Spago
User Guide

Authors

Luigi Bellio,
Gianfranco Boccalon,
Daniela Butano,
Monica Franceschini,
Luca Fiscato,
Andrea Zoppello

1 Document Goal

The document presents the Spago framework functionalities and the guidelines about the project configuration implemented with Spago.

Versions History

Version/Release n° :	1.0	**Version Date/Release** :	March, 10th 2005
Description:	First release (english version)		
Version/Release n° :	1.1	**Version Date/Release** :	April, 21th 2005
Description:	Some grammatical and linguistic changes		
Version/Release n° :	1.5	**Version Date/Release** :	August, 10th 2005
Description:	Fixed the Navigator paragraph: removed some unsupported options (NAVIGATOR_BACK_TO_SERVICE).		
Version/Release n° :	1.6	**Version Date/Release** :	October 10, 2005
Description:	Added the “splitting option” to the configuration handling. Added the possibility to substitute the validation engine.		
Version/Release n° :	1.7	**Version Date/Release** :	November 30, 2005
Description:	Modified the documentation about the validation, and added the paragraph on the agent dependent publishing(AZ)		

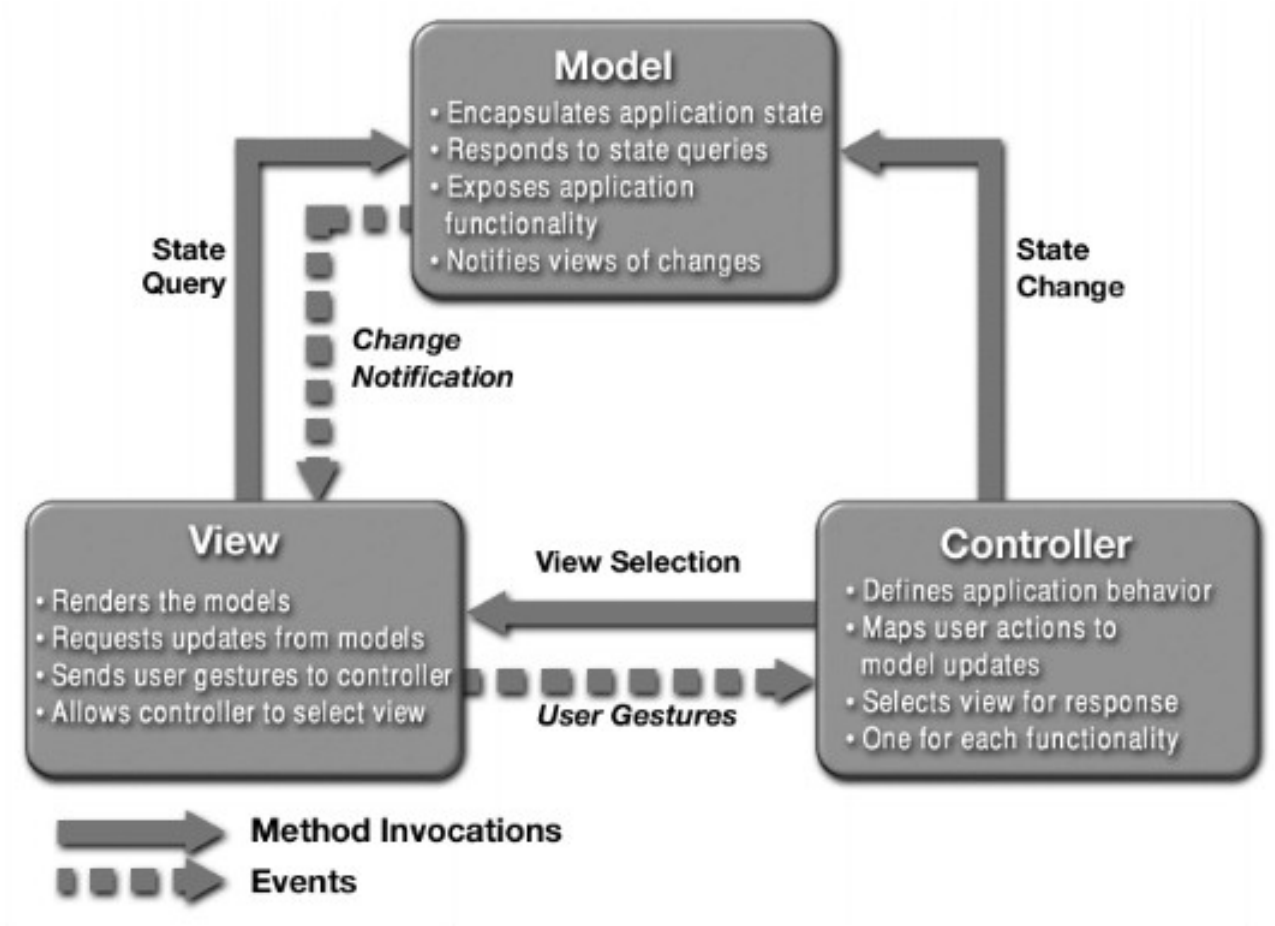
2 Introduction

The framework's development started in 1999 to meet the requirements of a complex project for enterprise business.
The main goal was to use a software layer abstracting the common requirements for a web application like database management, caching mechanisms and so on. During these years the software has been expanded and improved as the result of the development of many enterprise projects. The result of the framework's reengineering and coding refactoring is the Spago Framework.
The framework implements the MVC (Model-View-Controller) architectural pattern and can be considered not only a good product but also a working methodology. By means of Spago designers and developers responsibilities are separated. The main advantage of using a framework (like Spago) is to gain a higher productivity because in this way your are forced to use an application structure that enhances code extensibility and maintenance.
Actually Spago is a free open source project and it can grow with new modules to increase functionalities and it can be improved in order to avoid any limit and to fix bugs.

2.1Reference Architecture

Spago implements the Model-View-Controller architectural pattern, organized by three tiers:

- **Presentation tier** : HTTP to web container, SOAP, WAP, EJB (soon: HTTP to portlet container, TCP/IP)
- **Business tier** : controls, elaborations
- **Integration tier** : towards data source or transactional services.



2.2Multichannel

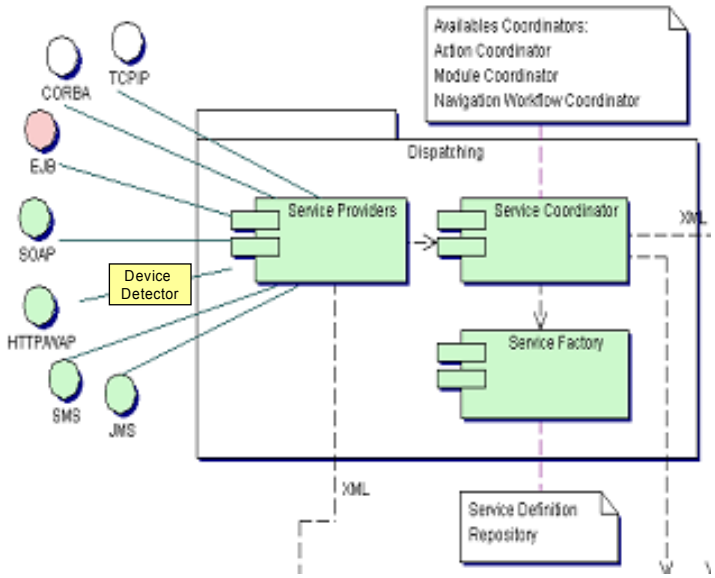
Spago supports client interaction via different channels/protocols, using Spago you can easily dispatch your services to different channels: HTTP, WAP, SOAP, EJB.
Currently the channels managed from the framework, with specific adapters, are:

- **HTTP/WAP**: allows to send requests to the framework using the http protocol, which is the default protocol for web and WAP browser.
- **SOAP**: allows to send requests to the framework using the SOAP protocol (over http).
- **EJB**: allows to invoke service exposed as EJB (Enterprise Java Bean).
- **JMS**: allows the framework to receive the service requests through a JMS bus.

The adapters for the following channels are still in development:

- **Portlet**: will permit to expose the service on portlet container

Spago developers can add new adapters with a plug-in mechanism.



The adapter takes responsibility for acquiring request data from a specific channel, transforming request parameters into a format compliant with the Model module, and for choosing the correct view. It also makes the binding of conversational context in the specific container. For example, on the HTTP channel, the AdapterHTTP receives the *HttpSession* and

HttpServletRequest, which are the java objects associated to the HTTP protocol, extracts all the information and encapsulates them into internal request and response object, equals for all channels.

3 Services and Components

The framework provides various services and components, the next sections will describe them.

3.1SourceBean

The base of the framework is the XML. All the Spago subsystems communicate between them using XML streams. All these stream aren't a simply String object containing the xml but a framework object, called **SourceBean,** which can be view as a container for other objects, structured like an xml node. An xml file or string can be parsed in order to construct the correspondent SourceBean, each xml envelope is transformed in a SourceBean while the attribute are transformed in a SourceBeanAttribute object that contains the value of the attribute. So, a SourceBean is a hierarchical container of other SourceBeans and SourceBeanAttributes exactly as an xml node is a hierarchical container for other nodes and attributes. A SourceBean can be created also as a normal java object but in every moment it can be transformed into an xml document. The value of the attributes are always string if the SourceBean is constructed from a parsed file, but if it is built from developers its attributes can contain every object.
The SourceBean object is similar to the standard java object *org.w3c.dom.Document* because it is constructed as a DOM object, but it adds simpler navigation and recover services as the punctual notation navigation, for example.
As an example consider the following xml node:

```
<EVENT_CONDITIONS>
  <CONDITION driver="it.eng.spago.event.condition.impl.DefaultEventConditionDriver">
    <CONFIG
      key="new_doc"
      class="it.eng.spago.event.condition.impl.DefaultEventCondition">
    </CONFIG>
  </CONDITION>

  <CONDITION driver="it.eng.spago.event.condition.impl.DefaultEventConditionDriver">
    <CONFIG
      key="download_doc"
      class="it.eng.spago.event.condition.impl.DefaultEventCondition">
    </CONFIG>
  </CONDITION>
</EVENT_CONDITIONS>
```

Once read the xml node and constructed the correspondent SourceBean, it is possible to access to all the *CONDITION* envelopes with the instruction:

```
Vector conditions = sourceBean.getAttributeAsVector("CONDITION");
```

Note that is not necessary to specify the principal envelope ***EVENT_CONDITIONS*** because it is the root of the object. The result of the operation is a vector of SourceBean objects relative to the CONDITION envelopes. Each one of them can be analyzed and navigated recursively in order to recover information on conditions. From this example can be noticed that a SourceBean is a multi-value container: it can contain a sequence of elements with the same name.

It is possible to serialize a SourceBean into an XML document, but, because the SourceBean attributes can contains whichever type of object, there's the need of a mechanism that transforms the objects into their xml representation. Based on the object contained into the attribute, Spago solves the problem in this way:

1. If the SourceBean attribute object implements the interface *it.eng.spago.base.XMLObject* and so it implements a method to transform itself into an XML document, the system calls this method.
2. If the SourceBean attribute object doesn't implement the interface *it.eng.spago.base.XMLObject* the system calls the method *toString* of the object (every java object has the toString method), in order to recover the string representation an put it into the xml node of the SourceBean.

It is necessary to implement the interface *XMLObject* when the contents are used for various channels or better when the SourceBean information have to be presented in different way for different channels, using XSL stylesheets, for example. Otherwise, if the presentation is made only for the HTTP channel, for example, it is not necessary to implement the interface *XMLObject* because the information of the SourceBean can be recovered using directly the method of the SourceBean, without the need to transform the object into an xml document. Developers can create objects implementing the *XMLObject* interface and define all its five methods, or more simply they can extend the *AbstractXMLObject* and implement the method *toElement*(*Document*).

3.1.1Accented letters

An xml stream which has to be published must contains the doc-entity definitions for all the characters not included into the ASCII 127 standard, as the accented letters for example. The framework has a configuration file *xhtml-lat1.ent* which contains all the characters that are not part of the ASCII 127 standard. This file is used in order to produce a doc-entity to put into the xml document produced by the method *toXML* of the SourceBean object, but, because this doc-entity is too large, the system includes it only when the xml document must be passed to a transcoder

This automatism is available into the SourceBean but developers, who wants to create new objects that implement the *XMLObject* interface, have to perform the same functionality, in order to create a correct and complete xml document.

3.2Configuration

The framework is based on a set of configuration files defined into a fix and well know principal configuration file *master.xml.* This file declares all paths of the configuration files that have to be loaded. At start-up time the system analizes the *master.xml* and loads into a SourceBean all the data contained into each file of the list
Below there's an example of the *master.xml* file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MASTER>
<CONFIGURATOR
    path="/WEB-INF/conf/data_access.xml"/>
  <CONFIGURATOR
    path="/WEB-INF/conf/actions.xml"/>
  <CONFIGURATOR
    path="/WEB-INF/conf/modules.xml"/>
  <CONFIGURATOR
    path="/WEB-INF/conf/pages.xml"/>
  <CONFIGURATOR
    path="/WEB-INF/conf/presentation.xml"/>
  <CONFIGURATOR
    path="/WEB-INF/conf/publishers.xml"/>
</MASTER>
```

The xml structure and data of this file, and all those contained, are loaded into an object *ConfigSingleton,* which is a Singleton SourceBean that allows to easily access to all the configuration information of the application. In every point of the application code it is possible to retrieve configuration data using the navigation services and methods of the SourceBean. This mechanism allows to split the configuration into different files, for an easier way to find and extend information, but in the application code this division is transparent because all the elements can be recovered using a single object.
For example if the files *actions.xml* and *modules.xml* contains the following xml:

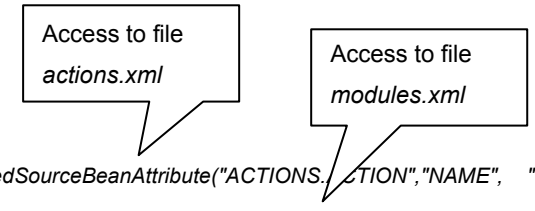
actions.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
        <ACTION
    name="LIST_USERS_ACTION"
    class="it.eng.spago.demo.action.ListUsersAction"
    scope="REQUEST">
    <CONFIG>
    </CONFIG>
  </ACTION>
</ACTIONS>
```

modules.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MODULES>
  <MODULE
    name="AutomaticListUsers"
class="it.eng.spago.dispatching.module.impl.DefaultListModule"/>
</MODULES>
```

Using the unique Configurator object is possible to access to both set of information




```
ConfigSingleton config = ConfigSingleton.getInstance();
        String actionClassName = (String) config.getFilteredSourceBeanAttribute("ACTIONS.ACTION","NAME", "
LIST_USERS_ACTION");
        String moduleClassName = (String) config.getFilteredSourceBeanAttribute ("MODULES.MODULE", "NAME","
AutomaticListUsers");
```

In case of the application needs a new configuration file it is necessary to registry it into the file *master.xml* in order to access its information with the *ConfigSingleton* object. The name of the file *master.xml* is a default name configured into the file *web.xml* with the parameter AF_CONFIG_FILE , which defines the path of the master file, relative to the application web root.
The path of the application web root can be set up using the AF_ROOT_PATH parameter of the web.xml file, but, normally, this parameter is empty and the system takes the context root.

3.2.1Configuration reset

It's possible to force the re-reading of the configuration calling the static method *ConfigSingleton.**release**()*. The configuration will be read at the first access to any configuration information.

3.2.2Configuration organization

In complex applications you can split the configuration files. In this way the file *actions.xml*, for example, becomes the folder *actions* that contains the files *actions1.xml* and *actions2.xml*.
With this feature you can organize the configuration files according the logical areas of your application.

Here you find an example regarding the file *actions.xml*. The file *master.xml* is untouched:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MASTER>
    <CONFIGURATOR path="/WEB-INF/conf/spago/data_access.xml" />
    <CONFIGURATOR path="/WEB-INF/conf/spago/actions.xml" />
    <CONFIGURATOR path="/WEB-INF/conf/spago/authorizations.xml" />
    .....
</MASTER>
```

The content of file *actions.xml* differs from usual syntax: in this case it has the same syntax of *master. xml*, as you can see in this example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MASTER>
```

```
        <CONFIGURATOR path="/WEB-INF/conf/spago/actions/actions1.xml" />
        <CONFIGURATOR path="/WEB-INF/conf/spago/actions/actions2.xml" />
    </MASTER>
```

In the folder ***actions*** there are the files *actions1.xml* and *actions2.xml* with the classic syntax.

Here you find a sample of *actions1.xml*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
        <ACTION name="TEST1" class="it.eng.TestAction" scope="REQUEST">
                <CONFIG param="Sample configuration"></CONFIG>
        </ACTION>
</ACTIONS>
```

3.3Tracing

The framework provides a logging service that can be configured into the file *tracing.xml,* which has the following structure:

```
<TRACING>
        <LOGGER name="DEFAULT_LOGGER" class="it.eng.spago.tracing.DefaultLogger">
                <CONFIG trace_min_log_severity="0" debug="true" trace_path="/demo/log/" trace_name="demo-" append="true"
                trace_thread_name="false" />
         </LOGGER>
        < LOGGER name="Framework" class="it.eng.spago.tracing.DefaultLogger">
                <CONFIG trace_min_log_severity="4" debug="false" trace_path="/demo/log/" trace_name="framework-"/>
        </LOGGER>
        <LOGGER name="Application" class="it.eng.spago.tracing.Log4JLogger">
                <CONFIG trace_min_log_severity="0" debug="true"/>
        </LOGGER
  </TRACING>
```

Each <LOGGER> envelope define a “logger” which is a class that implements the IFaceLogger interface and exposes the necessary logging services. The attributes of the LOGGER tag are:

- name: logical name of the logger. Developers can specify different logger for different components of the application in order to distinguish the trace activity of each component. It is possible to registry a default logger DEFAULT_LOGGER for trace messages of all components for which isn’t defined a logger.
- class: complete name of the class that implements the logger.

The CONFIG envelope defines the configuration of each logger and specifies the following attributes:

- trace_min_log_severity: minimun severity of tracing (values comprise from 0 to 4, extremes included).
- debug: flag to activate the DEBUG level. If set to “true” all the debug information are traced.
- trace_path: define the path of the log files.
- trace_name: define the initial part of the log file (Some logger can add information to this name, like the date for example) in order to distinguish them from others application log files.
- trace_thread_name (only for the DefaultLogger provided from Spago): flag to activate the tracing of the threads name. If true the thread name that executes the log instruction is traced with the message.

Developers, in order to trace messages, can use the class TracerSingleton. This object, using the design pattern Factory, creates an instance for each logger class defined into the configuration file and exposes a static log method. The method takes in input the logic name of the logger so it can use the appropriate instance of logger class in order to tracing the messages. Currently the framework provides two different loggers:

- DefaultLogger: traces messages on a log file.
- Log4JLogger: usign the libraries of Log4J it is able to trace messages on all the Appender defined into the log4j.properties (see log4J documentation for major details).

Developers can define their own logger classes.

It is important to notice that through the definition of the couple *name / class* into the configuration file, it is possible to differentiate the tracing modality and to separate the log messages for each application or component that interact with the framework. In order to obtain this behaviour developers must only define an association name / logger for each component, that has to be managed separately, and use the name as a parameter for the log method of the TracerSingleton object.

3.3.1DefaultLogger

The framework provides a default logger for the logging and tracing of information, warnings and errors. The particularity of this system is that to midnight it performs the switch on a different log file, in order to permits the adoption of some backup policies and to compress the log files.

An example of use can be:

TracerSingleton.log("application name", TracerSingleton.DEBUG, "message");

Obviously the "application name" must be associated (into the configuration file) to the class implementing the default logger.
Every call to the tracer needs to specify the type of message to logging, the different types are:

- INFORMATION: information messages, not errors.
- WARNING: warnings
- MINOR: errors that have little importance
- MAJOR: grave errors not blocking
- CRITICAL: blocking errors.
- DEBUG: debug information

The message will be traced only if the error level of the instruction is major respect to the minimum tracing level defined into the configuration file. For example, if the minimum level is MINOR the tracer will log only the messages with error levels MAJOR and CRITICAL. A special case is DEBUG which can be able and disable also with the flag "debug" of the configuration file.
This tracer is able to trace message strings and XML streams

3.3.2Log4JLogger

This class performs the logging and tracing activities using the Log4J libraries. The class is only a wrapper over the Log4J system and allows only to configure the minimum level of tracing and the logging of debug information, defining the relative parameters into the configuration file, as for the DefaultLogger class. All the operations and other configurations are delegated to Log4J system, so developers, in order to use it, must configure the Log4J property file *log4j.properties* (see Log4J user guide for details).

3.4Errors Handler

Spago maintains a stack of applicative and non-applicative errors. Every error contains a severity indicator. Two different type of errors are managed:

- *it.eng.spago.error.**EMFInternalError***: these errors are produced by components external to the current development context: for instance, a JDBC SQLException. When an external error occurred the developers can generate a new *EMFInternalError* associating it a severity and, eventually, an error message and the native exception. It is also possible to associate to the error a generic object that represent an *additionalInfo* property, which can be used to set additional and context specific data that can be useful, during the presentation phase, for example. The unique constrain of the object is that it must be serializable because it has to be passed between the framework subsystems as an xml stream. Typically these errors are not published on the user interface
- *it.eng.spago.error.**EMFUserError***: related to business logic and referring to a code segment; a pre-defined description is related to this code at runtime. The code correspond to a multilanguage message defined into the messages files (The mechanism of multilanguage application messages is explained into the section Language Management. So, in order to build a new EMFUserError developers can use the constructor:

  *public EMFUserError(String severity, int **code**)*

  The error is associated with a message contained into a properties file, so the developers can easily change the message error of the application, simply modifying the properties file. Each message string can contains some placeholders which can be replaced by the string value of some parameters. This functionality allows to reuse the same message in different context or to enrich the error message with specific details. In order to use this mechanism Spago provides this construct for the error class:

  *public EMFUserError(String severity, int code, Vector **params**)*

  In this constructor *params* is a vector of parameters containing the values to substitute to the message placeholders. Into the message the placeholders must be defined with this syntax: %1,%2, ….

Once generated the error, developers can store it into the errors stack *it.eng.spago.error.**EMFErrorHandler***, which can be access, into some specific framework contexts, using the method *getErrorHandler()*. The stack contains all the application errors registered and it allows to retrieve them, in a second time, in order to using them for various scope, like the presentation to the user for example.

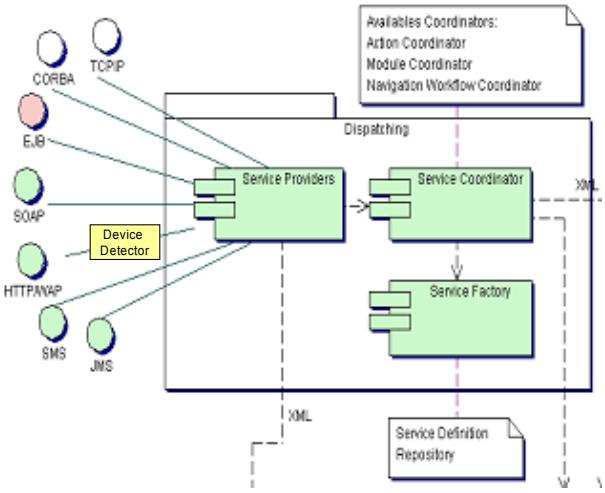
As an example of presentation errors suppose the application has a JSP page (which shows the results of a service) and that this JSP inherited from *it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage* class. In order to retrieve the stack of errors it is possible to simply write:

```
<%@ page extends="it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage"%>
<% EMFErrorHandler errorHandler = getErrorHandler(request);
    Collection errors = errorHandler.getErrors();
   // do something %>
```

The presentation of the errors can be entirely implemented from the developers but the framework provides a tag in order to carry out this task . This tag is *<spago:error/>* and is explained into the section Presentation.

3.5Dispatching

The framework provides one adapter for each channel, because the information and their formats are different for each channel. The adapters are objects that listen the service requests and translate them into a common internal format (XML) before starting the service.



Different adapters can be created in different Virtual Machine:

- The adapters HTTP/WAP, SOAP live into the Web Container: they are servlets (which can be found in *spago-web.jar*)
- The adapter EJB lives in the EJB Container (can be found in *spago-ejb.jar*) and is composed from:
  - *it.eng.spago.dispatching.ejbchannel.**AdapterEJBBean***: is a statefull session bean with a remote interface CMP (container managed persistence) BMT (bean managed transaction).
  - *it.eng.spago.dispatching.ejbchannel.**SessionFacadeEJBBean***: is a stateless session bean with a local interface. Because it is a stateless bean the EJB container can manage it with some pooling mechanisms for the load balancing, and its state isn't serialized on a database between different requests.
  - *it.eng.spago.dispatching.ejbchannel.**RemoteSessionFacadeEJBBean***: is a stateless session bean with a remote interface.
- The JMS adapter lives into the EJB Container: it is a queue subscriber of type Message Driven Bean, implemented by the class *it.eng.spago.dispatching.jmschannel.**JmsAdapterMDB*** (*spago-ejb.jar*).

The framework provides two mode for dispatching the business logic:

- Actions
- Modules

3.5.1Action

Actions are business objects which totally carry-out a request of an applicative service. In other words, one service corresponds to one business object (but the same object can carry out different requests), which must implement the *ActionIFace* interface (Starting from this point the document will refer to this object simply using “action”). This mechanism is the same offered from the popular open source framework struts

3.5.1.1Action definition

The actions are defined into a configuration file called *actions.xml.* An example of this file can be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
        <ACTION
            name="LIST_USERS_ACTION"
            class="it.eng.spago.demo.action.ListUsersAction"
            scope="REQUEST">
            <CONFIG>
            </CONFIG>
        </ACTION>
</ACTIONS>
```

Each action has the following attributes:

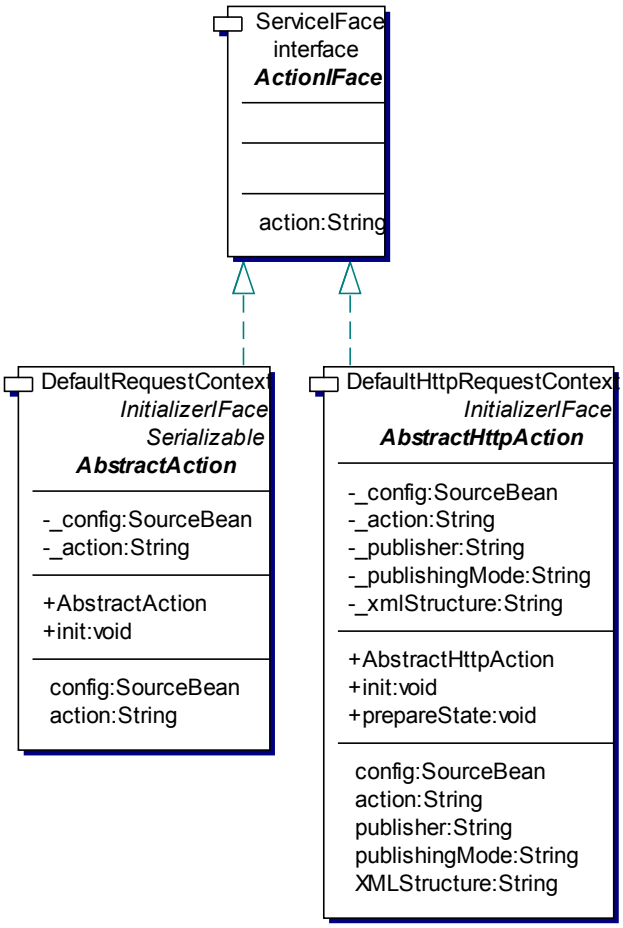
- Logical name: action’s name to identify it. The service request specify the logical name of the action in order to invoke it for produce the response.
- Class: name of the class implementing the action.
- Scope: the life context of the object. The possible values are:
  - REQUEST: a new action is activated for every new request
  - SESSION: the same action carries out the service corresponding to all the requests of the same conversation (session).
  - APPLICATION: the same action carries out the service corresponding to all the requests sent to the same container (JVM). Notice that a business object with this scope is not a real singleton, because there is a different instance of it for every JVM (at instance, for cluster nodes).

The advised modality is REQUEST: the only contraindication for this modality is when into the constructor of the action are performed heavy activities. In this case, in fact, each new creation of the class involve the code execution with a consequent degradation of the performances.

3.5.1.2Implementation

Developers can implement an action in two way:

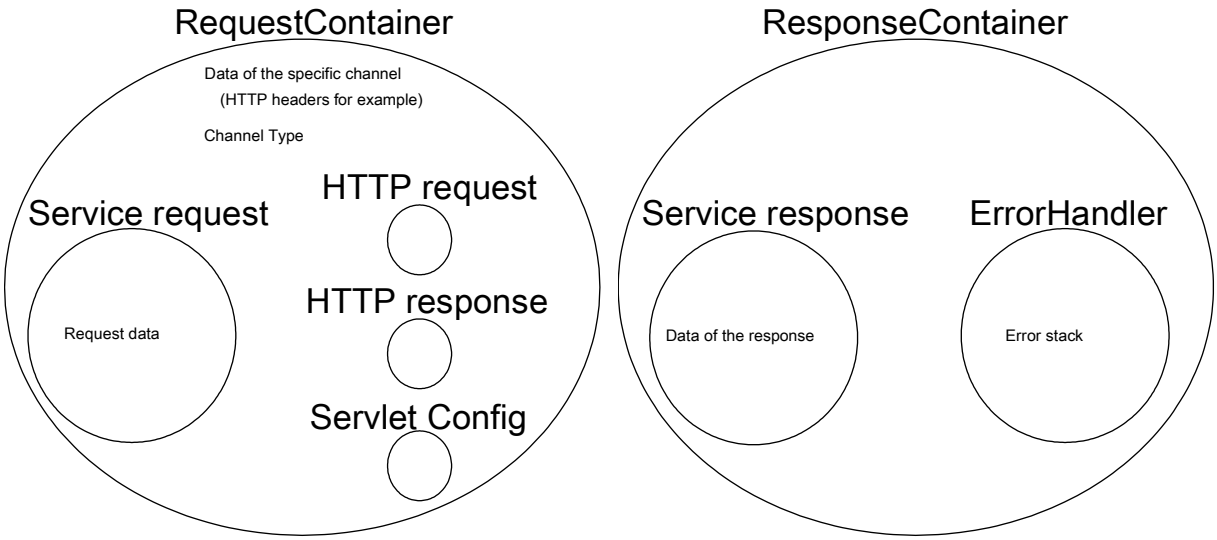
- Create a class that implements the *ActionIFace* interface.
- Create a class that extends the class *AbstractAction* or *AbstractHTTPAction*. (This is the simpler way)



In both cases the main method to implement is *service*:

*void **service**(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception;*

This method has two input parameters: the request and the response (sourcebean objects). This mechanism allows, independently from the channel, to access to the request parameters and to valorized the response with an xml stream, which can be used from the presentation part (if present). In this way, the business logic is independent from the channel. The service method can access also the specific channel parameters (the HTTP header for example) using an object *RequestContainer.*



The object RequestContainer, accessible from the service method, is a container for all the information about the request. It contains the object *ServiceRequest* (the real request data, passed to the method service) and other information relative to the channel (For example, in case of HTTP channel, it contains the java HTTP request / response and the servlet config). This object allows also to recover an object *SessionContainer* which represents the current session, so supposing that in the session context is stored the identifier of the current user, developers can retrieve it using the following instructions:

```
public void service(SourceBean request, SourceBean response) {
        RequestContainer requestContainer = getRequestContainer();
        SessionContainer sessionContainer = requestContainer.getSessionContainer();
        String userId = (String)sessionContainer.getAttribute("userId");
        .....
}
```

The object *RequestContainer* is a container single-value (sourcebean is a container multi-value). The external envelope contains the channel type, the specific channel information (the HTTP headers for example), the current user attributes,

and, only for the HTTP channel, it contains the HTTP request, the HTTP response and the ServletConfig. The most inner container is the *Service request* which is the object passed to the method service containing request's parameters.

The object *ResponseContainer* is a container single-value, that maintains the response objects produced by services and the error stack. Because the request container and the response container are container single-value, every time an attribute is set it will override the previous version of the same attribute. This behaviour is different from that one of the SourceBean object which is a multi-value object (every time an attribute is set it is created independently if it already exists, so the same attribute can compare more times).

3.5.1.3Initialization

During the action initialization can be retrieved some configuration information defined into the *CONFIG* envelope of the action xml file. This is an example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
    <ACTION name="LIST_USERS_ACTION" class="com.engiweb.demo.action.ListUsersAction" scope="REQUEST">
        <CONFIG pool="afdemo" title="List Users" rows="2">
            <QUERIES>
                <SELECT_QUERY statement="LIST_USERS"/>
                <DELETE_QUERY statement="DELETE_USER"/>
            </QUERIES>
        </CONFIG>
    </ACTION>
</ACTIONS>
```

These information are passed to the *init* method, as a SourceBean object, every time the action is created. Developers can use these information for different scope inside the method.

```
public void init(SourceBean config) {
    // ...
    // do something
    // ...
} // public void init(SourceBean config)
```

3.5.1.4Specificity of the channel

The framework incapsulates data, coming from different channels, providing the request and the response as a SourceBean object. In some cases, however, it is necessary to access to the channel native objects using some specific objects, also if the application becomes not portable to other channels.

An example is an action that has to upload a file; in this case the action has to access to the original java HTTP request to extract the information about the multipart form.

The framework provides the possibility to access to HTTP specific channel information extending the class *AbstractHTTPAction* instead of the class *AbstractAction.* This class allows to recover the original java request and response objects. Using this class, developers must pay attention to not make conflict with the Spago presentation

mechanism. To avoid the conflict, the developers must notify to the framework that the response will be produced by the action, simply calling the method *freezeHttpResponse()*.

The framework doesn't provide a similar mechanism to access specific information of different channels from HTTP, so, for example, it is not possible to get data from the EJB context of the EJB container. However, the only case encountered until today is the need to upload a file in an HTTP channel.

3.5.2Modules

Every module is a business object; it can cooperate with other modules to carry out a service request. All modules cooperating for the same service comprise a logical unit called a **page**. The service response is the union of all modules responses. An easy business logic execution workflow describes the order and conditions of execution for all the modules of one page. At the end of execution of each module, the framework identifies the next modules to execute according to request parameters.

3.5.2.1Modules definition

Modules are defined in the *modules.xml* configuration file. See an example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MODULES>
  <MODULE
    name="ModuleClient"
    class="it.eng.spago.demo.module.ModuleClient"/>

  <MODULE
    name="ModuleSupplier"
    class="it.eng.spago.demo.module.ModulsSupplier"/>
</MODULES>
```

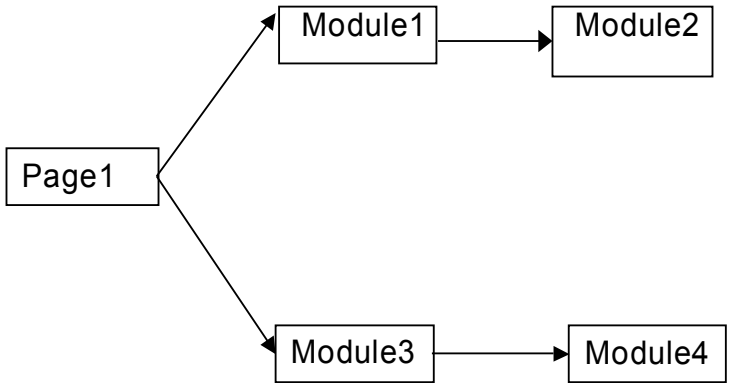
Each module has the following attributes:

- Logic name: module's name for identify it.
- Class: name of the java class implementing the module.

It is not possible to specify the module scope, in fact, this is inherited from the page containing it.

3.5.2.2Page

A page is a logic modules composition; it is a graph that defines the modules and their invocation sequence. An example of a page graph can be:

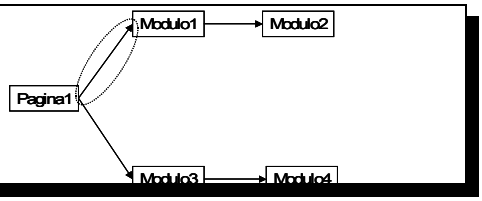


The image shows a simple graph: the starting point is the page. The graph evaluation is recursive and the navigation goes in depth; the modules are invoked in this order: Module1, Module2, Module3, Module4. The response is the concatenation of the four modules responses.

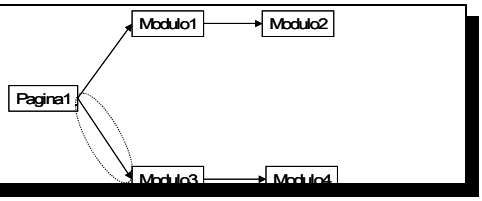
3.5.2.3Pages definition

The page structure is defined in the *pages.xml* configuration file. The configuration file, for the example page defined in the previous section, is:

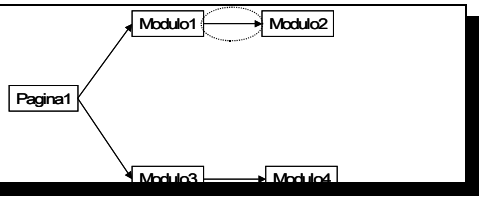
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PAGES>
  <PAGE name="Pagina1" scope="SESSION">
    <MODULES>
      <MODULE name="Module1"/>
      <MODULE name="Module2"/>
      <MODULE name="Module3"/>
      <MODULE name="Module4"/>
    </MODULES>
    <DEPENDENCIES>
      <DEPENDENCE source="Page1" target="Module1">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```



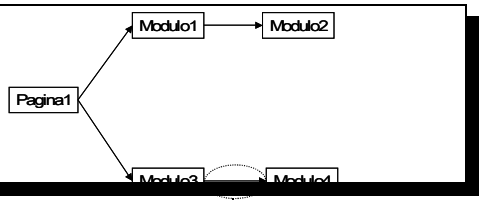

```
      <DEPENDENCE source="Page1" target="Module3">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```




```
      <DEPENDENCE source="Module1" target="Module2">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```

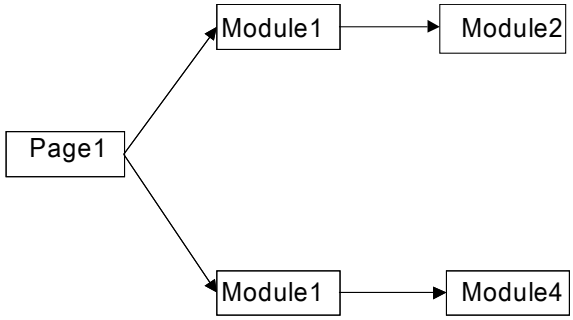



```
      <DEPENDENCE source="Module3" target="Module4">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```




```
    </DEPENDENCIES>
  </PAGE>
</PAGES>
```

Each arc is a *DEPENDENCE* envelope that specifies the start node (*source* attribute) and the final node (*target* node). A module can be into different branches of the graph, so, the following graph, where the Module1 is present in two branches is allowed:



In these cases, if the same module is executed more time, only the last module response is stored into the page response.

Normally the module extends the page scope but this behaviour can be bypassed adding the *keep_instance* parameter.

```
<MODULES>
   <MODULE name="Module1" keep_instance="true"/>
   <MODULE name="Module2"/>
   <MODULE name="Module3"/>
   <MODULE name="Module4"/>
</MODULES>
```

If the parameter exist or if the value is *false*, the module extends the scope of page that contains it. If the parameter value is *true*, the module will have a session scope, independently from the page scope.

Another parameter that could be defined is *keep_response:* if valorized to *true* allows to re-obtain the module response produced during the first invocation of the page, in case of the module won't be activated during the next page invocations. The default value is *false*. The page scope has to be *SESSION*.

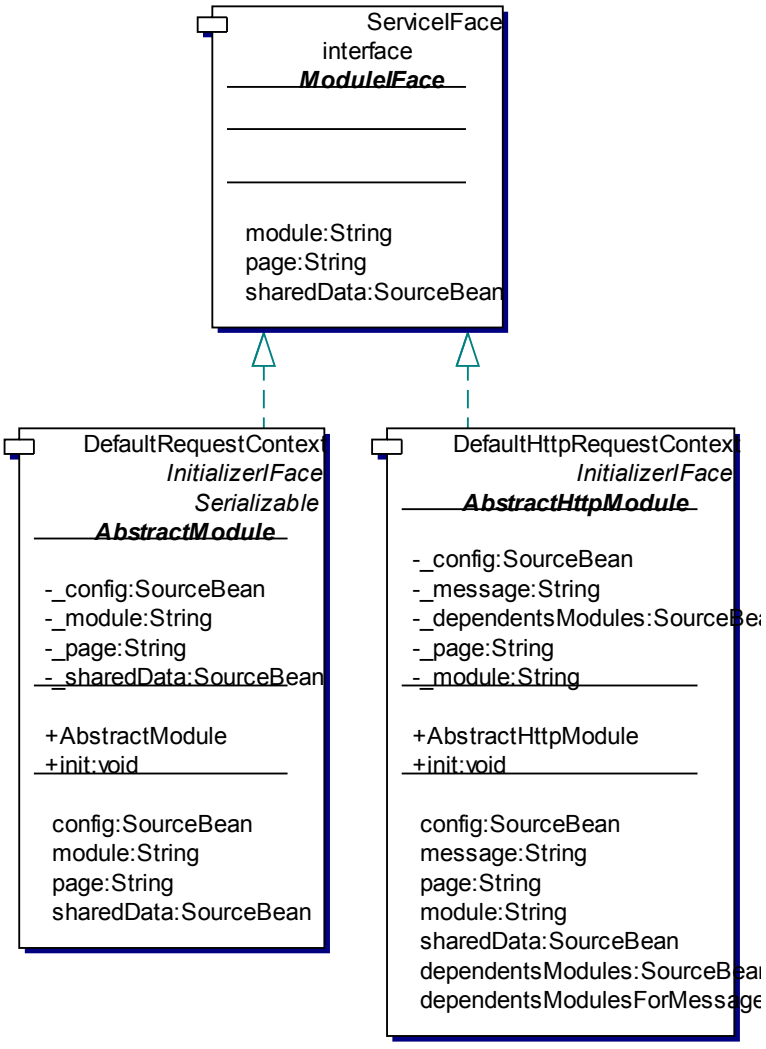
```
<MODULES>
   <MODULE name="Module1" keep_response="true"/>
   <MODULE name="Module2"/>
</MODULES>
```

If a module is used in more pages, developers can use the method *getPage()* to retrieve the logical name of the page that is currently using it and then differentiate the business logic. Furthermore, the same class can be used to implement different logical modules, so developers have also the possibility to know which logical page is currently executing the class.

3.5.2.4Implementation

Developers can implement a module in two way:

- Create a class that implements the *ModuleIFace* interface.
- Create a class that extends the class *AbstractModule* or *AbstractHttpModule*. (this is the simpler way).



In both cases the *service* method must be implemented:

void ***service****(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception;*

Also in this case, there's the possibility to access directly to the HTTP channel information, extending the *AbstractHTTPModule* module instead of the *AbstractModule* module.

3.5.2.5Conditions

The rules to verify that a transition from one module to another exists are called **conditions**.
In the example above described, all the modules are invoked independently from request parameters or from the results of previous modules.
This is the default behaviour and to obtain it , the developers must only leave empty the *CONDITIONS* envelope of the graph branch relative to the module.

```
<DEPENDENCE source="Page1" target="Module1">
    <CONDITIONS>
    </CONDITIONS>
    <CONSEQUENCES/>
</DEPENDENCE>
```

However, the modules invocation must be performed only if some conditions are verified (the presence of some parameters to the request, the presence of some parameters with some values and so on). All these conditions can be defined into the *CONDITIONS* envelope. If all conditions are true, the module is executed. The syntax is:

```
<CONDITIONS>
  <PARAMETER
    name="nome parametro"
    scope=" USER | ADAPTER_REQUEST | SERVICE_REQUEST | SESSION  | APPLICATION | ADAPTER_RESPONSE
|
                SERVICE_RESPONSE | ERROR"
    value=" AF_DEFINED  | AF_NOT_DEFINED | ..."/>
</CONDITIONS>
```

The *name* attribute contains the name of the parameter to analyse; the *scope* attribute is the container where it can be found. The possible scopes are:

- USER: allows to verify the user properties, like the user identifier, for example, recovering them from the authentication module (which will be explained later).
- ADAPTER_REQUEST: allows to verify the parameters contained into the RequestContainer, like the http headers for example.
- SERVICE_REQUEST: allows to verify the parameters contained into the service request.
- SESSION: allows to verify the parameters contained into the session container.
- APPLICATION: allows to verify the parameters contained into the application container.

- ADAPTER_RESPONSE : allows to verify the parameters contained into the ResponseContainer. This scope is reserved for future release because at this moment the framework don't store anything into the Response Container.
- SERVICE_RESPONSE : allows to verify the parameters contained into the service response, valorised from the previous modules .
- ERROR : allows to verify the presence of Internal or User error into the Error Handler.

The value attribute indicates if the parameter must be present / or not or the value that it must contain:

- AF_DEFINED: the parameter must be present into the specified scope.
- AF_NOT_DEFINED: the parameter must not be present into the specified scope.
- "parameter value": value (not allowed in case of scope ="ERROR") of parameter

An example of condition could be the presence, into the request, of the parameter *module* and the parameter *user* valorised with value "Mario Rossi". The conditions must be specified like this:

```
<CONDITIONS>
 <PARAMETER name="module" scope="SERVICE_REQUEST" value="AF _DEFINED"/>
 <PARAMETER name="user" scope="SERVICE_REQUEST" value="Mario Rossi"/>
</CONDITIONS>
```

Otherwise, for verify the presence into the response of the *sample* parameter (which has to be inserted from a previous module), the condition is:

```
<PARAMETER name="Modulo1.sample" scope="SERVICE_RESPONSE" value="AF _DEFINED"/>
```

A specific case of conditions is the search of some errors into the error stack; in fact, the attribute *scope* must contains the "ERROR" constant and the attribute *name* must contains one of the following values:

- AF_INFORMATION : indicates all errors with "INFORMATION" severity level.
- AF_WARNING : indicates all errors with "WARNING" severity level.
- AF_ERROR : Indicates all errors with "ERROR" severity level.
- AF_BLOCKING : indicates all errors with "BLOCKING" severity level.
- "error code" : indicates all errors of type EMFUserError with the code specified.
- "empty value": if the name attribute is empty the system considers all *EMFUserError* or *EMFInternalError* errors, with whichever severity. In practise the system controls if there are errors into the stack.

For the ERROR scope the following conditions are valid:

```
<PARAMETER name="" scope="ERROR" value="AF_DEFINED" />
   true if the error stack contains at least one error

<PARAMETER name="" scope="ERROR" value="AF_NOT_DEFINED" />
   true if the error stack doesn't contain errors

<PARAMETER name="<severity>" scope="ERROR" value="AF_DEFINED" />
   true if the error stack contains at least one error with severity level equals to  <severity>
  which is a value chosen between  AF_INFORMATION,        AF_WARNING, AF_ERROR, AF_BLOCKING
```

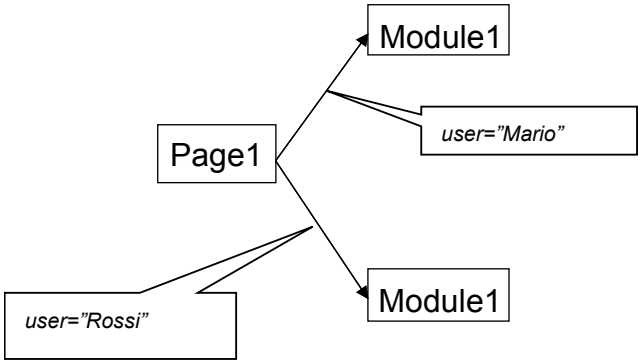
<PARAMETER name="<severity>" scope="ERROR" value="AF_NOT_DEFINED" />
true if the error stack doesn't contain error with severity level equals to <severity>

<PARAMETER name="<code>" scope="ERROR" value="AF_DEFINED" />
true if the error stack contains at least one error of type "EMFUserError" with code equals to <code>

<PARAMETER name="<code>" scope="ERROR" value="AF_NOT_DEFINED" />
true if the error stack doesn't contain errors of type "EMFUserError" with code equals to <code>

3.5.2.6OR Conditions

The dependencies value the conditions using AND logical operator so all the conditions must be verified for continue the execution. To define OR conditions it is necessary to insert new braches into the graph, one for each combination of the OR condition. For example, if the condition is that the *user* parameter must contain the value *"Mario"* or the value *"Rossi"* , it is necessary to define two branches, one for each part of the OR expression.



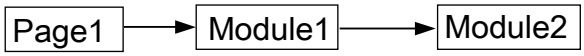
The xml configuration of the page:

```
<PAGE name="Page" scope="SESSION">
  <MODULES>
    <MODULE name="Module1"/>
  </MODULES>
  <DEPENDENCIES>
    <DEPENDENCE source="Page" target="Module1">
                                                    <CONDITIONS>
          <PARAMETER name="user" scope="SERVICE_REQUEST" value="ciccio"/>
      </CONDITIONS>
      <CONSEQUENCES/>
    </DEPENDENCE>
```

```
        <DEPENDENCE source="Page" target="Module1">
                                                                    <CONDITIONS>
            <PARAMETER name="user" scope="SERVICE_REQUEST" value="pelliccio"/>
          </CONDITIONS>
          <CONSEQUENCES/>
        </DEPENDENCE>
    </DEPENDENCIES>
</PAGE>
```

3.5.2.7Module Response

The result produced from a module is inserted into an xml envelope which has the same logical name of the module. All the xml envelopes produced are inserted into the response of the page service, so, for example, if the system executes a service page configured like this



and each module set into its response one attribute as below:

```
void service(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception {
        .....
        serviceResponse.setAttribute("test", " test string");
        ....
}
```

the final response produced will be:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE RESPONSE (View Source for full doctype...)>
<RESPONSE>
        <SERVICE_RESPONSE>
            <MODULE1 test=" test string"/>

            <MODULE2 test=" test string"/>

        </SERVICE_RESPONSE>
<ERRORS/>
</RESPONSE>
```

3.5.2.8Consequences

Once defined the branches of the graph and the conditions associated to each branch, it is possible to specify some additional parameters to those already present into the request, which have to be passed to the target module, if the transaction is activated. It is possible to define new parameters with fixed values or with values recovered from different scope. This activity can be performed configuring opportunely the *CONSEQUENCES* envelope into the *DEPENDENCE* xml node

```
< CONSEQUENCES >
 <PARAMETER
    name="parameter name"
```

```
        type="ABSOLUTE | RELATIVE"
        scope=" USER | ADAPTER_REQUEST | SERVICE_REQUEST | SESSION | APPLICATION | ADAPTER_RESPONSE
|
                    SERVICE_RESPONSE"
        value="parameter value"/>
    </ CONSEQUENCES >
```

The *name* attribute define the name of the parameter, the *value* attribute contains its value while the scope attribute specify the container from which retrieve the parameter value (the possible values for this attribute are explained into the conditions section). The *type* attribute can assume this values:

- ABSOLUTE: indicates that the parameter has the value of the attribute *value* (Only in this case the *scope* parameter can be omitted). For example, in order to pass the *user* parameter with a fixed value *"Mario Rossi"* the consequence must be defined like this:

```
< CONSEQUENCES >
  <PARAMETER name="user" type="ABSOLUTE" value="Mario Rossi"/>
</ CONSEQUENCES >
```

- RELATIVE: indicates that the parameter has the value of the variable which has the name defined into the name attribute and it is contained into the container identified by the scope attribute. For example, in order to pass the *user* parameter with the value equals to the session parameter *userSession,* the consequence must be defined like this:

```
< CONSEQUENCES >
  <PARAMETER name="user" type="RELATIVE" scope="SESSION" value="userSession"/>
</ CONSEQUENCES >
```

All parameters defined into the consequences are accessible from the module using the request SourceBean, so to access a parameter defined like this:

```
< CONSEQUENCES >
  <PARAMETER name="user" type="ABSOLUTE" value="Mario Rossi"/>
</ CONSEQUENCES >
```

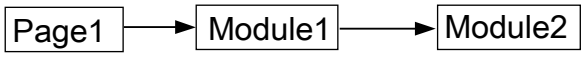
the developers must use this instruction inside the module:

```
void service(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception {
        .....
        String param = serviceRequest.getAttribute("user");
        ....
}
```

Into the request are present all the parameters defined by the invocation URL and all the parameters defined by the consequences.

3.5.2.9Modules cooperation

Modules are components that can cooperate between them exchanging some information. The main way to exchange data between modules is the use of consequences. For example, supposing that the system must execute a page configured like this:



Supposing also that the *Module2* needs to access to the information inserted into the response from the *Module1.* It is necessary to introduce a consequences tag into dependencies from *Module1* to *Modulo2* like this:

```
<CONSEQUENCES>
 <PARAMETER name="paramModule1" type="RELATIVE" scope="SERVICE_RESPONSE" value="Module1.example"/>
</CONSEQUENCES>
```

In this way if the *Module1* inserted into the response a parameter called *example* with these instructions:

```
void service(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception {
        .....
         serviceResponse.setAttribute("example", "example string");
        ....
}
```

Module2 can access to this parameter in the following way:

```
void service(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception {
        .....
        String param = serviceRequest.getAttribute("parameterModule1");
        ....
}
```

If the consequences mechanism is not sufficient for the module cooperation, it is possible to use a memory area, called *shared data,* shared from all modules.

```
public SourceBean getSharedData();
```

The shared memory is a SourceBean object that developers can access with the method *getSharedData()* and it is recreated every new service request. This is a powerful mechanism but increases the ties between modules, so when possible it is better to use the consequences for exchanging modules data.

3.5.2.10Graphs

It is possible to define, into the *graphs.xml* configuration file, some graphs template associated with the logical unit *Pages.* The association between graphs and pages is specified into the *pages.xml* file. The following xml is an example of graph configuration:

```
<GRAPH id="1" >
        <MODULES>
                <MODULE id="1" keep_instance="TRUE" keep_response="FALSE" />
                <MODULE id="2" keep_instance="TRUE" keep_response="FALSE" />
        </MODULES>
        <DEPENDENCIES>
                <DEPENDENCE id="1" source="" target="{1}">
                        <CONDITIONS>
                                <PARAMETER name="module" scope="SERVICE_REQUEST"
                                value="AF_NOT_DEFINED" />
```

```
                    </CONDITIONS>
                    <CONSEQUENCES />
                </DEPENDENCE>
                <DEPENDENCE id="2" source="" target="{1}">
                    <CONDITIONS>
                        <PARAMETER name="module" scope="SERVICE_REQUEST" value="{1}" />
                    </CONDITIONS>
                    <CONSEQUENCES />
                </DEPENDENCE>
                <DEPENDENCE id="3" source="" target="{2}">
                    <CONDITIONS>
                        <PARAMETER name="module" scope="SERVICE_REQUEST" value="{2}" />
                    </CONDITIONS>
                    <CONSEQUENCES />
                </DEPENDENCE>
            </DEPENDENCIES>
        </GRAPH>
```

Each graph has an unique identifier and defines a set of dependencies between some abstract modules which must be replaced into the page configuration from real modules. This mechanism allows reusing the same graph logic with different pages that can associate different modules. In this example the graph has the identifier “1” and it is composed by two modules and three dependencies. Note that each module has an id attribute, which is the logical name, and that the *source* and *target* attributes references modules using this logical name.

A page configuration that uses this graph becomes:

```
        <PAGE name="BasicPaginaUtenti" class="it.eng.spago.dispatching.module.ConfigurablePage" scope="SESSION">
        <GRAPH id="1">
            <MODULES>
                <MODULE id="1" name="BasicListUsers" />
                <MODULE id="2" name="DetailUser" />
            </MODULES>
            <DEPENDENCIES>
                <DEPENDENCE id="2" source="{2}" target="{1}">
                    <CONDITIONS>
                        <PARAMETER name="{2}.result" scope="SERVICE_RESPONSE" value="val2" />
                    </CONDITIONS>
                    <CONSEQUENCES />
                </DEPENDENCE>
            </DEPENDENCIES>
        </GRAPH>
        </PAGE>
```

The configuration defines that the page uses the graph with id equals to “1” (the graph of this example) and associates the modules classes to the modules logical names. Logical module “1” is associated to the module "BasicListUsers" and

logical module “2” is associated to the module “DetailUser”. Into the page configuration it is also possible to redefine the dependencies, in this case it has been redefined the dependency “2”. This mechanism recall the object “overriding” service.

3.5.2.11Exceptions management

If the execution of the module *service* method raises an exception; the default behavior is to interrupt page evaluation so the next modules are not executed. Developers can change this behavior simply catching all the possible exceptions inside the service method and eventually transform them into *EMFUserError* or *EMFInternalError,* which can be added to the *ErrorHandler* (errors stack). In this way, the page evaluation ends and the errors are stored to be showed in the presentation phase. The system can show to the user the result obtained and the errors occurred.

3.5.2.12Consideration

The modules dispatching is more complex than the actions dispatching, but it is also more flexible and powerful. A best practice is to use the modules dispatching when it is possible to identify some components that can be reused by more services; in this way, once write the necessary components the developers have only to compose them into a page graphs.

3.6Data validation

The validation of data introduced by the users is a common problem for a web application. To facilitate this operation, Spago implements a service that allows to partially automating the validation process. The use of this component is optional and configurable, it is possible to have some pages using the validation and other that don't use it. The validation can be blocking or not blocking, so, in the first case the execution is stopped, otherwise the execution continues. The component executes a validation server side; at the moment if developers wont apply the validation client side, reducing the communication with the server, they must write ad hoc javascript functions.

The component provides two types of validation:

1. Automatic validation of the common data types.
2. Validation performed from some java class written for this scope.

3.6.1Configuration

In Spago to configure the validation two configuration files must be created:

- The *field-validators.xml* file to configure Basic Field Validator.
- The *validation.xml* file to configure the validation rules for Spago services.

3.6.1.1Field Validator's Configuration

A Field Validator is a Java class with the following responsibility:

- To make syntax validation for service request fields.
- Optionally make a type conversion of the value if the validation succeeds.
- To save in a child of the original service request (TYPED_SERVICE_REQUEST) the value (original or type converted)with an alias.

The Field Validators are bound to fields by the value of TYPE attribute in FIELD section of the validation.xml (see next paragraph).

To define a new Field Validator is necessary that a Java Class inherit from the *it.eng.spago.validation.fieldvalidators.**AbstractFieldValidator,*** and associate this class with a type in the *fieldvalidators.xml* configuration file.

By default Spago provide a set of standard Field Validator that are associated to the type that can be specified in the TYPE attribute of the validation.xml file.

The fieldvalidators.xml file for the standard Spago Field Validator looks like:

```
<FIELD-VALIDATORS>

        <FIELD-VALIDATOR fieldType="GENERIC"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.GenericFieldValidator"/>
```

```
        <FIELD-VALIDATOR fieldType="FISCALCODE"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.FiscalCodeValidator"/>

        <FIELD-VALIDATOR fieldType="EMAIL"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.EMailValidator"/>

        <FIELD-VALIDATOR fieldType="URL"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.UrlValidator"/>

        <FIELD-VALIDATOR fieldType="DATE"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.DateValidator">
                <CONFIG>
                        <DATE-FORMAT dateFormat="dd/MM/yyyy"/>
                </CONFIG>
        </FIELD-VALIDATOR>

        <FIELD-VALIDATOR fieldType="NUMERIC"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.NumericValidator">
                <CONFIG>
                        <LANGUAGE language="it"/>
                        <COUNTRY country="IT"/>
                </CONFIG>
        </FIELD-VALIDATOR>

        <FIELD-VALIDATOR fieldType="ALFANUMERIC"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.AlphaNumericValidator"/>

        <FIELD-VALIDATOR fieldType="LETTERSTRING"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.LetterStringValidator"/>

        <FIELD-VALIDATOR fieldType="REGEXP"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.RegExpValidator"/>

        <FIELD-VALIDATOR fieldType="RE"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.RegExpValidator"/>

        <FIELD-VALIDATOR fieldType="IMPORTO"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.DecimalValidator">
                <CONFIG>
                        <LANGUAGE language="it"/>
                        <COUNTRY country="IT"/>
                </CONFIG>
        </FIELD-VALIDATOR>

        <FIELD-VALIDATOR fieldType="DECIMAL"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.DecimalValidator">
                <CONFIG>
                        <LANGUAGE language="it"/>
                        <COUNTRY country="IT"/>
                </CONFIG>
        </FIELD-VALIDATOR>

        <FIELD-VALIDATOR fieldType="NAME"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.NameValidator"/>

</FIELD-VALIDATORS>
```

It can be noticed that each FIELD-VALIDATOR can have a *CONFIG* section and from Java code you can get by calling the method*: **public SourceBean FieldValidatorIFace.getConfig().***

The standard types and associated Field Validators defined by Spago are:

- **1 o GENERIC (it.eng.spago.validation.fieldvalidators.GenericFieldValidator)**
  This validator doesn't make syntax control and doesn't apply type conversion; eventually it will save the field with an alias. This validator can be applied for example to do only length control on fields.
- **2 o FISCALCODE: (it.eng.spago.validation.fieldvalidators.FiscalCodeValidator)**

This validator checks if a field is a valid Italian Fiscal Code or Partita IVA. It doesn't apply any type conversion. The field will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.

- **3 o EMAIL: email (it.eng.spago.validation.fieldvalidators.EMailValidator)**
  This validator checks if a field is a valid email address. It doesn't apply any type conversion. The field will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **4 o URL: Internet Address (it.eng.spago.validation.fieldvalidators.URLValidator)**
  This validator checks if a field is a valid URL. If validation succeed the field will be converted to a *java.net.URL* and then it will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **5 o DATE: date (it.eng.spago.validation.fieldvalidators.DateValidator)**
  This validator checks if a field is a valid Date according to the date format specified in *DATEFORMAT.dateFormat* on the CONFIG section. If the CONFIG section is not found the Italian format *dd/MM/yyyy* will be used. If validation succeed the field will be converted to a *java.util.Date* and then it will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **6 o NUMERIC: Numeric Field (it.eng.spago.validation.fieldvalidators.NumericValidator)**
  This validator checks if a field is a valid Number according to the locale for the country and language specified in
  COUNTRY.country and *LANGUAGE.language* attributes on the CONFIG section. If the CONFIG section is not found the Italian locale will be used . If validation succeed the field will be converted to a *java.lang.Double* and then it will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **7 o ALFANUMERIC: Alphanumeric String (it.eng.spago.validation.fieldvalidators.AlphaNumericValidator)**
  This validator checks if a field is composed of letters and numbers. It doesn't apply any type conversion. The field will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **8 o LETTERSTRING: LetterString (it.eng.spago.validation.fieldvalidators.LetterStringValidator)**
  This validator checks if a field is composed of letters. It doesn't apply any type conversion. The field will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **10 o REGEXP: regular-expression (it.eng.spago.validation.fieldvalidators.RegExpValidator)**
  This validator checks if a field match the regular expression specified in the "REGEXP" attribute on the FIELD section. . It doesn't apply any type conversion. The field will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.
- **11 o DECIMAL (it.eng.spago.validation.fieldvalidators.DecimalValidator)**
  This validator checks if a field is a valid Number according to the locale for the country and language specified in
  COUNTRY.country and *LANGUAGE.language* attributes on the CONFIG section. Then it is checked that the number of decimal digit is equal to the number specified on the DECIMALS attribute. If validation succeeds the field will be converted to a *java.lang.Double* and then it will be put in the TYPED_SERVICE_REQUEST, eventually with an alias.

It'is important to remark that the definition of FieldValidator as Java class and the association to field type by a configuration files make the Spago Validation Architecture powerful and flexible. With this architecture is possible:

- Define new Validators and types , implementing simple Java class
- Change the implementation of the Standard Spago Field Validators.

- Changing the validator associated to a field can be changed only editing a configuration file.

3.6.1.2Configuring Validation Rules for Spago Services

To activate the data validation of the page, developers must create a new configuration xml envelope into the file *validation.xml*. The new envelope defines the information to validate:

```
<VALIDATIONS>
        <SERVICE name="insertEmployee" type="ACTION">
                <VALIDATION blocking="true" validators="">
                        <CONDITIONS>
                                <PARAMETER name="age" scope="SERVICE_REQUEST" value="1" />
                        </CONDITIONS>
                        <FIELDS>
                                <FIELD name="name" maxLength="10" mandatory="true" type="8" "
                                        aliasAfterValidation="Firstname" />
                                <FIELD name="age" maxLength="5" mandatory="false" type="6" />
                                <FIELD name="department" maxLength="20" mandatory="false" type="7"
                                        aliasAfterValidation= "Department"/>
                        </FIELDS>
                </VALIDATION>
        </SERVICE>
</VALIDATIONS>
```

The SERVICE envelope has the following attributes:

- NAME: logical name of the action or page to which apply the validation.
- TYPE: can have two values: ACTION or PAGE, based on the service modality.

Inside the SERVICE envelope, the VALIDATION envelope has the following attributes:

- VALIDATORS: list of the logical name of the java validators to apply to the page. This param is optional. The java validators will be explained in the next paragraph.
- BLOCKING: flag for blocking the execution after a validation error. If true the execution is stopped, if false the execution continue.

The CONDITION envelope defines the conditions that must be satisfy for execute the validation, the FIELDS envelope defines all the fields to check.

Each FIELD envelope contains the following attributes:

- NAME: name of form field contained into the HTML page.
- TRIM: "true | false", if the value is set to true, Spago delete the starting and ending spaces of the field value.
- TOUPPERCASE: "true | false", if the value is set to true, Spago transforms the value of the field to the correspondent uppercase value.
- TYPE: this attribute is mandatory and it is used to match the Field Validator to apply to this field. The value of this attribute will be used to search a *FIELD-VALIDATOR* element in the *FIELD-VALIDATORS* envelope that has the attribute *fieldType* equals to the specified value.
- MANDATORY: "true | false if the value is set to true, the field is mandatory.
- MAXLENGTH: indicates the maximum chars number for the field value (the value must be an integer).

- REGEXP: regular expression.
- DECIMALS: number of decimals for an import type (the value must be an integer).
- ALIASAFTERVALIDATION: This is the name used to save the field in the (eventually after a type conversion) TYPED_SERVICE_REQUEST envelope.

If a field don't pass the validation, Spago inserts a *it.eng.spago.validation.**EMFValidationError*** object in the error handler and youn can use the *EMFValidationError.getFieldName()* method to get the name of the field that has produced the error.
The range code reserved, into the messages file, for these errors types is from 10100 to 10200.

3.6.2Java Validation

If the mechanism of automatic validation is not sufficient, because it is necessary to perform cross controls on various information, or because the control is complex, it is possible to associate to the pages some specific validation classes.
A class for a validation function, that we will be call "validator", is a class that extends *it.eng.spago.validation.**AbstractValidator*** and implements the following abstract methods:

- *public abstract boolean **check**(SourceBean request, EMFErrorHandler errorHandler):* Perform the formal controls on form data. Normally this method implements the controls for the syntax validity of the fields. If the method return *true,* the validation process continues with others validators.
- *public abstract boolean **validate**(SourceBean request, EMFErrorHandler errorHandler):* perform the domain controls on the data fields. Normally this method implements the controls for cross validation with data of the others fields. This method is invoked only if the formal controls have not produced errors. If the method return *true,* the validation process continues with others validators.

A page can have more validators so the system performs a chain of validation operations following this rules:

1. First, all the **check** methods of all the page validators are called. If one method return false, the validation chain is interrupted.
2. If no check method return false, Spago call the **validate** methods of all the page validators. If one method return false, the validation chain is interrupted.
3. At the end of the validation process, into the error stack are present the errors that occurred during the validation process.

The validators are defined into a file *validators.xml* with the following sintax:

```
<VALIDATORS>
        <VALIDATOR name="EMISSIONE_BIGLIETTO" class="spago.eng.it.validation.TestValidator"/>
</VALIDATORS>
```

Each VALIDATOR envelope has two attributes:

- NAME: logical name of the validator that will be referenced into the file *validation.xml*.
- CLASS: complete name of the class that perform the validation operations.

It is possible to use a validator for more pages or to define a complex validation job using more elementary validators.

3.6.3Changing the validation engine

It's possible to configure the validation engine used by Spago. In the file *validators.xml* there is an attribute called *ENGINE* that contains the class name of the class that implements the validation. This class implements the interface *it.eng.spago.validation.**ValidationEngineIFace***. This is a sample of the *validators.xml* file:

```
<VALIDATORS>
   <ENGINE class="it.eng.spago.validation.impl.ValidationImpl"/>
   .......
   <VALIDATOR name="<some validator>" class="<some class>"/>
   .......
</VALIDATORS>
```

If this parameter is missing the validation is not applied.

3.7Multipart form handling

Spago uses the library *commons-fileupload* to handle the multipart forms. The configuration file *upload.xml* contains the list of the file-upload handlers, so you can choose the implementation that best suit the needs of your application.
The attribute *UPLOAD-MANAGER* contains the name of the implementation to use.
Here is the structure of the file *upload.xml*:

```
<UPLOAD>
    <UPLOAD_METHODS>
        <UPLOAD_METHOD name="SAVE_FILE_AS_ATTRIBUTE"
            class="it.eng.spago.dispatching.httpchannel.upload.SaveAsAttributeUploader"/>
        <UPLOAD_METHOD name="SAVE_FILE_ON_DISK"
            class="it.eng.spago.dispatching.httpchannel.upload.SaveOnDiskUploader">
            <CONFIG PATH="C:\" />
        </UPLOAD_METHOD>
    </UPLOAD_METHODS>
    <UPLOAD-MANAGER name="SAVE_FILE_ON_DISK"/>
</UPLOAD>
```

The existing handlers are:

- *SAVE_FILE_AS_ATTRIBUTE*: saves the file of the form, as an attribute in the service request. For each file, two attributes are created, with the following names:
    - *<INPUT_FIELD_NAME>_FILENAME*: contains the file name. *<INPUT_FIELD_NAME>* is the field name as defined in the form.
    - *< INPUT_FIELD_NAME >_FILEDATA*: it's a byte array (byte[]) where Spago stores the content of the file.
- *SAVE_FILE_ON_DISK*: saves the file of the form, on disk, in a directory configured on the attribute PATH of the envelope CONFIG in the configuration file. For each file, in addition to saving it on disk, Spago creates an attribute with the name:

- <*INPUT_FIELD_NAME*>*_FILENAME*: contains the file name. < *INPUT_FIELD_NAME* > is the field name as defined in the form.

The default handler is the disk saving one.
It's possible to implement a custom handler, implementing the interface *it.eng.spago.dispatching.httpchannel.upload.**IUploadHandler*** defined as follows:

```
public interface IUploadHandler {
        public void upload(FileItem fileItem) throws Exception;
}
```

FileItem is the object created by *commons-fileupload*. The handler has to be configured in *upload.xml* and defined as main handler referencing it in the attribute *UPLOAD-MANAGER*.

3.8Prevent Resubmit

Spago provides an implementation of the pattern PreventResubmit to prevent the multiple submissions to server of the same form data. To every request, the *Coordinator* generate a new id and puts it into the *PermanentContainer.* The tag *GenerateToken (*specified inside a JSP page*)* generates, into the html form, the hidden field *SPAGO_TOKEN* that contains the id produced. After the form submit operation, the request service logic can control if the form id field value is equal to that stored into the PermanentContainer. If the two values are different the system generates a *EMFInternalError* containing the message "Navigation not permitted".
It is possible to disable this mechanism at run-time, simply setting the SPAGO_GENERATE_TOKEN=FALSE attribute into the request URL The Framework uses automatically this mechanism, like for example for the automatic creation of list / detail pages (see Automatic Generation of list / detail pages (one - shot) section), to prevent the resubmission of the data submitted on click of button declared in the<`DELETE_CAPTION`> (list) and <`SUBMIT_BUTTON` > (detail).

3.9Publication

Spago redirects the output data of business logic to the View module. It carries out publishing according to the request. When no publisher is configured for a specific channel request, the framework's response to the client is in XML format. In this way, it is possible to test the correct execution of business logic in the development phase before the View module is available. You can use one or more stylesheets for XML data description for all available channels:

- XML/XSLT : it is possible to transform the XML response stream using an XSLT stylesheet or a XSLT stylesheets cascade.
- JSP: the presentation is delegated to a JSP page. The page can access to the response envelope and to the possible errors.
- Servlet: the presentation is delegated to a Servlet that usually retrieves the information to show from the session (information setted from the actions and the modules previously executed).
- JAVA : a java class is called to determine the presentation modality to use. This choice is not properly a presentation but a switch for other real publication services, based on the conditions.

- LOOP (only for http channel): the service result is not really published but the AdapterHTTP invokes a new service using data contained into the service result.
- None: some services not need the presentation phase: a SOAP service could insert directly into the response the XML stream produced, for example. This modality is also very useful during the application debug.

Not all modalities are right to publish in all channels: the Servlet/JSP modality can't be used for SOAP or EJB channel, for example.

3.9.1Configuration

The first activity to publish a service response is associate the service action or page to the correspondent publication object. This mapping is defined into the *presentation.xml* file, as the follow example shows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PRESENTATION>
  <MAPPING
    business_name="LIST_USERS_ACTION"
    business_type="ACTION"
    publisher_name="ListUsersAction"/>
  <MAPPING
    business_name="AutomaticPageUsers"
    business_type="PAGE"
    publisher_name="AutomaticPageUsers"/>
</PRESENTATION>
```

Each action or page has a publisher object associated; in this way a single publisher can be used by more actions or pages.

The publisher objects are defined into the *publishers.xml* file.

```
<PUBLISHERS>
  <PUBLISHER name="ListUsersAction">
    <RENDERING channel="HTTP" type="XSL" mode="">
      <RESOURCES>
        <ITEM prog="0" resource="/WEB-INF/xsl/HTTPListUsersAction.xsl"/>
      </RESOURCES>
    </RENDERING>
    <RENDERING channel="WAP" type="XSL" mode="">
      <RESOURCES>
        <ITEM prog="0" resource="/WEB-INF/xsl/WAPListUsersAction.xsl"/>
      </RESOURCES>
    </RENDERING>
  </PUBLISHER>
```

```
  <PUBLISHER name="AutomaticPageUsers">
    <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
      <RESOURCES>
        <ITEM prog="0" resource="/jsp/demo/listdetail/PageUsers.jsp"/>
      </RESOURCES>
    </RENDERING>
  </PUBLISHER>
</PUBLISHERS>
```

For each publisher, the file defines a *PUBLISHER* envelope that, with the attribute *name*, assigns it a logical name:

```
<PUBLISHER name="publisher logical name">
```

The presentation subsystem needs to know the channel where the response will be published, a *RENDERING* envelope will be defined for every channel used.

```
<RENDERING channel="HTTP" type="XSL" mode="">
```

The channels are:

- HTTP
- WAP
- SOAP
- EJB

Once established the channel, the presentation modality must be specified. The possible values (explained in the section Publication) are:

- NOTHING
- SERVLET
- JSP
- XSL
- JAVA
- LOOP

Not all the modalities are valid for all the channels, the servlet modality have not meaning in the EJB channel, for example.
The following table shows the possible combinations:

Channel	type	Mode	prog	resource
HTTP	NOTHING		0	
	SERVLET	FORWARD	0	Servlet name
		SENDREDIRECT	0	Servlet name
	JSP	FORWARD	0	JSP page name
		SENDREDIRECT	0	JSP page name
	XSL		"0, 1, 2, …"	XSL stylesheets name
	LOOP		0	Parameters of the new request
WAP	NOTHING		0	
	SERVLET	FORWARD	0	Servlet name
		SENDREDIRECT	0	Servlet name
	JSP	FORWARD	0	JSP page name
		SENDREDIRECT	0	JSP page name
	XSL		"0, 1, 2, …"	XSL stylesheets name
SOAP	NOTHING		0	
	XSL		"0, 1, 2, …"	XSL stylesheets name
	LOOP		0	Parameters of the new request
EJB	NOTHING		0	
	XSL		"0, 1, 2, …"	XSL stylesheets name
JMS	NOTHING		0	
	XSL		"0, 1, 2, …"	XSL stylesheets name

3.9.2USER-AGENT DEPENDENT PUBLISHING

Some applications need to publish on different resources (jsp, xsl ect..) depending on the user-agent of the request call.

To handle these problems in Spago is possible to have more than one RENDERING section child of PUBLISHER elements for the same channel, each one different only for the value of the optional *user-agent* attribute.
The value of *user-agent* attribute identify a logic user-agents inside Spago. A Mapping between strings that arrives in request and logical user agent is made creating a *USER-AGENT* element on the *useragents.xml* configuration file.

For example the following section defines two logical user agents user agents called IE6 and Firefox.

```
<USER-AGENTS>

   <USER-AGENT logicalName="IE6"
               identifierString="Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
               1.1.4322)"/>
   <USER-AGENT logicalName="Firefox"
               identifierString="Mozilla/5.0 (Windows; U; Windows NT 5.1; it-IT; rv:1.7.10)
               Gecko/20050717 Firefox/1.0.6"/>
</USER-AGENTS>
```

that can be used for agent dependent publishing in the publisher.xml configuration file:

```
<PUBLISHER name="AGENT_DEPENDENT_PUBLISHER">
```

```
        <RENDERING channel="HTTP" type="JSP" mode="FORWARD" user-agent="IE6">
                <RESOURCES>
                        <ITEM prog="0" resource="/testAgentIE.jsp"/>
                </RESOURCES>
        </RENDERING>

        <RENDERING channel="HTTP" type="JSP" mode="FORWARD" user-agent="Firefox">
                <RESOURCES>
                        <ITEM prog="0" resource="/testAgentFirefox.jsp" />
                </RESOURCES>
        </RENDERING>

</PUBLISHER>
```

In that case, the service (ACTION, o PAGE) that is associated to *AGENT_DEPENDENT_PUBLISHER* in *presentation.xml* file, will result in the rendering of */testAgentIE.jsp* in the case that the request come from Internet Explorer or */testAgentFirefox.jsp* in the case we're using firefox.

If the request string that identify user-agent doesn't identify a logical user agent in Spago or the matching logical user-agent doesn't have a specific RENDERING section, the first RENDERING section available for the channel will be used.

3.9.3XML/XSLT

This modality allows to transform the xml response using XSLT stylesheets. It is possible to associate more than one stylesheet for every service. The XML/XSL transformation is an onerous task if the XSLT stylesheet is compiled every time, so the framework compile the stylesheets at start-up time and put them in cache. If a modify occurs to a stylesheet, the framework recompile it automatically without reload the application. The transformation result stream cannot be only HTML but also any other set of semantic tags or instructions.
Spago developers performed some test whit Crystal Clear and Jreport Engines to produce a PDF output. The tests with FOP (Formatting Objects Processor) demonstrated that is a complex mechanism to manage.
Spago use Xalan libraries as transcoder system.

3.9.4Servlet/JSP

The service result can be showed by a JSP page or by a Servlet that can produce the entire HTML stream.
To build a new JSP it is possible to extend the class *it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage*. In this way it is possible to access to the objects requestContainer, responseContainer, errorHandler:

- public RequestContainer getRequestContainer(HttpServletRequest request);
- public SourceBean getServiceRequest(HttpServletRequest request);
- public ResponseContainer getResponseContainer(HttpServletRequest request);
- public SourceBean getServiceResponse(HttpServletRequest request);
- public EMFErrorHandler getErrorHandler(HttpServletRequest request);

3.9.5JAVA

The JAVA modality allows to execute a java class that, at run time, can redirect the presentation to different publishers. Developers can declare their own classes or extend the framework class AbstractPublisherDispatcher that implements services for recover the publisher configuration. The configuration is defined into the CONFIG envelope as show below:

```
<PUBLISHER name="DefaultPublisherDispatcher">
    <RENDERING channel="HTTP" type="JAVA" mode="">
        <RESOURCES>
            <ITEM prog="0" resource="it.eng.spago.presentation.DefaultPublisherDispatcher">
            <CONFIG>
              ...
            </CONFIG>
            </ITEM>
        </RESOURCES>
    </RENDERING>
</PUBLISHER>
```

The java class can retrieve from response the information to decide the publisher to use. A typical example of this mechanism, it is to check if into the stack errors are stored some error for publishing the JSP error page or the JSP page associated with the service.

Spago defines a default java publisher for simple publisher selection, configurable into the configuration file. The publisher class (extends the class AbstractPublisherDispatcher) to registry is:

```
it.eng.spago.presentation.DefaultPublisherDispatcher
```

The structure should look like this:

```
<CHECKS>
  <CHECK target="DefaultJavaPublisher1">
    <CONDITIONS>
        <PARAMETER name="param1" scope="SERVICE_REQUEST" value="0" />
        <PARAMETER name="param2" scope="SESSION" value="AF_DEFINED" />
    </CONDITIONS>
  </CHECK>
  <CHECK target="DefaultJavaPublisher2">
     <CONDITIONS>
        <PARAMETER name="param1" scope="SERVICE_REQUEST" value="1" />
     </CONDITIONS>
  </CHECK>
</CHECKS>
```

The CHECKS envelope must contain all controls that the class has to perform to establish the publisher to use. For every different logical publisher target there's a CHECK envelope which define the conditions that satisfy it. The class will call the publisher that verifies all conditions.

The *name* attribute defines the name of the attribute to search inside the containers; the scope attribute identifies the container where search the attribute and its possible values are:

- USER
- ADAPTER_REQUEST
- SERVICE_REQUEST
- SESSION
- APPLICATION
- ADAPTER_RESPONSE
- SERVICE_RESPONSE

The value attribute can define two constants to specify the presence / absence into the container or the value:

- AF_DEFINED
- AF_NOT_DEFINED
- param_value

3.9.6LOOP

The LOOP modality is available only in HTTP and SOAP channels and allows to execute a loop_back operation. The scenery is:

- The browser invokes a service calling the AdapterHTTP
- The business logic is executed
- The presentation is delegated to the AdapterHTTP that invokes a new service
- The business logic of the new service is executed
- The presentation is delegated to the publisher associated to the new service
- The response of the service returns to the browser

The parameters of the new serviceRequest are defined into the <PARAMETER> envelopes with the usual grammar used for the dynamic parameters declaration:

```
<PUBLISHER name="LoopBasicListUsersAction">
        <RENDERING channel="HTTP" type="LOOP" mode="">
                <RESOURCES>
                        <PARAMETER name="PAGE" type="ABSOLUTE" value="SmartPageUsers" scope="" />
                </RESOURCES>
        </RENDERING>
</PUBLISHER>
```

It is also possible to generate the second request at run-time as in the following example:

```
 <PUBLISHER name="NewLoopPublisher">
      <RENDERING channel="HTTP" type="LOOP" mode="">
          <RESOURCES>
                <PARAMETER name="" type="RELATIVE" value="MODULE_RESPONSE.SERVICE_REQUEST"
          scope="SERVICE_RESPONSE" />
          </RESOURCES>
       </RENDERING>
</PUBLISHER>
```

3.9.7An example

This paragraph describe, with an example, the possibility to associate different publishers to an actions according to the channels. The following action performs a query to retrieve a users list.

```
public class ListaUtentiAction extends AbstractAction {
    public ListaUtentiAction() {
    } // public ListaUtentiAction()

    public void service(SourceBean request, SourceBean response) {
        DataConnection dataConnection = null;
        SQLCommand sqlCommand = null;
        DataResult dataResult = null;
        try {
            DataConnectionManager dataConnectionManager = DataConnectionManager.getInstance();
            dataConnection = dataConnectionManager.getConnection("spagodemo");
            String statement = SQLStatements.getStatement("LISTA_UTENTI");
            sqlCommand = dataConnection.createSelectCommand(statement);
            dataResult = sqlCommand.execute();
            ScrollableDataResult scrollableDataResult = (ScrollableDataResult)dataResult.getDataObject();
            response.setAttribute(scrollableDataResult.getSourceBean());
        } // try
        catch (Exception ex) {
            TracerSingleton.log(Constants.NOME_MODULO, TracerSingleton.CRITICAL, "QueryExecutor::executeQuery:", ex);
        } // catch (Exception ex) try
        finally {
                                Utils.releaseResources(dataConnection, sqlCommand, dataResult);
        } // finally try
    } // public void service(SourceBean request, SourceBean response)
} // public class ListaUtentiAction extends AbstractAction
```

We not analyse the action code but the response produced by the action:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE RESPONSE (View Source for full doctype...)>
<RESPONSE>
        <SERVICE_RESPONSE>
                <ROWS>
                <ROW COGNOME="Bellio" EMAIL="luigi.bellio@eng.it" ID_USER="10" NOME="Luigi" />
                <ROW COGNOME="Ferrari" EMAIL="andrea.ferrari@eng.it" ID_USER="254" NOME="Andrea" />
                <ROW COGNOME="Butano" EMAIL="daniela.butano@eng.it" ID_USER="250" NOME="Daniela" />
                </ROWS>
        </SERVICE_RESPONSE>
<ERRORS/>
</RESPONSE>
```

In the next paragraphs, the following type of publisher will be applicated to the response.

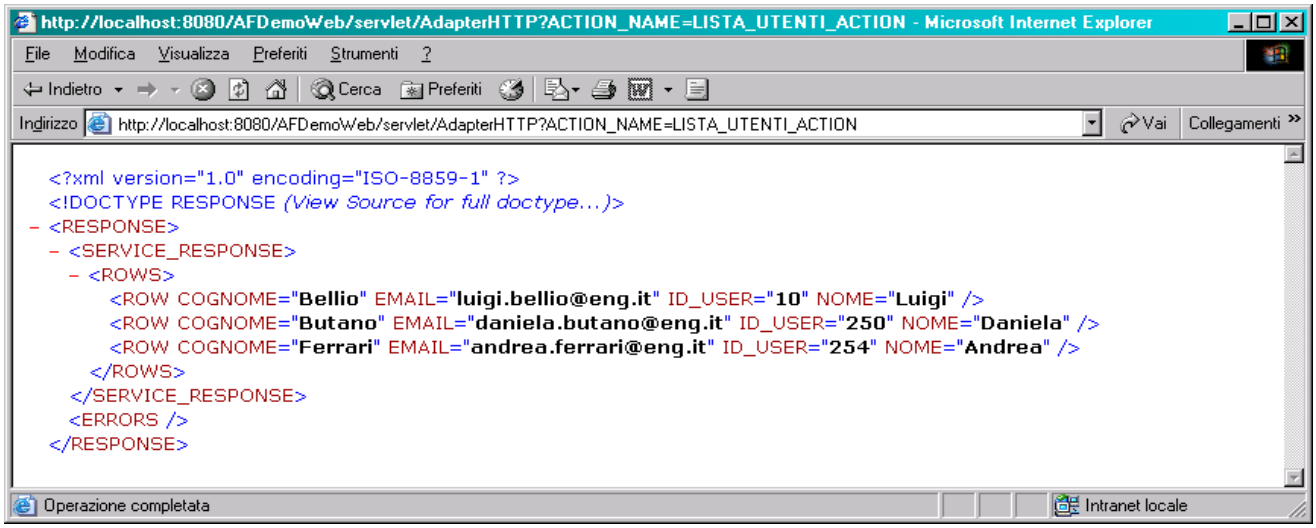
- None
- XSL
- JSP

3.9.7.1HTTP channel and none publisher

If the action doesn't have any publisher, the XML stream produced is sent to the browser without any modify. The actual browsers are able to show an xml document inside their editor. To obtain this result you should NOT indicate a publisher associated to the action into the *presentation.xml* file. You have to comment the association envelope:

```
<MAPPING
   business_name="LISTA_UTENTI_ACTION"
   business_type="ACTION"
   publisher_name="ListaUtentiAction"/>
```

The result of the actions invocation, with an Internet Explorer browser, is the following:



3.9.7.2HTTP channel and XSL transformation

This paragraph describes how configure an XSL type publisher. It is necessary associate a publisher to the action *ListaUtentiAction* (remove comments into the xml mapping showed in the previous paragraph). By invoking the following URL:

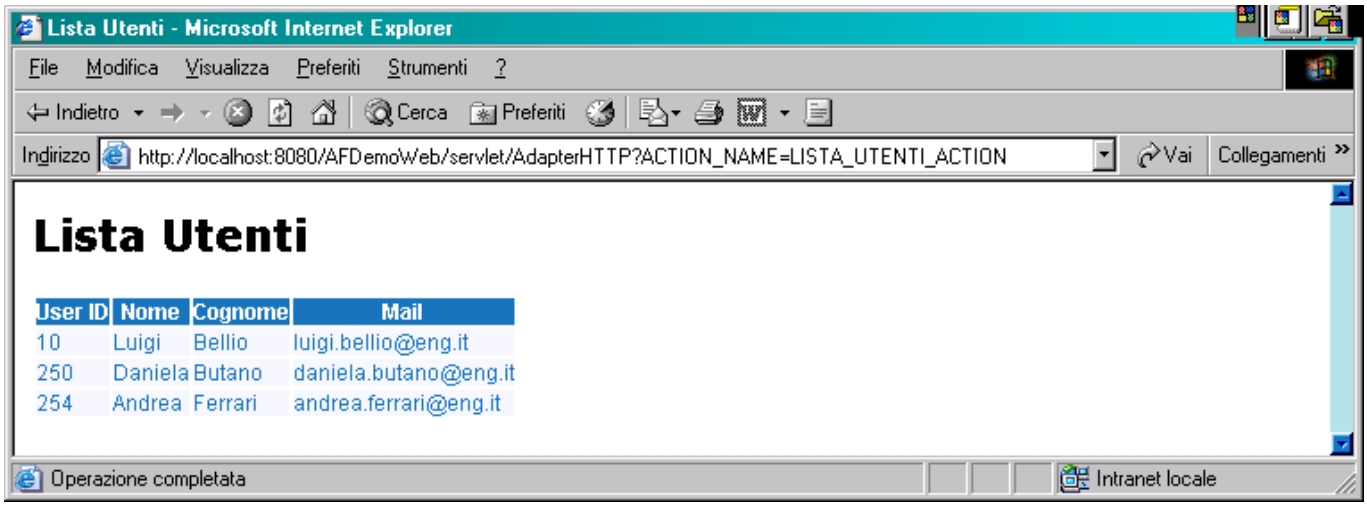
http://localhost:8080/AFDemoWeb/servlet/AdapterHTTP?ACTION_NAME=LIST_USERS_ACTION

the framework retrieves the logical publisher name associated to the action and looks into the publishers configuration to perform the presentation. The publisher is configured into the *publishers.xml* file as below:

```
<PUBLISHERS>
   <PUBLISHER name="ListaUtentiAction">
      <RENDERING channel="HTTP" type="XSL" mode="">
         <RESOURCES>
            <ITEM prog="0" resource="/WEB-INF/xsl/HTTPListaUtentiAction.xsl"/>
         </RESOURCES>
      </RENDERING>
   </PUBLISHER>
</PUBLISHERS>
```

The configuration specifies that the publisher is an XSL transformation and will use the *HTTPListaUtentiAction.xsl* stylesheet. It is possible to define more than one stylesheets to apply, each one distinguished by a progressive number (*prog* attribute) that determines the order.

For the HTTP channel, the service produces the HTML page showed below, using the *ListaUtentiAction* response and the XSL transformation stylesheet *HTTPListaUtentiAction.xsl.*



The XSL transcoding file should look like as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="RESPONSE/SERVICE_RESPONSE">
<html>
   <head>
      <link rel="stylesheet" type="text/css" href="../css/spago/listdetail.css"/>
      <title>Lista Utenti</title>
   </head>
   <body>
      <h2>Lista Utenti</h2>
      <table class="LISTA" border="0" cellpadding="0" cellspacing="2">
         <tr class="LISTA">
            <th class="LISTA">User ID</th>
            <th class="LISTA">Nome</th>
            <th class="LISTA">Cognome</th>
            <th class="LISTA">Mail</th>
         </tr>
         <xsl:apply-templates/>
      </table>
   </body>
</html>
</xsl:template>

<xsl:template match="RESPONSE/SERVICE_RESPONSE/ROWS/ROW">
         <tr class="LISTA">
            <td class="LISTA"><xsl:value-of select="@ID_USER"/></td>
            <td class="LISTA"><xsl:value-of select="@NOME"/></td>
            <td class="LISTA"><xsl:value-of select="@COGNOME"/></td>
            <td class="LISTA"><xsl:value-of select="@EMAIL"/></td>
         </tr>
</xsl:template>

</xsl:stylesheet>
```

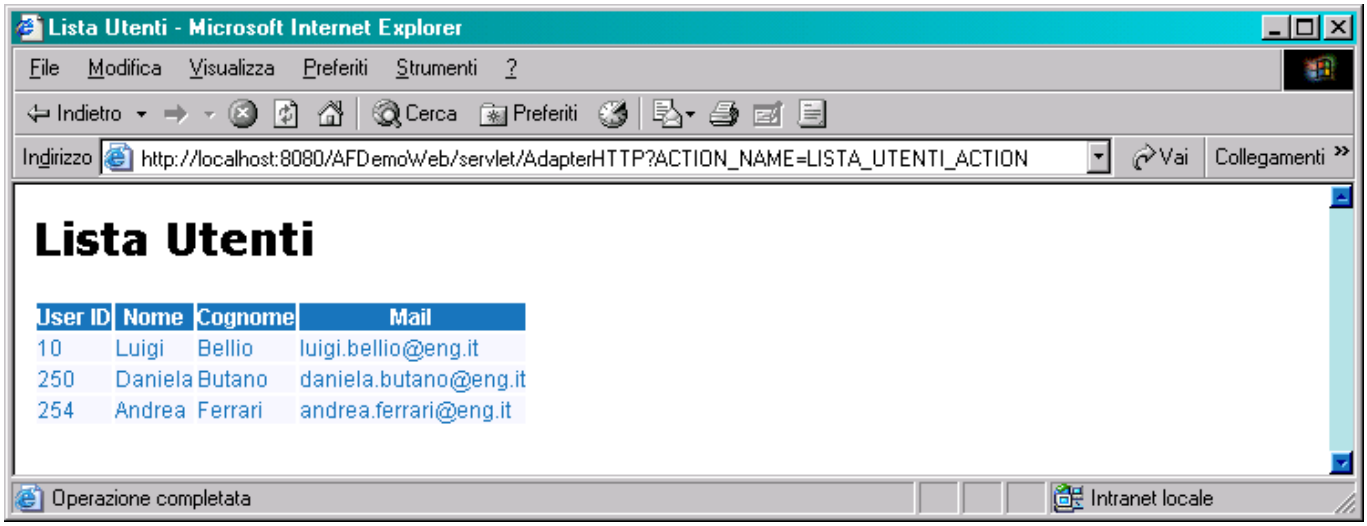
The stylesheet simply produces a table row for every database rows contained in the response.

3.9.7.3HTTP Channel and JSP Publisher

The same result can be obtained using a JSP page. It is not necessary to modify the mapping between the action and the logical publisher into the file *presentation.xml* but only to change the publisher configuration, into the file *publishers.xml,* to declare that the publisher is a JSP page.

```
<PUBLISHERS>
   <PUBLISHER name="ListaUtentiAction">
      <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
         <RESOURCES>
            <ITEM prog="0" resource="/jsp/demo/listdetail/ListaUtenti.jsp"/>
         </RESOURCES>
      </RENDERING>
   </PUBLISHER>
</PUBLISHERS>
```

The JSP produces an HTML page like this (the same obtained with XSL transformation):



The JSP code is the following:

```
<%@ page import="it.eng.spago.base.*, it.eng.spago.configuration.ConfigSingleton,
        it.eng.spago.tracing.TracerSingleton, java.lang.*,  java.text.*,  java.util.*"%>
<%@ taglib uri="spagotags" prefix="spago"%>
<%
   ResponseContainer responseContainer = ResponseContainerAccess.getResponseContainer(request);
   SourceBean serviceResponse = responseContainer.getServiceResponse();
%>

<html>
   <head>
      <link rel="stylesheet" type="text/css" href="../css/spago/listdetail.css"/>
      <title>Lista Utenti</title>
   </head>
   <body>
      <h2>Lista Utenti</h2>
      <table class="LISTA" border="0" cellpadding="0" cellspacing="2">
         <tr class="LISTA">
            <th class="LISTA">User ID</th>
            <th class="LISTA">Nome</th>
            <th class="LISTA">Cognome</th>
            <th class="LISTA">Mail</th>
         </tr>
```

```
<%  Vector rows = serviceResponse.getAttributeAsVector("ROWS.ROW");
   for (Enumeration enum = rows.elements(); enum.hasMoreElements();)
   {
          SourceBean row = (SourceBean)enum.nextElement();  %>
            <tr class="LISTA">
               <td class="LISTA"><%= row.getAttribute("ID_USER") %></td>
               <td class="LISTA"><%= row.getAttribute("NOME") %></td>
               <td class="LISTA"><%= row.getAttribute("COGNOME") %></td>
               <td class="LISTA"><%= row.getAttribute("EMAIL") %></td>
            </tr>
<% } %>
      </table>
   </body>
</html>
```

The JSP can access to the object responseContainer and serviceResponse to retrieve users attribute to show.

3.9.7.4WAP Channel and XSL Transformation

The publication system has to know the channel used for the presentation because, with various channels, the information to show can be different or can be visualized in different way. This example show how the same action, used in previous paragraphs with HTTP channel, can be activated from a WAP browser. The result will be constructed transforming the xml service stream response with a transcoding XSL stylesheet, obviously different from which one used in http example. Into the *publishers.xml* file it is necessary to add a new *RENDERING* envelope for manage the WAP channel.

```
<PUBLISHER name="ListaUtentiAction">
   <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
      <RESOURCES>
         <ITEM prog="0" resource="/jsp/demo/listdetail/ListaUtenti.jsp"/>
      </RESOURCES>
   </RENDERING>
   <RENDERING channel="WAP" type="XSL" mode="">
      <RESOURCES>
         <ITEM prog="0" resource="/WEB-INF/xsl/WAPListaUtentiAction.xsl"/>
      </RESOURCES>
   </RENDERING>
</PUBLISHER>
```

To try this presentation modality it has been used a WAP emulator, not a really WAP browser, so the request doesn't pass through a WAP gateway. In this way the framework can't establish that the request comes from a WAP channel. To provide this information to the framework it has been necessary to add a new parameter ***CHANNEL_TYPE=WAP*** into the URL request. So the URL called in order to try the action is:

*http://localhost:8080/AFDemoWeb/servlet/AdapterHTTP?ACTION_NAME=LISTA_UTENTI_ACTION&**CHANNEL_TYPE=WAP***

Invoking the request from a WAP emulator we obtain the result showed in the following image:

The XSL used has the same structure of that one used for the HTTP channel, but it does the rendering of less information.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" doctype-public="-//WAPFORUM//DTD WML 1.1//EN" doctype-
system="http://www.wapforum.org/DTD/wml_1.1.xml"/>
```

```
<xsl:template match="RESPONSE/SERVICE_RESPONSE">
<wml>
   <card id="ListaUtenti" title="Lista Utenti">
      <p>
         <table columns="2" align="LCC">
            <tr>
               <td>User ID</td>
               <td>Mail</td>
            </tr>
```

```
            <xsl:apply-templates/>
         </table>
      </p>
   </card>
</wml>
</xsl:template>
```

```
<xsl:template match="RESPONSE/SERVICE_RESPONSE/ROWS/ROW">
         <tr>
            <td><xsl:value-of select="@ID_USER"/></td>
            <td><xsl:value-of select="@EMAIL"/></td>
         </tr>
</xsl:template>
```

```
</xsl:stylesheet>
```

3.10Tag Libraries

Spago provides the following tag libraries:

- **list**: used for the automatic generation of list pages. The tag retrieves the layout information from the xml envelope <*CONFIG*>, relative to the module or action that produces the list data, defined into the configuration xml node (into the *actions.xml* or *modules.xml*). It recovers the list data from the *serviceResponse* produced by the action or module.
- **detail**: used for the automatic generation of a detail page. The tag retrieves the layout information from the xml envelope <*CONFIG*>, relative to the module or action that produces the list data, defined into the configuration xml node (into the *actions.xml* or *modules.xml*). It recovers the detail data from the *serviceResponse* produced by the action or module.
- **error**: the tag implements the javascript function *checkError()* that shows, using a pop-up, the error stack contained into the ErrorHandler.
- **lookup** and **lookupClient**: Spago implements the lookup functionality through two tags. The JSP page that contains the lookup field must contain the **lookupClient** tag. The "onClick()" event handler, defined in the lookup field, has to call the function *lookup_valueidLookup()*. The javascript function, defined from the tag, will open a new window showing the lookup list. The second JSP page, visualized into the new window, must use the **lookup** tag for the creation of the lookup list. The tags Lookup and lookupClient use the configuration parameters defined into the file *lookup.xml.*

```
<LOOKUPS>
    <LOOKUP idlookup="ListUsers">
            <PARAMETERS pageName="ListUsersLookupPage" moduleName="ListUsersLookupModule"
            formName="StartLookup">
                    <FIELDS>
                            <FIELD lookupTableField="userid" formField="userid" likeField="true" />
                    </FIELDS>
            </PARAMETERS>
    </LOOKUP>
</LOOKUPS>
```

The lookupClient tag uses the <*PARAMETERS*> envelope to invoke the module that executes the list rendering; the <*FIELD*> envelopes are useful to associate the form fields with the list fields.

- **navigationToolbar**: this tag shows a navigation toolbar to select one of the services already executed during the user / application interaction. For more details see Navigation Toolbar.
- **comboNavigationToolbar**: this tag shows a navigation combo field to select one of the services already executed during the user / application interaction. For more details see Navigation Toolbar.
- **generateToken**: this tag generates the hidden field *SPAGO_TOKEN* holding the last id, produced by the server, for the prevent resubmit mechanism.
- **message**: this tag shows the message, associated to the *code* attribute, hold into the properties file *message_languagecode_countrycode.properties.*
- **reloadField**: retrieves from the *serviceRequest* the value of the *reloadField* attribute. It is used to show, in case of error, the inserted values.

- **writeString**: this tag shows the string passed in input or the optional *nullValue* attribute (default value is blank) if the parameter is null.

3.11Exceptions Handler

The framework handles the contextual service request anomalies in the following way:

- If the client requests a service but the session is expired the framework execute an action registered with the name "**SESSION_EXPIRED_ACTION**" into the actions.xml file
- If the client requests a service and the relative "action" or "module" throws an exception, the framework publishes it in the "**SERVICE_ERROR_PUBLISHER**" registered into the publishers.xml file.
- If the client requests a service and the JSP throws an "exception" the framework publishes the page declared as jsp error: **"/jsp/spago/jspError.jsp**"

3.12Multi-Language

Spago implements a mechanism that permits to change the application messages handling different languages.
The errors multilanguage catalogue is stored into the properties files placed into the application classpath, following the java standard mechanism. The files have a structure that should look like this:

10001 = The list doesn't contain lines
10002 = Cancellation executed correctly
10003 = Error during the cancellation

Every message has a code used for retrieve it. The code is used to instance the EMFUserError object:

*public EMFUserError(String severity, int **code**)*

The files holding the messages in different languages, are placed in the application classpath and they are called *message_languagecode_countrycode.properties*, where *languagecode* and *countrycode* are the codes assigned to the language and country (constants defined into the java class *java.util.Locale*). For example, the file for the american language must be called *messages_en_US.properties* while the one for the italian language must be call *messages_it_IT.properties.* To use a specific language, the developers must define, into the *permanent container,* two strings containing the language and country code, as showed:

```
SessionContainer permanentContainer = requestContainer.getSessionContainer().getPermanentContainer();
permanentContainer.setAttribute("AF_LANGUAGE", "it");
permanentContainer.setAttribute("AF_COUNTRY", "IT");
```

The language will be the default for the session. Normally these values are set during the login phase based on the users preferences.

3.12.1Multi-language presentation

The framework allows to manage the application messages in a multilanguage modality using a property file for each language. The framework can retrieve the message, using the object *MessageBundle*:

*public static String **getMessage**(String **code**)*

This mechanism permits to handle messages multi-language and the display items too. See the ***message*** tag for more details.
For the SOAP and EJB channel the language problem is not important because the services should provide only data, without presentation logic.

3.13Distribution of the business logic

An important Spago feature is the possibility to choice the execution of the business logic (actions and modules) on a web container or on an EJB container, using a local or remote interface. It is possible simply changing some parameters contained into the *distribution.xml* files. The main differences are relative to the database management, see the Transaction management for more details.
The distribution of services execution is set into the configuration file *distribution.xml* as show in the following snippet :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<DISTRIBUTIONS>
    <DISTRIBUTION business_type="ACTION" business_name="LIST_USERS_ACTION"
        distribute_coordinator_class="it.eng.spago.dispatching.distribute.DistributeCoordinator" />
    <DISTRIBUTION business_type="PAGE" business_name="AutomaticPageUsers"
        distribute_coordinator_class="it.eng.spago.dispatching.distribute.DistributeCoordinator" />
    <DISTRIBUTION business_type="PAGE" business_name="BasicListUsers"
        distribute_coordinator_class="it.eng.spago.dispatching.distribute.RemoteDistributeCoordinator" />
</DISTRIBUTIONS>
```

This file define all the actions and pages that must be executed on an EJB container. All the modules of a page, associated to an EJB container, are executed in this environment. It is not possible to have an hybrid page which has some modules executed on web container and other on EJB container. Using the local interfaces (the DistributeCoordinator) the performances don't decrease too much. If an operation fails, during the generation of a page, the system makes a rollback of all the page modules, because the transaction is associated to the page and not to the single modules.

3.14Data Access

The Spago Data Abstract Layer is a module, implemented over JDBC, that virtualizes the data access, providing only one interface for access data in different databases. The benefit of this architecture is a unique model to access data and an easy way to import an application from one database to another. The module, furthermore, implements a set of new functionalities, like the management of connection pool or a scrollable result set.
It is also possible using native types of different database vendors (like Oracle CLOB type for example) but this choice make more difficult, or in some case impossible, the application porting.
Developers can use it inside Spago actions or modules and all data result are stored inside a SourceBean object that can be easily transform into XML.

3.14.1Requirements

The Data Access Layer has been implemented for satisfy the follow requirements:

1. Meet the SQL '92 standard that define the commons sql commands like SELECT, INSERT, UPDATE, DELETE.
2. Possibility to use different database in the same application and to execute sql commands on each one of them.
3. Independence from the database vendor and lack of database specific code.
4. Independence from the JDBC driver used.
5. Manage "native connection pool" (created at application start – up using configuration parameters define in a configuration file) and "managed connection pool" (created from an application server and retrieve using a JNDI context)
6. Manage JDBC transactions, considering that all sql operations external to one transaction will be executed with autocommit modality.
7. The execution result of queries or operations must be of one type self-describing (different operations will return different result objects that implement an unique interface which defines a method for identify the type of result)
8. String format of Date or Timestamp JDBC object must be unique and comply with the one defined in the data access layer configuration file. These requirements arise because with different database the string format of Timestamp or Date can be different , so the application can't be portable. This mechanism allows to use an only format to set and retrieve information of Date and Timestamp type.
9. All module operations must throw exception of type "EngInternalError" if an error occurs during the execution.

3.14.2General Principles of planning

The planning of the data access module needed to use some best practices and specific **Design Pattern,** like:

- *The object DataConnectionManager* , that manage all the application connection pools, has been implemented like a Singleton.
- Use of **AbstractFactory** and **FactoryMethod** patterns for the creation of all objects that have different implementation but are compliant to one interface.

- The objects of type *DataField*, *ScrollableDataResult*, *SQLCommand* are examples of objects created by an AbstractFactory or a FactoryMethod.
- The objects of type *SQLCommand* are compliant to the **Command** pattern.
- The objects that implement the **ConnectionPoolInterface** are created using introspection methods in order to avoid the inclusion of all vendor libraries during the code compilation phase.

3.14.3Management of connections and pools

The fundamental object of data access module is *DataConnectionManager* that implements the following functionalities*:*

1. Create and configure connection pool objects compliant to the interface *ConnectionPoolInterface*.
2. Provide connection for all the managed databases.

The connection pools are defined by a xml file (*data_access.xml*) that contains all the necessary parameters to instantiate and to initialize them. The most important parameter is the complete name of the java class that take care to build a connection pool for a specific database.

This object increase the performance of the application and allows the developers to not worry about the creation of JDBC connections. The developers, who want to get a connection to a database, must simply write:

Code Example 2-1: get a database connection

```
DataConnection dcDefault = DataConnectionManager.getInstance().getConnection();
DataConnection dcOracle = DataConnectionManager.getInstance().getConnection("oracle");
DataConnection dcDB2 = DataConnectionManager.getInstance().getConnection("DB2");
//do something
dcDefault.close();
dcOracle.close();
dcDB2.close();
```

An example of the connection pool configuration file (*data_access.xml*) can be:

Code Example 2-2: data_access.xml

```
<DATA-ACCESS>
  <DATE-FORMAT format="DD-MM-YYYY"/>
  <TIMESTAMP-FORMAT format="DD-MM-YYYY hh:mm:ss"/>
  <CONNECTION-POOL connectionPoolName="demo"
                   connectionPoolFactoryClass="it.eng.spago.dbaccess.factory.
                                               OracleConnectionPoolDataSourceFactory">
    <CONNECTION-POOL-PARAMETER parameterName="connectionString"
      parameterValue="jdbc:oracle:thin:@xxx.xxx.xxx.xxx:xxxx:XXX"
      parameterType=""/>
    <CONNECTION-POOL-PARAMETER parameterName="jdbcDriver"
      parameterValue="oracle.jdbc.OracleDriver" parameterType=""/>
    <CONNECTION-POOL-PARAMETER parameterName="driverVersion" parameterValue="2.1"/>
    <CONNECTION-POOL-PARAMETER parameterName="user" parameterValue="PIPPO"/>
```

```
    <CONNECTION-POOL-PARAMETER parameterName="userPassword" parameterValue="PIPPO"/>
    <CONNECTION-POOL-PARAMETER parameterName="poolMinLimit" parameterValue="0"/>
    <CONNECTION-POOL-PARAMETER parameterName="poolMaxLimit" parameterValue="0"/>
    <CONNECTION-POOL-PARAMETER parameterName="cacheScheme"
        parameterValue="DYNAMIC_SCHEME" />
    <CONNECTION-POOL-PARAMETER parameterName="sqlMapperClass"
        parameterValue="it.eng.spago.dbaccess.sql.mappers.OracleSQLMapper"/>
  </CONNECTION-POOL>
  <CONNECTION-MANAGER>
    <REGISTER-POOL registeredPoolName="demo"/>
  </CONNECTION-MANAGER>
</DATA-ACCESS>
```

Each pool is defined by an envelope *CONNECTION-POOL* with the following attributes*:*

- *connectionPoolName*: name of the pool. This name is used from *DataConnectionManager.getConnection* because this method needs to know which pool to use (If no name is specified the method access the first pool of the list).
- *connectionPoolFactoryClass*: factory that instantiates the connection pool object. The framework provides this implementations:
  - *it.eng.spago.dbaccess.factory.**AppServerManagedConnectionPoolFactory*** that instantiates an object *it.eng.spago.dbaccess.pool.**AppServerManagedConnectionPool*** that access, using JNDI name and context, to a connection pool managed by the application server. In this case most of the configuration parameters aren't obbligatory because often they are already defined in the configuration file of the application server.
  - *it.eng.spago.dbaccess.factory*. ***OracleConnectionPoolDataSourceFactory*** that instantiates a native Oracle connection pool compliant to JDBC 2.0 specification.
- *connectionString*: the string connection to the database in the format defined by JDBC specification (only for native pool).
- *JNDIName*: the JNDI name of the pool managed from the application server (only for managed pool).
- *jdbcDriver*: driver JDBC uses for access to the database (only for native pool).
- *driverVersion*: version of the driver (only for native pool).
- *user*: user name used to open a connection.
- *userPassword*: password used to open a connection.
- *poolMinLimit*: the minimal number of open connections to maintain.
- *poolMaxLimit*: the maximum number of open connections to maintain.
- *sqlMapperClass*: java class for the conversion of a Timestamp / Date value to or from the respective string format. The string value of Date / Timestamp objects has different format if recovered from different databases and this difference make the application not completely portable. Map can erase this problem because they transform the different string value format to the one define in the configuration file. This mechanism allows to solve the porting problem simply changing the mapper class. The framework provides the following mapper:
  - *it.eng.spago.dbaccess.sql.mappers.**DB2SQLMapper***: mapper for DB2.
  - *it.eng.spago.dbaccess.sql.mappers.**MySQLMapper***: mapper for MySQL.

- *it.eng.spago.dbaccess.sql.mappers.**OracleSQLMapper***: mapper for Oracle.

It is possible to add new mappers implementing the SQLMapper interface.

Some specific parameters have been introduced for the Oracle native connection pool:

- *cacheTimeToLiveTimeout (optional)*
- *cacheInactivityTimeout (optional)*
- *cacheScheme*:

The meaning of this parameters can be found looking at the Oracle documentation.

The Oracle native connection pool provides also the possibility to enchiper the user passwords stored inside the configuration file. This functionality renders necessary a mechanism for dechiper the passwords so Spago contains a *PasswordProvider* class that implement the dechiper service. Developers must include, into the cofiguration xml envelope a new parameter:

- Name: *algorithm*
- Value: *it.eng.spago.dbaccess.encript.PasswordProvider*

The chypher / dechypher algorithm used is a DES/ECB/PKCS5Padding specified by SUN JCE. The used key is stored inside the jar file of the framework.
In order to generate new keys and new passwords Spago includes a simple graphical interface which is possible to launch using the command "java *it.eng.spago.dbaccess.encript.gui.CriptGui*" from a shell of an operating system.
The algorithm used from this tool in part of JDK1.4 so for use it with JDK 1.3/1.2 it is necessary to insert into the classpath the jar: "jce1_2_2.jar" included with the framework distribution .

3.14.4Execution of SQL Commands

The DataConnection class is a wrapper of the well know object *java.sql.Connection* and it has the important job to create command objects, which are use to execute SQL statements. Because of the support for JDBC 1.0 and JDBC 2.0 the command object for the two specification are different, so the DataConnection expose the factory methods for create the right commands for the JDBC specification used.

Code Example 2-3: execution of sql commands

```
//
// You've already obtained the DataConnection dc Object
//
String strSelect = "SELECT * FROM TAB_PROVA WHERE A=?, B=?";

SQLCommand cmdSelect = dc.createSelectCommand(str)
List inputParameter = new ArrayList(2);
DataResult dr = cmdSelect.execute(inputParameters);
```

3.14.5Census of the statements

Spago give the possibility to store all the SQL statements used by the application in one xml configuration file . Each statement is represented by an xml envelope with two attributes:

- Name: name of the query that permits to identify and retrieve it.
- Value: the sql query expression

The configuration file that contains the statements is *statements.xml* and an example part of this file is showed below*:*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<STATEMENTS>
  <STATEMENT
    name="LISTA_UTENTI"
    query="SELECT userid,nome,cognome,mail,telefono,indirizzo FROM utenti ORDER BY userid"/>
</STATEMENTS>
```

Developers can retrieve the query string simply calling the *SQLStatements.getStatement* method, passing the query name: *String statement = SQLStatements.getStatement("LISTA_UTENTI");*

3.14.6Get the result of a SQL command

One ot the requirements for the data access module was that the execution of all different SQL command had to return the same object result type. To satisfy this requirement Spago implements an object *DataResult* that is a container for one of the SQL command result objects, which are different objects but compliant to the same interface *DataResultInterface*. After the execution of a SQL command a specific result object is created and it is put inside the DataResult, which know in every moment the specific type of the contained object.
The example below show that the execution of a query return a DataResult type and that this object contains scrollable result set, which is the specific result object for a SELECT command.

Code Example 2-4: result of a sql command

```
//
// You've already obtained the DataConnection dc Object
//
String strSelect = "SELECT * FROM TAB_PROVA WHERE A=?, B=?";
SQLCommand cmdSelect = dc.createSelectCommand(str)
List inputParameter = new ArrayList(2);

DataResult dr = cmdSelect.execute(inputParameters);
if (dr.getDataResultType().equals(DataResultInterface.SCROLLABLE_DATA_RESULT)){
      ScrollableDataResult sdr = (ScrollableDataResult)dr.getDataObject();
}
```

The different result objects compliant to the interface *DataResultInterface* are *four:*

1. **ScrollableDataResult**: this object is a wrapper for *java.sql.ResultSet* so probably is the most used. The service provides are those of the JDBC resultset and services of direct / inverse navigation and absolute positioning. For compatibility reasons with JDBC 1.0 the object doesn't support direct update of resultset data.
2. **InformationDataResult**: this object is created after the execution of an insert/update/delete command and it contains information that tell if the statement was executed correctly and how much rows have been interested.
3. **PunctualDataresult**: this object is a container for a SQL type (different from ResulSet) that can be returned from a store procedure.
4. **CompositeDataResult:** this object is a container for other result object and it is necessary because the execution of store procedures can produce multiple return values of different type. For example, if a store procedure returns an int value and a resultSet the *CompositeDataResult* will contain one *PunctualDataResult* and one *ScrollableDataResult*.

Code Example 2-5: composite dataResult

```
// Open the connection and build the object for
// the invocation of the PL/SQL procedure
DataConnection conn = DataConnectionManager.getConnection();
String storeProcedureString = "{ call " + getPackageName() + ".get_all(?,?,?,?) }";
SQLCommand command = conn.createStoredProcedureCommand(storeProcedureString);

int paramIndex = 0;

// create an array of null parameters
ArrayList parameters = new ArrayList(4);
parameters.add(conn.createDataField("NumPerPage",java.sql.Types.INTEGER, null));
((SroredProcedureCommand)command).setAsInputParameters(paramIndex++);
parameters.add(conn.createDataField("Page",java.sql.Types.INTEGER, null));
((SroredProcedureCommand)command).setAsInputParameters(paramIndex++);

// create the param for store the number of read records
parameters.add(conn.createDataField("RecordCount",java.sql.Types.INTEGER, null));
((SroredProcedureCommand)command).setAsOutputParameters(paramIndex++);

parameters.add(conn.createDataField("ObjRecordset",java.sql.Types.CLOB, null));
((SroredProcedureCommand)command).setAsOutputParameters(paramIndex++);

// call the store procedure
DataResult result = command.execute(parameters);

// Retrieve the CLOB of the result
if ((result.getDataResultType()).equals(DataResultInterface.COMPOSITE_DATA_RESULT))
CompositeDataResult resultInterface =(CompositeDataResult)result.getDataObject();
List outputParams = resultInterface.getContainedDataResult();
```

```
PunctualDataResult pdr =(PunctualDataResult)outputParams.get(1);
DataField df = pdr.getPunctualDatafield();
Clob resultClob = (Clob)df.getObjectValue();
xmlString = resultClob.getSubString(1L,(int)resultClob.length());
}
```

3.14.7 Transaction management

The transaction management is implemented in a different way if the business logic is executed on a web container or on an EJB container.

3.14.7.1Business logic on Web Container

In the Web Container environment all the database operations, that are not part of one transaction, are considered with autocommit enabled, respecting the default modality defined by JDBC 2.0. Developers who wants to insert database operations into a transaction must use methods ***initTransaction***, ***commitTransaction*** e ***rollBackTransaction*** declared by the *it.eng.spago.dbaccess.sql.**DataConnection*** class. This mechanism not cover distributed transactions so the operations are part of the same transaction only if they use the same JDBC connection. In case of an error occurs during the execution of one operation the Spago Framework won't perform no one commit or rollback commands because this job is delegated to the developers..

3.14.7.2Business logic on EJB Container

In the EJB Container environment the transactions are managed from the Transaction Manager of the EJB Container. The methods *initTransaction*, *commitTransaction* and *rollBackTransaction* of the DataConnection object have no utility because the correspondent operations are executed by the Framework with appropriate calls to the EJB Container.
Spago for each service request start a new distributed transaction, so all the service used for reply to the request are part of the same transaction event if they use different connection. The commit operation is executed automatically at the end of the request elaboration if no errors are occurred, otherwise the framework call a rollback instruction. Considering that the transaction is distributed the commit and rollback instructions have effect on all connections.
This mechanism is based on EJB Containers that implement the distributed transaction protocol *"two phase commit"* using objects compliant to the JTA XA interface.

3.14.8Release of the resources

The policy for release resources is the same of that one define by JDBC: release resources in inverse order of allocation. For example, after a SELECT operation, the JDBC objects used must be release in this order: Cursor, Statement and finally the Connection. In order to simply the release process Spago implements a method that perform this task automatically with the correct order.

```
Utils.releaseResources(dataConnection, sqlCommand, dataResult);
```

3.15Hibernate

Spago can be integrated with Hibernate and using it like a module for object/relational persistence. Hibernate allows to abstract a database like an objects model. Every table of the relational schema is represented by an object which is build starting from specific mapping configuration files. Hibernate can abstract different databases only changing some parameters inside its configuration, so the application become portable and developers have only to interact with Hibernate API.

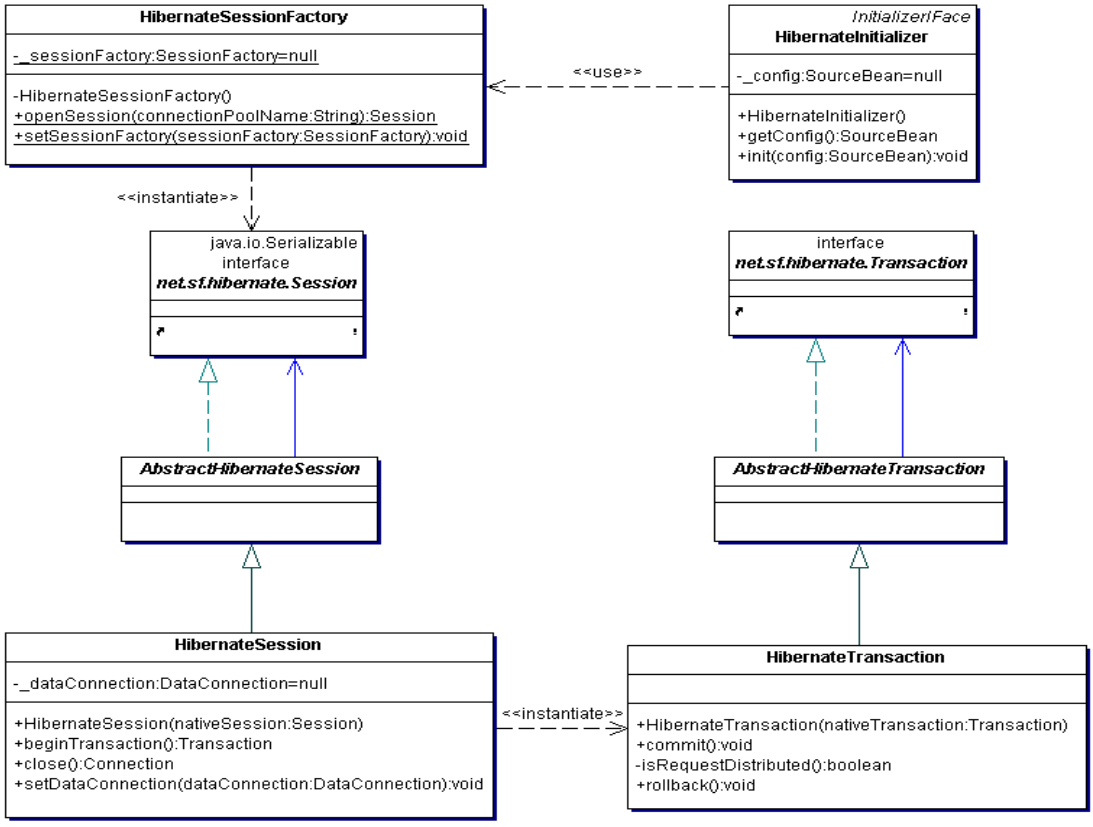
Write the mapping files can be a long activity so developers can use some open source tools that create the mapping files and the relative java class starting from the database definition. The explanation of hibernate or the various tool is not an objective of this documents, readers who wants other information can look at the Hibernate Site http://www.hibernate.org/.

The major problem to solve during the integration process was:

- integrate Hibernate with the Spago connection pool logic.
- Integrate Hibernate with the Spago transaction management

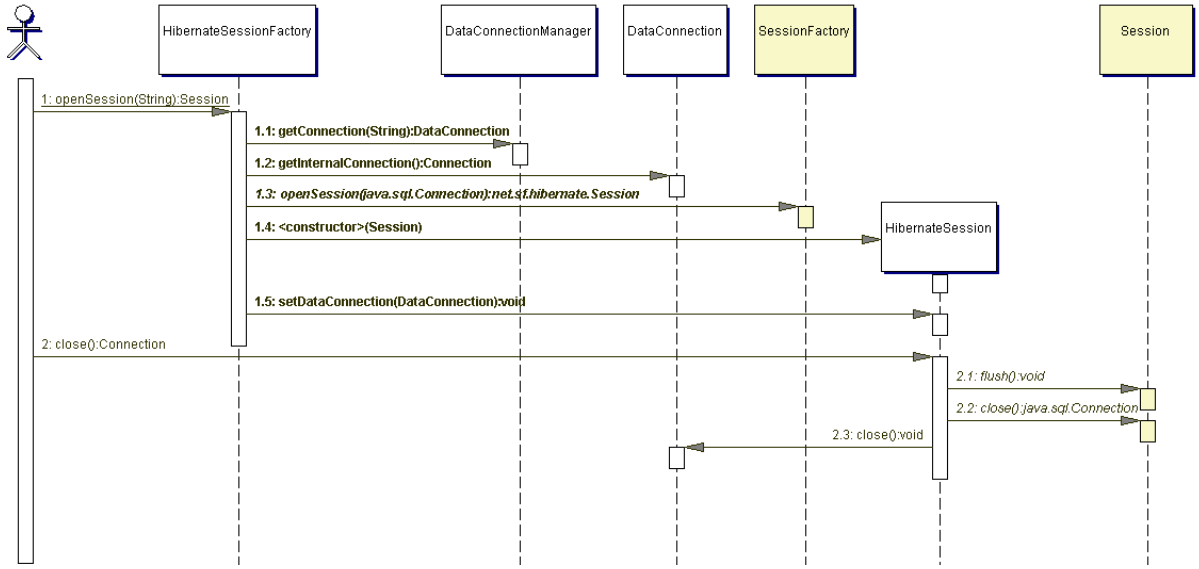
3.15.1Hibernate Integration

The Hibernate base object is Session, which represent a connection to the repository managed, so, in order to integrate it, Spago needs a mechanism to get an hibernate Session from a connection pool defined in the configuration file *data_access.xml*. This mechanism is implemented by a set of classes with this architecture:

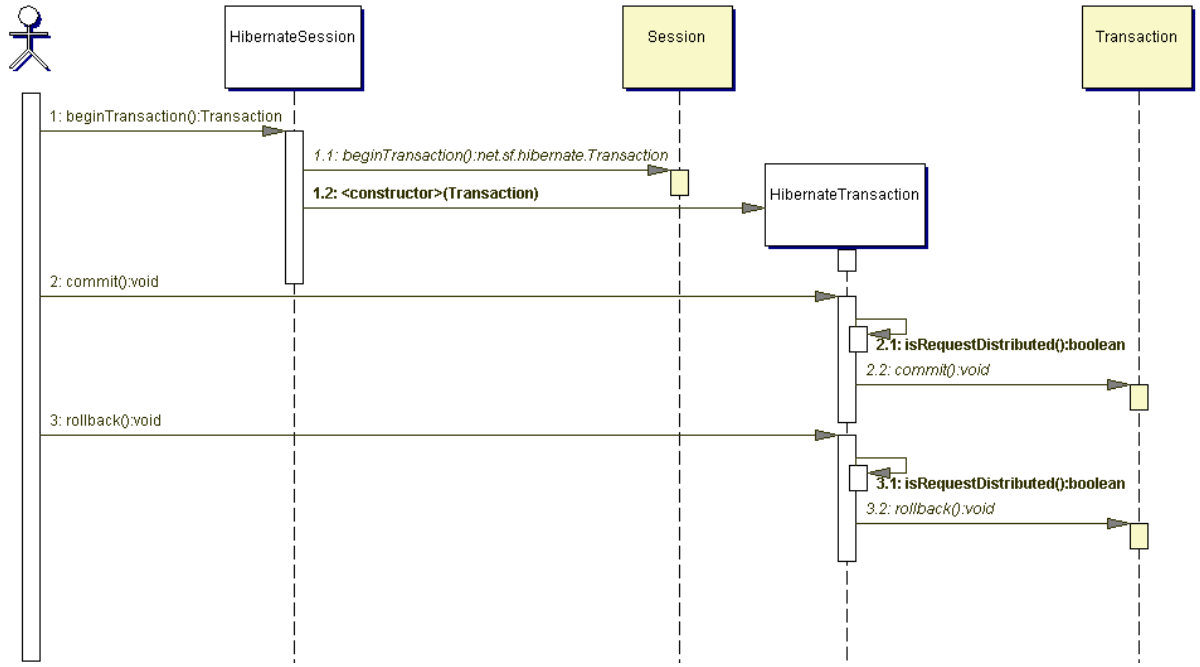


Spago builds an hibernate session factory at the application start – up time, so inside the configuration file must be present an xml envelop that load the class *it.eng.spago.dbaccess.hibernate.**HibernateInitializer.*** This class initialize an Hibernate configuration object and set it inside an object *it.eng.spago.dbaccess.hibernate.**HibernateSessionFactory*** that represent a session factory for the hibernate module. HibernateSessionFactory implements a method *public static Session openSession(String connectionPoolName)* that, starting from the name of one connection pool defined in *data_access.xml,* returns an ***HibernateSession*** object. HibernateSession is a class of spago that contains a native hibernate Session and implements all the methods necessary to interact with an hibernate repository.

The following UML sequence diagram show the interaction between Spago objects and Hibernate objects (***SessionFactory*** and ***Session***) for open and close sessions.



Hibernate has a proprietary JTA transactions manager but this mechanism can't be used with distributed transactions that involves more than one hibernate session. To solve the problem it has been necessary to implement two *decorator,* one for the object hibernate Session and the other for the object hibernate Transaction. These objects intercept all the request *beginTransaction*, *commit* and *rollback* and, if the transaction is not distributed, they redirect them to the native hibernate Transaction, or otherwise they ignore the request because the transaction management is perform from the framework and must be transparent to the user. The second case happens when the application is executed in a EJB container environment with transactions management because in this scenery the transactions are distributed and always managed by the framework.



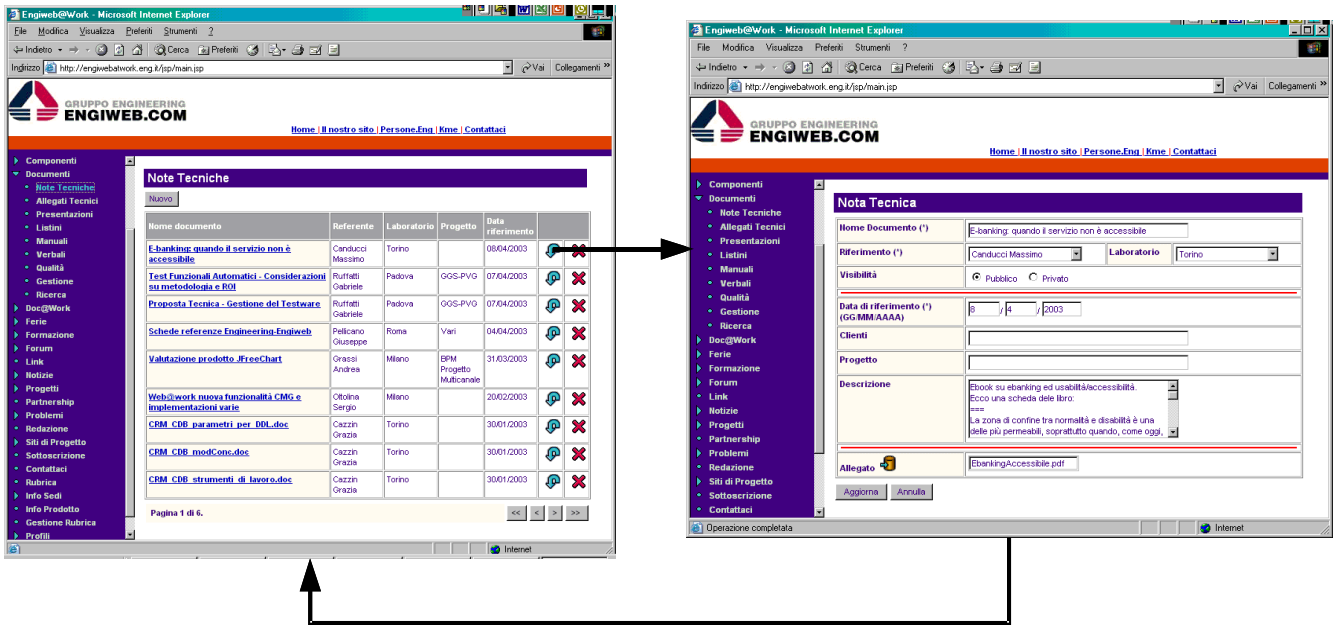
3.16Navigation

The navigator is one of the various transversal services, supplied from Spago, that allows to store the application requests in a stack and to submit them again in every moment Using this functionality it is possible to construct a navigation service more powerful that the classic browser navigation, because this one reestablish automatically all the request parameters and also the server session.

Suppose, for example, to build an application that shows a list of elements and that each one of them can be selected for view details. With Spago this simple program can be implemented using an action that retrieves all the elements of the list, and a JSP that shows the list. Each element is a link that allows to call a new action for retrieve details to show in a new page.

The First actions needs some parameters passed with the request. When a user needs to return back from the detail page to the list, the action has to be executed again with the same parameters. Spago permits to do this operation simply adding a NAVIGATION_BACK_TO_MARK=1 parameter.

Navigator.signAction(getRequestContainer());



The action, that build the list, ask to the navigator to store the current request and session using the method *signAction.* After, when a user needs to return to the list page from the detail page, the application ask to the navigator (using a *NAVIGATOR_BACK_TO_MARK=1 parameter)* to execute again the action with the request and session stored in the stack.

The navigator can be activated setting in the request URL some variable "*NAVIGATOR_XXX*" where the "XXX" string is the name of the command to exec:

- NAVIGATOR_RELOAD: execute again the last service, the same page is reload.
- NAVIGATOR_BACK=?:return back of a number of pages (and relative services) equals to the value of the parameter.

- NAVIGATOR_FORCEBACK=?: returns back of a number of pages (and relative services) equals to the value of the parameter, considering also the pages (or better the relative services) not stored in the navigation history (see navigator_disabled below).
- NAVIGATOR_BACK_TO=?: returns back to the service (page or action) with the name equals to the parameter value returning back to the first called service.
- NAVIGATOR_BACK_TO_MARK=?: returns back of a number of services (and relative pages) equals to the value of the parameter, counting only the services marked. Each request can be mark using Navigator class method

  *public static void **signService**(RequestContainer request).*
- NAVIGATOR_RESET: reset the navigation history. This command when executed erases all the data stored inside session and, in some case, this is an unwanted behaviour. To solve this issue the framework implements a container, call permanent container, accessible from the session container, which is not affect by this problem and isn't erased after a navigation reset:

  *public synchronized SessionContainer **getPermanentContainer**();*
- NAVIGATOR_DISABLED=TRUE: the service call is not stored in the navigation history. This command is useful for select which services (and relative pages) are part of the navigation chain. A typical example of use is with a multiframe site that can refresh more than one frame at the same time. In this case the server has to create a number of threads equal to the number of requests that can be executed in a "casual" order, so the navigation history can be different from which expected by developers
- NAVIGATOR_BACK_TO_SERVICE=? : returns back to the service (and the relative page) identified by the label specified. Developers can assign a label to a request using the Navigator class method:

  *public static void **signService**(RequestContainer request, String serviceLabel)*
- NAVIGATOR_FREEZE="TRUE": permits to register the request without create a new item in the stack but replacing the previous.
- NAVIGATOR_BACK_TO_SERVICE_LABEL="LABEL": returns back to the service (and the relative page) identified by the label specified. Developers can assign a label to a request using the Navigator class method:

  *public static void **signService**(RequestContainer request, String serviceLabel)*

It is possible to manage errors caused by the navigation system recording a new Action with a fixed name NAVIGATION_ERROR_ACTION.

The use of the navigator has a collateral effect: the request and session stack maintenance is an heavy activity. All the data contained in the session is serialized and stored like XML nodes inside HTTP session, for the http channel, or like an attribute value for the classes AdapterSoap and AdapterEJB, for the SOAP and EJB channel. The serialized data can be useful also for mechanisms of load balancing and failure recovery on a cluster server.

For the IBM philosophy the session hasn't to contain more than 4kbyte to serialize/deserialize, it has to contain only the name of the objects to retrieve. Developers can follow this advise turning off the navigation system simply setting to false the attribute *navigator_enabled* of the configuration file *common.xml* .

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<COMMON
   file_uri_prefix="file:"
```

```
    xhtml_lat1_ent="/WEB-INF/conf/xhtml-lat1.ent"
    xsl_reload="TRUE"
    navigator_enabled="TRUE | FALSE">
</COMMON>
```

Into the AdapterHTTP class it has been added the possibility to calculate the dimension of the session data and write it in the log file. This operation verify at the same time if some objects contained are not serializable and in this case an exception is raised. The functionality can be able / disable simply setting to TRUE / FALSE the value of the attribute *serialize_session* present in the configuration file *common.xml:*

```
<COMMON
        .....
        serialize_session="TRUE" >
</COMMON>
```

The default value is FALSE. The functionality can be very useful to control if all the session objects are serializable (before store them in a database, for example), but It is advised to disable it when the application is not used for test or debug because the serialization of the session is an heavy operation.

3.16.1Navigation Toolbar

Using the system navigation, and specially his stack of service requests, it has been possible to implement a navigation toolbar that allows to show the navigation path follow and to return back to one of the step visualized.
The two custom tags:

- *it.eng.spago.tags.DefaultNavigationToolbarTag*
- *it.eng.spago.tags.ComboNavigationToolbarTag*

can be used for rendering (with to different modalities) the navigation toolbar. The first tag shows a path of links composed by the service names, while the second show the same service names inside an html combo element. The only services rendering are those mark with the call to the method:

- *public static void signService(RequestContainer request, String serviceLabel)*

of the Navigator class, where “serviceLabel” is the label showed in the navigation toolbar.

3.17Pagination

The pagination functionality is a service that permits to split a list of elements in blocks of a fixed dimension ad to show each block in a chain of pages. One page rendering all the elements of a block with a table and allows the user to select one item to view its detail page. This mechanism needs a Paginator object that permits to get the block of elements correspondent to one page.
The framework allows to define the logic for manage list / detail pages simply configuring and using a specific module. Using this option developers haven't to worry about the Paginator and how the data is retrieve, but, in some case, this automation is impossible, for example if the data is stored in an LDAP server. In this case developers must write the code to recover the data and put it inside the Paginator object.
The Paginator is a container for a collection of objects that acts like a buffer between the retrieve and presentation operations. It implements methods to add data dynamically so developers can perform different operations to recover information and put all the results inside it. The presentation logic will get information from the paginator and will show them in the appropriate way. The paginator object has two implementation:

- *One-shot*: the simpler implementation that require less work for the programmer. The entire collection of elements to present is recover only one time, the pagination of the data is perform during the rendering of the list not during the retrieve operation. This solution is easy but can suffer two types of problem: the initial time to retrieve all the elements (that can be considerable if the list is long) and the memory occupation (all the data is retrieves one time and maintained in memory).
- *With-cache*: with this modality the pagination is perform during the retrieve operation, managing a "window" on the result data to show. The elements are not recover all in an only moment but each request of a new page of the list involves a new operation for recover only the elements to show in the new page. This solution reduces the problem of execution time and memory allocation but is more complex to use.

3.17.1One-shot

The pagination subsystem is composed from some base interfaces and default implementations:

- *it.eng.spago.paginator.basic.**PaginatorIFace***: interface that defines all functionalities of one paginator. Declares all service necessary to manage the contents of a paginated list.
  - *void addRow(Object row)*: add a new row to the list of objects managed from the paginator (obviously a new row is a new object).
  - *int getPageSize()*: returns the dimension of the page which is the number of elements visualized in a single page.
  - *void setPageSize(int size)*: set the dimension of the page.
  - *int rows()*: returns the total number of the paginator rows (the number of objects contained).
  - *int pages()* : returns the number of pages necessary to show all the paginator elements.
  - *SourceBean getPage(int page)* : returns all the elements (serialized in a SourceBean) correspondent to the page index specified.
  - *SourceBean getAll()*: returns all the elements of the paginator (serialized in a SourceBean).

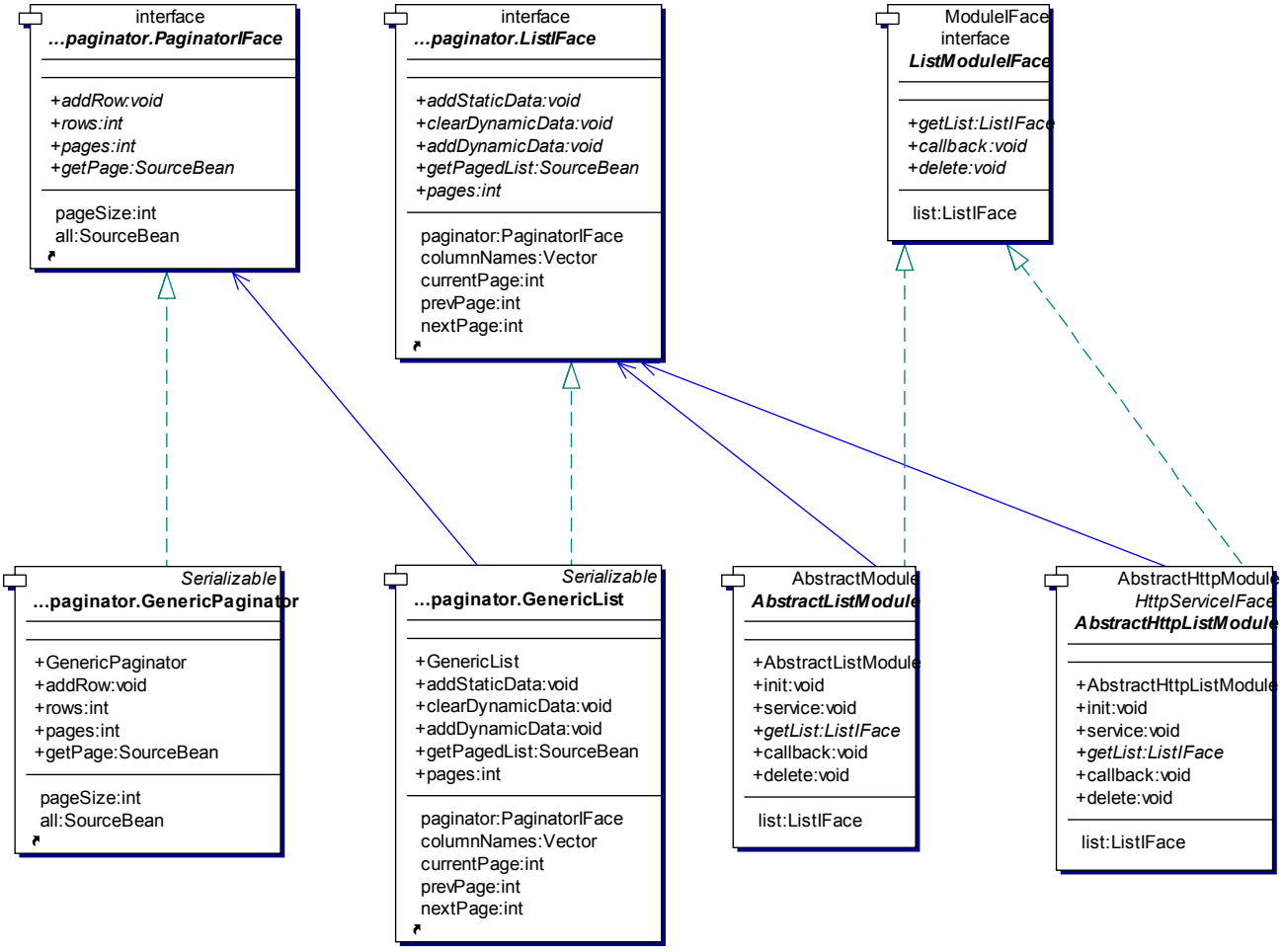
  The framework class *it.eng.spago.paginator.basic.impl.**GenericPaginator*** is an implementation of this interface.

- *it.eng.spago.paginator.basic.**ListIFace**:* interface that defines the necessary methods to navigate the list of pages, to retrieve elements of one page and to dynamically add data during the presentation phase.
  - *Int getCurrentPage():* returns the index of the current page.
  - *void setCurrentPage(int page):* set the index of the current page.
  - *int getPrevPage():* returns the index of the previous page.
  - *int getNextPage():* returns the index of the next page.
  - *int pages():* returns the total number of pages.
  - *SourceBean getPagedList(int i):* returns the elements (serialized in a sourceBean object) correspondents to a page index.
  - *void addDynamicData(SourceBean data):*
  - *void clearDynamicData():*
  - *void addStaticData(SourceBean data):* insert static information

  The framework class *it.eng.spago.paginator.basic.impl.**GenericList*** is an implementation of this interface**.** This object uses an instace of the GenericPaginator for manage the pagination.
- *it.eng.spago.dispatching.service.list.basic.IFaceBasicListService*: interface that defines a module services for the management of a paginated list:
  - *public ListIFace **getList()***: retrieves a ListIFace object that can be used for rendering the elements blocks and navigate the chain of pages.
  - *public void **callback(SourceBean request, ListIFace list, int page)***: this method is call every time the pagination service is invoked during the navigation of the presentation pages, so every time a user changes page. In this method is possible to add some information useful for the rendering activity.
  - *public void **delete**(SourceBean request)*: delete an element from the list.

  The framework classes *it.eng.spago.dispatching.module.list.basic.**AbstractBasicListModule*** and *it.eng.spago.dispatching.module.**AbstractHttpListModule*** are implementation of this interface. The class *AbstractHttpListModule* is a variant of *AbstractListModule* for access to the HTTP context. However these class are abstract and implement only, with an empty method, the two services *callback* and *delete.* The remaining service *getList()* must be implemented by the developers.

  The *getList()* service is called only one time (or all the times that is necessary a list refresh which is request with an explicit command) and the data recovered can't be modified. Every time a user changes page the *getList()* is not recalled, otherwise the *callback* service is called.



Developers have two possibilities to use the pagination subsystem:

1. The simpler possibility is to use the framework functionalities for automatic generation of list and detail pages, recovering data from a database. This mechanism is explained in the paragraph Automatic Generation of list/detail pages.

2. The other modality is to create a new module that extends the abstract class *AbstractListModule* or *AbstractHttpListModule*. This module has to implement the method *public ListIFace getList(SourceBean request)* which has to instantiate a *GenericPaginator,* for add new element to the list, and a *GenericList* to which associate the Paginator. The module *it.eng.spago.dispatching.module.list.basic.impl.**DefaultBasicListModule*** implements an example of this activities for the automatic generation of list / detail pages. A part of this module is showed in the Code Example 2-6. The module will be used by a JSP containing all the necessary logic for rendering the elements of the list. The framework provides some tags, to use inside the JSP, for perform automatically the rendering operation, but they are explained in details in the paragraph Automatic Generation of list / detail pages.

Code Example 2-6: creating a new list (DefaultBasicListModule)

```
// creation of the paginator
PaginatorIFace paginator = new GenericPaginator();
paginator.setPageSize(<number of rows for each page>);

// Add all rows to the paginator
for (int i = 0; i < <number of rows>; i++)
   paginator.addRow(<i - row>);

// creation of the list
ListIFace list = new GenericList();
// Association list-paginator
list.setPaginator(paginator);
list.setColumnNames(<vector containing the columns name>);
return list;
```

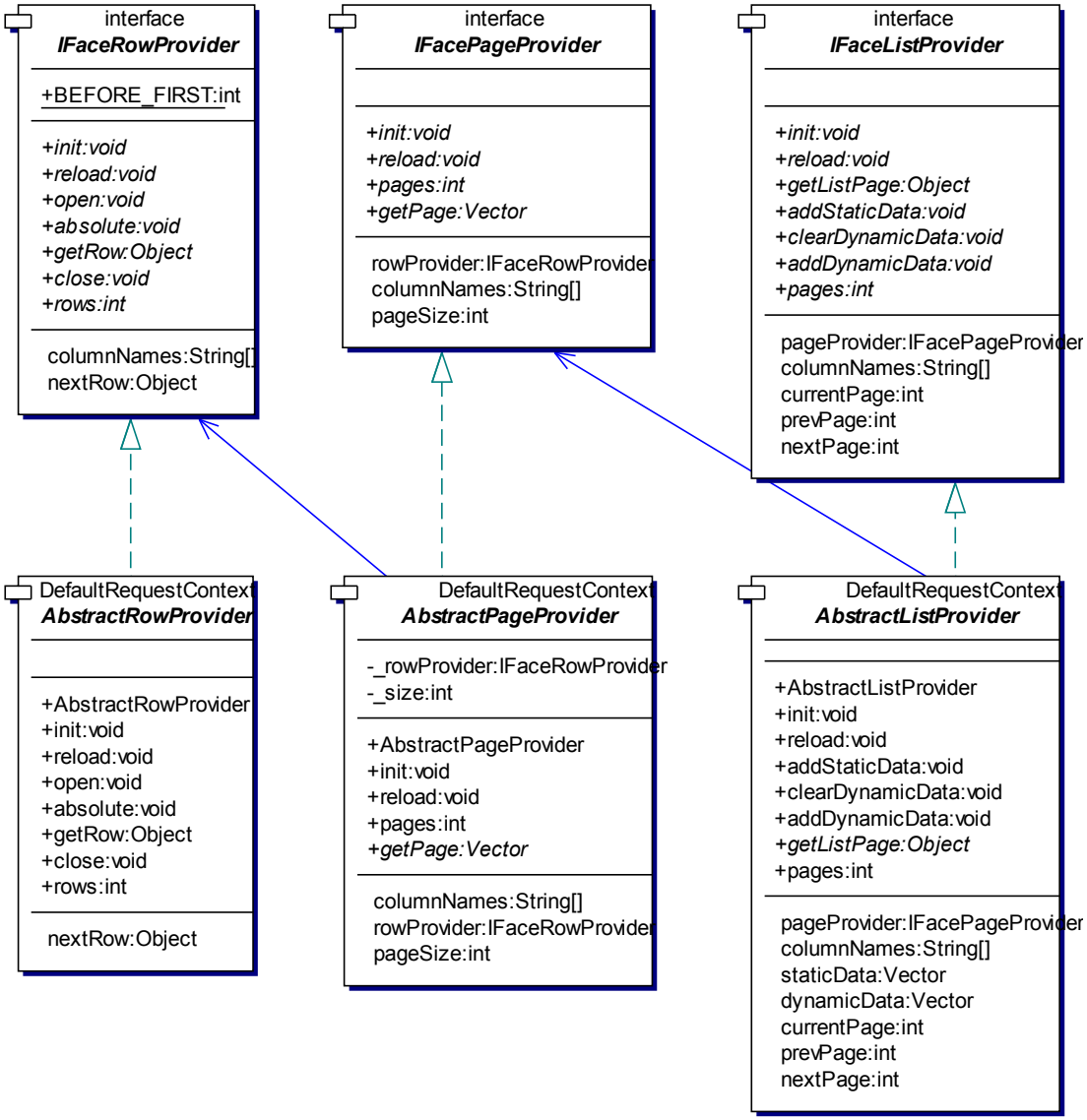
3.17.2With - cache

The one shot paginator is easy to use but has an important limit: it retrieves the entire list of elements and put it in memory, with evident problems of time execution (if the list is too long) and memory allocation. To solve this problems, and trying to implement a mechanism for retrieve data from any datasource, the framework provide a new paginator. The necessity is to recover data from database, LDAP server or others data sources, simply modifying some configuration parameters. The pagination subsystem is composed from some interfaces that define the methods for the pagination functionalities:

- *IFaceRowProvider*: defines the method for retrieve a row of the list (*getRow*).
- *IFacePageProvider*: defines all services for page management. A page is a “window” or a subset of the entire collection of elements. it uses the interface *IFaceRowProvider* for recover rows of the list. Objects that implements this interface have to define chaching alghoritms more efficient than the total cache.
- *IFaceListProvider*: defines all services for list management. Methods exposed allow to retrieve “pages” of data and reply to the callback (send when users change page), with the possibility to add dynamic and static data useful for the rendering phase, like the column labels, for example.

The difference regarding the one-shot paginator is that this system perform a data page split operation during the data recover phase. The paginator never retrieves all the list elements, but only those that have to be show in the list page selected by the user. Implementation, every time a page change is requested, executes the necessary operation to retrieve all the elements of the list from the datasource (so, for example, if the datasource is a database, the system executes the SELECT query to recover all data), but, after, executes a filter operation to extract only the items necessary to the page requested.

This mechanism not reduce the time to recover data from the datasource, but considerably diminishes the transfer time to the client and the memory allocation, because it maintains only a subset of the items.



The principal limit of this mechanism is that currently it is able to retrieve data only from a database, so the implementations provided by the framework not cover all the initial requirements. The framework implementation class are:

- *DBRowProvider*: allows to recover data from the database using the configuration into the <ROW_PROVIDER> envelope of the file *actions.xml,* which define how to retrieve the SQL statements to execute in order to extract information. Two modalities exist for recover the statement:
  - From the file *statements.xml,* using the logical name defined by the attribute "statement" of the <*_QUERY> envelopes contained inside the tag <CONFIG>, as the following example shows:

```
<ROW_PROVIDER class="it.eng.spago.paginator.smart.impl.DBRowHandler">
        <CONFIG pool="afdemo">
                <LIST_QUERY statement="LIST_USERS" ></LIST_QUERY>
                <DELETE_QUERY statement="DELETE_USER">
                        <PARAMETER type="RELATIVE" value="userid" scope="SERVICE_REQUEST" />
                </DELETE_QUERY>
        </CONFIG>
</ROW_PROVIDER>
```

  - Delegating the task to a specific class, which implements the IfaceQueryProvider interface, the dynamic construction of the statement. The name of the class is defined by the attribute "class" of the <*_QUERY> tags, as the following example shows:

```
<ROW_PROVIDER  class="it.eng.spago.paginator.smart.impl.DBRowHandler">
        <CONFIG pool="afdemo">
                <LIST_QUERY statement=""
                        class="it.eng.spago.query.UtentiQueryProvider">
                </LIST_QUERY>
                <DELETE_QUERY statement="DELETE_UTENTE">
                        <PARAMETER type="RELATIVE" value="userid"
                                scope="SERVICE_REQUEST" />
                </DELETE_QUERY>
        </CONFIG>
</ROW_PROVIDER>
```

    Obviously the statement attribute mustn't be valorized.

- *CacheablePageProvider*: implements the paginator functionalities and maintains in memory the data of some "lateral" pages of the current one, assuming that the user will navigate in the next and previous pages (it is a simple predictive mechanism). It is possible to configure the number of lateral pages to maintain in memory. Currently the page loading is contextual to the user click on the navigation buttons or elements, so if the user navigates to a page already contained into the paginator, this will return it, otherwise the paginator retrieve the data of page and its lateral from the database. We are working for implements an asynchronous load mechanism, retrieving data of the lateral page when the system notices that the navigation is going out of the "window" memory.

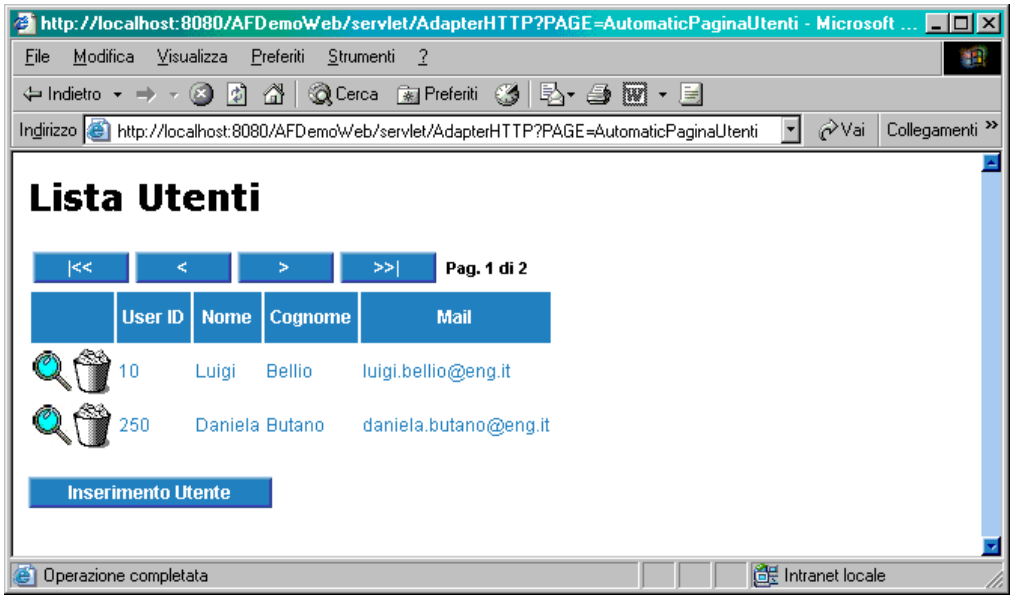
- *DefaultListPRovider*: implements the list functionalities in order to slide pages and to add static or dynamic information to the page data. This class has an own configuration file where it reads the necessary instructions for build the column labels.

An example of this default implementation can be found into the section Automatic Generation of list / detail (with - cache).

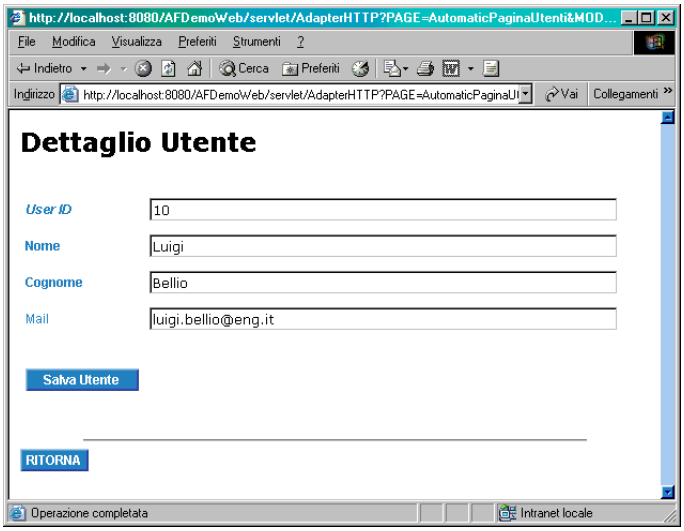
3.18Automatic Generation of list / detail pages (one - shot)

A common application requirement is to manage a set of elements with a paginated list. Developers wants to retrieve data from a database and rendering fixed blocks of items with a table. Each row is correspondent to an element and allow the user to access the detail page associated or to cancel the self item. Different blocks of elements are rendered in different pages, and each one of these allows users to navigate to the previous or next. The last important function for a list page is to insert a new element.
Spago framework implements an automatically mechanism to build a paginated list, developers have only to configure the necessary parameters, like the items retrieve query, the number of rows for page and others, inside xml configuration files. The rendering operation can be delegated to some implemented tags that have only to be declared inside a JSP, but developers can easily write new tags in order to personalize the data visualization .
The framework can build a paginated list using modules or indifferently actions modality, but this paragraph describes an example of paged list build with one-shot paginator, using only Spago Modules and tag libraries. Readers have to know that the same task can be executed in a similar way using actions (see Spago example).

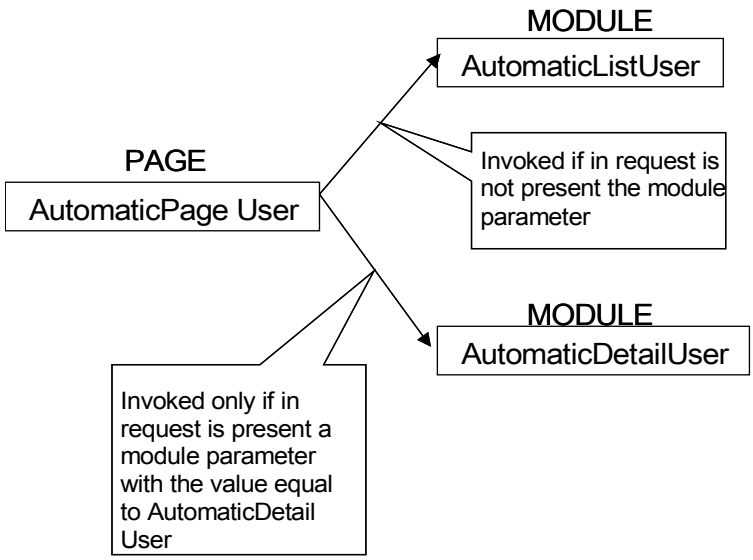
This example use the modules *it.eng.spago.dispatching.module.list.basic.impl.**DefaultBasicListModule,*** for the list generation, and *it.eng.spago.dispatching.module.detail.impl.**DefaultDetailModule*** for the detail generation. These classes implement the functionalities of the one – shot paginator. With some configuration parameters these modules, and the rendering tag libraries implemented, can retrieve items from a database and show them in a paged list like this:



Like the image shows, the page renders the elements with a table, each rows correspond to one items. The page has some buttons that permit to move to next / previous / last / first page and to insert a new element. Each rows have a button, drawn like a lens, that redirect to the correspondent item detail page, showed below.



For the following example it is used an only page to visualize the list and the detail. It is configured as:



3.18.1List

For this example suppose to retrieve data from a database table structured like this:

Columns of UTENTE

Name	Type	Nullable	Default	Comments
ID_USER	NUMBER(10)			
COGNOME	VARCHAR2(30)			
NOME	VARCHAR2(30)			
EMAIL	VARCHAR2(100)	Y		

Developers have to configure the module for the list construction in the file *modules.xml*:

```
<MODULE
  name="AutomaticListaUtenti"
  class="it.eng.spago.dispatching.module.list.basic.impl.DefaultListModule">
  <CONFIG pool="afdemo" title="Lista Utenti" rows="2">
    <QUERIES>
      <SELECT_QUERY statement="LIST_USER"/>
      <DELETE_QUERY statement="DELETE_USER">
        <PARAMETER type="RELATIVE" value="id_user" scope="SERVICE_REQUEST"/>
      </DELETE_QUERY>
    </QUERIES>

    <COLUMNS>
      <COLUMN name="id_user" label="User ID"/>
      <COLUMN name="nome" label="Nome"/>
      <COLUMN name="cognome" label="Cognome"/>
      <COLUMN name="email" label="Mail"/>
    </COLUMNS>
    <CAPTIONS>
      <SELECT_CAPTION image="" label="Detail User" confirm="FALSE">
        <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPageUser" scope=""/>
        <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDetailUser" scope=""/>
        <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
      </SELECT_CAPTION>
      <DELETE_CAPTION image="" label="Erase User" confirm="TRUE">
        <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
      </DELETE_CAPTION>
    </CAPTIONS>

    <BUTTONS>
      <INSERT_BUTTON image="" label="Insert User" confirm="FALSE">
        <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPageUser" scope=""/>
        <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDetailUser" scope=""/>
```

```
        </INSERT_BUTTON>
      </BUTTONS>
    </CONFIG>
  </MODULE>
```

The module configuration contains a CONFIG session that define the following attributes:

- *pool*: the name of the connection pool to use for retrieve the connection to the database. For an explanation of how configure the connection pool take a look to the Data Access paragraph.
- *title*: title of the list

- *rows*: number of rows for each page.

The CONFIG session contains also the following xml envelops:

- QUERIES: contains the names of the statements (defined in the file *statements.xml*) for the recover of the entire set of elements (*SELECT_QUERY*), and for the deletion of one list item (*DELETE_QUERY)*.

```
<QUERIES>
  <SELECT_QUERY statement="LIST_USER"/>
  <DELETE_QUERY statement="DELETE_USER">
    <PARAMETER type="RELATIVE" value="id_user" scope="SERVICE_REQUEST"/>
  </DELETE_QUERY>
</QUERIES>
```

  In this case, in the file *statements.xml,* the queries are defined like*:*

```
<STATEMENT name="LISTA_UTENTI"
        query="SELECT id_user,nome,cognome,email FROM UTENTE ORDER BY id_user"/>
<STATEMENT name="DELETE_UTENTE"
        query="DELETE FROM UTENTE WHERE id_user = ?"/>
```

  In case of the queries need some placeholder, like for example the deletion query, it is possible to retrieve the parameters from the request, the session or the application scope, with the same grammar used for define the module conditions (see Conditions paragraph). The queries are used from the select and delete captions defined in the *CAPTIONS* envelop (explain below).

- COLUMNS: contains the list columns definition. For each column is define the name of the database table column (*name*) and the correspondent label to show in the page (*label*).

```
<COLUMNS>
  <COLUMN name="id_user" label="User ID"/>
  <COLUMN name="nome" label="Nome"/>
  <COLUMN name="cognome" label="Cognome"/>
  <COLUMN name="email" label="Mail"/>
</COLUMNS>
```

The query that retrieve database data can return a resultset with more database columns than those showed in the page. These hide information can be used for different scope like unique identifier for each elements or condition control.

- CAPTIONS: contains the configuration for the buttons that allow to cancel an element of the list (*DELETE_CAPTION*) or to access to an item detail page (*SELECT_CAPTION*).

```
<CAPTIONS>
  <SELECT_CAPTION image="" label="Dettaglio Utente" confirm="FALSE">
    <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPageUser" scope=""/>
    <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDetailUser" scope=""/>
    <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
  </SELECT_CAPTION>
  <DELETE_CAPTION image="" label="Erase User" confirm="TRUE">
    <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
  </DELETE_CAPTION>
</CAPTIONS>
```

The default possible captions are:

- SELECT_CAPTION: button used to access the detail page of one element. The *PAGE* parameter defines the page that execute all the operations for recover and redirect to the JSP the detail information. The MODULE parameter define which module of the page has to be executed. All the parameters necessary to the detail page can be specified inside new parameter envelope (like id_user for example).
- DELETE_CAPTION: button used to delete a row of the table or better an element of the list. The operation recalls the self-module performing the *DELETE*_QUERY defined in the QUERY envelop. It is necessary to specify all parameters to replace the query's placeholders. The value for this parameter can be recovered from columns (also hidden columns) of the table list. For this button is applied the prevent resubmit for default but it is possible to change this behaviour setting the attribute prevent_resubmit to "FALSE"

It is possible to add a specific caption using the follow syntax (obviously inside the *CAPTIONS* envelop):

```
<CAPTION image="URL IMAGE" label="LABEL" confirm="TRUE | FALSE">
  <PARAMETER name="PAGE" type="ABSOLUTE" value="PAGE NAME " scope=""/>
  <PARAMETER name="MODULE" type="ABSOLUTE" value=" MODULE NAME" scope=""/>
  PARAMS LIST
</CAPTION>
```

The external envelope has tree attributes:

- image: the URL of the image to show in place of the classical button.
- label: the label of the caption. (the button label)
- confirm: contains TRUE or FALSE value, if true the system ask to the user to confirm the operation execution with a *javascript confirm,* if false the operation is executed directly.

Inside it is possible to define some parameters, like the name of the page that implement the business logic associated to the caption, the name of the module to execute (a page can contain more than one module so it is necessary to specify which to use), and all other parameters useful for the caption functionality execution.

- BUTTONS: contains the configuration of the buttons to show under the list.

```
<BUTTONS>
  <INSERT_BUTTON image="" label="Insert User" confirm="FALSE">
    <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPageUser" scope=""/>
    <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDetailUser" scope=""/>
  </INSERT_BUTTON>
</BUTTONS>
```

The possible buttons are:

- INSERT_BUTTON: button for insert a new row or better a new element into the list.

It is possible to add new specific button using the follow syntax (obviously inside the BUTTONS envelope) :

```
< BUTTON image="URL IMAGE" label="LABEL" confirm="TRUE | FALSE">
  <PARAMETER name="PAGE" type="ABSOLUTE" value="PAGE NAME" scope=""/>
  <PARAMETER name="MODULE" type="ABSOLUTE" value="MODULE NAME" scope=""/>
  LIST PARAMETERS
</BUTTON>
```

The external envelop has tree attributes:

- image: the URL of the image to show in place of the classical button.
- label: the label of the caption. (the button label)
- confirm: contains TRUE or FALSE value, if true the system ask to the user to confirm the operation execution with a *javascript confirm,* if false the operation is executed directly.

Inside it is possible to define some parameters, like the name of the page that implement the business logic associated to the button, the name of the module to execute (a page can contain more than one module so it is necessary to specify which to use), and all other parameters useful for the execution. Of the button functionality.

3.18.2List Commands

It is possible to send some messages to the module list inserting some parameters into the request:

- Previous page: in the request must be inserted the parameter ***MESSAGE=LIST_PREV***.
- Next page: in the request must be inserted the parameter ***MESSAGE=LIST_NEXT***.
- First page: in the request must be inserted the parameter ***MESSAGE=LIST_FIRST***.
- Last page: in the request must be inserted the parameter ***MESSAGE=LIST_LAST***.
- Page with index i: in the request must be inserted the parameter ***MESSAGE=LIST_PAGE&LIST_PAGE=i*** , the *i* value is the index of the page to show.
- Reload the list: in the request must be inserted the parameter ***LIST_NOCACHE=TRUE***. A typical case that require this message can be when the list is a search result dependent from some search parameters. Every time the user changes the search parameters also the list change so it must be refreshed, but the request hasn't to be register as a new element in the navigation history.

The first four messages are used for the default navigation toolbar, the fifth can be used to implement a “go to page” functionality and the last is used to refresh the list.

3.18.3Detail

This paragraph analyse the configuration of the detail module *AutomaticDetailUser* contained into the *modules.xml* file.

```
<MODULE
   name="AutomaticDetailUser"
   class="it.eng.spago.dispatching.module.detail.impl.DefaultDetailModule">
   <CONFIG pool="afdemo" title="Detail User">
      <QUERIES>
         <INSERT_QUERY statement="INSERT_USER">
            <PARAMETER type="RELATIVE" value="id_user" scope="SERVICE_REQUEST"/>
            <PARAMETER type="RELATIVE" value="nome" scope="SERVICE_REQUEST"/>
            <PARAMETER type="RELATIVE" value="cognome" scope="SERVICE_REQUEST"/>
            <PARAMETER type="RELATIVE" value="email" scope="SERVICE_REQUEST"/>
         </INSERT_QUERY>
         <SELECT_QUERY statement="SELECT_USER">
            <PARAMETER type="RELATIVE" value="id_user" scope="SERVICE_REQUEST"/>
         </SELECT_QUERY>
         <UPDATE_QUERY statement="UPDATE_USER">
            <PARAMETER type="RELATIVE" value="nome" scope="SERVICE_REQUEST"/>
            <PARAMETER type="RELATIVE" value="cognome" scope="SERVICE_REQUEST"/>
            <PARAMETER type="RELATIVE" value="email" scope="SERVICE_REQUEST"/>
            <PARAMETER type="RELATIVE" value="id_user" scope="SERVICE_REQUEST"/>
         </UPDATE_QUERY>
      </QUERIES>
```

```
      <FIELDS>
         <FIELD name="id_user" label="User ID" widget="" size="60">
            <INSERT default_type="" default_value="" default_scope=""
               is_readonly="FALSE" is_mandatory="TRUE" is_visible="TRUE"/>
            <UPDATE is_readonly="TRUE" is_mandatory="TRUE" is_visible="TRUE"/>
         </FIELD>
         <FIELD name="nome" label="Nome" widget="" size="60">
            <INSERT default_type="" default_value="" default_scope=""
               is_readonly="FALSE" is_mandatory="TRUE" is_visible="TRUE"/>
            <UPDATE is_readonly="FALSE" is_mandatory="TRUE" is_visible="TRUE"/>
         </FIELD>
         .....
      </FIELDS>
```

```
      <BUTTONS>
         <SUBMIT_BUTTON  image="" label="Salva Utente" confirm="TRUE"/>
      </BUTTONS>
   </CONFIG>
</MODULE>
```

As for the list module the xml envelop CONFIG has the following attributes:

- *pool*: contains the name of the connection pool to use for effect the connection to the database. For major information on connection pool configuration see Data Access section.
- *title*: title of the detail (can be visualized into the JSP as a title).

Inside it contains other xml configuration envelopes:

- QUERIES: contains the references to the statements (defined into the files *statements.xml*) used for insert / read / update a single row into the database:
  - INSERT_QUERY
  - SELECT_QUERY
  - UPDATE_QUERY
- FIELDS: contains the definition of all the fields visualized in the JSP detail page, which are used to show the detail information and to allow the user to change them. For each filed must be specified a envelop *FIELD* with the following attributes:
  - *name*: name of the database column associated to the field
  - *label*: label of the field.
  - *widget*: widget to use for the representation. Currently the only widget allowed is the html text field, so this parameter has an empty value, but in the future releases Spago will support other widgets .
  - *size*: dimension of the field.
  - *is_readonly*: flag that signal if the field value can be only read .
  - *is_mandatory*: flag that signal if the field is mandatory. The obligatory fields are represented with bold label and the system generates for them a javascript code which controls if the field is empty when the form is submitted.
  - *is_visible*: flag that signal if the field is visible or not .
- BUTTONS: contains the configuration of the buttons showed under the list. The possible default buttons are:
  - SUBMIT_BUTTON: button used for submit the insert / update form. Developers can configure the image to show in place of the classical html button, the label of the button and a confirm flag that if set to TRUE force the system to show a javascript confirm operation pop-up when the button is selected.

For this button is applied the prevent resubmit mechanism but this default behaviour can be disable setting the *prevent_resubmit* attribute to false.

As for the list page is possible to add new specific buttons simply adding the correspondent *BUTTON* envelopes. (see the List section for better information).

3.18.4Presentation

In the previous paragraphs it has been explained the list / detail module configuration, in this one is explain the presentation of the data. In the example it has been used an only framework page for the visualization of the list and the detail, distinguishing the two situation based on the presence of the parameter *module* in the session. For the presentation will be used a JSP page, configuring the file *presentation.xml* as below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PRESENTATION>
  <MAPPING
    business_name="AutomaticPageUser"
    business_type="PAGE"
    publisher_name="AutomaticPageUser"/>
</PRESENTATION>
```

This configuration maps the Spago page to an homonym Spago publisher which, into the file *publishers.xml,* is configured as below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PUBLISHERS>
  <PUBLISHER name="AutomaticPageUser">
    <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
      <RESOURCES>
        <ITEM prog="0" resource="/jsp/demo/listdetail/PageUsers.jsp"/>
      </RESOURCES>
    </RENDERING>
  </PUBLISHER>
</PUBLISHERS>
```

This configuration force the system to show the execution result of the page ***AutomaticPageUser*** into a JSP called ***PageUsers.jsp.***

The JSP uses a framework tag library to perform the rendering operation:

```
<%@ taglib uri="spagotags" prefix="spago"%>
```

this library defines the following tags:

- *<spago:**error**/>*: access to the error stack (*EMFErrorHandler*) and, using the errors contained, it generates a javascript function call *checkError()*. This function must be call after the JSP page load event, using this instruction:

  ```
  <body onload="checkError();">
  ```

  into the html *body*. The function, if there's errors into the stack, produces a javascript alert pop-up that shows the list of errors. An example of the produced code and the relative alert pop-up can be:

  ```
  <SCRIPT language="Javascript" type="text/javascript">
  <!--
  function checkError() {
  alert("\nLa lista non contiene righe");
  }
  // -->
  </SCRIPT>
  ```

in this case the list retrieved is empty so the module ha inserted a new error into the *EMFErrorHandler* and consequently the error tag produce a function that shows an alert pop-up with the error message.

- *<spago:**list** moduleName="module name" />*: performs the rendering of the list.
- *<spago:**detail** moduleName="nome del modulo"/>*: performs the rendering of the detail page.

The list and detail modules can be configured to force the system to show a javascript confirm pop-up before executing some operations, like the delete function for example. In this case the last two tags generate a javascript function *goConfirm(url,alertFlag)* that is called when an operation to confirm is request. The function, before send the request to the server, shows a javascript confirm pop-up and send the request only if the user answer positively to the confirm question. An example of the produced code and the relative confirm pop-up can be::

```
<SCRIPT language="Javascript" type="text/javascript">
   function goConfirm(url, alertFlag) {
      var _url = "AdapterHTTP?";
      _url = _url + url;
      if (alertFlag == 'TRUE' ) {
         if (confirm('Confermi operazione'))
            window.location = _url;
      }
      else
         window.location = _url;
   }
</SCRIPT>
```

Obviously the JSP, which is the rendering target for the list and detail module, needs a mechanism to decide if use the list or detail tag. This control can be easily perform looking at the presence into the request of a param module.
The entire JSP code is showed below.

```
<%@ page import="it.eng.spago.base.*,java.lang.*,java.text.*,java.util.*"%>
<%@ taglib uri="spagotags" prefix="spago"%>
<%
   ResponseContainer responseContainer = ResponseContainerAccess.getResponseContainer(request);
   SourceBean serviceResponse = responseContainer.getServiceResponse();
%>
<html>
<HEAD>
<LINK rel="stylesheet" type="text/css" href="../css/spago/listdetail.css" />
</HEAD>
<body onload="checkError();">
<spago:error/>
<% if (serviceResponse.getAttribute("module") == null) { %>
   <spago:list moduleName="AutomaticListUser" />
<% } else { %>
   <spago:detail moduleName="AutomaticDetailUser" />
   <hr width="80%" size="2">
```

```
    <table>
      <tr><td>
        <input class="ListButtonChangePage" type="button" value="RETURN"
          onclick="window.location='AdapterHTTP?NAVIGATOR_BACK=1&LIST_NOCACHE=TRUE'" />
      </tr></td>
    </table>
<% } %>
</body>
</html>
```

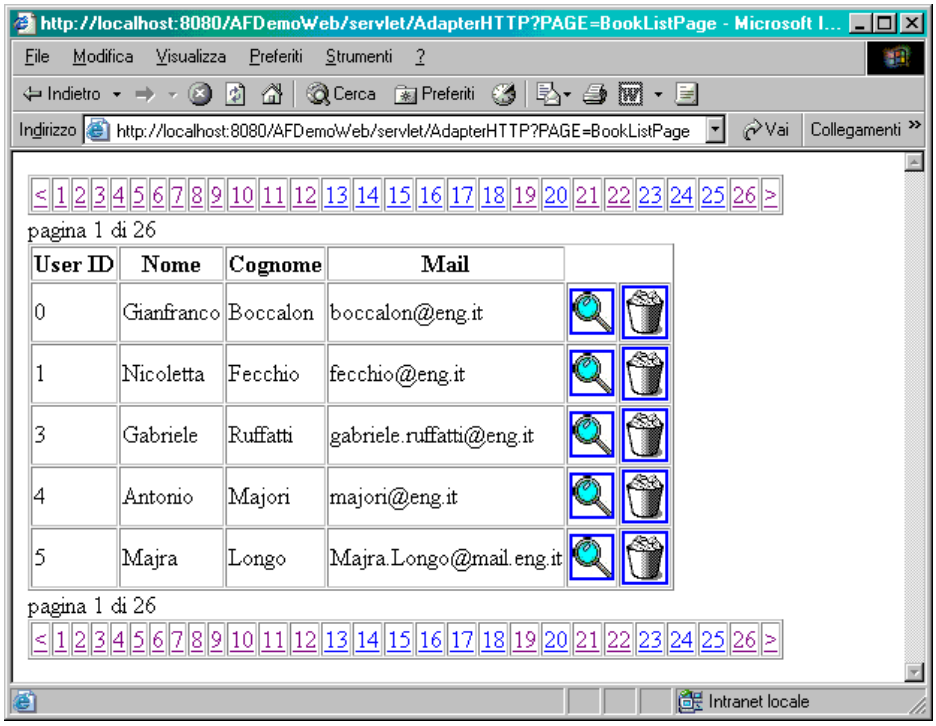
To change the page layout developers have to modify the tags of the library or to produce new tags.

3.19Automatic Generation of list / detail (with - cache)

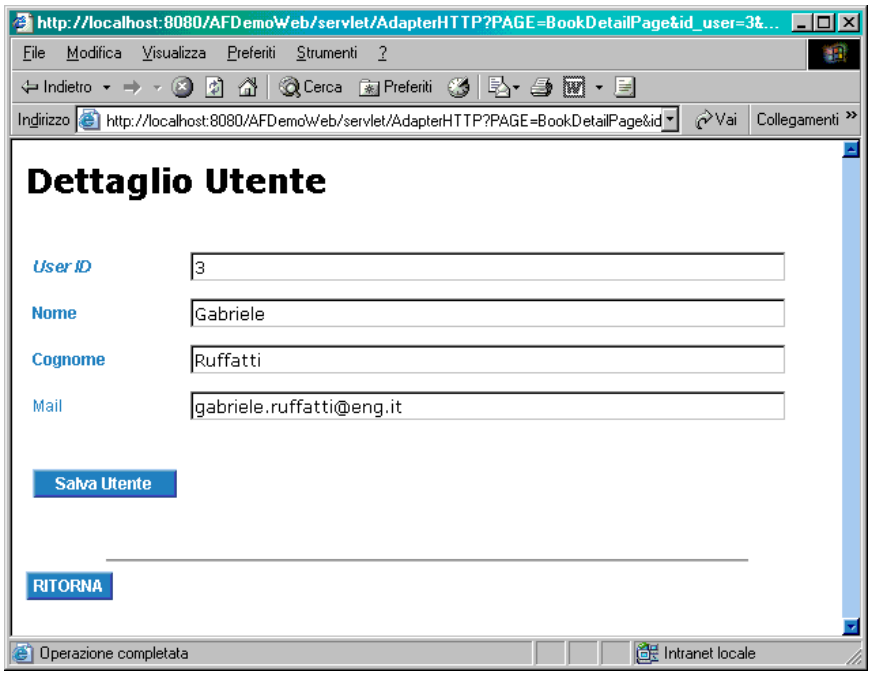
In the previous session it has been described an example of a paginated list build using Spago modules, tag library and a one-shot paginator, in this one, instead, the same example will be constructed using a Paginator with-cache. (See the section Pagination for more information about the different type of framework paginators, and the section Automatic Generation of list / detail pages (one - shot) for an introduction on the paginated list and the specific example)

This example uses the modules *it.eng.spago.dispatching.module.list.smart.impl.**DefaultSmartListModule,*** for the list generation, and *it.eng.spago.dispatching.module.impl.**DefaultDetailModule*** for the detail generation (which is the same module used in the on-shot paginator example). The class DefaultSmartListModule uses the paginator with-cache functionalities and for this reason it extends the abstract module *AbstractSmartListModule* (see section With - cache).

With some configuration parameters, these modules, and the rendering tag libraries implemented, can retrieve items from a database and show them in a paged list like this:

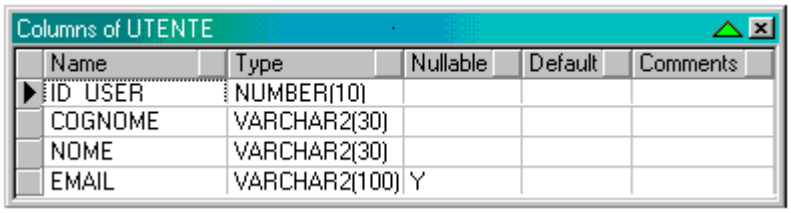


Like the image shows, the page renders the elements with a table, each rows correspond to one items. The page shows the index of all the list page, users can easily jump from one page to another simply selecting the page index. Each rows have a button, drawn like a lens, that redirect to the correspondent item detail page showed below.



3.19.1List

For this example suppose to extract data from a database table structured like this::



Developers must configure the module for the list construction in the file *modules.xml*:

```
<MODULE name="SmartListUsers" class="it.eng.spago.dispatching.module.list.smart.impl.DefaultSmartListModule">
    <CONFIG title="Smart List Users Module">
        <ROW_PROVIDER class="it.eng.spago.paginator.smart.impl.DBRowHandler">
            <CONFIG pool="afdemo">
                <LIST_QUERY statement="LIST_USERS"></LIST_QUERY>
                <DELETE_QUERY statement="DELETE_USER">
                    <PARAMETER type="RELATIVE" value="userid"
                    scope="SERVICE_REQUEST" />
                </DELETE_QUERY>
            </CONFIG>
        </ROW_PROVIDER>
        <PAGE_PROVIDER class="it.eng.spago.paginator.smart.impl.CacheablePageProvider">
            <CONFIG page_size="10" side_pages="2" />
        </PAGE_PROVIDER>
        <LIST_PROVIDER class="it.eng.spago.paginator.smart.impl.DefaultListProvider">
            <CONFIG />
        </LIST_PROVIDER>
        <COLUMNS>
            <COLUMN name="userid" label="User ID" />
            <COLUMN name="nome" label="Nome" />
            <COLUMN name="cognome" label="Cognome" />
            <COLUMN name="mail" label="Mail" />
        </COLUMNS>
        <CAPTIONS>
            <SELECT_CAPTION image="" label="Detail User" confirm="FALSE">
                <PARAMETER name="PAGE" type="ABSOLUTE" value="SmartPageUseri"
                scope="" />
                <PARAMETER name="MODULE" type="ABSOLUTE" value="DetailUser" scope="" />
                <PARAMETER name="userid" type="RELATIVE" value="userid" scope="LOCAL" />
            </SELECT_CAPTION>
            <DELETE_CAPTION image="" label="Elimina Utente" confirm="TRUE">
                <PARAMETER name="userid" type="RELATIVE" value="userid" scope="LOCAL" />
            </DELETE_CAPTION>
        </CAPTIONS>
        <BUTTONS>
            <INSERT_BUTTON image="" label="Inserimento Utente" confirm="FALSE">
                <PARAMETER name="PAGE" type="ABSOLUTE" value="SmartPageUser" scope=""
                />
                <PARAMETER name="MODULE" type="ABSOLUTE" value="DetailUser" scope="" />
            </INSERT_BUTTON>
        </BUTTONS>
    </CONFIG>
</MODULE>
```

```
<MODULE
  name="BookListModule"
  class="it.eng.spago.dispatching.module.DefaultListModule">
  <CONFIG rowHandler="DBRowHandler" pageProvider="CacheablePageProvider"
listProvider="DefaultListProvider"/>
  </MODULE>
```

the module contains a configuration session *CONFIG* in which the following xml envelopes are defined:

- *ROW_PROVIDER*: defines the class that implements the interface *IfaceRowProvider,* that acts like a row handler, and the queries necessary to manage the list. Contains an envelope *CONFIG* that declare, with the attribute *pool,* the name of the connection pool from which get a connection to the database, and the logical name of the queries for return the list elements (*LIST_QUERY*) and for delete a list item (*DELETE_QUERY*) . These queries must be defined into the file *statements.xml* like this example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<STATEMENTS>
```

```
<STATEMENT name="LIST_USERS"
   query="SELECT id_user,nome,cognome,email FROM UTENTE ORDER BY id_user"/>
<STATEMENT name="DELETE_USER"
   query="DELETE FROM UTENTE WHERE id_user = ?"/>
</STATEMENTS>
```

- *PAGE_PROVIDER*: defines the class that implement the interface *IfacePageProvider,* that acts like a page handler, and the page configuration parameters. The contained *CONFIG* envelope defines the number of rows for each page (*page_size*) and the number of pages to maintain in cache (*side_pages*).
- *LIST_PROVIDER*: defines the class that implements the interface *IfaceListProvider,* that acts like a list handler.

3.19.2List Commands

It is possible to send some messages to the module list inserting some parameters into the request. This messages are the same used for the one-shot paginator, so readers can look at the section List Commands for a complete messages description.

3.19.3Detail

The detail module is configured exactly like the correspondent one of the one-shot paginator example.

3.19.4Presentation

This example uses a Spago page in order to implement the logic that builds the list and the detail. The system uses an only JSP page for the visualization of the two operations result. The configuration of the page, into the file *pages.xml,* looks as:

```
<PAGE name="SmartPageUsers" class="it.eng.spago.dispatching.module.ConfigurablePage" scope="SESSION">
        <GRAPH id="2">
            <MODULES>
                <MODULE id="1" name="SmartListUsers" />
                <MODULE id="2" name="DetailUser" />
            </MODULES>
        </GRAPH>
</PAGE>
```

The graph declared is configured into the *graphs.xml* file as showed below:

```
    <GRAPH id="2" >
        <MODULES>
            <MODULE id="1" keep_instance="TRUE" keep_response="FALSE" />
            <MODULE id="2" keep_instance="TRUE" keep_response="FALSE" />
        </MODULES>
        <DEPENDENCIES>
            <DEPENDENCE id="1" source="" target="{1}">
                <CONDITIONS>
                    <PARAMETER name="module" scope="SERVICE_REQUEST"
value="AF_NOT_DEFINED" />
                </CONDITIONS>
                <CONSEQUENCES />
            </DEPENDENCE>
            <DEPENDENCE id="2" source="" target="{1}">
                <CONDITIONS>
                    <PARAMETER name="module" scope="SERVICE_REQUEST" value="{1}" />
                </CONDITIONS>
                <CONSEQUENCES />
            </DEPENDENCE>
            <DEPENDENCE id="3" source="" target="{2}">
                <CONDITIONS>
                    <PARAMETER name="module" scope="SERVICE_REQUEST" value="{2}" />
                </CONDITIONS>
                <CONSEQUENCES />
            </DEPENDENCE>
        </DEPENDENCIES>
    </GRAPH>
```

The page, into the *presentation.xml,* is associated to the correspondent publisher:

<MAPPING business_type="PAGE" business_name="SmartPageUsers" publisher_name="SmartPageUsersAction" />

the publisher is defined inside the file *publishers.xml*:

```
<PUBLISHER name="SmartListUSersAction">
        <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
            <RESOURCES>
                <ITEM prog="0" resource="/jsp/demo/action/SmartListUUsers.jsp" />
            </RESOURCES>
        </RENDERING>
    </ PUBLISHER >
```

3.20Profiling

The security subsystem is composed only from interfaces: different authentication and authorization systems can be integrated with specific wrapper. Spago developers have made this choice because it was impossible to build a security system for cover all different requirements. In this way it is possible to define an own specific service follow some guide lines dictated from Spago.

The interface of the framework to implement for manage the profiling is *it.eng.spago.security.IengUserProfile,* which defines methods for:

- Get the unique identifier of the user
- Get an attribute value associated to a user
- Get the collection of user rules
- Verify if a user has a rule
- Get user functionalities collection. A functionality is a logic operation performed by an user. So, a functionality is the name of a module or an action (both are user services demands), but it is also a generic application.
- Control if a user can exec a module
- Control if a user can exec an action

The framework introduce also an abstract class *it.eng.spago.security.AbstractEngUserProfile* that implements the security interface and extend the class *DefaultRequestContext.*

public abstract class AbstractEngUserProfile extends DefaultRequestContext implements IEngUserProfile

Developers can extend this class, implementing all the interface methods, with the possibility to access to user profile data stored in the request.

Applications developers, during the login phase, have to insert an object, that implements the *IengUserProfile* interface or extends the *AbstractEngUserProfile,* into the permanent container (so that it is not erased after a NAVIGATOR_RESET) like an attribute name *IEngUserProfile.ENG_USER_PROFILE*:

```
IEngUserProfile userProfile = new ...;
getRequestContainer().getSessionContainer().getPermanentContainer().setAttribute(IEngUserProfile.ENG_USER_PROFILE, userProfile);
```

The system, for each service request (indifferently if executed from a module or from an action), verify the user permission to execute the request, and only if he has it the service is executed. In case of the user has no permission:

- If the service is executed with actions modality the system is redirect to a “SECURITY_ERROR_PUBLISHER”, which must be defined inside the file *publisher.xml,* and can be used to show a JSP error page for example.
- If the service is executed with modules modality the module isn’t executed but no one exception is raised and the the execution continues. This mechanism allows to perform correctly the page task executing only the modules allowed to the user.

All the logic and configuration files that defines the user identifier, rules and functionalities associated, and in general all profile information, are not part of Spago because this information and activities are delegated to an external system.

3.20.1Integration with external system

The easier way for integrate Spago with external profiling system is to implement the interface IEngUserProfile but another one exists and allows to split the authentication and authorization functionalities.
This new integration system is based on two interfaces:

- Authentication: *it.eng.spago.security.IAuthenticationHandler*. Defines a service for authenticate the user with the method *public Object* ***authenticate****(SourceBean request, SourceBean response) throws EMFInternalError.* This method recovers the user information from the request and build a generic object that contains the authentication information.
- Authorization: *it.eng.spago.security.IAuthorizationHandler*. Defines a service for create an put, in the application context (the permanent session container), the current user profiling information, represented by the object that implements *IengUserProfile* interface. The method for perform this operation is:

  public void ***putAuhtorizationProfile****(Object userIdentifier,RequestContainer requestConatiner) throws EMFInternalError;*

To create a login service developers must build two classes for implement these interfaces and use the framework class *GenericLoginHandler.* This class defines an only method *service(RequestContainer,SourceBean,SourceBean)* that must be called after all the authentication information have been stored in request. The method instances the authentication object, calls the object service for authenticate user with the information in request, instances the authorization object (if the user is correctly authenticate) and puts it in the permanent session. After the method execution, in the permanent session there's an object of type *IengUserProfile.*
The *GenericLoginHandler* class must be configured (specifying the name of the class that implements the authentication and authorization interfaces) using the file *security.xml.* All the parameters necessary to the class methods can be defined in this file. An example configuration is show below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SECURITY>
  .....
  <AUTHENTICATION_HANDLER>
    it.eng.spago.security.profilemanager.EngiwebProfileManagerAuthenticationHandler
  </AUTHENTICATION_HANDLER>

  <AUTHORIZATION_HANDLER>
    it.eng.spago.security.profilemanager.EngiwebProfileManagerAuthorizationHandler
  </AUTHORIZATION_HANDLER>

  <ENGIWEBPM
    urlProvider="iiop://172.20.100.109:900"
    ctxInitialFactory="com.ibm.websphere.naming.WsnInitialContextFactory"
    realm="PADOVA TEST"/>
</SECURITY>
```

The framework contains an example implementation of this mechanism which retrieves the user information from an xml file. The classes that implement the authentication and authorization interfaces are:

- *it.eng.spago.security.xmlauthorizations.XMLAuthenticationHandler*
- *it.eng.spago.security.xmlauthorizations.XMLAuthorizationHandler*

This example can be found into the *spago-profile-core.jar*.file.

3.20.2Example implementation using XML files

The framework contains an example implementation of an authentication/authorization system based on XML files. The example classes are part of the package *it.eng.spago.security.xmlauthorizations* contained into the *spago-profile-core.jar* file. The classes that implement the authentication and authorization interfaces are:

- *it.eng.spago.security.xmlauthorizations.XMLAuthenticationHandler*
- *it.eng.spago.security.xmlauthorizations.XMLAuthorizationHandler*

In order to use this implementation the *security.xml* file must be configured like follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SECURITY>
  .....
  <AUTHENTICATION_HANDLER>
    it.eng.spago.security.xmlauthorizations.XMLAuthenticationHandler
  </AUTHENTICATION_HANDLER>

  <AUTHORIZATION_HANDLER>
    it.eng.spago.security.xmlauthorizations.XMLAuthorizationHandler
  </AUTHORIZATION_HANDLER>

  <XMLPM authorizationFileLocation="C:\Progetti\ServiziSportivi\sviluppo\web\WEB-INF\conf\authorizations.xml"/>
</SECURITY>
```

This implementation require an xml configuration envelope call *XMLPM* which contains the absolute path of the file containing the users information. An example of this *authorizations.xml* file can be:

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTHORIZATIONS>
    <ENTITIES>
        <USERS>
            <USER userID="sduca" password="7fLbNfT7Y5q6KPu4OJZVaACnvIM="
                nome="sonia" cognome="duca" quartiere="2"/>
            <USER userID="azoppello" password="7oMUvcvVv0jjZFbGJHs9i/uDU3A="
                nome="andrea" cognome="zoppello" quartiere="2"/>
        </USERS>

        <ROLES>
            <ROLE roleName="RuoloBase" description="RuoloBase"/>
        </ROLES>

        <FUNCTIONALITIES>
            <FUNCTIONALITY functionalityName="InserimentoDomanda" description="InserimentoDomanda"/>
            <FUNCTIONALITY functionalityName="AssegnazioneDomanda"
description="AssegnazioneDomanda"/>
            <FUNCTIONALITY functionalityName="GestioneDecodifica" description="GestioneDecodifica"/>
        </FUNCTIONALITIES>

    </ENTITIES>
```

```
<RELATIONS>
    <BEHAVIOURS>
        <BEHAVIOUR userID="sduca" roleName="RuoloBase"/>
        <BEHAVIOUR userID="azoppello" roleName="RuoloBase"/>
    </BEHAVIOURS>

    <PRIVILEDGES>
        <PRIVILEDGE roleName="RuoloBase" functionalityName="InserimentoDomanda"/>
        <PRIVILEDGE roleName="RuoloBase" functionalityName="AssegnazioneDomanda"/>
    </PRIVILEDGES>
</RELATIONS>
</AUTHORIZATIONS>
```

The file contains the following xml envelopes :

- *ENTITIES*: define all users, rules and functionalities into the respective envelope *USERS*, *ROLES* and *FUNCTIONALITIES*.
- *RELATIONS*: define the association between rules and users into the *BEHAVIOURS* envelope, and the association between functionalities and rules into the *PRIVILEDGES* envelope.

The user password are codify using a one-way hash (sh1) algorithm that performs an encrypt operation on a digest calculated from the original password. All the password have a fixed length an it is almost impossible to decrypt them in a reasonable time.

In this case the functionalities defined correspond to the application use-cases so the example use the *business_map.xml* file to manage the mapping between the functionalities and the correspondent Spago services (actions or pages).

3.21Initialization

The framework provides a service for load, at application start-up time, some modules or applicative objects. The configuration of the initialisation process is defined into the *initializers.xml* file. An example of this file can be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<INITIALIZERS>
   <INITIALIZER class="it.eng.spago.event.condition.EventConditionInitializer" config="EVENT_CONDITIONS"/>
   <INITIALIZER class="it.eng.spago.event.handler.EventHandlerInitializer" config="EVENT_HANDLERS"/>
</INITIALIZERS>
```

Each class defined by the *class* attribute must implement the interface *it.eng.spago.init.InitializerIFace* which defines the method *void **init**(SourceBean config)*. This method is called during the application start – up phase and contains the logic for initialise the module.

The init method receives in input a SourceBean object containing the specific module configuration parameters. In the configuration file is possible to specify the name of an xml envelope containing the module configuration, which must be defined in one file census by the *master.xml* . Spago automatically recovers this envelope and builds the correspondent SourceBean in order to pass it to the init method.

In this example to the object *EventConditionInitializer* is passed a configuration xml envelope *EVENT_CONDITIONS* while to the object *EventHandlerInitializer* is passed the *EVENT_HANDLERS* envelope.

3.22 Monitoring

The JAMon product, available at site http://www.jamonapi.com/, has been integrated inside Spago for monitor the application business activities, data access, and presentation services, in order to determine performance bottlenecks, user/application interactions, and application scalability.

The use of JAMon can be able / disable changing to true / false the *monitor_enabled* attribute contained into the *common.xml* file. The administration interface can be found inside the folder “web/root/jamon” of the distribution.

4 Deploy on cluster

This paragraph reports some consideration about Spago's developing applications that will be executed on cluster systems. In this context it is necessary to note that the application's processes can be execute on different Virtual Machine and potentially on different server hosts; it depends from how the application is deployed. This scenery can cause some problems to the Spago Containers:

- Application Container: it is an application container shared from all actions and modules. Technically is implemented like a singleton: this means that this object lives inside the class loader from which is invoked, therefore if there's more than one application server (like in cluster system) more than one application container will be created. In a classical configuration there could be one into the Web Container and one into the EJB Container, for example.

  For this reason, in these context, the container can be only used like a repository for read only information. It is not possible to use it for pass information between business objects (actions e modules).

  If the application needs a real singleton container also in a cluster system, the SUN guide lines advise to use an entity bean or a remote service, like a CORBA SERVER, for example. The CORBA solution has a problem because the remote server would not be considered during the fail over mechanism of the application server, so the entity bean is the best solution.
- Session Container: it is a container associated to the current user. A critical element of this object is the dimension. The IBM philosophy identify in 4K the maximum dimension for the http session because, with a mechanism of session management enable, all the data contained is serialized and de-serialized. With a lot of objects contained this process can be a performance issue. IBM guide lines advise to store into the session only the name of the object to retrieve. Spago implements some mechanism that make an intensive use of the session, like the navigator, for example. We are working for update the framework in order to allow users to configure parameters for control how much data can be put in the session.