorService</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>langDetectorService Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>langDetectorService</finalName>
  </build>
</project>
```

Listing 3.1: The original pom.xml file

To adapt this file to our needs, you should add the dependency to the WebLab libraries. For this, you can copy the lines provided below into the pom file at the root level under `<modelVersion></modelVersion>` markups. It will enables Maven to load the proposed jars and all its dependencies.

```xml
 [...]
  <parent>
    <groupId>org.ow2.weblab.webservices</groupId>
    <artifactId>parent</artifactId>
    <version>1.2</version>
  </parent>
 [...]
```

Now we have to add dependencies of the language detection API we will use. We also add a standard logging and the WebLab RDF-Helper-Jena which provides a set of functionnalities to annotate WebLab resources with RDF/xml.

To do this, you can copy the lines provided below under the `<dependencies></dependencies>` markups. It will enables Maven to load the jar librairies and all its dependencies.

```
[...]
 <dependency>
   <groupId>de.spieleck.app.ngramj</groupId>
   <artifactId>cngram</artifactId>
   <version>1.0-0.060327</version>
 </dependency>
 <dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-log4j12</artifactId>
   <version>1.5.6</version>
   <scope>runtime</scope>
 </dependency>
 <dependency>
   <groupId>org.ow2.weblab.core.helpers</groupId>
   <artifactId>rdf-helper-jena</artifactId>
   <version>1.3</version>
 </dependency>
[...]
```

The final POM file looks like the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.ow2.weblab.webservices</groupId>
    <artifactId>parent</artifactId>
    <version>1.2</version>
  </parent>

  <groupId>org.tutorial.ws</groupId>
  <artifactId>langDetectorService</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>

  <name>langDetectorService Maven Webapp</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
```

```
        <groupId>de.spieleck.app.ngramj</groupId>
        <artifactId>cngram</artifactId>
        <version>1.0-0.060327</version>
      </dependency>
      <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.6</version>
        <scope>runtime</scope>
      </dependency>
      <dependency>
        <groupId>org.ow2.weblab.core.helpers</groupId>
        <artifactId>rdf-helper-jena</artifactId>
        <version>1.3</version>
      </dependency>
    </dependencies>

    <build>
      <finalName>langDetectorService</finalName>
    </build>
</project>
```

Listing 3.2: The final pom.xml file

The next step is the actual implementation of the service. In the WebLab platform, services interfaces have been standardized in a small set of generic interfaces that rely on the same data exchange model. They define the method that are used by the multiple components that can be used in a multimedia processing chain (see on WebLab documentation on the wiki[6] for more information). Thus a “WebLab-compliant” service need to implement one of these generic interfaces.

For our new service, we will use the Analyser interface. A UML description of this interface can be found in WebLab documentation whereas the WSDL itself can be found in the source repository. This WSDL has been used to generate the JAVA interface (through JAXWS) which will be used in our example. It includes the JAVA classes associated with elements of the data exchange model.

3.4 Implementation of the Web service class

The implementation itself will be quite easy. Our new service is an `Analyser`, so we have to implement this interface which force us to add only one unique method. It is called to parse resources and named `process(ProcessArgs args)`. It receive a `Resource` as input and answer a `Resource` as output (the `ProcessArgs` could optionally contain an extra object called `UsageContext` but it will be ignored for this tutorial).

[6] http://weblab-project.org

First you need to create the sources folder where you will put the java class. Simply create the folder `/src/main/java` in the root folder of your project

```
mkdir your-service-path/src/main/java
```

Note that this is imposed by the structure of a maven project[7]. Keeping on this structure will ease the packaging and delivery through maven.

Then you need to create a class in the newly created source folder. The listing below gives an example of implementation.

```java
package org.ow2.weblab.tutorial;

import java.io.IOException;
import java.util.List;

import javax.jws.WebService;

import org.apache.commons.logging.LogFactory;
import org.ow2.weblab.core.extended.factory.AnnotationFactory;
import org.ow2.weblab.core.extended.ontologies.DublinCore;
import org.ow2.weblab.core.extended.util.ResourceUtil;
import org.ow2.weblab.core.helper.PoKHelper;
import org.ow2.weblab.core.helper.RDFHelperFactory;
import org.ow2.weblab.core.model.Annotation;
import org.ow2.weblab.core.model.Document;
import org.ow2.weblab.core.model.Resource;
import org.ow2.weblab.core.model.Text;
import org.ow2.weblab.core.services.AccessDeniedException;
import org.ow2.weblab.core.services.Analyser;
import org.ow2.weblab.core.services.ContentNotAvailableException;
import org.ow2.weblab.core.services.InsufficientResourcesException;
import org.ow2.weblab.core.services.InvalidParameterException;
import org.ow2.weblab.core.services.ServiceNotConfiguredException;
import org.ow2.weblab.core.services.UnexpectedException;
import org.ow2.weblab.core.services.UnsupportedRequestException;
import org.ow2.weblab.core.services.analyser.ProcessArgs;
import org.ow2.weblab.core.services.analyser.ProcessReturn;

import de.spieleck.app.cngram.NGramProfiles;

/**
 * Basic service that annotate Text unit with the 2-char code of the detected
 * language
 * @author WebLab team
 */
@WebService(endpointInterface = "org.ow2.weblab.core.services.Analyser")
```

[7] http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

```
public class LangDetectorService implements Analyser {

  private NGramProfiles nps = null;
  private NGramProfiles.Ranker ranker = null;

  /**
   * Simple constructor that initiate the language profiles for NGramJ
   */
  public LangDetectorService() {
    try {
      nps = new NGramProfiles();
      ranker = nps.getRanker();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  @Override
  public ProcessReturn process(ProcessArgs args)
      throws AccessDeniedException, ContentNotAvailableException,
      InsufficientResourcesException, InvalidParameterException,
      ServiceNotConfiguredException, UnexpectedException,
      UnsupportedRequestException {
    // check we received a valid request
    checkArgs(args);
    // Get text units in the Document received
    List<Text> texts =
      ResourceUtil.getSelectedSubResources(args.getResource(),
          Text.class);
    StringBuilder sb = new StringBuilder();
    Resource res = args.getResource();
    // Concatenation of the document text sections
    for (Text text : texts) {
      if (text.getContent() == null || text.getContent().isEmpty()) {
        LogFactory.getLog(this.getClass())
            .debug("Text '" + text.getUri()
                + "' has no content; ignored.");
        continue;
      }
      sb.append(text.getContent());
      sb.append("\n\n\n"); // insert line break to separate successive texts
    }
    // detecting of the language of the text
    String textLanguage = this.detectLanguage(sb.toString());
    // integration of the annotation dc:language in the document
    Annotation annot = AnnotationFactory.createAndLinkAnnotation(res);
    PoKHelper pokH = RDFHelperFactory.getPoKHelper(annot);
    pokH.setAutoCommitMode(false);
    pokH.createLitStat(res.getUri(), DublinCore.LANGUAGE_PROPERTY_NAME,
        textLanguage);
    pokH.commit();
    LogFactory.getLog(this.getClass()).info(
        "Document '" + res.getUri()
            + "' annotated with the dc:language property : "
            + textLanguage);

    ProcessReturn pr = new ProcessReturn();
    pr.setResource(res);

    return pr;
  }

  /**
   * Detecting the language of the text content passed as a String with NGramJ
   * @param text : the text content
   * @return a string that contains the 2 char code for the detected language
   * or xx if the language is unkown
   */
  private String detectLanguage(String text) {
    String retVal = "xx";
    if (!ranker.equals(null)) {
      ranker.reset();
      ranker.account(text);
      NGramProfiles.RankResult res = ranker.getRankResult();

      if (res.getScore(0) > 0.1)
        retVal = res.getName(0);
    }
    return retVal;
  }

  /**
   * Check the request sent to the service, mostly for null content.
   * @param args the arguments sent to the service
   * @throws InvalidParameterException
   * @throws UnsupportedRequestException
   */
  private void checkArgs(final ProcessArgs args)
      throws InvalidParameterException, UnsupportedRequestException {
    if (args == null) {
      throw new InvalidParameterException("ProcessArgs was null.",
          "ProcessArgs was null.");
    }
    final Resource res = args.getResource();
```

```
      if (res == null) {
        throw new InvalidParameterException("Resource was null.",
             "Resource was null.");
      }
      if (!(res instanceof Document)) {
        throw new UnsupportedRequestException(
             "Resource was not a document.",
             "Resource was not a document.");
      }
   }
}
```

Listing 3.3: Implementation of the Language detector Service.

The web service part of this class is simply managed through the annotation `@WebService` and the implementation of the `Analyser` interface which also hold extra annotations to describe the methods exposed as web service (more information about these annotations could be found in JAXWS documentation[8]).

The process implemented is also easy to understand: it uses the classes of the model to exploit the method argument (here a ProcessArgs as defined in the WSDL and extended from the service interface).

First the process defined will test if the `Resource` in the `ProcessArgs` is a `Document` and then extract the text content of this document. After that, the text is sent to 'cngrmj' API which returns its proposed language. Finally, the language code (en, fr, es...) is integrated in the resource as value of the `dc:language` property using the WebLa RDF Helper.

The standard `web.xml` (located in `your-service-path/WEB-INF`) file has to be configured to describe the URL which are listened by our application and the classes that will handle this listening process. In our case, the classes will be the one from JAXWS and thus the file will in most case be the same. However, we should define the name of our service in it as presented below :

```
<web-app>
  <display-name>Tutorial web service sample.</display-name>
  <listener>
    <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</
         listener-class>
  </listener>
  <servlet>
    <servlet-name>langDetector</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>langDetector</servlet-name>
     <url-pattern>/*</url-pattern>
```

[8]http://java.sun.com/webservices/jaxws/docs.html

```
    </servlet-mapping>
    <session-config>
      <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

What you can see is that we associated the JAXWS servlet listener to a servlet named `langDetector` and configured a url-mapping. One more configuration file has to be created to fix the endpoint name: `sun-jaxws.xml` (in the same location). It will define the endpoint name and the service implementation which will be used by the listener to realize the process (thus associating the servlet listener to our class). This is again very simple as shown below :

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint name="langDetector" implementation="org.ow2.weblab.tutorial.
      LangDetectorService" url-pattern="/*" />
</endpoints>
```

So the code is ready, let's test it.

3.5 Compilation, packaging and test

The next step is to package our web service into a war file in order to deploy it on a web server and to test it. For this, you can use maven and type the below command :

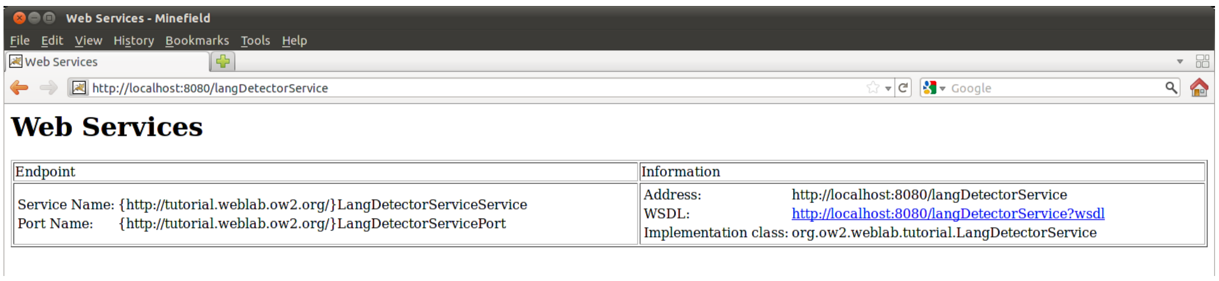
```
mvn package
```

This command will compile the project and create the war file of your web service under the directory `langDetectorService/target`. You can now deploy it on your tomcat server (we assume you used the tomcat web server provided with the demo) : simply copy the resulting WAR file from `langDetectorService/target/langDetectorService.war` to `<tomcat-home>/webapps/`. To ensure the service is correctly deployed on tomcat, check the log (open `<tomcat-home>/logs/catalina.out`) or go to the service welcome page URL `http://localhost:8080/langDetectorService`. If the service is correctly statred you should see :

To test this service, you have to install SoapUI (see SoapUI documentation[9]). Then simply :

1. Create a New soapUI project ;
2. Copy the URL of the service in the 'Initial WSDL/WADL' box (i.e. `http://localhost:8080/langDetectorService?wsdl`) ;
3. Edit the 'process' request and copy the request provided hereafter ;
4. Send the request and check the results.

[9]`http://www.soapui.org/`


Web Services - Minefield

File Edit View History Bookmarks Tools Help

http://localhost:8080/langDetectorService

Web Services

Endpoint	Information
Service Name: {http://tutorial.weblab.ow2.org/}LangDetectorServiceService Port Name: {http://tutorial.weblab.ow2.org/}LangDetectorServicePort	Address: http://localhost:8080/langDetectorService WSDL: http://localhost:8080/langDetectorService?wsdl Implementation class: org.ow2.weblab.tutorial.LangDetectorService

Figure 3.1: Welcome page of the service correclty deployed on tomcat.

```
<soapenv:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:analyser="http://weblab.ow2.org/core/1.2/services/analyser"
xmlns:model="http://weblab.ow2.org/core/1.2/model#" >
    <soapenv:Header/>
    <soapenv:Body>
        <analyser:processArgs>
            <resource xsi:type="model:Document" uri="weblab://aaa/1">
                <mediaUnit xsi:type="model:Text" uri="weblab://aaa/1#0">
                    <content>The WebLab platform is the generic name of the development
                        and execution environment
        platform provided by EADS in several research projects (Vitalas, WebContent, e-
            WokHub, Citrine...)
        involving the process of several types of media.</content>
                </mediaUnit>
            </resource>
        </analyser:processArgs>
    </soapenv:Body>
</soapenv:Envelope>
```

You can see here the result of this query, Fig 3.2:

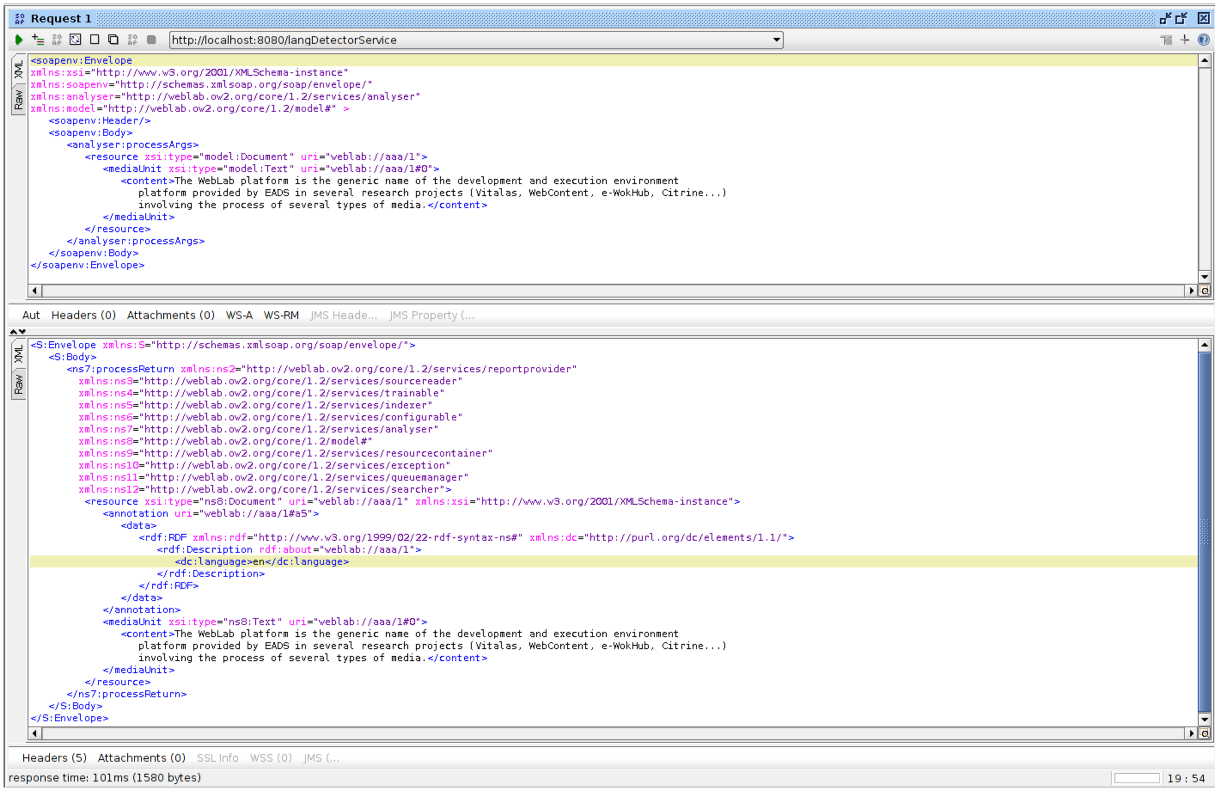
Moreover if you check the logs of your apache tomcat server (default logger write in `<tomcat-home>/logs/catalina.out`), you should see :

```
INFO [http-9080-2] (LangDetectorService.java:85) - Document 'weblab://aaa/1'
annotated with the dc:language property : en
```

So that's the end of the service tutorial. The next section will provide you some insight that may be useful when trying to go further and implement your own service.

3.6 Going further

After this tutorial is achieved, the next step is to implement your own service in accordance to any service specification. Hopefully, the procedure is really similar to the proposed



http://localhost:8080/langDetectorService

```
<soapenv:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:analyser="http://weblab.ow2.org/core/1.2/services/analyser"
xmlns:model="http://weblab.ow2.org/core/1.2/model#" >
   <soapenv:Header/>
   <soapenv:Body>
      <analyser:processArgs>
         <resource xsi:type="model:Document" uri="weblab://aaa/1">
            <mediaUnit xsi:type="model:Text" uri="weblab://aaa/1#0">
               <content>The WebLab platform is the generic name of the development and execution environment
                  platform provided by EADS in several research projects (Vitalas, WebContent, e-WokHub, Citrine...)
                  involving the process of several types of media.</content>
            </mediaUnit>
         </resource>
      </analyser:processArgs>
   </soapenv:Body>
</soapenv:Envelope>
```

Aut Headers (0) Attachments (0) WS-A WS-RM JMS Heade... JMS Property (...
XML
Raw

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns7:processReturn xmlns:ns2="http://weblab.ow2.org/core/1.2/services/reportprovider"
        xmlns:ns3="http://weblab.ow2.org/core/1.2/services/sourcereader"
        xmlns:ns4="http://weblab.ow2.org/core/1.2/services/trainable"
        xmlns:ns5="http://weblab.ow2.org/core/1.2/services/indexer"
        xmlns:ns6="http://weblab.ow2.org/core/1.2/services/configurable"
        xmlns:ns7="http://weblab.ow2.org/core/1.2/services/analyser"
        xmlns:ns8="http://weblab.ow2.org/core/1.2/model#"
        xmlns:ns9="http://weblab.ow2.org/core/1.2/services/resourcecontainer"
        xmlns:ns10="http://weblab.ow2.org/core/1.2/services/exception"
        xmlns:ns11="http://weblab.ow2.org/core/1.2/services/queuemanager"
        xmlns:ns12="http://weblab.ow2.org/core/1.2/services/searcher">
         <resource xsi:type="ns8:Document" uri="weblab://aaa/1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <annotation uri="weblab://aaa/1#a5">
               <data>
                  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:dc="http://purl.org/dc/elements/1.1/">
                     <rdf:Description rdf:about="weblab://aaa/1">
                        <dc:language>en</dc:language>
                     </rdf:Description>
                  </rdf:RDF>
               </data>
            </annotation>
            <mediaUnit xsi:type="ns8:Text" uri="weblab://aaa/1#0">
               <content>The WebLab platform is the generic name of the development and execution environment
                  platform provided by EADS in several research projects (Vitalas, WebContent, e-WokHub, Citrine...)
                  involving the process of several types of media.</content>
            </mediaUnit>
         </resource>
      </ns7:processReturn>
   </S:Body>
</S:Envelope>
```

response time: 101ms (1580 bytes) 19 : 54

Figure 3.2: Test of the langDetectorService using soapUI

tutorial so we will only provide you the information needed to your own `whatYouWant-Project`.

3.6.1 Building custom project

First of all, you need to build a new Maven project using your own names. So the command can be change to:

```
mvn archetype:create -DgroupId=anyGroupId
   -DartifactId=newNameOfTheProject
   -DarchetypeArtifactId=maven-archetype-webapp
```

Note that this will create a new folder which will be named after the project name and that the group and artifact ID will be in the `pom.xml`.

3.6.2 Using an IDE

There are plenty of maven plugins in order to enable the use of your service project in your favorite IDE (including Eclipse; Netbeans, JDeveloper and more ...). Since this tutorial tries to be "IDE-indenpendent" we do not document this part. Sorry but this is just out of the scope. Just check on the web for more information.

3.6.3 Custom implementation

To implement the service, just create a new class in the project (in the package you want) and make it implementing the right interface previously generated. In the tutorial it was `org.ow2.weblab.core.services.Analyser`, but if you change it, it may use another name (and probably package). Remember, that you also need to add the annotation concerning the web service implementation like:

```
@WebService(endpointInterface = "package.plus.name.of.interface")
```

The class is now specifying that it implements the service interface both to Java (with classing `implements` statement) and `jaxws` (with annotation). You should now implement the needed methods in accordance to the interface definition. This is where you can do "what you want". Which means that this is the place of your code which can call your own external applications, use your own libraries... Just make sure that it achieve the right process.

When you develop your own service, there are two possibilities:

- add your own code to the existing project created with the tutorial procedure. It means that your business classes will be directly in the project. Thus you can easily package the application.
- package your code in a JAR. It will allow your code to be independent from the service implementation and thus to be easily reusable in other application. However, you should follow the procedure to add correctly your jar as a dependency in the project (see hereafter).

In both cases, `SoapUI` will only test that the communication with the service is valid (i.e. it can handle the arguments and provide response) and that the arguments and output are valid or not against the data exchange model. However no validation can be made on the content of the response to test if the process that should be done by your service has been correctly done. You should either valid the results manually or test it outside the `soapUI` (using `JUnit` for instance).

3.6.4 Adding dependencies to the project

If your project depends on external libraries (classic Java libraries or your own libraries), two step are required to use the benefits of Maven. First you need to install the jar libraries into your local repository and then declare the dependence in your project POM file.

As explained in the Maven documentation, libraries are identified by their group id, artifact id and version number. Thus for each libraries you depends on you should choose those references. For instance if you have a `myLibrary.jar`, you can choose this configuration:

- `groupId` = `my.organisation.group`
- `artifactId` = `myLibrary`
- `version` = `1.0`

Then you have to install it in your local repository using the Maven command:

```
mvn install:install-file -Dfile=path/to/your/jar/myLibrary.jar
  -DgroupId=my.organisation.group -DartifactId=myLibrary
    -Dversion=1.0 -Dpackaging=jar -DgeneratePom=true
```

Then, add the following dependency to the POM file of your project:

```
  [...]
 <dependency>
    <groupId>my.organisation.group</groupId>
```

```
        <artifactId>myLibrary</artifactId>
        <version>1.0</version>
    </dependency>
    [...]
```

Finally, run the following command to ensure that Maven correctly take this new dependency in account:

```
mvn clean compile
```

If you have several dependencies, you can install all of them, add all dependencies to the POM file and execute only one time the previous command.

⚠ If the JAR you want to install depends itself on other libraries, the easiest way is to include them directly in the JAR itself. The other procedure can be to convert your entire project to the Maven structure, but this task is out of the scope of this tutorial. However, as already mentioned, one can look into Maven documentation for more information on Maven dependencies management.

⚠ You may have set maven to work "offline" if you follow the steps proposed in the tutorial. It may be time to go back online and configure some repositories if you want to use extra dependencies.

3.6.5 Packaging and tests

Nothing change here except that the provided arguments should be in accordance with the new interface implemented (and perhaps your own service pre-conditions).

3.6.6 Delivery

To deliver your component embedded in a web service, you have to provide of course the WAR file, ready to be deployed on Tomcat application server. However some more data are needed to ensure it will be integrable easily.

A complete description of the services provided by the component is needed in order to build the correct processing chain without misunderstanding on any component functionality. A dedicated sheet is provided and should be filled by partners for each components they will develop.

Then a test kit should be provided in order to be able to replay the internal tests that have been made to validate the component functionality. It involves data to build requests and valid the results of each service methods:

- SOAP request samples,
- SOAP responses (one for each requests),
- configuration procedures,

- testing data (i.e. multimedia content for most of the services) provided in input and/or expected results.

Without those information and data, the integration of any component can be very time consuming and one should take car on provided the right information in them.

3.6.7 Extra documentation

For more information, follow the links hereafter:

- Service interfaces:
  http://weblab-project.org/index.php?title=WebLab_services_interfaces
- Data exchange model:
  http://weblab-project.org/index.php?title=WebLab_data_exchange_model
- Document structure:
  http://weblab-project.org/index.php?title=WebLab_Structure_Representation
- Document annotation:
  http://weblab-project.org/index.php?title=WebLab_Document_Annotation
- Anything:
  http://weblab-project.org/forum/yabb2/YaBB.pl

CHAPTER 4

Tutorial for the integration of a service in the WebLab demo

4.1 Goal of the tutorial

In this chapter we describe the integration of the web service developed above in a WebLab application. The objective is to integrate the language detection function in the processing chain of the demonstration application in order to add a language attribute to the crawled documents.
This feature is then used by the visualisation portlet to display a 'country flag' based on the language detected next to each document in the result portlet. Next sections will detail :

1. Integration of the langDetectorService into petals ESB
2. Modification of the BPEL chain service unit
3. Re-deployment on the petals ESB
4. Tests of the new WebLab demo

Before going deeper in this chapter, some explanation about ESB, JBI, service assembly, service unit, etc. are needed. It the the goal of the next section.

4.2 What is an ESB

An ESB is a software bus used to integrate heterogeneous applications and applications parts. JBI is a Java specification used to integrate software components. Of course there are some ESB which implement JBI, and PEtALS is one of them.

An ESB is able to compose heterogeneous components using normalized installing, requesting and messaging paradigms.

To do this it uses:

- **Service Engines (SE)**: used to do business processes like XSL transformation, orchestration, composition, load balancing…
- **Binding Components (BC)**: used to get inside and outside the bus. For example a MAIL BC, a SOAP BC, an XMPP BC…
- **Endpoints**: used for service generalisation, each endpoint can be found and requested through the bus.
- **Service Units (SU)**: used to configure SE and BC, each SU is linked to an SE or BC.
- **Service Assembly (SA)**: deployable in the bus, used to configure SUs.

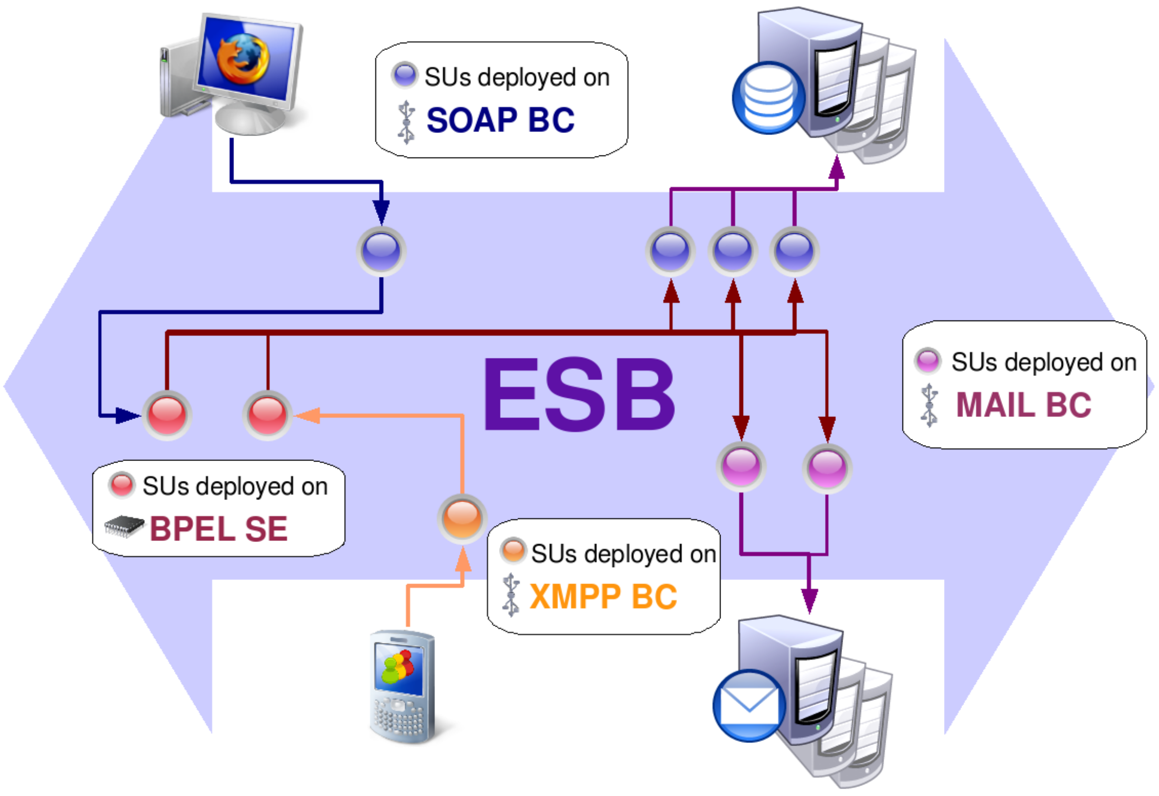


Figure 4.1: ESB architecture example.

In this guide you are going to create SUs, package them in a SA and deploy this SA in the bus. The figure 4.1 presents the bus capabilities:

- BCs provide protocol abstraction (multiples client / component integration possibilities);
- Endpoint provide service abstraction, reusing and composition;

- Service composition provides flexibility and scalability;
- Many others things that can't be listed here…

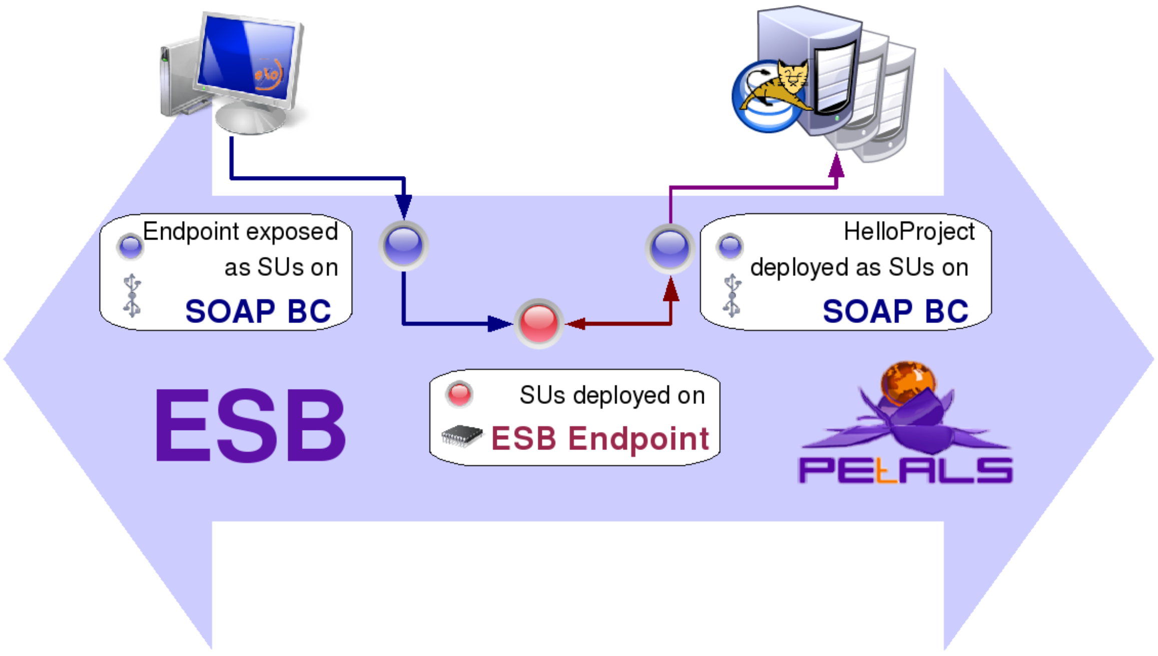


Figure 4.2: Architecture deployed during the integration on ESB.

4.3 Integration of the LangDetector Service

The first integration step is to expose the langDetectorService as an endpoint. To do this, we need to deploy a SU on the SOAP BC. When the SU will be deployed, it will create a new endpoint on the bus "connected" to the langDetectorService. It means that when the endpoint will be called, the langDetectorService will be threw the SOAP BC.

Go to your workspace directory and create an SU artefact using this command:

```
mvn archetype:create -DarchetypeGroupId=org.objectweb.petals
   -DarchetypeArtifactId=maven-archetype-petals-jbi-service-unit
     -DarchetypeVersion=1.0.0 -DgroupId=org.weblab.example
       -DartifactId=langDetectorProvideSU -Dversion=1.0-SNAPSHOT
```

This command creates the folder langDetectorProvideSU:

```
langDetectorProvideSU/
|-- pom.xml
`-- src
    `-- main
        |-- jbi
        |   `-- jbi.xml
        `-- resources
```

So the first thing to change is the generated `jbi.xml` file by the following one:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi  version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jbi="http://java.sun.com/xml/ns/jbi"
xmlns:petalsCDK="http://petals.ow2.org/components/extensions/version-4.0"
xmlns:soap="http://petals.ow2.org/components/soap/version-3.1"
xmlns:analyser="http://weblab-project.org/services/analyser">
<!-- Import a Service into PEtALS or Expose a PEtALS Service => use a BC. -->
<jbi:services  binding-component="true">
<!-- Import a Service into PEtALS => provides a Service. -->
<jbi:provides  interface-name="analyser:Analyser"
        service-name="analyser:LanguageDetector" endpoint-name="lg1">

        <petalsCDK:wsdl>wsdl/Analyser.wsdl</petalsCDK:wsdl>
        <soap:address>http://localhost:8080/langDetectorService
        </soap:address>
        <soap:synchronous-timeout>0</soap:synchronous-timeout>
        <soap:mode>SOAP</soap:mode>
    </jbi:provides>
</jbi:services >
</jbi:jbi >
```

This file is the service unit configuration file and as you can see, the only parameter which depend on the service are the service URL and the WSDL URL.

So using this file, the SOAP BC is able to know that when the endpoint

PEtALSneeds the WSDL file to correctly deploy the service unit. So you have to copy `langDetectorService/src/wsdl` directory into `src/main/jbi`.

Edit the `Analyser` WSDL file located in `langDetectorProvideSU/src/main/jbi/wsdl`. You can now copy these lines at the end of the document before the `</definitions>` markup. It is the definition of our new service.

```
         [...]
        <service name="LanguageDetector">
    <port name="LangPort" binding="tns:AnalyserSOAPBinding">
            <soap:address location="http://www.example.org/"/>
    </port>
        </service>
         [...]
```

Now, we have to install it in our local repository, to do this just type this command in the `langDetectorProvideSU` folder:

```
mvn clean install
```

Now to deploy it on the bus, you need to create a service assembly which contains this service unit. To do this type this command in your workspace:

```
mvn archetype:create -DarchetypeGroupId=org.objectweb.petals
   -DarchetypeArtifactId=maven-archetype-petals-jbi-service-assembly
     -DarchetypeVersion=1.0.0 -DgroupId=org.objectweb.example
       -DartifactId=langDetectorProvideSA -Dversion=1.0-SNAPSHOT
```

This command creates the folder `langDetectorProvideSA`:

```
langDetectorProvideSA/
|-- pom.xml
`-- src
    `-- main
        |-- jbi
        |   `-- jbi.xml
        `-- resources
```

First we have to add a dependency to the newly created service unit (`langDetectorProvideSU`) in the `pom.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- =============== -->
  <!-- Identification -->
  <!-- =============== -->
  <name>SA :: langDetectorProvideSA</name>
  <artifactId>langDetectorProvideSA</artifactId>
  <groupId>org.weblab.example</groupId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jbi-service-assembly</packaging>
  <description>langDetectorProvideSA description.</description>

  <!-- ============ -->
  <!-- Dependencies -->
  <!-- ============ -->
  <dependencies>
    <dependency>
      <groupId>org.weblab.example</groupId>
      <artifactId>langDetectorProvideSU</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>jbi-service-unit</type>
    </dependency>
  </dependencies>
  <!-- ===== -->
  <!-- Build -->
  <!-- ===== -->
  <build>
    <plugins>
      <plugin>
        <groupId>org.objectweb.petals</groupId>
        <artifactId>maven-petals-plugin</artifactId>
        <version>1.0.0</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>

</project>
```

The next step is to edit the `jbi.xml` file and copy this content:

```
<?xml version="1.0" encoding="UTF-8"?>

<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <service-assembly>
    <identification>
      <name>langDetectorProvideSA</name>
      <description>langDetectorProvideSA description</description>
    </identification>
    <service-unit>
      <identification>
        <name>langDetectorProvideSU</name>
        <description>langDetectorProvideSU description</description>
      </identification>
      <target>
        <artifacts-zip>langDetectorProvideSU-1.0-SNAPSHOT.zip</artifacts-zip>
        <component-name>petals-bc-soap</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

Take care about the correctness of the name in the identification part. It must match the name of the service unit previously created.
Before deploying this service assembly on the bus, we must package it. Go to the `langDetectorProvideSA` folder and type:
`mvn clean package`

4.4 Modification of the BPEL chain SU

Waiting for update

4.5 Modification of the visualisation portlet

Waiting for update

4.6 Deployment on the petals ESB and tests

Waiting for update

CHAPTER 5

Conclusion