Document type PUB = public

**ABSTRACT:** Guidelines for development of services and portlets using the WebLab Core as a baseline.

**KEYWORD LIST:** Development guide, architecture, technical baseline, web services, WebLab, orchestration, ESB, portlets
## Modification control

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2007-10-01</td>
<td>Creation</td>
</tr>
<tr>
<td>0.2</td>
<td>2007-10-19</td>
<td>First version of tutorial for the project WebContent</td>
</tr>
<tr>
<td>0.5</td>
<td>2007-10-22</td>
<td>Corrections</td>
</tr>
<tr>
<td>1.0</td>
<td>2007-10-26</td>
<td>First version of tutorial for the project Vitalas</td>
</tr>
<tr>
<td>1.1</td>
<td>2007-11-30</td>
<td>Document reorganisation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WebLab generalisation (project independence)</td>
</tr>
<tr>
<td>1.2</td>
<td>2007-12-20</td>
<td>Corrections</td>
</tr>
<tr>
<td>2.0</td>
<td>2008-04-01</td>
<td>1\textsuperscript{st} complete version</td>
</tr>
<tr>
<td>2.1</td>
<td>2009-01-23</td>
<td>Adaptation to model 1.0 (was previously on 0.7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Adding the Orchestration chapter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Change of portal, from eXo to Liferay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Many corrections.</td>
</tr>
</tbody>
</table>
## CONTENTS

4.4 Going beyond the tutorial ........................................... 42
  4.4.1 Building custom project ...................................... 43
  4.4.2 Customising the POM ........................................... 43
  4.4.3 Code generation for other WSDL .............................. 43
  4.4.4 Custom implementation ....................................... 43
  4.4.5 Adding dependencies to the project ......................... 44
  4.4.6 Packaging and tests .......................................... 45
  4.4.7 Delivery ...................................................... 45
4.5 The annotation data field problem (in Java) ................... 46

5 Portlet Development .................................................. 47
  5.1 Overview on Portlet development ................................ 47
  5.2 Technical steps .................................................. 48
  5.3 Overview of Portlets technology ................................. 48
    5.3.1 Portals and Portlet architecture .......................... 48
    5.3.2 Portlet vs Servlets ......................................... 49
    5.3.3 Portlet Life Cycle .......................................... 50
  5.4 First portlets in Java ............................................ 50
    5.4.1 Creating a java web application scheme ..................... 51
    5.4.2 Implementation .............................................. 55
    5.4.3 Compilation and packaging .................................. 65
    5.4.4 Deployment, validation and test ............................ 67
  5.5 Going beyond the tutorial ...................................... 69
    5.5.1 Building custom project .................................... 70
    5.5.2 Customising the POM ....................................... 70
    5.5.3 Code generation for other WSDL ........................... 70
    5.5.4 Custom implementation ..................................... 70
    5.5.5 Adding dependencies to the project ......................... 71
    5.5.6 Packaging and tests ........................................ 71
    5.5.7 Dispatch to JSP or servlet .................................. 72
    5.5.8 Resource serving ............................................ 73

6 Integration Guide .................................................... 74
  6.1 PEtALS integration .............................................. 74
    6.1.1 What is an ESB? ............................................. 74
    6.1.2 Endpoint creation .......................................... 76
    6.1.3 Exposing an endpoint to the outside ....................... 83
  6.2 Portlet integration .............................................. 86
    6.2.1 Updating portlets to connect to services exposed in the ESB . 86

7 Orchestration .......................................................... 88
  7.1 BPEL overview .................................................... 88
  7.2 Endpoint configuration .......................................... 88
    7.2.1 Scope of the orchestration tutorial ....................... 88
    7.2.2 Installing the Orchestra service engine .................. 88

2009-01-23 4/114
CHAPTER 1

Introduction

1.1 Objectives

The “WebLab platform” is the generic name of the development and execution environment platform provided by EADS in several research projects (Vitalas, WebContent, e-Wok Hub, Citrine...) involving the process of several types of media.

This document is composed of 8 chapters. It can be read by anyone interested in the process of web services or portlets development since it browses many of web services and portlets features.

One can note that this tutorial is not intended to be read by non-programmer since it relies on (not so) strong basic capability in programming languages and web application architecture.

This document is composed of the 8 following chapters:

- Introduction to the WebLab Development Kit (current chapter),
- Architecture of the WebLab platform,
- Tutorial overview and description of the development kit content,
- WebLab services development tutorial,
- WebLab portlets development tutorial,
- WebLab services and portlets integration tutorial,
- WebLab services orchestration tutorial,
- General tools installation informations.
1.2 RELATED DOCUMENTS

1.2 Related documents

- Evolution request sheet
- Component description sheet

1.3 Development guide pictorial semantic

⚠️ Focus on a critical step.

ℹ️ Use to highlight some information.

⚠️ Step reserved to Linux users.

⚠️ Step reserved to Windows users.

1.4 Vocabulary and definitions

In the following chapters some ambiguous terms will appear. Here is the reference for their sense in the context of this document:
## 1.4. VOCABULARY AND DEFINITIONS

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application programming interface.</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language which is an XML language using to orchestrate services (W3C standard).</td>
</tr>
<tr>
<td>Component</td>
<td>Piece of software developed by a partner and which should be integrated into the platform.</td>
</tr>
<tr>
<td>ESB</td>
<td>Enterprise Service Bus which is a software infrastructure to connect a set of heterogeneous software components.</td>
</tr>
<tr>
<td>Orchestrator</td>
<td>Component dedicated to the orchestration of services and thus the definition of processing chain.</td>
</tr>
<tr>
<td>POM</td>
<td>Project Object Model, the fundamental unit of work in Maven.</td>
</tr>
<tr>
<td>Service</td>
<td>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 be a part of a processing chain and requires some pre-processing.</td>
</tr>
<tr>
<td>Service engine</td>
<td>Software component embedded in the ESB which realises technical process such as distribution or orchestration.</td>
</tr>
<tr>
<td>Service interface</td>
<td>Definition of the input and output data for each method proposed by a service (see service interface definitions).</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol (W3C standard).</td>
</tr>
<tr>
<td>Technical service</td>
<td>Service which could be embedded or not in the ESB and providing a common technical functionality used by most of the services.</td>
</tr>
<tr>
<td>WebLab</td>
<td>Name of the platform composed among other by the ESB, the data exchange model and the portal.</td>
</tr>
<tr>
<td>Web server</td>
<td>A server which can handle HTTP request/response and thus host web services.</td>
</tr>
<tr>
<td>Web service</td>
<td>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.</td>
</tr>
<tr>
<td>WSDL</td>
<td>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).</td>
</tr>
<tr>
<td>XQuery</td>
<td>W3C standard defined to Query XML data.</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition (W3C standard).</td>
</tr>
</tbody>
</table>
CHAPTER 2

Architecture

This chapter presents some basic idea about the architecture of the WebLab platform. Developers of components should read it carefully. The main ideas should be very useful while designing and/or building a component or converting an existing application into a deployable service on the platform.

2.1 Architecture guidelines

Regarding the scope of the WebLab platform, the architecture is “Service Oriented” (SOA).

Each component that could be integrated into the platform shall implement one or several functionalities that are described by service interfaces. The function workflow needed to provide user applications will be done by putting together services able to communicate.

Each component, implementing one or several services interfaces won’t have any knowledge of the other services and their capabilities. They will provide to the other one or several processing capabilities which will be driven by the orchestrator to define the business processes.

As a consequence, the service definition and conception is a key step in the platform. The granularity of the services should be one of the main concerns during the design and development of a WebLab component.

In order to provide a flexible architecture, the service design should handle the following features:

- *Loosely-coupled*: It means that services should be as autonomous as possible and that dependencies between components should be avoided. Communication between services should be done using standards.
2.2. MAIN ARCHITECTURE

- **Coarse-grained**: A service should provide a coherent set of functions and should hide implementation complexity.

- **With well defined interfaces**: A service should implement one or several interfaces giving access to functions it is able to deal with.

- **Composable**: A service can be easily integrated in a global application and can interact with other services.

Every service is provided by a “producer” to a “consumer”. The interaction between producer and consumer is carried out by a software bus that is responsible of the mediation and the communication between the services. As a consequence, a consumer invokes a service to carry out a function of which he is the producer.

A directory of service is provided in the platform so as to allow the producers to publish their service offers and the consumers to know them.

A service call is done by sending a message through the software bus from the consumer to the publisher either in an asynchronous or a synchronous way.

In the architecture, we will consider:

- **Business services** that provide business functions (such as video segmentation, text clustering),

- **Technical services** that are part of the baseline provided (such as security, data access layer, etc.),

- **GUI services** that will interact with users on one side and with the service bus on the other side to request process or data.

Business applications and use cases realisations shall be implemented by combining business services and technical services within the same process thanks to orchestration tools.

2.2 Main architecture

The figure 2.1 (page 11) provides a full overview of the WebLab platform architecture based on SOA.

2.2.1 Description of each layer and each module

The general architecture can be seen as a multi-layer architecture (see figure 2.1). Having a top down view, we can describe the architecture with the following layers:

- An user layer providing access to:
  - specific applications such as batch processing functions,
2.2. MAIN ARCHITECTURE

- graphical user interfaces that may be included in a web portal. This enable the user to access services in an hidden way through a simple web browser: the activation of processing chain is then called,
- business process design application such as a workflow editor enabling to define service chain and tools to support deployment, and configuration,
- administration and monitoring application that enable to control the execution of the processing chains and to check the correct execution of the processing chains (service availability, lack of errors, etc.).

- An orchestration layer that is responsible for acting as an intermediary between services in order to perform a business process. Orchestration scripts can be defined by an external graphical tool that generates process description in languages such as BPEL or XPDL. The orchestration layer embeds an execution engine that is able to run process description in the previous languages.
- A communication and distribution layer able to route messages between services.
- A business services layer composed of the services that will be developed for the platform and covering the complete information lifecycle.
- A technical services layer composed of the element presented above: semantic entity storage, service directory, security and supervision.
2.2. MAIN ARCHITECTURE

2.2.2 Exchange data model

As components will share data through the workflow, a common data exchange model must be designed. It will be used to define a common exchange format, and thus the processing services could be easily chained: a producer service will encode its result in the format and provide them to a consumer service which will then be able to process them. The orchestration will then be rationalised since it does not need to develop specific interfaces between each service. The unique data format will allow reducing the computational effort on model. The service chaining task will also be simplified and the introduction of new services will not need too much adaptation.

Figure 2.2: Data exchange model version 1.0.

One can note that the data model (see figures 2.2 and 2.3 [pages 12 and 14]) will allow describing the structure and parts of content of any document without defining a new document format. The main issue is to build a description which permit to each service to easily process the document. Thus the model must handle a description of the document structure in order to provide easy access to any section or sub-section of it, in accordance to any segmentation strategy. Then, an annotation mechanism should be designed in order to provide metadata on any of those sections. Those metadata could be from high to very low level of granularity. They could be processed by other services
2.2. MAIN ARCHITECTURE

(i.e. for low level picture description feature) or directly available to the end-user (i.e. for semantic annotation).

However it should not provide any description of the rendering features of the original document (such as bold and italic for text document) when it is not related to information extraction tasks. The model is also not intended to transport data, such as the original content.

Its main capabilities should be:

- Allowing to describe different views of the same document or part of document without necessarily involving the full document content.
- Adaptability and extensibility to any multimedia document.
- Allowing to annotate the document content with various level of abstraction (low level feature to high semantic annotation) through an annotation framework, interoperable with standards.
- Scalable to very large document corpus.

Thus the recent document models used in state-of-the-art tools — IPTC/EXIF, MPEG7, FDL, KLV or MXF — will not be well adapted to the platform scope. Most of them are limited to certain document types (i.e. pictures for IPTC), does not provide a sufficient level of abstraction for annotation and/or are intended to transport content. However, theses initiatives should be taken into account and conversion model should take those multimedia standard initiatives as a starting point and interoperability with some of them should also be studied. The technical support will be able to also integrate XSLT sheets or conversion services from and toward these various formats. One can note that other issues will arise on the content exchange and storage format and the standards could be involved to resolve them.

For technical purposes, the model grammar will be expressed through an XML-scheme (XSD files) and the service interface definition (WSDL files). Thus data could be exchange through the SOAP messages since the model is based on XML. This reference will describe the object classes which will enable the description of any data type and their respective annotation.
2.2. MAIN ARCHITECTURE

Figure 2.3: Data exchange model with packages.

2.2.3 Document example in the exchange model

Figure 2.4: Document example.

This is introduction
First Section
This is a text:
Conclusion
This is a very simple example
2.2. MAIN ARCHITECTURE

Its representation in accordance to the data exchange model should be as presented in the following figure 2.5:

Figure 2.5: UML object model describing the document.

It will lead to the following XML serialisation that follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<mediaUnit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="wl:document" synchronised="false" uri="weblab://test/doctest0">
  <mediaUnit xmlns:text="http://weblab-project.org/core/model/text"
xsi:type="text:text" uri="weblab://test/doctest0#0">
    <content>This is an introduction.</content>
  </mediaUnit>
  <mediaUnit xsi:type="wl:composedUnit" synchronised="false" uri="weblab://test/doctest0#1">
    <mediaUnit xmlns:text="http://weblab-project.org/core/model/text"
xsi:type="text:text" uri="weblab://test/doctest0#1-0">
      <content>First section.</content>
    </mediaUnit>
    <mediaUnit xmlns:text="http://weblab-project.org/core/model/text"
xsi:type="text:text" uri="weblab://test/doctest0#1-1">
      <content>This is a text.</content>
    </mediaUnit>
  </mediaUnit>
  <mediaUnit xsi:type="wl:composedUnit" synchronised="false" uri="weblab://test/doctest0#2">
    <mediaUnit xmlns:text="http://weblab-project.org/core/model/text"
xsi:type="text:text" uri="weblab://test/doctest0#2-0">
      <content>Conclusion.</content>
    </mediaUnit>
    <mediaUnit xmlns:text="http://weblab-project.org/core/model/text"
xsi:type="text:text" uri="weblab://test/doctest0#2-1">
      <content>This is a very simple example.</content>
    </mediaUnit>
  </mediaUnit>
</mediaUnit>
```

Listing 2.1: XML example of a simple document representation.

Annotated documentation
2.2. MAIN ARCHITECTURE

Listing 2.2: XML example of an annotated document representation.

Multimedia Document
2.2. MAIN ARCHITECTURE

2.2.4 Data exchange model updates

The data exchange model has evolved and minor changes must be taken into account in order to process correctly the content and to produce valid service answer. Figure 2.2 (page 12) shows the current version. Model changes are:

- Version 1.0

- WebLabR1 modification to be more generic (xs:anyURI).
2.3 IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

- Resource modification: Resource can now optionally contain a list of LowLevelDescriptor.
- LowLevelDescriptor (previously in multimedia.xsd) modification: Feature have now optionally a label and handles directly a list of simple types. They are now described in model.xsd.
- Query (query.xsd) modification: Specialisation of each type of queries (Structured, Semantic, Composed, Universal, FullText and Similarity).
- In interfaces, modification of URI parameters to be more generic (xs:anyURI).
- The samples take in account new interfaces definition.
- Creation of the WebLab ontology providing some specific properties for the annotable (Resource) part of the model.

- Version 0.7
  - Renaming LargeContentMediaUnit to BinaryMediaUnit.
  - Set the string content of Text unit optional.
  - Adding a regex restriction on URI for resources in accordance to the URI and WRI specification.
  - Adding Content object to handle any kind of essence content within Content package.
  - Adding BinaryContent as sub-class of Content to handle raw bytes arrays content within Content package.
  - Adding TextContent as sub-class of Content to handle string content within Content package.
  - Change the list item of ResourceCollection from resources to resource.

- Version 0.6
  - Suppression of Position and creation of various Segment specialisations instead (TemporalSegment, LinearSegment, SpatialSegment)
  - Suppression of shapes for SpatialSegment
  - Parent link suppression (not visible on illustration but was present in XSD)
  - URI assignment guidelines (discussed in the next section)

Complete evolution request sheet are available on demand for detailed explanation on those changes. Further evolutions are under study and anyone who want to suggest an evolution is invited to fill an evolution request sheet.

2.3 Implementation recommendations and best practices

This section will provide some useful advises and guidelines for the development of web service and moreover for the development of component included in the platform.
2.3. IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

2.3.1 The “URI” thing or how to handle “MediaUnit”

Many issues have been raised through the URI handling especially on URI assignment. During any process chain of document, many changes can occur on a document. Many objects (MediaUnit, Annotation...) added/deleted to enrich the structural and semantic description of documents. Those changes will be made by the services and thus are only controlled by them. Regarding the multiple possibilities of process some recommendations are needed. However, one can note that those recommendations are not stable so far and that some changes may occur in the near future.

The main idea is to kept the URI valid against the last RFC (RFC3986) and to propose a restriction ensuring that URI over the WebLab platform are unique. This restriction defined the WRI (Weblab Resource Identifier) based on the URI syntax. A distinction is proposed between the URI of the ‘classic’ resources which will have an absolute URI and those which could be embedded in another Resource (see hereafter the explicit list of those resources). Absolute URI syntax is defined as:

\[
\text{URI} = \text{scheme } "." \text{ hier-part } [ "?" \text{ query } ]
\]

As presented, it is composed of two mandatory parts, scheme and hier-part, and an optional query part. The WRI restriction will impose to fix the scheme to `weblab` to ensure that this URI is defined in the WebLab platform scope. In the RFC, the hier-part syntax is composed of multiple optional parts as presented hereafter:

\[
\text{hier-part} = ["/" \text{ authority}] \text{ [path]}
\]

To keep it simple, the idea is that the URI should contain either an authority or a path (or both) and that a path may be defined through several ways. For our purpose the hier-part will be composed of an id based on authority syntax, called `id_ref` and a unique identifier of the resource under the responsibility of the authority. Finally the absolute WRI syntax is defined as:

\[
\text{absoluteWRI} = "weblab:="/\text{id_ref }"/\text{id_res}
\]

\[
\text{id_ref} = \text{authority}
\]

\[
\text{id_res} = \text{path-absolute [?query]}
\]

One can note that the complete syntax is based on URI syntax. Complete description is available in annex, but for the sake of simplicity, details on valid and invalid characters are not provided here. The scheme stand for the `weblab` character string itself. The `<id_ref>` will generally be assimilated as the identifier of the service that will fix the URI, following the authority syntax defined by URI recommendations. The `<id_res>` will be an identifier assigned by the authority to this resource respecting the syntax of path absolute optionally expanded by a query part as defined by the URI recommendations. There is no need that this path refer to an existing path (i.e. in a file system) in the authority scope, but this identifier must be unique in the authority scope (i.e. service scope). The following examples show some resources WRI assignment:

- `weblab://video.service/video23`
- `weblab://crawler.cluster.12/document129`
- `weblab://feedbacklearning.service/user?213@scope`
- `weblab://xmlrepository/corpus?pictures`
- `weblab://search-q2003b/results?queryid=another_uri`
- `weblab://wsd/persons.ontology`
- `weblab://formatting.service_89/video/cctv4:20071202?hour=1234`

\[^1\]http://gbiv.com/protocols/uri/rfc/rfc3986.html

2009-01-23
Some resources of the model could be embedded in other resources. Regarding the model, those are:

- MediaUnit and any sub class of MediaUnit (excepting Document itself),
- Annotation,
- Segment and any sub class.

One can note that any resource could be embedded in a resource collection, however this case will be addressed as an exception and only the resources cited in the previous list will be subject to a restriction of WRI. Thus the WRI of such objects will use the optional fragment part of URI to define a unique identifier of the resource in the scope of the container resource. The syntax will then be:

\[
\text{WRI} = \text{WRIOfContainer} "\#" \text{id\_res} \\
\text{WRIOfContainer} = \text{absoluteWRI} \\
\text{id\_res} = \text{fragment}
\]

The WRI of the container will be a valid absolute WRI of a resource and the \(<\text{id\_ref}>\) a unique identifier in the scope of the container resource, respecting the fragment syntax defined in URI recommendations. The following examples show some resources WRI assignment on embedded resources:

- \(\text{weblab://video.service/video23#audio/english?timestamp=2341}\)
- \(\text{weblab://crawler.cluster.12/document129#composedunit/text12}\)
- \(\text{weblab://feedbacklearning.service/user?213##scope#implicitinterest/cluster/picture?id=af6b}\)
- \(\text{weblab://search-q293b/results?queryid=another_uri#}\)
- \(\text{weblab://wsd/persons.ontology#concept/person/role/president/jose_e_dos_santos}\)
- \(\text{weblab://formatting.service_89/video/cctv4:20071202?hour=1234#segment_1feg}\)

Some samples of new XML serialisation of document and resources sing the recommended URI syntax are provided in appendix. Each service adding or modifying elements in a Resource is thus responsible for the validity of the WRI assigned to each new resource created. It means that absolute WRI must be unique in the authority scope and that fragment, when it is used, must be unique in the resource scope.

**URI grammar**

\[
\text{WRI} = \text{absolute-WRI} ["\#" \text{idSubRes}] \\
\text{absolute-WRI} = \text{"weblab" \:" idRef \:" idRes} \\
\text{idRef} = \text{authority} \\
\text{idRes} = \text{path-absolute} [?\text{query}] \\
\text{idSubRes} = \text{fragment}
\]

;description out of RFC 3986

; (bold variables are preferred when choices are possible)

\[
\text{authority} = [ \text{userinfo \:" @\:"} \text{host} [ ":\:" \text{port} ] \\
\text{path-absolute} = "/" [ \text{segment-nz} *( "/" \text{segment} ) ] \\
\text{query} = *( \text{pchar} / "/" / "?" ) \\
\text{fragment} = *( \text{pchar} / "/" / "#" ) \\
\text{userinfo} = *( \text{unreserved / pct-\text{encoded} / sub-delims / ":\:"} )
\]

\text{host} = \text{IP-literal / IPv4address / reg-name}
2.3. IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

```
2.3. IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

```

WRI regex

This regex should be tested against each new WRI to ensure their validity. Note that capturing group will allow to easily select part of the WRI which are relevant for a quick mapping to any internal identifier.

```
^weblab://([^/\?#\n]+)+/((([^/\?#\n][^?#\n]+)+)(\?([^#\n]+))?)+(#([^\n]+))?
```

Semantic of each capturing group:

- group 1: reference identifier including // at the beginning
- group 2: reference identifier
- group 3: resource identifier
- group 4: resource identifier without query part
- group 5: query part of resource identifier including ? at the beginning
- group 6: query part of resource identifier
- group 7: fragment part of WRI including # at the beginning
- group 8: fragment part of WRI
2.3. IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

2.3.2 Extending service interfaces

Some service interface will certainly need extensions and/or adaptation to specific needs or breakthroughs made during the project. For compatibility and stability reasons, the interfaces cannot be changed for every needs. A dedicated evolution request sheet will be provided in order to have a common model on evolution request. Those will be studied and discussed within the developer partners and decision on evolution will collaboratively taken.

Remember that components can also implemented several services. Actually the development kit contains the following interfaces (presented in figure 2.6 page 24):

- **Analyser** which is dedicated to services that can process **Resource** in order to alter its structure and/or its annotations.
- **Configurable** which enables the service to have some parameters to modify its behavior on its provided services.
- **Trainable** which is dedicated to services than need to be trained on a set of **Resource**.
- **Indexer** allows to index **Resource**.
- **Searcher** allows to search for **Resource** with a **Query**.
- **ContentProvider** which allow to retrieve a **Content** object as defined in the model, which can contains the essence of a **MediaUnit**.
- **ContentConsumer** which allows services to work on content that have been previously setted.
- **ResourcesContainer** which defines a method that allow the service to store **Resource** and provide a simple get functionality.
- **ReportProvider** which allows to build a **Document** based on a set of **Resources**.

Based on those interfaces, it contains also some service example that include one or more of this generic interfaces. Those are defined in WSDL files which depends on the interfaces WSDL. Note that the functionalities addressed by this services should be accessible through the interfaces methods, but the validity of the process engaged is only dependant on the service implementation. Moreover, a service can define other methods not present in any interfaces.

- **DocumentAnalysis** process **MediaUnits** for any kind of functionality by implementing **Analyser**.
- **DocumentSegmentation** process **MediaUnits** in order to discover segment of interests in the content by implementing **Analyser**, **Configurable**, **ContentConsumer** and **Trainable**.
2.3. IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

- **FullTextIndexing** is a simple indexing service on full text. Note that the “text” aspect is only referred in the name of the service by implementing Configurable and Indexer.

- **FullTextSearching** provides search functionality on full text by implementing Searcher.

- **NamedEntitiesExtraction** processes MediaUnits in order to extract segments of interests and annotating them to refer to the recognised named entity by implementing Analyser.

- **Normalisation** processes MediaUnit and Content in order to normalise their format and eventually to convert metadata into annotations by implementing Analyser, ContentConsumer and ContentProvider.

- **OntologyAlignment** provides methods to generate an ontology from the alignment of two others (and with a given knowledge base) by creating two methods for setting the knowledge and the ontologies to align.

- **OntologyAnalyser** provides methods to generate an ontology from a given ontology and a knowledge base by creating two methods for setting the knowledge and the source ontology to enrich.

- **ResourceRepository** provides methods to store, search and have access to Resources and Contents by implementing ContentConsumer, ContentProvider, ResourceContainer and Searcher.

- **WebCrawling** provides methods to crawl some places in accordance to its configuration (not limited to web). This is an example of service that import methods from interfaces and that defines new methods.

These provided examples are only “examples” and thus any new services implemented will have its own WSDL definition.
2.3. IMPLEMENTATION RECOMMENDATIONS AND BEST PRACTICES

Figure 2.6: Generic interfaces for the services.
3.1 Overview of the whole tutorial

This section aims at providing a global view of the tutorial. This tutorial is built upon several parts. The complete objective of the tutorial is to enable you to build a complete application among the WebLab platform:

1. Development of a service that manipulate the exchange model (cf. chapter 4 Service Development)
2. Development of two portlets communicating together and communicating with the previous service (cf. chapter 5 Portlet Development)
3. Integration of those components on an ESB and a complete portal. (cf. chapter 6 Integration)

The functional part of this tutorial is very simple:

1. The service creates a Text section to a given Document and adds to this Text a content. If a Text already exists, it adds extra content to the existing Text section.
2. The first portlet enables the user to pass a Document to the previous service and to display the content of the Document and to send the Document to a second portlet.
3. The second portlet enables the user to display, the content of the Document sent by the first portlet.
3.2 Content of the development kit

Just unzip the zip files in a devKit folder to obtain the following content:

```
devKit
|-- licence.txt
|-- WDK-2.1.pdf
|-- schemas
  |-- interfaces
  |  |-- Analyser.wsdl
  |  |-- AnalyserTypes.xsd
  |  |-- Configurable.wsdl
  |  |-- ConfigurableTypes.xsd
  |  |-- ContentConsumer.wsdl
  |  |-- ContentConsumerTypes.xsd
  |  |-- ContentProvider.wsdl
  |  |-- ContentProviderTypes.xsd
  |  |-- Indexer.wsdl
  |  |-- IndexerTypes.xsd
  |  |-- ReportProvider.wsdl
  |  |-- ReportProviderTypes.xsd
  |  |-- ResourceContainer.wsdl
  |  |-- ResourceContainerTypes.xsd
  |  |-- Searcher.wsdl
  |  |-- SearcherTypes.xsd
  |  |-- Trainable.wsdl
  |  |-- TrainableTypes.xsd
  |  `-- exception.xsd
  `-- model
    |-- content.xsd
    |-- model.xsd
    |-- multimedia.xsd
    |-- ontology.xsd
    |-- query.xsd
    |-- service.xsd
    `-- structure.xsd
     `-- text.xsd
     `-- user.xsd
  `-- services
    |-- DocumentAnalysis.wsdl
    |-- DocumentSegmentation.wsdl
    |-- FullTextIndexing.wsdl
    |-- FullTextSearching.wsdl
    |-- NamedEntitiesExtraction.wsdl
    |-- Normalisation.wsdl
    |-- OntologyAlignment.wsdl
```

Figure 3.1: Overview of the complete EADS tutorial.
3.2. CONTENT OF THE DEVELOPMENT KIT

|-- OntologyAlignmentTypes.xsd
|-- OntologyAnalyzer.wsdl
|-- OntologyAnalyzerTypes.xsd
|-- ResourceRepository.wsdl
|-- WebCrawling.wsdl
|  `-- WebCrawlingTypes.xsd
|-- sources
| |-- ESB-projects-src.zip
| |-- helloPortlets-v1-src.zip
| |-- helloPortlets-v2-src.zip
| |-- helloProject-src.zip
|  `-- testProject-src.zip
|-- tools
| |-- JWebServiceClient-2.0-SNAPSHOT.zip
| |-- extended-1.0.1.zip
| |-- model-1.0.1.zip
| |-- petals-se-orchestra-1.0-patch3.zip
|  `-- repository.zip
|  `-- war
|   |-- portlets
|    | `-- V1
|    |   | `-- helloPortlets.war
|    |  `-- V2
|    |   | `-- helloPortlets.war
|    `-- services
|    |-- helloProject.war
|    `-- testProject.war

12 directories, 57 files
4.1 Overview on Web service development

Not only the development of web services is possible in Java and many languages but and moreover there are several ways to do this in Java. This section will try to describe the steps of a web service development process, trying to be as much language independent as possible. It supposes that the WSDL interface of the service is defined and thus that it will be implemented through the service development.

The scheme proposed in page 30 provides a global overview on those development steps, which can be briefly describe as:

1. Design of an UML model for data and services. This should be done as a part of the architecture design. But evolutions can be proposed and should be studied during the whole process in order to test their validity against the platform architecture and components interoperability.

2. Translation to WSDL/XSD. The data exchange model, as a master piece of the platform, is already proposed as XSD files in this development kit. WSDL generic interfaces are also provided and other definitions are provided in samples.

3. Code generation step will focus on parsing the WSDL and XSD files to generate source code for the proposed elements (i.e. model and services). This can generally be done automatically depending on the language wanted.

4. Implementation and compilation are the main development tasks and will be addressed by developers. As illustrated, external code and components (libraries, applications...) can be involved in the generated code. Finally, the code is compiled.

5. Packaging will adapt the compiled code to the web server.
4.1. **OVERVIEW ON WEB SERVICE DEVELOPMENT**

6. Deployment of the web service.

7. Test of the web service invocation.

As explained, some steps, especially the definitions steps, are already achieved and are parts of the platform specification. Then some others steps can be automated depending on the programming language used. The implementation is where the "work is done" regarding the development since this is where some programming code is written to achieve the service functionality.
4.1. OVERVIEW ON WEB SERVICE DEVELOPMENT

Figure 4.1: Web Services development.
4.2 Technical steps

Technically, developers of platform component will only be facing to a reduced number of tasks from basic tools installation to advanced testing. The following list should resume all of them:

1. Installation of development tools
2. Code generation from WSDL and XSD
3. Implementation
4. Compilation and packaging
5. Deployment
6. Validation and test

This document will focus on helping the developers through those tasks by providing useful information and advising on tools selection. Some recommendations on how to exploit the platform capabilities will also be discussed here.

The provided information will be splitting in accordance to each language/development tools specificities. However some tasks, such as the validation procedure will be common. The following table give a simple overview on what task can be merged. Tasks non related to Java/JAXWS tutorial will be completed later. But other tasks will certainly be merged.

For example the packaging with Axis can also used Maven.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation of development tools</td>
<td>JDK, Maven and Tomcat</td>
<td>With AXIS</td>
</tr>
<tr>
<td>Code generation from WSDL and XSD</td>
<td>wsimport (i.e. Maven plugin)</td>
<td>AXIS library</td>
</tr>
<tr>
<td>Implementation</td>
<td>Still manual work</td>
<td></td>
</tr>
<tr>
<td>Compilation and packaging</td>
<td>Maven</td>
<td>Maven</td>
</tr>
<tr>
<td>Deployment</td>
<td>Tomcat</td>
<td>Tomcat</td>
</tr>
<tr>
<td>Validation and test</td>
<td>JWebServiceClient (provided by EADS)</td>
<td></td>
</tr>
</tbody>
</table>

4.3 “Hello world!” service in Java with JAXWS

⚠️ This part is reserved to Java developers.

We assume here that you have already installed the following tools: Java, Maven (and the provided repository), Tomcat, JWebServiceClient. If it is not the case, please refer to the chapter 8: Tools Installation.
4.3. "HELLO WORLD!" SERVICE IN JAVA WITH JAXWS

4.3.1 Code generation from WSDL and XSD

The two first sections will describe how to build the project and then how to generate the Java code from the WSDL and XSD.

Creating a Java web application scheme

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

```bash
mvn archetype:create -DgroupId=org.tutorial.ws -DartifactId=helloProject -DarchetypeArtifactId=maven-archetype-webapp
```

Well, not so simple in fact so some explanations are needed. We call the Maven script and ask for archetype plug-in which is able to build project scheme, in our case using web application scheme. Then we provide information on the names of our project:

- groupId: a group in which our project will be available (i.e. it will be deployed in sub folder of our repository following this group name). Here `org.tutorial.ws`.
- artifactId: the name of our project, which will also be the name of the folder created and containing our project. Here `helloProject`.
- archetypeArtifactId: name of the maven plug-in used to generate the project. Here the required plugin is `maven-archetype-webapp` which will produce the skeleton of a web application project.

To customise your service you can change the group ID, the artifact ID, but keep the archetype artifact ID. The configuration file will change and you will need to keep track of your change (what names do you used during the creation), but nothing difficult.

After this a folder called `helloProject` (or any artifact ID you gave) should be available and should contain the following files and subdirectories:

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

\[ info \]
There are numerous plugins in Maven that allows to import the generated project into your preferred IDE (Eclipse, Netbeans...). However, to avoid any dependency to an IDE, we suggest to only use simple text editor for this tutorial. More information on project artifact generation for your IDE can be found on Maven website\[^1\].

\[^1\]http://maven.apache.org/
4.3. "HELLO WORLD!" SERVICE IN JAVA WITH JAXWS

Modifying the Project Object Model (POM)

The Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. For instance, the build directory is target; the source directory is src/main/java; the test source directory is src/main/test; and so on.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other informations such as the project version, description, developers, mailing lists and such can also be specified.

The next step is then to configure the project in order to set the right dependencies to JAXWS and to set the right plug-in configuration for the generation of the Java code. Open the file helloProject/pom.xml, which should look like this:

```xml
<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>helloProject</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloProject 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>helloProject</finalName>
  </build>
</project>
```

Listing 4.1: Original generated POM configuration file of the project.

One can refer to the Maven documentation to have explanation of each component of this file. You can simply note that the option passed to Maven (group ID and so on...) are written in this file and thus any change to those parameter will affect the pom.xml.

To adapt this file to our needs, you should add the dependencies to the JAXWS libraries. For this you can copy the line provided in listing 4.2 in to the pom file between the `<dependencies>` mark-up (without removing the old dependencies). It will enables Maven to load the proposed jar and all its dependencies.

```xml
[...]
<dependency>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-tools</artifactId>
  <version>2.1.3</version>
</dependency>
[...]
```

Listing 4.2: Example of inclusion of a dependency in the pom.xml.
4.3. “HELLO WORLD!” SERVICE IN JAVA WITH JAXWS

Then plugins informations have to be added too between the <build></build> (again without removing the previously generated information). Copy/paste the data provided in listing 4.3. As you can see 2 plugins are inserted:

```xml
[...] <plugins>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxws-maven-plugin</artifactId>
    <configuration>
      <sourceDestDir>src/main/java</sourceDestDir>
      <verbose>true</verbose>
      <wsdlFiles>
        <wsdlFile>services/DocumentAnalysis.wsdl</wsdlFile>
      </wsdlFiles>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins>

Listing 4.3: Plugins configuration.

The only important configurations are the source destination directory which is here standardised with Maven project structure (i.e. source in helloProject/src/main/java and then sub-folder with package names) and the path to the WSDL interface. Note that the configuration of this plugin do not contain the full path to the WSDL file of the service interface (here DocumentAnalysis.wsdl) but only its name. It is expected in the default folder which is helloProject/src/wsdl (which will be the case later while achieving the next section instructions to copy the right files in the right place). However it can be changed here if you want any other custom path (but remember it) or a sub-path can be specified as it is done in this configuration (in the services sub-folder).

More information on this plugin configuration can be found here: https://jax-ws-commons.dev.java.net/jaxws-maven-plugin

```
4.3. “HELLO WORLD!” SERVICE IN JAVA WITH JAXWS

Listing 4.4: Final pom.xml.

```xml
<groupId>com.sun.xml.ws</groupId>
<artifactId>jaxws-tools</artifactId>
<version>2.1.3</version>
</dependency>
<dependency>
<groupId>org.codehaus.mojo</groupId>
<artifactId>jaxws-maven-plugin</artifactId>
<configuration>
<sourceDestDir>src/main/java</sourceDestDir>
<verbose>true</verbose>
<wsdlFiles>
<wsdlFile>services/DocumentAnalysis.wsdl</wsdlFile>
</wsdlFiles>
</configuration>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>
</plugin>
</plugins>
</build>
```

About WSDL

Now the real work begins. You need to copy the WSDL folder provided in the development kit (in `<devKitFolder>/sources/schemas`) to a newly created folder `helloProject/src/wsdl` which is the default place where the Maven plugin will search the WSDL. The WSDL depends on some XSD and WSDL files which define the exchange data model format and the service interface. Thus those files are in the model and interfaces subfolders.

For our Hello service, the WSDL used is `DocumentAnalysis.wsdl`. It imports another WSDL file `Analyser.wsdl` which depends on an XML schema `AnalyserTypes.xsd`, which defines the complex types involved by the service. As you can see, the schema itself import other external XSD in order to avoid to redefine the object included in the data exchange model. For our tutorial, we include all the additional packages and thus all the model objects. This is the safest way in order to not forget any objects that are not necessarily manipulated by the service but that could be contained in an argument.

A simple example is a text processing service. It is obvious that it will only process `Text` unit, however, it can receive a global `Document` and process only the `Text` parts. But, if this `Document` contains `Audio` or `Video` units, the service must know their definitions in order to keep them in the processed `Document` (they won’t be altered but definition is needed to “pass” them through the service).

Well again, not so simple... But keep in mind that including all the model packages has no real consequences so that is the safest way. Experts can remove import statements
when they are sure that the service will never receive any objects defined in the imported package. It will only simplify the services WSDL.

Generating the service interface from WSDL definition

Set the current directory to your project root folder:

```
  cd <pathToYourProject>/helloProject
```

Then call the following command to generate the code of service interface:

```
  mvn jaxws:wsimport
```

If everything is fine, the project folder should contain files and directories presented listed page 36.

As you can see all the classes from the exchange data model have been generated in the `org.weblab_project.core.model` package; classes and interfaces related to the service are in `org.weblab_project.services.documentanalysis` package. All the Java classes have been pre-compiled in the `helloProject/target` folder (but that's not really important).

The most important generated file is the service interface which will need to be implemented to define the “real process” done by our Hello service. It is located at `helloProject/src/main/org/weblab_project/services/documentanalysis/Analyser.java`.

```
helloProject/
|-- pom.xml
  `-- src
    |-- main
      `-- java
        `-- org
          `-- weblab_project
            `-- core
              `-- model
                `-- Annotation.java
                `-- ComposedUnit.java
                `-- Document.java
                `-- Feature.java
                `-- LowLevelDescriptor.java
                `-- MediaUnit.java
                `-- ObjectFactory.java
                `-- Resource.java
                `-- ResourceCollection.java
                `-- Segment.java
                  `-- content
                    `-- BinaryContent.java
                    `-- Content.java
                    `-- ObjectFactory.java
                    `-- TextContent.java
                    `-- package-info.java
                  `-- multimedia
                    `-- Audio.java
                    `-- BinaryMediaUnit.java
                    `-- Coordinate.java
                    `-- Image.java
                    `-- ObjectFactory.java
                    `-- SpatialSegment.java
                    `-- TemporalSegment.java
                    `-- Video.java
                    `-- package-info.java
                  `-- ontology
                    `-- ObjectFactory.java
                    `-- Ontology.java
                    `-- Intelegy.java
```
4.3. "HELLO WORLD!" SERVICE IN JAVA WITH JAXWS

```java
|-- package-info.java
| `-- query
|   |-- Attribute.java
|   | `-- ComposedQuery.java
|   | `-- FullTextQuery.java
|   | `-- ObjectFactory.java
|   | `-- Operator.java
|   | `-- Query.java
|   | `-- SemanticQuery.java
|   | `-- SimilarityQuery.java
|   | `-- StructuredQuery.java
|   | `-- UniversalQuery.java
|   `-- package-info.java
| `-- service
|   |-- ObjectFactory.java
|   | `-- Service.java
|   | `-- package-info.java
|   `-- structure
|     |-- Cell.java
|     | `-- Line.java
|     | `-- Table.java
|     `-- package-info.java
| `-- text
|   |-- LinearSegment.java
|   | `-- ObjectFactory.java
|   | `-- Text.java
|   `-- package-info.java
| `-- user
|   |-- HumanResource.java
|   | `-- ObjectFactory.java
|   | `-- UsageContext.java
|   `-- package-info.java
|-- services
| `-- analyser
|   |-- types
|     |-- ObjectFactory.java
|     | `-- ProcessArgs.java
|     | `-- ProcessReturn.java
|     `-- package-info.java
| `-- documentanalysis
|     |-- Analyser.java
|     | `-- DocumentAnalysisService.java
|     | `-- ProcessException.java
|     `-- exception
|         |-- ObjectFactory.java
|     `-- WebLabException.java
|         `-- package-info.java
|-- resources
| `-- webapp
|   |-- WEB-INF
|     | `-- web.xml
|     `-- index.jsp
|-- wsdl
| `-- interfaces
|   |-- Analyser.wsdl
|   | `-- AnalyserTypes.xsd
|   | `-- Configurable.wsdl
|   | `-- ConfigurableTypes.xsd
|   | `-- ContentConsumer.wsdl
|   | `-- ContentConsumerTypes.xsd
|   | `-- ContentProvider.wsdl
|   | `-- ContentProviderTypes.xsd
|   | `-- Indexer.wsdl
|   | `-- IndexerTypes.xsd
|   | `-- ReportProvider.wsdl
```

2009-01-23

4.3. "HELLO WORLD!" SERVICE IN JAVA WITH JAXWS

| `-- ReportProviderTypes.xsd |
| `-- ResourceContainer.wsdl |
| `-- ResourceContainerTypes.xsd |
| `-- Searcher.wsdl |
| `-- SearcherTypes.xsd |
| `-- Trainable.wsdl |
| `-- TrainableTypes.xsd |
| `-- exception.xsd |
| `-- model |
| `-- content.xsd |
| `-- model.xsd |
| `-- multimedia.xsd |
| `-- ontology.xsd |
| `-- query.xsd |
| `-- service.xsd |
| `-- structure.xsd |
| `-- text.xsd |
| `-- user.xsd |
| `-- services |
| `-- DocumentAnalysis.wsdl |

4.3.2 Implementation

The interface generated is quite easy to understand. It is very similar to a classic interface, but with specific Java annotations to declare that it should be used as a web service. Implementing such an interface is just as easy as usual with only a minor addition: the class implementing the service interface must declare that it should also be exploited as a service. The listing 4.5 gives an example of implementation.

```java
package org.hello.ws.impl;

import javax.jws.WebService;
import org.weblab_project.core.model.ComposedUnit;
import org.weblab_project.core.model.Document;
import org.weblab_project.core.model.Resource;
import org.weblab_project.core.model.text.Text;
import org.weblab_project.services.analyser.types.ProcessArgs;
import org.weblab_project.services.analyser.types.ProcessReturn;
import org.weblab_project.services.analyser.types.ProcessException;
import org.weblab_project.services.documentanalysis.Analyser;
import org.weblab_project.services.documentanalysis.DocumentAnalysis.Analyser;
import org.weblab_project.services.exception.WebLabException;

@WebService(endpointInterface = "org.weblab_project.services.documentanalysis.Analyser")
public class HelloService implements Analyser {

    public ProcessReturn process(ProcessArgs args) throws ProcessException {

        Resource resource = args.getResource();
        // add a text unit if it is a Document
        if (resource instanceof Document) {
            ComposedUnit original = (ComposedUnit) resource;
            Text hello = new Text();
            // setting a fake content
            hello.setContent("Hello !");
            // adding URI to the new Text
            // \ service should ensure that this URI is unique
            // in the composedUnit scope
            hello.setUri(original.getUri() + "#" + original.getMediaUnit().size());
            // adding the Text to the composedUnit
            original.getMediaUnit().add(hello);
            // if not a Document throw a WebLab exception
            } else {
```

2009-01-23 38/114
4.3. "HELLO WORLD!" SERVICE IN JAVA WITH JAXWS

As one can see, the process described is quite easy to understand: it uses the classes generated for the model and thus exploits the method argument (here a ProcessArgs as defined in the WSDL and extended from the service interface). The classes generated for the model are simple Java bean with method to access each class variables. Here the process defined will test if the MediaUnit in the ProcessArgs is a Document and add a Text unit with some dumb things inside. Finally it will send back the modified object as requested by the method structure. This implementation could then use other Java classes, external libraries or any tools accessible from Java code to do the (more complex) process wanted. Roughly speaking, it's means that this is where "the job is done!".

The only "exotic" thing is the annotation @WebService [...] which declares that the class is a type of Analyser service and thus that wrapping classes should be used while accessing the service. Watch out that this should point to the correct service interface. If not, error won't be generated at compilation and packaging but the service will not be able to work properly.

This is only an example since no test are achieve to ensure that the new Text section URI is unique in the scope of its container (here the Document named as unit). Testing that created elements have valid URI is under service responsibility.

Don't forget to create the right class file in the right folder, according to the class and package names. For instance, HelloService.java in helloProject/src/main/java/org/hello/ws/impl.

4.3.3 Compilation and packaging

Configuration of the service deployment

The standard web.xml (located in helloProject/src/main/webapp/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"
4.3. “HELLO WORLD!” SERVICE IN JAVA WITH 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 in Listing 4.6.

```xml
com/dtd/web-app_2_3.dtd">
  <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>hello</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>hello</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

Listing 4.6: web.xml — Configuration of servlet listener.

The understanding of this file is quite easy and the only thing that is customisable
is the URL pattern which correspond to the relative URL of the servlet that will handle
the service. Note that the servlet name defined in the servlet mark-up should absolutely
match with the servlet name defined in the servlet mapping mark-up.

But one more configuration file is needed to fix the endpoint name: sun-jaxws.xml
(same location). It will define the service implementation which will be used by the listener
to realise the process and the endpoint name. This is again very simple as shown in listing
4.7. Note that the endpoint name should absolutely match with the servlet name defined
in web.xml.

```xml
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint name="hello" implementation="org.hello.ws.impl.HelloService" url-pattern="/" />
</endpoints>
```


Finally, the absolute URL of the service will include the server URL, final name (as
defined in pom.xml) and URL pattern in something like: <server address>/<final
name>/</URL pattern>

For instance:

- • http://localhost:8080/helloProject/* — for our case or,
- • http://localhost:8080/helloProject — since * pattern include all sub URL
  or,
- • http://127.0.0.1:8080/helloProject — using the local IP address or the real
  IP address when accessing from another computer over a network.
4.3. "HELLO WORLD!" SERVICE IN JAVA WITH JAXWS

Packaging the service

The last step before deployment consists in wrapping the service implemented with web services classes compatible with the web application server used. Then the code needed to handle the SOAP message must be added and all the classes must be compiled. Finally the global project should be packaged into a web archive (WAR file) which can be deployed on a web server.

Hopefully, thanks to Maven, a simple command will do all those boring tasks:

`mvn package`

It will go through all the Maven tasks: code generation, compilation, unit testing (if some have been defined in `helloProject/src/test/java`) and packaging the web application. During this process, according to class path defined in the POM file, JAXWS will build the WAR file (web archive) in the `helloProject/target` directory.

Note that further packaging will sometimes need a cleaning of the project. This can be done easily through the command: `mvn clean package`. It will erase the target folder, then download the dependencies from the repository, recompile all classes and finally package the application.

4.3.4 Deployment, validation and test

Finally the service could be deployed on a Java web application server using the same procedure as for the test-service in the installation validation step. To request your newly deployed web service, you could use the JWebServiceClient. You can look at its manual and at tools installation chapter to have a complete description of its scope, capabilities and uses.

To start, just launch the application and select (or add) the following parameters:

- **URL**: `http://localhost:8080/helloProject` (i.e. server location and final name as declared in the `pom.xml` file)
- **Operation**: `process` (name defined in `AnalyserTypes.xsd` included in `AnalyserTypes.wsdl` included in `DocumentAnalysis.wsdl`)

A `Resource` must be send as operation argument to get a correct response from the web service, see listing 4.8.

```xml
<resource xmlns:model="http://weblab-project.org/core/model/
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
synchronised="false" uri="weblab://tutorial/my.document"
xsi:type="model:document" />
```

Listing 4.8: Example of processArgs.

The service should answer the same `Resource` with a `Text` section added containing the “Hello” message as presented in figure 4.2. Any wrong parameter will raised an error from the client (is the service URL is wrong for example) or from the service itself (if the arguments are not valid).
4.4 GOING BEYOND THE TUTORIAL

Figure 4.2: Successful call to the Hello service.

Note that the response content rely on the service implementation and that the client could not validate this since it depends on the service processing aims. However a manual validation can be made by looking into the response XML code.

Well done! Now you know how to use a provided WSDL to implement your own service and test it with the provided client. Now, it's up to you to develop the needed service and their processes.

4.4 Going beyond the tutorial

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 tutorial so we will only provide you the information needed to change the helloProject to your whatYouWantProject.
4.4. GOING BEYOND THE TUTORIAL

4.4.1 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:

```bash
cmv 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.

4.4.2 Customising the POM

After the project creation, the POM file can be modified in the same way presented in the tutorial. However watch out to not erase the part which describe the group and artifact id since you just create a new project with new names. So just copy the dependency to JAXWS and the plugin part to ensure the correct use of wsimport and compilation options.

In the configuration of the wsimport plugin, one can note the reference to the WSDL services/DocumentAnalysis.wsdl. If you plan to implement another kind of service, simply change the name of this WSDL file to the one you want. However you have to ensure that the file is present in the src/wsdl folder.

4.4.3 Code generation for other WSDL

If you changed the configuration of the wsimport plugin in the POM file and correctly copy the WSDL (and all other files it depends on such as XSD) in the right folder, code generation can be done in the same way as presented in the tutorial.

4.4.4 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.weblab_project.services.Analyser, but if you change the WSDL, it has another name (and probably package). You also need to add the annotation concerning the web service implementation like:

```java
@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.
4.4. GOING BEYOND THE TUTORIAL

When you develop your own service, two way are possible:

- add your own code to the existing project created with the tutorial procedure. It means that your business classes you will 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, the JWebServiceClient 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 JWebServiceClient (using JUnit for instance).

4.4.5 Adding dependencies to the project

If your 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>
```

2009-01-23 44/114
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 the directly in the JAR itself. The other procedure can be to convert your all project to the Maven structure, but this task is out of the scope of this tutorial and moreover is not in the philosophy which is “keep what you have, wrap it into a valid service implementation”. However, as already mentioned, one can look into Maven documentation for more information on Maven dependencies management.

### 4.4.6 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).

### 4.4.7 Delivery

To deliver your component developed with jaxws, you provide of course the WAR file, deployable on Tomcat application server. However some more data are needed to ensure its easy integration.

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 results.

Without those information and data, the integration of any component can be very difficult and one should take care on provided the right information in them.
4.5. THE ANNOTATION DATA FIELD PROBLEM (IN JAVA)

4.5 The annotation data field problem (in Java)

When you look at the Annotation class in the generated sources, the data field type is Object. So you have to be very careful with this Object because if you want to set the data field with a String containing RDF statements, you can't directly use the data field setter.

⚠️ Never use something like:

```java
Annotation annot = new Annotation();
annot.setUri("http://test.org/document#1-a1");
annot.setData("<data><rdf:rdf xmlns ...></rdf:rdf></data>");
```

In fact the field data in the Annotation class is an org.w3c.dom.Element, so you have to manipulate it with the DOM API, using transformers like suggested below.

💡 Use something like:

```java
//creates an Annotation
Annotation annot = new Annotation();
annot.setUri("http://test.org/document#1-a1");
// string containing the rdf statements
String rdfString="<data><rdf:rdf xmlns ...></rdf:rdf></data>";
try{
    //transformer instantiation
    Transformer trans = TransformerFactory.newInstance().newTransformer();
    //creates a StringReader containing the RDF string
    StringReader strRead = new StringReader(rdfString);
    //instantiate a DOMResult
    DOMResult res = new DOMResult();
    //do the transformation of a String to a DOM document
    trans.transform(new StreamSource(strRead), res);
    //set annotation data field with the DOM documentElement
    annot.setData((org.w3c.dom.Document)res.getNode()).getDocumentElement());
} catch (Exception ex){
    //...
}
```

Of course RDF String have to be a well formed XML String and valid against RDF schema.

💡 A complete example of a service which annotates a Resource is available in the devKitFolder/sources/testProject-src.zip archive.
The development ofportlet is not only possible in Java but moreover there are several ways to do this in Java. This section will try to describe the steps of a portlet development process.

5.1 Overview on Portlet development

Here it is a global overview on the steps involved in portlet development. describe as:

1. Design of an UML model of the portlet. This should be done as a part of the architecture design. But evolutions can be proposed and should be studied during the whole process in order to test their validity against the platform architecture and components interoperability.

2. Implementation and compilation is the main development task and will be addressed by developers. As illustrated, external code and components (libraries, applications...) can be involved in the generated code. Finally, the code is compiled.

3. Packaging will adapt the compiled code to the web server.

4. Deployment of the portlet on a Portlet Container (Liferay Portal for instance).

5. Test of the portlet invocation.

As explained, some steps, especially the definitions steps, are partly achieved. Then some others steps can be automated depending on the programming language used. The implementation is where the “work is done” regarding the development since this is where some programming code is written to achieved the portlet functionality.
5.2 Technical steps

Technically, developers of platform component will face only a reduce number of tasks from basic tools installation to advanced testing. The following list should resume all of them:

1. Installation of development tools,
2. Project Creation,
3. Implementation,
4. Compilation and packaging,
5. Deployment,

This document will focus on helping the developers through those tasks by providing useful information and advising on tools selection. Some recommendations on how to exploit the platform capabilities will also be discussed here.

<table>
<thead>
<tr>
<th>Task</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation of development tools</td>
<td>JDK, Maven, Liferay Portal</td>
</tr>
<tr>
<td>Project generation</td>
<td>Maven</td>
</tr>
<tr>
<td>Implementation</td>
<td>Still manual work</td>
</tr>
<tr>
<td>Compilation and packaging</td>
<td>Maven</td>
</tr>
<tr>
<td>Deployment</td>
<td>Liferay Portal</td>
</tr>
<tr>
<td>Validation and test</td>
<td>Still manual work</td>
</tr>
</tbody>
</table>

5.3 Overview of Portlets technology

5.3.1 Portals and Portlet architecture

The following diagram page 49 provides a preliminary overview of what is useful to understand in portlets. Various concepts are described below:

- **Portals**: An HTTP-based site hosted with special portal software that allows the aggregation of several different back-end systems, processes, or sites brought together through a single portal page. Portals may provide additional services such as single sign-on security, customisation, personalisation, and back-end administrative/declarative application development.
5.3. CONTENT AGGREGATION

- **Content aggregation**: The process of bringing together content from disjointed systems, via portlets, and controlled through the use of a portal.

- **Portlet container (PC)**: Controls the access, lifetime, and interaction of a single portlet. Provides the content returned from a portlet back to the portal for merging with the content of other portlets.

- **Portlet**: Provides content to its calling portal container for the purposes of being displayed on a portal page.

- **Fragments**: The content generated by a portlet is known as its fragment or fragment code. This is the HTML or XML code generated from the portlet rendering code.

- **WSRP**: Is a technology that enable portals to access remote portlets even developed in other languages (PHP, C#, etc.) thanks to an XML serialisation/deserialisation mechanism.

![Technologies and Architecture](image)

**Figure 5.1**: Portals and portlets technologies.

5.3.2 Portlet vs Servlets

Portlets are an extension of the servlet specification. As a consequence, it has some common part with servlets and also some differences:
5.4. FIRST PORTLETS IN JAVA

<table>
<thead>
<tr>
<th>Feature</th>
<th>Servlet</th>
<th>Portlet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are J2EE Web Component</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Are Managed by Container</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Generate dynamic Web content</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Generate fragments of pages (XML)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Are bound to an URL</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Have two phases processing scheme (action and render)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Manage states and modes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Alter HTTP headers</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Access the Referrer URL</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Allow advanced persistence mechanisms</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Allow access to user profile</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Manage scopes at the application and page level.</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

5.3.3 Portlet Life Cycle

Within a portal, a portlet has a lifecycle with the following methods:

- **init()** — called just the first time (at deployment or at portal start).
- **processAction()** — called in response to a user action such as clicking a button, hyperlink, submitting a form. In **processAction()**, a portlet may modify its own state as well as persistent information about a portlet.
- **processEvent()** — called if there is an event pending for this portlet.
- **render()** — generates markup depending on the portlet’s current state. Calls **doView()**.
- **destroy()** — called just the last time (at portal shutdown or when undeploying).

We present in the figure 5.2 (page 51) the lifecycle of the two portlets we are going to develop in this tutorial within a single portal.

5.4 First portlets in Java

⚠️ This part is reserved to Java developers.

We assume here that you have already installed the following tools: Java, Maven (and provide repository) and the Portlet Container. If it is not the case, please refer to the chapter 8: Tools Installation.

As we will use classes of the WebLab exchange model and call a web service using the generated client, it is better to have carried out the Service Development tutorial (chapter 4) before.
5.4. FIRST PORTLETS IN JAVA

Figure 5.2: Portlets lifecycle (with a call to a service).

5.4.1 Creating a java web application scheme

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

```
mvn archetype:create -DgroupId=org.tutorial.ws -DartifactId=helloPortlets
   -DarchetypeArtifactId=maven-archetype-webapp
```

Well, not so simple in fact so some explanations are needed. We call the Maven script and ask for archetype plug-in which is able to build project scheme, in our case using web application scheme. Then we provide information on the names of our project:

- **groupId**: a group in which our project will be available (i.e. it will be deployed in sub folder of our repository following this group name). Here `org.tutorial.ws`.

- **artifactId**: the name of our project, which will also be the name of the folder created and containing our project. Here `helloPortlets`.

- **archetypeArtifactId**: name of the Maven plugin used to generate the project. Here the required plugin is `maven-archetype-webapp` which will produce the skeleton of a web application project.

To customise your service you can change the group ID, the artifact ID, but keep the archetype artifact ID. The configuration file will change and you will need to keep track of your change (what names do you used during the creation), but nothing difficult.

After this a folder called `helloPortlets` (or any artifact ID you gave) should be available and should contain the following files and subdirectories:
There are numerous plugins in Maven that allow to import the generated project into your preferred IDE (Eclipse, Netbeans...). However, to avoid any dependency to an IDE, we suggest to only use simple text editor for this tutorial. More information on project artifact generation for your IDE can be found on maven website.

Modifying the Project Object Model (POM)

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this is the build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/main/test; and so on.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

The next step is then to configure the project in order to set the right dependencies to JAXWS and to set the right plug-in configuration for the generation of the Java code. Open the file helloPortlets/pom.xml, which should look like this:

```xml
<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.portlet</groupId>
  <artifactId>helloPortlets</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloPortlets 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>helloPortlets</finalName>
  </build>
</project>
```

Listing 5.1: Original generated POM configuration file of the project.

http://maven.apache.org/
5.4. FIRST PORTLETS IN JAVA

One can refer to the Maven documentation to have an explanation of each component of this file. You can simply note that the option passed to Maven (group ID and so on...) are written in this file and thus any change to those parameter will affect the pom.xml.

To adapt this file to our needs, you should add the dependencies to the Portlet API libraries. The official API for Portlet is 2.0 provided by Sun. For this you can copy the line provided in listing 5.2 into the POM file between the <dependencies></dependencies> markup (without removing the old dependencies). It will enables maven to load the proposed jar and all its dependencies.

```xml
<dependency>
    <groupId>javax.portlet</groupId>
    <artifactId>portlet-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope><!-- Prevents addition to war file -->
</dependency>

<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope><!-- Prevents addition to war file -->
</dependency>
```

Listing 5.2: Inclusion of portlets dependencies in the pom.xml.

As we will generate a client call to a web service, we also need to add the dependencies to the JAXWS libraries.

```xml
<dependency>
    <groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-tools</artifactId>
    <version>2.1.3</version>
</dependency>
```

Listing 5.3: Inclusion of JAXWS dependency in the pom.xml.

Then plugins informations have to be added too between the <build></build> (again without removing the previously generated information). Copy/paste the data provided in listing 5.4. As you can see 2 plugins are inserted:

- **jaxws-maven-plugin** which will permit to generate the java client classes for calling web services,
- **maven-compiler-plugin** which will enable Java 1.5 compilation. Note that if you want to use enhanced Java features from Java 1.6, you should change the compilation version to 1.6. However, you must have a 1.6 JDK and thus have followed the JDK 1.6 installation procedure presented in the chapter 8 of this tutorial.

```xml
<plugins>
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>1.10</version>
    </plugin>
</plugins>
```

2009–01–23
5.4. FIRST PORTLETS IN JAVA

```xml
<configuration>
  <wsdlDirectory>src/main/resources/wsdl</wsdlDirectory>
  <sourceDestDir>src/main/java</sourceDestDir>
  <verbose>true</verbose>
  <wsdlFiles>
    <wsdlFile>services/DocumentAnalysis.wsdl</wsdlFile>
  </wsdlFiles>
</configuration>
```

Listing 5.4: Plugins configuration.

The only important configurations are the source destination directory which is here standardised with Maven project structure (i.e. source in helloPortlets/src/main/java and then sub-folder with package names) and the path to the WSDL interface.

More information on this plugin configuration can be found here: https://jax-ws-commons.dev.java.net/jaxws-maven-plugin

```xml
<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.portlet</groupId>
  <artifactId>helloPortlets</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloPortlets 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>
    <dependency>
      <groupId>javax.portlet</groupId>
      <artifactId>portlet-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.xml.ws</groupId>
      <artifactId>jaxws-tools</artifactId>
      <version>2.1.3</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>helloPortlets</finalName>
  </build>
</project>
```

2009-01-23 54/114
5.4. FIRST PORTLETS IN JAVA

5.4.2 Implementation

We now have to implement two portlets in this tutorial:

1. The first one displays a WebLab Document and can send the results to a second portlet by an event mechanism, if you want to go further, this portlet can communicate with a web service to modify the content of a Document.

2. The second one receives an event from the first portlet and displays the Document.

Implementation of the first portlet

⇒ Generating the model class and client from the WSDL definition

In order to manipulate classes of the exchange model the first thing to do is to generate the classes from the exchange model (the XSD and WSDL should be added to the src/main/resources/wsdl directory). See Chapter 4 for further informations.

The service which will be called by the portlet is defined by the DocumentAnalysis.wsdl. To generate the Java code from this WSDL, set the current directory to your project root folder:

cd <pathToYourProject>/helloPortlets

Then call the following command to generate the code of service interface:

mvn jaxws:wsimport

If everything is fine, the project folder should contain files and directories listed page 57.
5.4. FIRST PORTLETS IN JAVA

As you can see all the classes from the exchange data model have been generated in the \texttt{org.weblab\_project.core.model} package; classes and interfaces related to the service are in \texttt{org.weblab\_project.services} package. All the java classes have been pre-compiled in the \texttt{helloPortlets/target} folder (but that’s not really important).

\begin{verbatim}
helloPortlets/
|-- pom.xml
  `-- src
     `-- java
        `-- org
           `-- weblab\_project
              `-- core
                 `-- model
                    `-- Annotation.java
                    `-- ComposedUnit.java
                    `-- Document.java
                    `-- Feature.java
                    `-- LowLevelDescriptor.java
                    `-- MediaUnit.java
                    `-- ObjectFactory.java
                    `-- Resource.java
                    `-- ResourceCollection.java
                    `-- Segment.java
                    `-- content
                        `-- BinaryContent.java
                        `-- Content.java
                        `-- ObjectFactory.java
                        `-- TextContent.java
                        `-- package-info.java
                    `-- multimedia
                        `-- Audio.java
                        `-- BinaryMediaUnit.java
                        `-- Coordinate.java
                        `-- Image.java
                        `-- ObjectFactory.java
                        `-- SpatialSegment.java
                        `-- TemporalSegment.java
                        `-- Video.java
                        `-- package-info.java
                    `-- ontology
                        `-- ObjectFactory.java
                        `-- Ontology.java
                        `-- package-info.java
                    `-- query
                        `-- Attribute.java
                        `-- ComposedQuery.java
                        `-- FullTextQuery.java
                        `-- ObjectFactory.java
                        `-- Operator.java
                        `-- Query.java
                        `-- SemanticQuery.java
                        `-- SimilarityQuery.java
                        `-- StructuredQuery.java
                        `-- UniversalQuery.java
                        `-- package-info.java
                    `-- service
                        `-- ObjectFactory.java
                        `-- Service.java
                        `-- package-info.java
                    `-- structure
                        `-- Cell.java
                        `-- Line.java
                        `-- ObjectFactory.java
                        `-- Table.java

2009-01-23 56/114
\end{verbatim}
5.4. FIRST PORTLETS IN JAVA

```java
protected static void setEndpointAddress(Object port, String newAddress,
    String SOAPAction) {
    BindingProvider bp = (BindingProvider) port;
    Map<String, Object> context = bp.getRequestContext();
    context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, newAddress);
    context.put(BindingProvider.SOAPACTION_USE_PROPERTY, true);
    context.put(BindingProvider.SOAPACTION_URI_PROPERTY, SOAPAction);
}
```

Just change the address in service call. See the implementation section for a sample in a portlet.

⇒ Change the URL of the called Web service

The service to call is actually deployed on Tomcat, to modify it URL programatically, use the following method that you should first add and then call on your port:
5.4. FIRST PORTLETS IN JAVA

⇒ **Implementation**

The first thing to do then is to create a class (let’s call it `HelloPortlet1` in package `org.weblab_project.portlet`) that extends `GenericPortlet` and to implement the `init` and `destroy` methods that are used at portal start and portal stop. We also add here the method `setEndpointAddress`.

```java
package org.weblab_project.portlet;

import java.util.Map;
import javax.portlet.GenericPortlet;
import javax.xml.ws.BindingProvider;
import org.weblab_project.core.model.Document;
public class HelloPortlet1 extends GenericPortlet {
    private Document doc = null;

    public void init() {
        doc = new Document();
        doc.setUri("weblab://myportlet/mydoc");
        Text text = new Text();
        text.setUri("weblab://myportlet/mydoc#T1");
        text.setContent("My first text content");
        doc.getMediaUnit().add(text);
    }

    public void destroy() {
        doc = null;
    }

    protected static void setEndpointAddress(Object port, String newAddress,
            String SOAPAction) {
        BindingProvider bp = (BindingProvider) port;
        Map<String, Object> context = bp.getRequestContext();
        context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, newAddress);
        context.put(BindingProvider.SOAPACTION_USE_PROPERTY, true);
        context.put(BindingProvider.SOAPACTION_URI_PROPERTY, SOAPAction);
    }
}
```

Listing 5.6: `HelloPortlet1` init and destroy

We are now going to add the rendering functionality to the portlet (generating the fragment of HTML) when we are in the mode view that display the content of the portlet; to do this we have to implement the `doView()` function.

Here we add the desired behavior to the portlet:

1. It displays the content of all text section of a given `Document`.

---

2Portlet API defines three default MODE: VIEW, EDIT, HELP and for these mode you can implement directly the function `doView`, `doEdit`, `doHelp`. Custom mode can also be defined using the `doDispatch` method and deployment descriptor.
2. It adds a form to send the Document to a service or to send an event containing the Document to a second portlet. Note that we use a `RenderResponse.createActionURL()` to get an URL that will be understood by the Portal as an action URL for the current portlet.

```java
package org.weblab_project.portlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletURL;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.weblab_project.core.model.Document;
import org.weblab_project.core.model.MediaUnit;
import org.weblab_project.core.model.text.Text;

/**
 * Render HTML corresponding to the view mode
 */
public void doView(RenderRequest req, RenderResponse res) throws IOException, PortletException {
    // portlet title
    res.setTitle("WEBLAB 1ST PORTLET");

    // setting Mime type
    res.setContentType("text/html;charset=UTF-8");

    // HTML code
    PrintWriter out = res.getWriter();
    out.println("<h2>WebLab 1st Portlet</h2><br/><br/>Document Text Content is :<br/><br/>");

    for (MediaUnit unit:doc.getMediaUnit()) {
        if (unit instanceof Text) {
            out.println(((Text) unit).getContent() + "<br/>");
        }
    }

    // ActionUrl used in form
    PortletURL submitUrl = res.createActionURL();

    // add a form to send the document to service and send event to a second
    // portlet
    out.println("<form method='POST' action='" + submitUrl + ">");
    out.println("<input type='submit' name='submitButton' value='SendToService'/>");
    out.println("<input type='submit' name='submitButton' value='SendEventToPortlet'/>");
    out.println("</form>");
}
```

Listing 5.7: HelloPortlet1 doView

The next step is to add the code corresponding to an action i.e. when a click on a button is done. To carry out this point, we have to implement the `processAction()` method. Note that we first have to retrieve the `submitButton()` parameter and check its value. If we are in the case of `SendToService` click, we just have to add an instance of the `DocumentAnalysis` client generated (`DocumentAnalysisService`), then set the URL...
of service to call using `setEndpointAddress()` method and call the `process()` method on its port. It will then call the service at the provided location (see previous section). Note that we handle the WSDL file as a portlet resource using the `PortletContext` and the method `getRealPath()`.

```java
import java.io.IOException;
import java.io.File;
import java.util.Map;
import java.net.URL;
import java.net.MalformedURLException;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.BindingProvider;

import java.io.IOException;
import java.io.File;
import java.util.Map;
import java.net.URL;
import java.net.MalformedURLException;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.BindingProvider;

public void processAction(ActionEvent req, ActionResponse res) throws IOException, PortletException {
    // get the content of the submitButton parameter,
    String value = req.getParameter("submitButton");
    System.out.println("Received " + value);
    if (value.equals("SendToService")) {
        // getting WSDL URL from portlet context
        URL WSDLServiceURL = null;
        try {
            WSDLServiceURL = new File(getPortletContext().getRealPath("WEB-INF/classes/wsdl/services/DocumentAnalysis.wsdl")).toURL();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        // creating service
        DocumentAnalysisService service = new DocumentAnalysisService(WSDLServiceURL,
                new QName("http://weblab-project.org/services/documentanalysis", "DocumentAnalysisService"));
        Analyser analyser = service.getAnalyserPort();
        // setting the service URL, here service on local tomcat
        setEndpointAddress(analyser, "http://localhost:8080/helloProject","process");
        ProcessArgs in = new ProcessArgs();
        in.setResource(doc);
        try {
            // call to web service
            doc = (Document) analyser.process(in).getResource();
        } catch (ProcessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
[...]
```

Listing 5.8: HelloPortlet1 processAction

We can now add the behavior when clicking on “Send Event”:

1. First we create an event
2. We add the action in the `processAction()` method

An event is simply a class that is `Serializable`. We add a `DocumentEvent` class that just contains a `Document`.

2009-01-23
5.4. FIRST PORTLETS IN JAVA

```java
package org.weblab_project.portlet;
import java.io.Serializable;
import org.weblab_project.core.model.Document;
public class DocumentEvent implements Serializable {
    /**
     * default generated id
     */
    private static final long serialVersionUID = 1L;
    private Document doc;
    public Document getDoc() {
        return doc;
    }
    public void setDoc(Document doc) {
        this.doc = doc;
    }
}
```

Listing 5.9: DocumentEvent

Then we can add the desired behavior to the `processAction()` method for a click on `sendEvent()`: we create a `DocumentEvent` instance, add to it the `Document` and send it by using the `setEvent()` method of `actionResponse()`. For this sake, simply add the following section after the first if section in the method.

```java
import javax.xml.namespace.QName;
...
if (value.equals("SendEventToPortlet"){
    DocumentEvent evt = new DocumentEvent();
    evt.setDoc(doc);
    QName qn = new QName("http://weblab-project.org/portlet/event_wsrp",
                        "DocumentEvent");
    res.setEvent(qn, evt);
    System.out.println("Event sent");
}
```

Listing 5.10: processAction SendEvent code

We have now completed the development of the first portlet (Resulting listing below).
5.4. FIRST PORTLETS IN JAVA

```java
import javax.portlet.RenderResponse;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.BindingProvider;
import org.weblab_project.core.model.Document;
import org.weblab_project.core.model.MediaUnit;
import org.weblab_project.core.model.text.Text;
import org.weblab_project.services.analyser.types.ProcessArgs;
import org.weblab_project.services.documentanalysis.DocumentAnalysisService;
import org.weblab_project.services.documentanalysis.Analyser;
import org.weblab_project.services.documentanalysis.ProcessException;

public class HelloPortlet1 extends GenericPortlet {
    private Document doc = null;

    public void init() {
        doc = new Document();
        doc.setUri("weblab://myportlet/mydoc");
        Text text = new Text();
        text.setUri("weblab://myportlet/mydoc#T1");
        text.setContent("My first text content");
        doc.getMediaUnit().add(text);
    }

    public void destroy() {
        doc = null;
    }

    /**
     * Render HTML corresponding to the view mode
     * @param req the request
     * @param res the response
     * @exception IOException
     * @exception PortletException
     */
    public void doView(RenderRequest req, RenderResponse res)
    throws IOException, PortletException {
        // portlet title
        res.setTitle("WEBLAB 1ST PORTLET");
        // setting Mime type
        res.setContentType("text/html;charset=UTF-8");

        // HTML code
        PrintWriter out = res.getWriter();
        out.println("<h2>");
        out.println("WebLab 1st Portlet");
        out.println("<br/><br/>Document Text Content is :<br/><br/>");
        for (MediaUnit unit : doc.getMediaUnit()) {
            if (unit instanceof Text) {
                out.println(((Text) unit).getContent() + "<br/>");
            }
        }
        // ActionUrl used in form
        PortletURL submitUrl = res.createActionURL();
        // add a form to send the document to service and send event to a second
        // portlet
        out.println("<form method='POST' action='" + submitUrl + ">");
        out.println("<input type='submit' name='submitButton' value='SendToService'/>");
        out.println("<input type='submit' name='submitButton' value='SendEventToPortlet'/>");
        out.println("</form>");
    }

    public void processAction(ActionEvent req, ActionResponse res)
    throws IOException, PortletException {
    }
```

2009-01-23
5.4. FIRST PORTLETS IN JAVA

```java
// get the content of the submitButton parameter,
String value = req.getParameter("submitButton");
System.out.println("Received "+ value);

if (value.equals("SendToService")) {
    // getting WSDL URL from portlet context
    URL WSDLServiceURL = null;
    try {
        WSDLServiceURL = new File(getPortletContext().getRealPath(
            "WEB-INF/classes/wsdl/services/DocumentAnalysis.wsdl")
            .toURL);
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }

    // creating service
    DocumentAnalysisService service = new DocumentAnalysisService(
        WSDLServiceURL,
        new QName(
            "http://weblab-project.org/services/documentanalysis",
            "DocumentAnalysisService"));
    Analyser analyser = service.getAnalyserPort();

    // setting the service URL, here service on local Tomcat
    setEndpointAddress(analyser, "http://localhost:8080/helloProject",
        "process");
    ProcessArgs in = new ProcessArgs();
    in.setResource(doc);
    try {
        // call to web service
        doc = (Document) analyser.process(in).getResource();
    } catch (ProcessException e) {
        e.printStackTrace();
    }
}

if (value.equals("SendEventToPortlet")) {
    DocumentEvent evt = new DocumentEvent();
    evt.setDoc(doc);
    QName qn = new QName(
        "http://weblab-project.org/portlet/event_wsrp",
        "DocumentEvent");
    res.setEvent(qn, evt);
    System.out.println("Event sent");
}
```

Listing 5.11: HelloPortlet1

We can now proceed with the development of the second portlet.

Create a class (let's call it HelloPortlet2 in package org.weblab_project.portlet) that extends GenericPortlet and to implement the EventPortlet interface as it receives event, we consider that this portlet has no specific behavior at init and destroy and has the same behavior than our first portlet in VIEW Mode.
import java.io.IOException;
import java.io.PrintWriter;
import javax.portlet.EventPortlet;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import org.weblab_project.core.model.Document;
import org.weblab_project.core.model.MediaUnit;
import org.weblab_project.core.model.text.Text;

public class HelloPortlet2 extends GenericPortlet implements EventPortlet {

    private Document doc;

    /**
     * Render HTML corresponding to the view mode
     */
    public void doView(RenderRequest req, RenderResponse res)
            throws IOException, PortletException {
        // setting portlet title
        res.setTitle("WEBLAB 2nd PORTLET");

        // setting Mime type
        res.setContentType("text/html;charset=UTF-8");
        PrintWriter out = res.getWriter();

        // HTML code
        out.println("<h2>");
        out.println("WebLab 2nd Portlet");
        out.println("<br/><br/>Document Text Content is :<br><br>");
        if (doc == null) {
            out.println("Document is empty");
        } else {
            for (MediaUnit unit : doc.getMediaUnit()) {
                if (unit instanceof Text) {
                    out.println(((Text) unit).getContent() + "<br>");
                }
            }
        }
    }
}

Implement the `processEvent` method reading the received event and setting the document attribute.

```java
 [...]
 /**
  * handle the incoming event
  */
 public void processEvent(EventRequest req, EventResponse resp)
 throws PortletException, IOException {
     doc = ((DocumentEvent) req.getEvent().getValue()).getDoc();
 }
 [...]
```

Now we also have a complete HelloPortlet2 implementation.

```java
package org.weblab_project.portlet;
import java.io.IOException;
import java.io.PrintWriter;
```
5.4.3 Compilation and packaging

Configuration of the portlet deployment

A first configuration file portlet.xml which has to be created in helloPortlets/main-/webapp/WEB-INF, will declare to the portal what are the portlets included in the web-app, which modes are supported and which portlets support the eventing mechanisms. Note
5.4. FIRST PORTLETS IN JAVA

d that this file is independent from the portal choice and only rely on the portlet generic specification.

```xml
  <!-- First portlet declaration -->
  <portlet>
    <description xml:lang="FR">HelloPortlet : First WebLab Portlet</description>
    <portlet-name>HelloPortlet1</portlet-name>
    <display-name xml:lang="FR">HelloPortlet : First WebLab Portlet</display-name>
    <portlet-class>org.weblab_project.portlet.HelloPortlet1</portlet-class>
    <expiration-cache>0</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
      <title>HelloPortlet : First WebLab Portlet</title>
      <short-title>HelloPortlet1</short-title>
      <keywords>JSR286 Portlet</keywords>
    </portlet-info>
  </portlet>

  <!-- Second portlet declaration supporting events -->
  <portlet>
    <description xml:lang="FR">HelloPortlet 2 : Second WebLab Portlet</description>
    <portlet-name>HelloPortlet2</portlet-name>
    <display-name xml:lang="FR">HelloPortlet 2 : Second WebLab Portlet</display-name>
    <portlet-class>org.weblab_project.portlet.HelloPortlet2</portlet-class>
    <expiration-cache>0</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
      <title>HelloPortlet 2 : Second WebLab Portlet</title>
      <short-title>HelloPortlet2</short-title>
      <keywords>JSR286 Portlet</keywords>
    </portlet-info>
    <!-- Supported event for this portlet -->
    <supported-processing-event>
      <qname xmlns:x="http://weblab-project.org/portlet/event_wsrp">x:DocumentEvent</qname>
    </supported-processing-event>
  </portlet>

  <!-- event declaration -->
  <event-definition>
    <qname xmlns:x="http://weblab-project.org/portlet/event_wsrp">x:DocumentEvent</qname>
    <value-type>org.weblab_project.portlet.DocumentEvent</value-type>
  </event-definition>
</portlet-app>
```

Listing 5.13: portlet.xml

The standard web.xml (located in helloPortlets/main/webapp/WEB-INF) file has to be configured too, to describe the container used. This file is similar for any portlet application that will be deployed in portlet container. Note that this file is the only part of the development that is portal dependant. See comment for deployment in Liferay Portlet Container.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc./DTD Web Application 2.3//EN"
2009-01-23 66/114
5.4. FIRST PORTLETS IN JAVA

Listing 5.14: web.xml — Configuration of servlet listener.

Packaging the portlet

The last step before deployment consists in wrapping the portlet implemented. Finally the
global project should be packaged into a web archive (WAR file) which can be deployed
on a web server.

Hopefully, thanks to Maven, a simple command will do all those boring tasks:
mvn package

It will go through all the Maven tasks: code generation, compilation, unit testing
(if some have been defined in helloPortlets/src/test/java) and packaging the web
application. During this process, according to class path defined in the POM file, maven
builds the WAR file (web archive) in the helloPortlets/target directory.

Note that further packaging will sometimes need a cleaning of the project.
This can be done easily through the command: mvn clean package. It will
erase the target folder, then download the dependencies from the repository,
recompile all classes and finally package the application.

5.4.4 Deployment, validation and test

Finally the portlet could be deployed on the LiferayPortal. Just copy the generated
WAR file (in helloPortlets/target) into the <homeDir>/Liferay/deploy/ directory
or upload it using the LiferayPlugin portlet (see Tool installation chapter 8.5 page 110).

You can now go to your favorite browser and open http://localhost:8081/user/
joebloggs/home.

You have to Sign in using test@liferay.com as login and test as password. Then
you should see something like this:
5.4. FIRST PORTLETS IN JAVA

Create a new page, for example `helloPortlets` where you will add developed portlets.

You now can add `helloPortlets` using `add application` in the top right menu of the page. In the left menu opened, drag and drop `hello portlets` to the page.

You will obtain something like that:
5.5 GOING BEYOND THE TUTORIAL

Figure 5.6: helloPortlets on a page in LiferayPortal.

On the two portlets, you will see the following: the first portlet display the content of the Document, the second one “display document is empty” as no event has been received yet as presented in figure 5.6. Then you can test the two buttons:

1. **SendToService**: if a service is running at the specified address (cf. section 5.4.2, page 55 of this document), you will see that it will add content a new Text section with generated content.

2. **SendEventToPortlet**: it will send an event to the second portlet with the Document and display the content of the Document.

Figure 5.7: Portlets after several clicks on the two buttons: original text content has been replaced by several "Hello !" and sent to the second portlet.

5.5 Going beyond the tutorial

After this tutorial is achieved, the next step is to implement your own portlet in accordance to any portlet specification. Hopefully, the procedure is really similar to the proposed
tutorial so we will only provide you the information needed to change the helloPortlet to your whatYouWantPortlet.

5.5.1 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:

```bash
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.

5.5.2 Customising the POM

After the project creation, the pom file can be modified in the same way presented in this tutorial. However watch out to not erase the part which describe the group and artifact id since you just create a new project with new names. So just copy the dependency to portlet, jaxws and the plugin part to ensure the correct use of wsimport and compilation option.

In the configuration of the wsimport plugin, one can note the reference to the WSDL services/DocumentAnalysis.wsdl. If you plan to implement another kind of service, simply change the name of this WSDL file to the one you want. However you have to ensure that the file is present in the src/main/resources/wsdl folder.

5.5.3 Code generation for other WSDL

If you changed the configuration of the wsimport plugin in the pom file and correctly copy the WSDL (and all other files it depends on such as XSD) in the right folder, code generation can be done in the same way as presented in the tutorial.

5.5.4 Custom implementation

To implement any portlet, just create a new class in the project (in the package you want) that extends GenericPortlet and if needed EventPortlet.

You should now implement the needed methods in accordance to the portlet definition (init, destroy, doView, doEdit, doHelp, serveResource). 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.

If you need further information on portlet and portal technology, you can refer to existing book such as Professional Portal Development with Open Source Tools by Richardson et al. or the Liferay developpement guide (http://www.liferay.com/web/guest/community/documentation/5_1).
5.5. GOING BEYOND THE TUTORIAL

5.5.5 Adding dependencies to the project

If your projects depend on external libraries (classic Java libraries or your own libraries), two steps 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 the directly in the JAR itself. The other procedure can be to convert your all project to the Maven structure, but this task is out of the scope of this tutorial and moreover is not in the philosophy which is “keep what you have, wrap it into a valid service implementation”. However, as already mentioned, one can look into Maven documentation for more information on Maven dependencies management.

5.5.6 Packaging and tests


2009-01-23
5.5. GOING BEYOND THE TUTORIAL

5.5.7 Dispatch to JSP or servlet

In portlets developed in this tutorial, HTML code is generated in `doView`, `doEdit` or `doHelp` methods. To respect MVC model, best practice is to generate HTML code using JSP pages or Servlet to make view.

Dispatching is allow in `processAction`, `processEvent`, `render`, `serveResources` using a `PortletRequestDispatcher` and methods include or forward.

Here the adaptation of `HelloPortlet2`, dispatching to a jsp HTML corresponding to the view mode.

```java
package org.weblab_project.portlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.portlet.EventPortlet;
import javax.portlet.EventRequest;
import javax.portlet.EventResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import org.weblab_project.portlet.DocumentEvent;
import org.weblab_project.core.model.Document;

public class HelloPortlet2 extends GenericPortlet implements EventPortlet {
    private Document doc;

    /** Render HTML corresponding to the view mode */
    public void doView(RenderRequest req, RenderResponse res)
        throws IOException, PortletException {
        // setting document in request
        req.setAttribute("document", doc);

        // getting dispatcher
        PortletRequestDispatcher dispatcher = getPortletContext().
            getRequestDispatcher("/jsp/viewMode.jsp");

        // dispatch to a jsp
        dispatcher.include(req, res);
    }

    /** handle the incoming event */
    public void processEvent(EventRequest req, EventResponse resp)
        throws PortletException, IOException {
        doc = ((DocumentEvent) req.getEvent().getValue()).getDoc();
    }
}
```

Listing 5.15: Sample of dispatching to a jsp.

And the corresponding JSP, add in `src/main/webapp/jsp/viewMode.jsp` of your the project directory.

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8"
2009-01-23
```
5.5. GOING BEYOND THE TUTORIAL

```java
//getting the document from request
Document doc = (Document) request.getAttribute("document");
if (doc==null) {
    %>Document is empty<%}
} else {
    for (MediaUnit unit:doc.getMediaUnit()) {
        if (unit instanceof Text) {
            %><%=((Text) unit).getContent() %>
        } 
    }
}
```

Listing 5.16: Sample JSP use for dispatching.

5.5.8 Resource serving

In the new JSR286, the `serveResource()` method as been added to the API to handle resources. AJAX calls directly to the portlet are now allowed. Resource are accessible by an URL. Thus, you can use:

```java
ResourceURL resourceUrl = renderResponse.createResourceURL();
```

To get an URL that will enable you to call the `serveResource` method of a portlet (for instance in the `processAction()` method) and then implement `serveResource()` to send back the content to the AJAX client.

```java
public void serveResource(ResourceRequest request, ResourceResponse response)
throws PortletException, IOException {
    //setting the response MIME type (image, xml, html, etc)
    response.setContentType("text/html; charset=UTF-8");
    java.io.PrintWriter w = response.getWriter();
    //write response content
    w.println("...");
}
```

Listing 5.17: Sample of implementation of `serveResource()` method.
6.1 PEtALS integration

Service integration into the bus can be separated in two ways:

- creating an endpoint in the bus referencing the existing service, or
- exposing a bus endpoint to the outside (as an entry point for clients applications).

Before going deeper into the integration description, some explanations about ESB, JBI, endpoints... are needed.

6.1.1 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 is some ESB which implements JBI, and PEtALS is one of them.

An ESB is able to compose heterogeneous components using normalised 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 is findable and requestable through the bus.
6.1. PETALS INTEGRATION

- **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.

In this guide you are going to create SUs, package them in a SA and deploy this SA in the bus. The figure 6.1 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 6.1: ESB architecture example.
6.1. PETALS INTEGRATION

Figure 6.2: Architecture deployed during the integration on ESB.

Scope of the integration process

This tutorial aims at providing the information needed for the integration of a WebLab compliant service into the ESB and then make it available to external client.

As illustrated in figure 6.2, the HelloProject (see Chapter 4 page 28) will be hosted on a Tomcat server and then deployed on a SOAP binding component using service unit and service assembly to make it available as an ESB endpoint. Then the endpoint will be exposed through another service unit in order to make requestable for external clients. In our case, the client will be the portlets developed in Chapter 5 (page 47). Using this architecture will for sure only add complexity on the provided service/portlet example. However, the aim is to make available through the ESB and thus benefit from all ESB capabilities (orchestration, load balancing...).

6.1.2 Endpoint creation

⚠️ For this part you should have installed PEtALS, had a server with the hello service deployed.

Endpoint deployment

The first integration step is to expose the hello service as an endpoint. To do this, we need to deploy a SU on the SOAP BC. When the SU will be deployed, it’ll create a new endpoint on the bus “connected” to the hello service. That means when the endpoint will
be called, the hello service will be thrown the SOAP BC.

Before starting, check that you have installed PETALS and Maven.

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=helloProvideSU -Dversion=1.0-SNAPSHOT
```

This command creates the folder helloProvideSU:

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

So the only 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">
  <xsi:schemaLocation>
    http://www.w3.org/2001/XMLSchema-instance
    http://java.sun.com/xml/ns/jbi
    http://petals.ow2.org/components/extensions/version-4.0
    http://petals.ow2.org/components/soap/version-3.1
  </xsi:schemaLocation>
  <!-- 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:AnalyserService" endpoint-name="hello">
      <!-- CDK specific fields -->
      <petalsCDK:wsdl>interfaces/Analyser.wsdl</petalsCDK:wsdl>
      <!-- SOAP specific fields -->
      <soap:address>http://localhost:8080/helloProject</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 named hello is called, it have to send the request back the helloProject web service.

⚠️ There is another important point: interface-name and service-name have to match with there declaration in the WSDL file, only the endpoint name is changing.

PETALS needs the WSDL file to correctly deploy the service unit. So you have to copy the WSDL file and its dependencies (xsd files) into the src/main/jbi.

Now, we have to install it in our local repository, to do this just type this command in the helloProvideSU 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:
6.1. PETALS INTEGRATION

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

This command creates the folder `helloProvideSA`:

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

First we have to add a dependency to the newly created service unit (`helloProvideSU`) 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>
  OnCollisionIdentification
  <name>SA :: helloProvideSA</name>
  <artifactId>helloProvideSA</artifactId>
  <groupId>org.weblab.example</groupId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jbi-service-assembly</packaging>
  <description>helloProvideSA description.</description>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.weblab.example</groupId>
      <artifactId>helloProvideSU</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>jbi-service-unit</type>
    </dependency>
  </dependencies>

  <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:

```xml
<?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>helloProvideSA</name>
      <description>helloProvideSA description</description>
    </identification>
    <service-unit>
      <identification>
        <name>helloProvideSU</name>
        <description>helloProvideSU description</description>
      </identification>
      <target>
        <artifacts-zip>helloProvideSU-1.0-SNAPSHOT.zip</artifacts-zip>
        <component-name>petals-bc-soap</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

Before deploying this service assembly on the bus, we must package it. Go to the helloProvideSA folder and type:

```bash
mvn clean package
```

As you can see in the service assembly jbi.xml file, the service unit is configured to use the component petals-bc-soap. So we have to install it.

So now following this steps:

1. Launch PEtALS (if not already launched):
   ```bash
   <PEtALS Dir>/bin/startup.sh
   <PEtALS Dir>/bin/startup.bat
   ```

2. Install the soap binding component:
   Copy the file petals-bc-soap-3.1.2.zip into the PEtALS install folder <PEtALS Dir>/install.
   You must see logs on the PEtALS console:

3. Component 'petals-bc-soap' successfully installed
   Component initialized
   Start Component
   Starting Jetty server...
   Port : 8084 / Jetty Max poolsize : 50 / Jetty Min poolsize : 2 / Jetty Acceptors size : 4
   Component Information is available at http://10.134.150.23:8084
4. Now the service assembly is deployable:

Copy the archive located in `<workspace>/helloProvideSA/target/helloProvideSA-1.0-SNAPSHOT.zip` into the PETALS install folder: `<PETALS Dir>/install`.

You must see logs on the PETALS console:

- `[petals.DeploymentService]-INFO 2008-08-27 18:14:31,993` Service Assembly 'helloProvideSA' successfully deployed
- `[petals.container.components.petals-bc-soap]-INFO 2008-08-27 18:14:32,982` New Service Endpoint has been deployed: `{http://weblab-project.org/services/analyser}AnalyserService -> hello (INTERNAL):subdomain1/0/petals-bc-soap
- `[petals.container.components.petals-bc-soap]-INFO 2008-08-27 18:14:32,982` Providing access to external service 'http://localhost:8080/helloProject' in mode 'SOAP'
- `[petals.DeploymentService]-INFO 2008-08-27 18:14:33,000` Service Assembly 'helloProvideSA' successfully started

Testing the deployment

In order to test the hello endpoint, we can use an application called `sample-client`.

This application is a service engine (a component inside the bus) which is able to see all the deployed endpoints and to test them.

Install the component `sample-client`:

Copy the file `petals-sample-client-1.5.zip` into the PETALS install folder `<PETALS Dir>/install`.

You must see logs on the PETALS console:

- `[petals.InstallationService]-INFO 2008-08-27 18:01:38,427` Component 'petals-sample-client-1.5' successfully installed
- `[petals.container.components.petals-sample-client-1.5]-INFO 2008-08-27 18:01:40,025` init
- `[petals.container.components.petals-sample-client-1.5]-INFO 2008-08-27 18:01:40,027` start

The application is automatically launched (see figure 6.3, page 81). After select the `Query` tab, and click on the “Find all the endpoints” button. The hello Endpoint should appears (see figure 6.4, page 82) in the endpoint list. Double click on the hello Endpoint, the application automatically returns to the `Send` tab with the right parameters.

In order to call the endpoint, change the operation name to `process`, select the InOut message exchange pattern and write a request (with the parameter wrapper) for the endpoint (see figure 6.5, page 83).

Here is a valid request for the hello endpoint:

```xml
<typ:processArgs xmlns:ex="http://weblab-project.org/services/exception" xmlns:typ="http://weblab-project.org/services/analyser/types">
</typ:processArgs>
```

Now you can push one of the send buttons (either synchronous or asynchronous mode) and see the response (see figure 6.5, page 83).
6.1. PETALS INTEGRATION

Figure 6.3: Main PETALS sample client tab.
At this point, we have a working endpoint deployed on the bus.
6.1. PETALS INTEGRATION

![Sample Client after a calling on the hello endpoint.](image)

6.1.3 Exposing an endpoint to the outside

It's exactly the same process as before:

- building a SU,
- building a SA containing the SU,
- deploy the SA.
6.1. PETALS INTEGRATION

Endpoint exposition

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

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

This command creates the folder `helloExposeSU`:

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

```xml
<?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:consumes interface-name="analyser:Analyser"
service-name="analyser:AnalyserService" endpoint-name="hello">
<!-- CDK specific fields -->
<petalsCDK:mep>InOut</petalsCDK:mep>
<!-- SOAP specific fields -->
<soap:address>hello</soap:address>
<soap:synchronous-timeout>0</soap:synchronous-timeout>
<soap:mode>SOAP</soap:mode>
</jbi:consumes>
</jbi:services>
</jbi:jbi>
```

This file precise to the SOAP BC that it have to deploy a web service access (at the address `hello`) and when this service is called, it must call the endpoint `hello`.

We have to install it in our local repository, to do this just type this command in the `helloExposeSU` folder:

```bash
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:

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

This command creates the folder `helloExposeSA`:

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmns: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>
</project>
```
6.1. PETALS INTEGRATION

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

<!-- Dependencies -->
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.2</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.weblab.example</groupId>
<artifactId>helloExposeSU</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>
```

and the jbi.xml file:

```xml
<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>helloExposeSA</name>
<description>helloExposeSA description</description>
</identification>
<service-unit>
<identification>
<name>helloExposeSU</name>
<description>helloExposeSU description</description>
</identification>
</service-unit>
<target>
<artifacts-zip>helloExposeSU-1.0-SNAPSHOT.zip</artifacts-zip>
</target>
</service-assembly>
</jbi>
```

Now you can package the service assembly and deploy it in PETALS. Go to the helloExposeSA folder and type:

`mvn clean package`

Now copy the archive located in `<workspace>/helloExposeSA/target/helloExposeSA-1.0-SNAPSHOT.zip` into the PETALS install folder: `<PETALS Dir>/install/`. 

2009-01-23
6.2. PORTLET INTEGRATION

You should see logs on the PETALS console:

```
Service Assembly 'helloExposeSA' succesfully deployed

[petals.container.components.petals-bc-soap].INFO 2008-08-27 18:16:45,771
The Axis2 service 'hello' has been registered and is available at
'http://10.134.150.23:8084/petals/services/hello'
```

Now we can test this web service using the JWebServiceClient just like in the web service tutorial.

Just add a new service with this url:
```
```

⚠️ This exposed service do not provide the WSDL file needed by the client. So the client is going to ask you to localise a valid WSDL file for this service. Just point to the Analyser.wsdl file used by the helloProvideSU in the src/main/jbi/interfaces folder.

![Image of JWebServiceClient calling the ESB.](image)

#### Figure 6.6: JWebServiceClient calling the ESB.

6.2 Portlet integration

6.2.1 Updating portlets to connect to services exposed in the ESB

For this part, you must have developed the two portlets or used the provided archive. You just have to change location of service.
And now, you will have this code:

```java
public class HelloPortlet1 extends GenericPortlet {

    public void processAction(ActionRequest req, ActionResponse res)
        throws IOException, PortletException {
            // get the content of the submitButton parameter.
            String value = req.getParameter("submitButton");
            System.out.println("Received " + value);
            if (value.equals("SendToService")) {
                DocumentAnalysisService service = new DocumentAnalysisService();
                ProcessArgs in = new ProcessArgs();
                in.setMediaUnit(doc);
                try {
                    Analyser analyser = service.getAnalyserPort();
                    setEndpointAddress(analyser, "http://localhost:8084/petals/services/hello");
                    doc = (Document) analyser.process(in).getMediaUnit();
                } catch (ProcessException e) {
                    e.printStackTrace();
                }
            }
        }

    [...]
```

You now have to compile and package and deploy again your portlet application with `mvn clean package`.
This chapter presents how to create an orchestration chain using BPEL in the “WebLab platform”. To achieve it we use Orchestra as a service engine in the PEtALS ESB.

7.1 BPEL overview

BPEL is a powerful XML language used to orchestrate services. Thanks to it, we are able to chain services, assign and manipulate parameters and variables.

7.2 Endpoint configuration

7.2.1 Scope of the orchestration tutorial

For this section, you must have installed Tomcat with helloProject and testProject services online. We are going to chain this two services and expose this chain as a service on the bus.

To fully understand the orchestration part, you must have done the PEtALS integration chapter: 6.1, page 74. In this part we had to deploy the hello endpoint on PEtALS, now we are going to deploy the test endpoint.

7.2.2 Installing the Orchestra service engine

We provide a patched version of Orchestra service engine. To install it, simply get the zip file named petals-se-orchestra-1.0-patch3.zip and copy it to the PEtALS install folder.

You must see this logs:

```
 Component 'petals-se-orchestra' successfully installed
```
7.2. ENDPOINT CONFIGURATION

Component Framework configuration:
- JBI Acceptors size : 5
- JBI Workers size : 10
- Ignored Message Status : DONE_AND_ERROR_IGNORED
- Ack Timeout : 5000 ms
- JBI Listener : org.ow2.petals.engine.orchestra.listener.JBIListener
- External Listener :

Component parameters:
- Orchestra Initialization...
- Component initialized

Start Component

7.2.3 Creating a new service unit

Go to the same workspace directory as in the PEtALS part, 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=testProvideSU -Dversion=1.0-SNAPSHOT
```

This command creates the folder testProvideSU:

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

Change 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="analyser:AnalyserService" endpoint-name="test">
      <!-- CDK specific fields -->
      <petalsCDK:wSDL:interfaces/AnalyserService.wsd1</petalsCDK:wSDL>
      <!-- SOAP specific fields -->
  </jbi:services binding-component="true"/>
</jbi:jbi>
```

2009-01-23
7.2. ENDPOINT CONFIGURATION

This is exactly the same file as with the hello endpoint. The only changing thing is the endpoint name which is test and, of course, the service address http://localhost:8080/testProject. You also need to copy the WSDL file and its dependencies (xsd files) in the src/main/jbi folder.

Now you can install this service unit in your local maven repository.

To do this just type this command in the testProvideSU folder: mvn clean install

7.2.4 Creating the service assembly

To deploy it on PEtALS there is two solutions, you can create a new service assembly or add the new service unit to an already created service assembly.

The easiest way is to modify the existing one but for more clarity, we are going to create a new service assembly.

So 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.petals -DartifactId=testProvideSA -Dversion=1.0-SNAPSHOT

This command creates the folder testProvideSA:

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

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

```xml
<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>
   <name>SA :: testProvideSA</name>
   <artifactId>testProvideSA</artifactId>
   <groupId>org.objectweb.petals</groupId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jbi-service-assembly</packaging>
   ...
</project>
```
7.2. ENDPOINT CONFIGURATION

```xml
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.weblab.example</groupId>
    <artifactId>testProvideSU</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>jbi-service-unit</type>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.objectweb.petals</groupId>
      <artifactId>maven-petals-plugin</artifactId>
      <version>1.0.0</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

The next step is to edit the jbi.xml file:

```xml
<?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>testProvideSA</name>
      <description>testProvideSA description</description>
    </identification>
    <service-unit>
      <identification>
        <name>testProvideSU</name>
        <description>testProvideSU description</description>
      </identification>
      <target>
        <artifacts-zip>testProvideSU-1.0-SNAPSHOT.zip</artifacts-zip>
        <component-name>petals-bc-soap</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

Before deploying this service assembly on the bus, we must package it. Go to the testProvideSA folder and type:

```
mvn clean package
```

7.2.5 Deploying the service assembly

Be sure that PEtALS is running and that the bc-soap component is deployed (like explain in the PEtALS integration chapter).
7.2. ENDPOINT CONFIGURATION

Now just copy the zip file `<workspace>/testProvideSA/target/testProvideSA-1.0-SNAPSHOT.zip` into the PEnALS install folder: `<PEnALSDir>/install/`

You must see logs on the PEnALS vconsole:

```
[petals.DeploymentService]-INFO 2008-09-23 16:53:04,017
Service Assembly 'testProvideSA' succesfully deployed
[petals.container.components.pets-bc-soap]-INFO 2008-09-23 16:53:04,100
New Service Endpoint has been deployed:
   [http://weblab-project.org/services/analyser]AnalyserService
   ->test (INTERNAL):subdomain1/0/pets-bc-soap
[petals.container.components.pets-bc-soap]-INFO 2008-09-23 16:53:04,100
Providing access to external service 'http://localhost:8080/testProject' in mode 'SOAP'
[petals.DeploymentService]-INFO 2008-09-23 16:53:04,102
Service Assembly 'testProvideSA' succesfully started
```

7.2.6 Deployment testing

In order to test the endpoint deployment, as in the PEnALS integration chapter, we are going to use the pets-sample-client component. So the first step is to check that the new endpoint is correctly installed. In order to do this select the query tab, and click on the fill all the endpoints button (see figure 7.1 page 102).

You have to see two deployed endpoints: hello and test.

Now you can call the test endpoint using the sample client. Double click on the test endpoint, and the client switch automatically to the send tab. Set the operation field to process and the Message exchange pattern to InOut. Here is a valid request for the hello endpoint:

```
<typ:processArgs xmlns:ex="http://weblab-project.org/services/exception"
                xmlns:typ="http://weblab-project.org/services/analyser/types">
  <resource xmlns:wl="http://weblab-project.org/core/model/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            synchronised="false" uri="weblab://tutorial/my.document"
            xsi:type="wl:document" />
</typ:processArgs>
```

You can now click on the send - accept button. You shall see the same response as in the figure figure 7.2 page 103.

Here is the complete response for this endpoint:

```
<ns3:processReturn xmlns:ns3="http://weblab-project.org/services/analyser/types"
                   xmlns:ns2="http://weblab-project.org/services/exception">
  <resource xmlns:ns5="http://weblab-project.org/core/model/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            synchronised="false" uri="weblab://tutorial/my.document"
            xsi:type="ns5:document">
    <annotation uri="weblab://tutorial/my.document/annot1">
      <data>
        <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                 xmlns:weblab="http://weblab.eads.com/">
          <rdf:Description rdf:about="weblab://tutorial/my.document">
            <weblab:date>Tue Sep 23 17:25:00 CEST 2008</weblab:date>
          </rdf:Description>
        </rdf:RDF>
      </data>
    </annotation>
  </resource>
</ns3:processReturn>
```
7.3 Service Unit chain creation

In this section, we are going to create a BPEL chain which will be exposed as an endpoint on the bus and which will call two endpoints, hello and test.

7.3.1 Endpoints are not only web services

As the Orchestra engine is used as a JBI component and not as a standalone application, we are not going to chain directly web services but generic endpoints. Moreover JBI uses the same mechanism as web services to address endpoints (WSDL). So in BPEL engine we are going to use WSDL and Partner Links to point on a specific endpoint as we use it without the bus.

7.3.2 Partner Link and Orchestra patch

A Partner Link is used by the orchestrator to fetch the endpoint interface, in our case: {http://weblab-project.org/services/analyser/}Analyser. So the BPEL process must know this interface and its definition (the WSDL file).

So in BPEL when you call an endpoint, you must specify a partner link which defines the Interface to call. But the problem is that, in this case, you only precise an Interface and not the Service and the Endpoint. Normally, in BPEL you are able to assign dynamically this two values on a partner link. But, for the moment, Orchestra does not handle it.

In order to solve this issue, we have patched the Orchestra engine to be able to precise on each partner link, the service and endpoint values.

7.3.3 Service unit creation

As always, a BPEL process is a service unit and is deployed on a service engine (petals-se-orchestra). So you have to create a service unit which will contains the BPEL file:

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

This command creates the folder chainProvideSU:
7.3. SERVICE UNIT CHAIN CREATION

As we are going to create a chain that is an Analyser, you must modify the jbi.xml file like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:petals="http://petals.ow2.org/extensions"
  xmlns:wsdl="http://weblab-project.org/services/analyser"
  xmlns:jbi="http://java.sun.com/xml/ns/jbi" version="1.0">
  <jbi:services binding-component="false">
    <jbi:provides interface-name="wsdl:Analyser" service-name="wsdl:AnalyserService" endpoint-name="tutorialChain">
      <petals:wsdl>interfaces/Analyser.wsdl</petals:wsdl>
      <petals:su-interceptors></petals:su-interceptors>
      <petals:params>
        <petals:param name="bpel">AnalyserService.bpel</petals:param>
        <petals:param name="map">mapPL.xml</petals:param>
      </petals:params>
    </jbi:provides>
  </jbi:services>
</jbi:jbi>
```

7.3.4 Creating the BPEL artifact file

As we need the WSDL file which defines the Analyser interface, you have to copy interfaces and model folders inside the src/main/jbi folder.

After that you can start to define Partners Links which are going to be used by the process. In fact we need three of them, one for the hello endpoint, one for the test endpoint, and the last one for the chain itself, because don’t forget that the chain will be exposed as an endpoint.

To define a Partner Link you have to start by declaring its type inside a new WSDL file that we will call AnalyserServiceArtifacts.wsdl:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
  xmlns:tns="http://weblab-project.org/services/analyserArtifacts"
  xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
  xmlns:wsdl="http://weblab-project.org/services/analyser"
  name="AnalyserServiceArtifacts"
  targetNamespace="http://weblab-project.org/services/analyserArtifacts"
  xmlns:soap="http://schemas.xmlsoap.org/soap/">
  <!-- Partner link type definition refering to the portType
   {http://weblab-project.org/services/analyser}Analyser
   -->
  <plnk:partnerLinkType name="AnalyserPLType">
    <plnk:role name="AnalyserPLRole" portType="wsdl:Analyser" />
  </plnk:partnerLinkType>
  <!-- Import of Analyser.wsdl which defines
   {http://weblab-project.org/services/analyser}Analyser portType
   -->
</definitions>
```
7.3. SERVICE UNIT CHAIN CREATION

You can see that this definition is quite simple, it’s just namespaces declaration, after this Partner Link Type declaration and an import for the WSDL file containing the PortType definition.

So with this file we have declared a Partner Link Type called AnalyserPLType linked to the {http://weblab-project.org/services/analyser/} Analyser interface.

As the three needed partners links use the same interface, we are going to use this freshly created AnalyserPLType for the three partners.

7.3.5 Creating BPEL file

Here is the BPEL file, AnalyserService.bpel:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
This is a sample BPEL file which implements an AnalyserService.
For the moment, be careful that the process name and targetNamespace
are the same as the service definition in the jbi.xml file.

This chain is extract from the WebLab tutorial, it call first the hello endpoint
and second the test endpoint.

Note: for the moment, Orchestra do not support the "exitOnStandardFault"
attribute, so check that it is set to "no".
-->
<bpws:process exitOnStandardFault="no" name="AnalyserService"
  suppressJoinFailure="yes"
targetNamespace="http://weblab-project.org/services/analyser"
xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:ns="http://weblab-project.org/services/analyserArtifacts"
xmlns:tns="http://weblab-project.org/services/analyser">
  <!-- BPEL imports, here imports WSDL files
  -->
  <bpws:import importType="http://schemas.xmlsoap.org/wsdl/
  location="interfaces/Analyser.wsdl"
  namespace="http://weblab-project.org/services/analyser" />
  <!-- This one is used to get the PartnerLinkType "AnalyserPLType"
  used for the 3 defined partnerLinks.
  -->
  <bpws:import importType="http://schemas.xmlsoap.org/wsdl/
  location="AnalyserServiceArtifacts.wsdl"
  namespace="http://weblab-project.org/services/analyserArtifacts" />

  <!-- PartnerLinks definition

  The first one, "Client", is for the entire chain (myRole).
  The two others, "Hello" and "Test", are for the endpoints calls (partnerRole).
  -->
  <bpws:partnerLinks>
    <bpws:partnerLink myRole="AnalyserPLRole" name="Client"
      partnerLinkType="ns:AnalyserPLType" />
```

2009-01-23

7.3. SERVICE UNIT CHAIN CREATION

```xml
<bpws:partnerLink name="Hello"
    partnerLinkType="ns:AnalyserPLType" partnerRole="AnalyserPLRole" />
<bpws:partnerLink name="Test"
    partnerLinkType="ns:AnalyserPLType" partnerRole="AnalyserPLRole" />
</bpws:partnerLinks>

<!-- We need two variables, one for the processRequest, and one for the processResponse. The messageType is defined inside the WSDL file Analyser.wsdl. -->
<bpws:variables>
    <bpws:variable messageType="tns:processRequest" name="AnalyserRequest" />
    <bpws:variable messageType="tns:processResponse" name="AnalyserResponse" />
</bpws:variables>

<!-- Here the process starts. -->
<bpws:sequence>

    <!-- The receive element is the chain entry point. The partnerLink used is "Client"  (because a receive must use a "myRole" partnerLink. The parameter passed through the chain is written in the "AnalyserRequest" variable. This is the first receive, so you have to set "createInstance" attribute to "yes". -->
    <bpws:receive createInstance="yes" name="Receive"
        operation="process" partnerLink="Client" portType="tns:Analyser"
        variable="AnalyserRequest" />

    <!-- First endpoint invocation, here on "Hello" partnerLink (partnerRole). Note that it uses directly the "AnalyserRequest" variable, because the chain and the called endpoints share the same service interface and so the same service parameters. -->
    <bpws:invoke inputVariable="AnalyserRequest" name="InvokeHello"
        operation="process" partnerLink="Hello" portType="tns:Analyser"
        outputVariable="AnalyserResponse" />

    <!-- Here we copy the result of the first endpoint into the request of the second one. ".return" and ".args" names are set in the Analyser.wsdl file (with the service interface). -->
    <bpws:assign name="AssignTest" validate="no">
        <bpws:copy>
            <bpws:from><![CDATA[$AnalyserResponse.return/resource]]></bpws:from>
            <bpws:to><![CDATA[$AnalyserRequest.args/resource]]></bpws:to>
        </bpws:copy>
    </bpws:assign>

    <!-- Second call on the "test" endpoint. Exactly the same as the "hello" one except the partnerLink name ("Test"). -->
```

2009–01–23 96/114
7.3. SERVICE UNIT CHAIN CREATION

Read carefully the comments inside the file. Here are listed some important points (if you do not respect it, Orchestra engine will not be able to deploy the chain):

- attribute exitOnStandardFault on the whole chain must be set to “no”
- attribute on the receive element createInstance must be set to “yes”
- BPEL process name (targetNamespace and processName) must match the endpoint service definition (service-name inside the jbi.xml file)

7.3.6 Creating a mapping between partner links and endpoints

Orchestraservice engine patch need a file to map endpoints and partner links. This file location is already set in the jbi.xml file which you must have already replace:

<petals:param name="map">mapPL.xml</petals:param>

So you have to create a file called mapPL.xml in the src/main/jbi folder. Now lets explain the content of this file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mapPL>
  <!--
  This file is used by the patched orchestra service engine.
  Here you can precise the endpoint you want to call for one partner link.
  -->
  <!--
  Hello partner link will call the hello endpoint which is deployed with this service-name:
  {http://weblab-project.org/services/analyser}AnalyserService
  -->
  <entry plName="Hello">
    <service-ref xmlns="http://docs.oasis-open.org/wsbpel/2.0/serviceref">
      <EndpointReference xmlns="http://www.w3.org/2005/08/addressing">
        <Address>hello</Address>
      </EndpointReference>
    </service-ref>
  </entry>
</mapPL>
```
7.3. SERVICE UNIT CHAIN CREATION

In this file you have to list all partner links used in your BPEL processes. You just have to create an entry element for each partner link and precise its name (this name must match the partner link name inside the BPEL process) in the element attribute "plName".

Inside this element, add a service-ref element correctly filled with the endpoint information ("ServiceName" and "Address" in the map refers to "service-name" and "endpoint-name" in the jbi.xml file).

7.3.7 Deployment on the bus

So now the service unit is ready to be installed in your local repository.

mvn clean install

Now, like in the beginning of this chapter, you have to create a service assembly for this service unit.

Create it using the same maven command (type it inside your workspace folder):

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

After that change the pom.xml file to add the dependency to your service unit:

```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/xsd/maven-4.0.0.xsd">
```
7.3. SERVICE UNIT CHAIN CREATION

And the jbi.xml file to add the service unit declaration:

```xml
<?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>chainProvideSA</name>
      <description>chainProvideSA description</description>
    </identification>
    <service-unit>
      <identification>
        <name>chainProvideSU</name>
        <description>chainProvideSU description</description>
      </identification>
      <target>
        <artifacts-zip>chainProvideSU-1.0-SNAPSHOT.zip</artifacts-zip>
        <component-name>petals-se-orchestra</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```
Note the petals-se-orchestra component name and not petals-bc-soap.
Now you are able to package it:

mvn clean package

And at the end, deploy it on PEtALS (copy the generated zip from the service assembly to the PEtALS installation folder). Be sure that the petals-se-orchestra component is installed on PEtALS.

You have to see a lot of logs inside the PEtALS console like XSD and WSDL parsing.
Logs must finish by:

Using the sample client you have to see the freshly deployed endpoint tutorialChain (see figure 7.3 page 104).

7.3.8 Testing the chain

To test it, we are going to use the PEtALS sample client. Just select the right endpoint: “tutorialChain“, the right operation: “process“ and the right message exchange pattern: “InOut“. This part is already explain in the PEtALS integration chapter (see figure 7.4 page 105).

Use an empty WebLab resource wrapped in the processArgs as input:

The chain have to respond something like this:
And you can see PEtALS logs which confirm that the two endpoints have been called:

```
[petals.container.components.petals-sample-client-1.5]-INFO 2008-09-26 18:59:51,224
SampleClient try to send
[petals.container.components.petals-se-orchestra]-INFO 2008-09-26 18:59:51,435
Start synchronous process
[petals.container.components.petals-se-orchestra]-INFO 2008-09-26 18:59:51,435
Start the process on tutorialChain
[petals.container.components.petals-se-orchestra]-INFO 2008-09-26 18:59:51,435
Ready to launch the BPEL process
[petals.container.components.petals-se-orchestra]-INFO 2008-09-26 18:59:51,435
Launch the BPEL process
called endpoint service: {http://weblab-project.org/services/analyser}AnalyserService
called operation: process
MessageExchnage created: http://www.w3.org/2004/08/wsdl/in-out
Set the service to invoke: {http://weblab-project.org/services/analyser}AnalyserService
Set the endpoint to invoke: hello
Receive response from: hello
[petals.container.components.petals-se-orchestra]-INFO 2008-09-26 18:59:52,094
Receive response from: test
[petals.container.components.petals-se-orchestra]-INFO 2008-09-26 18:59:52,101
Launching of the BPEL process is done
Processing received message petals:uid:B93A0F3D-44D7B3CB1222448391419030001
```
### 7.3. SERVICE UNIT CHAIN CREATION

#### Figure 7.1: Sample client endpoints list with `hello` and `test` endpoints.

<table>
<thead>
<tr>
<th>Action on the selected endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get service description</td>
</tr>
<tr>
<td>Get interfaces</td>
</tr>
<tr>
<td>Get selected endpoint as reference</td>
</tr>
</tbody>
</table>
7.3. SERVICE UNIT CHAIN CREATION

Figure 7.2: Sample client calling the test endpoint.
7.3. SERVICE UNIT CHAIN CREATION

Figure 7.3: Sample client endpoints list with hello, test and tutorialChain endpoints.
### 7.3. SERVICE UNIT CHAIN CREATION

#### Figure 7.4: Sample client calling the tutorial chain.

![Sample client calling the tutorial chain](image)

- **Service Interface**: `http://weblab-project.org/services/analyserr/Analyserr`
- **Endpoint Operation**: `tutorialChain process`
- **Message Exchange Pattern**:
  - In Only
  - In Out
  - In Optional Out
  - Robust In Only
- **Request / Response Body**:
  - **In**:
    ```xml
    <ns3:processArgs>
        <resource xmlns:ns3="http://weblab-project.org/services/analyserr/types">
            <xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ns3:document="true"/>
        </resource>
    </ns3:processArgs>
    ```
  - **Out**:
    ```xml
    <ns3:processReturn>
        <mediaUnit xmlns:ns3="http://weblab-project.org/core/model/text" uri="tutorial/my.document#0" ns3:text="Hello">
            <content>Hello</content>
        </mediaUnit>
    </ns3:processReturn>
    ```

The diagram shows a sample client calling the tutorial chain, illustrating the interaction with the service interface and the message exchange pattern. The XML code snippet demonstrates the structure of the request and response bodies, highlighting the use of namespaces and XML elements.
8.1 Java Development Kit

To develop a Web service in Java, you need to have a Java Development Kit (JDK) installed. You can check this requirement by using the following command `javac -version`.

The current stable version of JDK is 1.6. But there is several different release of the same JDK which provide different version of the libraries already contained in the API. In our case, we are interested by two major libraries: JAXB (which enable to bind XML and Java objects) and JAXWS (which is the Java API for web services). To ease the development of web services and to keep compatibility with the major Java recommendation on web services, we need at least JAXB 2.1.x and JAXWS 2.1.x. The latest public release of JDK 1.6 is stable and contains the good version of the two needed API (JAXB 2.1 and JAXWS 2.1) but at this time there are still some annoying unresolved bugs (LocalString bug). Thus we recommend you to install the last JDK 1.5.x and further information will be delivered in order to update the kit when JDK 1.6 will be compatible.

To use a JDK version 1.5.x, you will include the required versions of JAXB and JAXWS directly in your project (remarks will be added for Java 5 users). Don’t forget the `JAVA_HOME` variable as explained below.
Many computer can host several Java Runtime Environment, ensure that the one called by java and javac commands is a 1.5.x one. For this, fix the environment variable JAVA_HOME.

```
export JAVA_HOME=<directory where you installed Java>
```

```
SET JAVA_HOME=<directory where you installed Java>
```

Or use the menu through:

- Click on Start button.
- Select MyComputer.
- Click right button on Windows background.
- Select and click Properties.
- Select Advanced System Parameters.
- Click Environment Variables button.

In the System Variables form:
- Select PATH variable.
- Click on Modify to change PATH variable add the path to JDK binaries.
  
  Example: add the chain C:\Program Files\Java\jdk1.5.x-x\bin to the \PATH. Don’t forget to add a semi-colon character at end of PATH before adding you JDK’s binary path.

- Click on New button to create JAVA_HOME variable.
  
  Example: create variable JAVA_HOME with value C:\Program Files-\Java\jdk1.5.x-x

- Then click twice on OK to validate your modifications.

Note that to keep the setted variables after reboot, those commands have to be written in the boot script loaded by your system.

## 8.2 Tomcat Application Server

Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages technologies. It is a Java based web server which can host web application and especially web services. Download the Zip package (platform independent) of Tomcat Core 6.0.x at:
8.3. MAVEN

Configure a login/password for Tomcat manager which allow to easily deploy services. This can be done by editing the `<tomcatfolder>/conf/tomcat-users.xml` file and adding the two bold lines presented in listing 8.1.

```
<!-
  NOTE: By default, no user is included in the "manager" role required
to operate the "/manager" web application. If you wish to use this app,
you must define such a user - the username and password are arbitrary.
-->
<tomcat-users>
  <user username="tomcat" password="tomcat" roles="tomcat" />
  <user username="role1" password="tomcat" roles="role1" />
  <user username="both" password="tomcat" roles="tomcat,role1" />
  <role rolename="manager" />
  <user username="admin" password="admin" roles="manager" />
</tomcat-users>
```

Listing 8.1: Configuration of Tomcat through the Tomcat-users.xml file.

Launching the Tomcat server is quite simple:

```
<Tomcat directory>/bin/startup.sh
<Tomcat directory>in\startup.bat
```

Test if the installation of Tomcat was successful, you can open the local server address `http://localhost:8080/` with your prefered browser which should display the Tomcat server homepage as in figure 8.1.

Advanced users of Tomcat may have defined the CATALINA_HOME environment variable to point to Tomcat home folder. This could induce conflict with embedded Tomcat in the portal, thus it is recommended to remove this variable.


8.3 MAVEN

Maven is a Java application (no OS dependency) which allows to manage software projects. Its main capabilities are to handle the dependencies and to generate project report and project bundle.

Download its 2.0.9 version (or later) as the provided repository is configured with this version implementation at `http://maven.apache.org/download.html` and follow the installation instruction provided.

Do not forget to include the Maven bin directory into your path, as explained in the installation instructions, in order to be able to call the `mvn` command from any folder.
Test the correct installation of Maven by calling `mvn -version` from a console. It should respond the Maven version installed.

In order to simplify the installation, the development kit contain a complete copy of a working Maven directory which contain the minimum libraries needed to achieve this tutorial in offline mode.

Just unzip the `<WDK>/tools/maven/repository.zip`. 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.

```
<settings>

<!-- 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 `setting.xml`, such as: `<offline>true</offline>`.

For further informations about Maven you might refer to the current online documentation, available at: [http://maven.apache.org/guides/index.html](http://maven.apache.org/guides/index.html).
8.4 PEtALS

PEtALS is an open source ESB. You can download it at http://forge.objectweb.org/project/showfiles.php?group_id=213. Choose the appropriate version of the PEtALS Standalone light distribution, the sample client and the binding component SOAP. In this tutorial we are using:

- petals-standalone-2.2.1-light.zip,
- petals-sample-client-1.5.zip and
- petals-bc-soap-3.1.2.zip.

Be careful with components version (especially BC-SOAP), because configuration in jbi.xml files may change with the versions.

To install PEtALS, just unzip the PEtALS archive into a folder that we will call (<PEtALS Dir>). Launch it using the following command lines:

PEtALS Dir>/bin/startup.sh -C

PEtALS prompt. Type 'help' for help.

petals@localhost:>

8.5 Liferay Portal


To install Liferay Portal, just unzip the package in a folder that we will call (<LiferayDir>), and launch it using the following commands:

<LiferayDir>/bin/startup.sh

<LiferayDir>/bin/startup.bat
Test the correct installation of Liferay by opening the local server address http://localhost:8081/web/guest/home to obtain something like in figure 8.2.

You can access to private page of the default user opening the address http://localhost:8081/user/joebloggs/home.

You have to sign in using test@liferay.com as login and test as password. Then you should see something like in figure 8.3:

To install new portlets, use the Plugin Installer Portlet (administrator rights needed) using the add application link (in the top right menu of the page see figure 8.4). When the left menu is opened, just drag and drop Plugin installer portlets to the page.

You should obtain something like in figure 8.5:

After that, just go on the “Upload File” tab and select the war corresponding to portlet to install and click “Install” (figure 8.6).

Check Liferay logs to be sure that the portlet deployment is completed and successful (see figure 8.7).

Deployed portlets are now accessible throw “add application” menu. Just drag and drop its on pages.

More information about Liferay Portal usage and administration can be found at http://www.liferay.com/web/guest/community/documentation/5_1.
8.6  JWebServiceClient

JWebServiceClient is a java application which can send and receive SOAP messaged. Thus it allows to validate:

- the service availability
- the WS-* recommandations respect
- the service inputs and outputs

Despite its Java implementation, the application can test any Web Service that provides a WSDL file in an online or offline basis.

The client is delivered as a compressed file JWebServiceClient-2.0-SNAPSHOT.zip. To install it, just uncompress the zip file into a directory. For the next steps, it will be called <JWebClientDir>. Open this folder and launch the start script adapted to your operating system:

- <JWebClientDir>/start.sh
- <JWebClientDir>/start.bat

For more information about the client, see it's own documentation.
8.6. JWEBSERVICECLIENT

Figure 8.4: Adding Plugin installer portlets on the page in Liferay Portal.

Figure 8.5: Plugin installer portlet on a page in Liferay Portal.
Figure 8.6: Plugin installer portlet after war upload.

Figure 8.7: Liferay logs after war deployment.