Spago
User Guide

Redatto da

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

Scopo del Documento

In questo documento vengono presentate alcune linee guida su come impostare i progetti sviluppati con Spago e su come utilizzarne le funzionalità.

Informazioni sulla versione

Versione/Release n°:	1.0	**Data**	08/10/2004
Descrizione modifiche:	Versione iniziale		
Versione/Release n°:	1.1	**Data**	29/10/2004
Descrizione modifiche:	Correzioni formali		
Versione/Release n°:	1.2	**Data**	03/11/2004
Descrizione modifiche:	Documentato il package di validazione Documentato il Prevent resubmit Eliminato il paragrafo sulla gestione dei job		
Versione/Release n°:	1.3	**Data**	13/12/2004
Descrizione modifiche:	Aggiornta la documentazione relativa alla validazione Aggiunto il paragrafo che descrive le tag libraries Minimi aggiornamenti alla struttura di *master.xml* e *data_-access.xml* Modificato il paragrafo relativo ad Hibernate Aggiunte alcune precisazioni nel par.2.2 relative a *web.xml* Minimi aggiornamenti al publisher di tipo LOOP Aggiornato il paragrafo Pubblicazione Multilingua		
Versione/Release n°:	1.4	**Data**	05/04/2005
Descrizione modifiche:	Sezione validazione: corretta la sintassi del contenuto di *validation.xml*. Minimi aggiornamenti alla sezione Accesso Ai Dati -> Gestione delle connessioni e dei pool		
Versione/Release n°:	1.5	**Data**	10/08/2005
Descrizione modifiche:	Sezione navigazione: rimozione del comando NAVIGATOR_BACK_TO_SERVICE in quanto inesistente.		

	Correzione del comando NAVIGATOR_BACK_TO.		
Versione/Release n°:	1.6	**Data**	10/10/2005
Descrizione modifiche:	Aggiunta la possibilità di spezzare i file di configurazione. Aggiunta la possibilità di sostituire il motore di validazione.		
Version/Release n°:	1.7	**Data**	30/11/2005
Description:	Modificata la documentazione sulla validazione per la nuova configurazione, aggiunta la sezione sui publisher dipendenti dall'agent.		
Version/Release n°:	1.8	**Data**	6/12/2005
Description:	Aggiunti paragrafi "Eventi sui servizi applicativi" e "Caching delle risposte sui sevizi applicativi".		
Version/Release n°:	2.1.0	**Data**	20/12/2005
Description:	Aggiunti paragrafi "Automatismo di supend Resume" e Form accessibili. Aggiunto paragrafo "Condizioni espresse tramite regole" relativo all'integrazione con Drools.		
Version/Release n°:	2.1.1	**Data**	27/09/2006
Description:	Aggiornato il paragrafo "Condizioni" nella sezione dei moduli.		

1 Introduzione

Il framework nasce nel 1999 per rispondere alle esigenze di un progetto complesso che si prefiggeva di costruire su web tutte le attività di una azienda.

Si voleva utilizzare un layer che permettesse di astrarre l'implementazione per quanto riguarda i requisiti più comuni: gestione del database, meccanismi di caching, e così via.

L'esperienza è stata messa a frutto per l'uso in altri progetti: il risultato è Spago, reingegnerizzato e riscritto, rispetto alla versione utilizzata in quel progetto.

Il risultato è un'implementazione del modello MVC (Model-View-Controller).

Il framework, comunque, è un progetto aperto a nuove estensioni che possano contribuire al suo miglioramento.

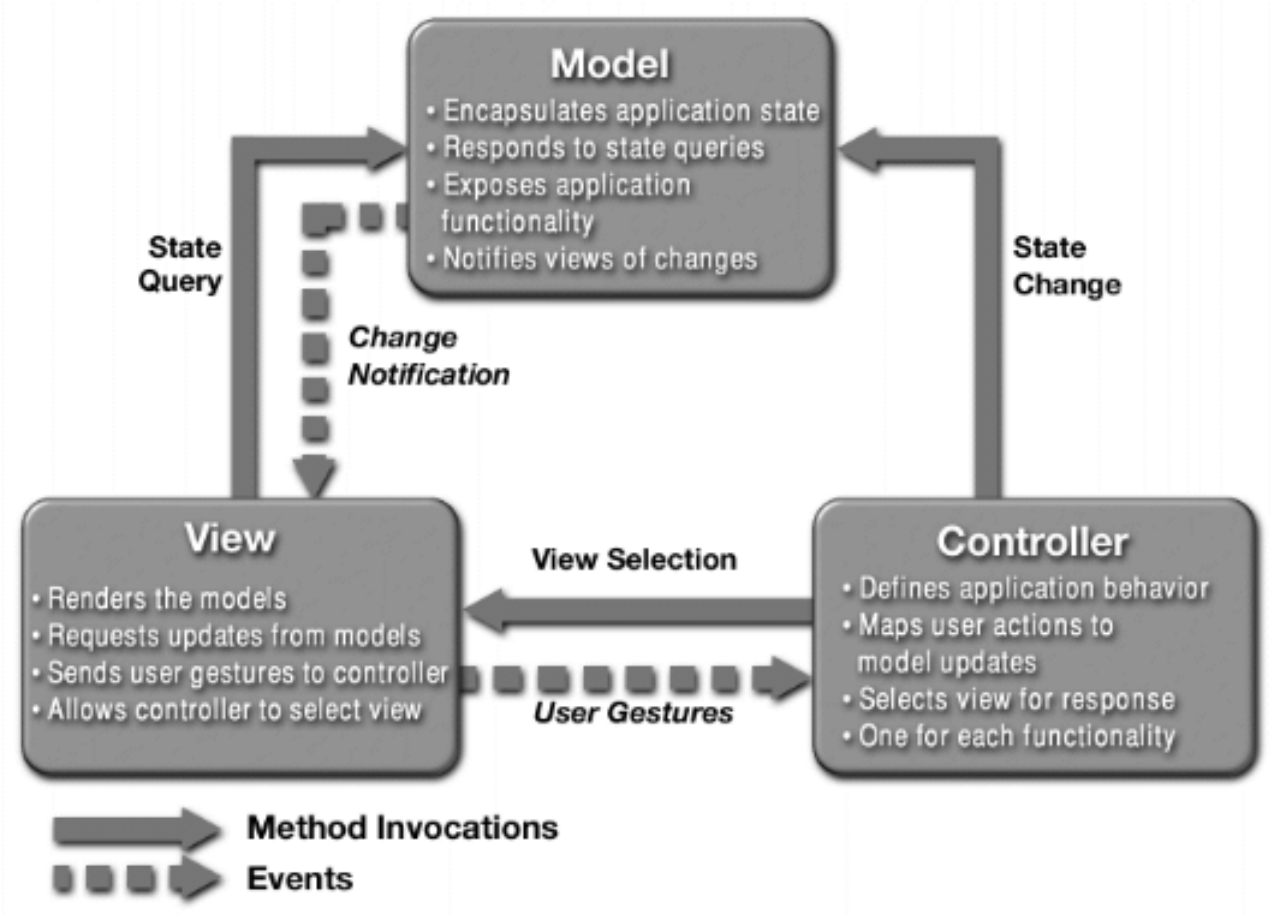
Spago non è semplicemente un prodotto ma è anche una metodologia di lavoro, in quanto il suo utilizzo implica innanzitutto una separazione delle responsabilità tra il progettista e lo sviluppatore.

L'uso di un framework (in generale, non necessariamente di Spago) comporta pertanto una maggiore produttività in quanto il "impone" una modalità di lavoro e una determinata struttura delle applicazioni, facilitando la manutenzione del codice.

1.1 Architettura di riferimento

Il modelllo MVC è il modello di riferimento del framework. Tale modello comporta la separazione di ogni applicazione in 3 componenti principali:

- **Model**: è la componente dell'applicazione che implementa la logica di business. Per applicazioni sviluppate con il framework, il model è costituito dai business object (gli oggetti che implementano la logica di business) scritti dallo sviluppatore.
- **View**: è la componente che effettua il rendering del modello: tale componente deve presentare i contenuti in modo coerente con il canale da cui proviene la richiesta. Il framework fornisce diverse modalità per questa attività; vedremo successivamente in dettaglio quali sono.
- **Controller**: è la componente che coordina l'esecuzione dei servizi forniti dal modello. Tale componente riceve le richieste di servizio ed effettua il dispatch verso le corrispondenti componenti del modello. Nell'Application Framework esiste un controller per ogni canale: è una servlet per il canale HTTP, un EJB per il canale EJB , etc. Pertanto lo strato di controller è già fornito dal framework.



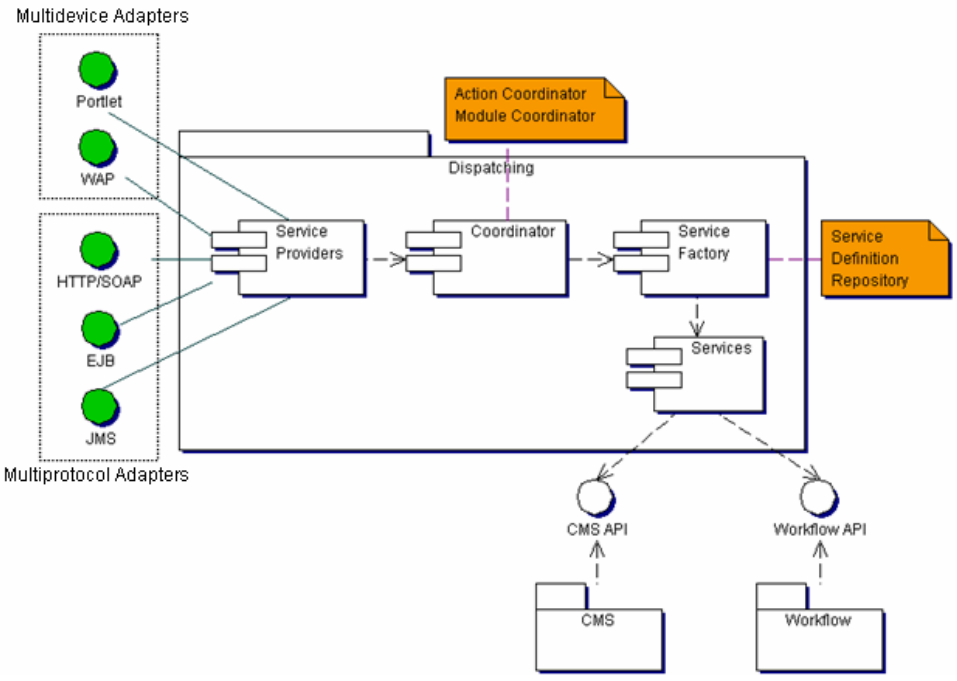
1.2 Modello per componenti

Insieme al framework viene fornito un documento di linee guida che contiene tutte le regole da seguire nel processo di sviluppo. Sono stati identificati dei passi che il progettista deve seguire: censimento use cases, definizione dei business object, definizione della presentazione, etc, e attraverso lo strumento visuale fornito con il framework il progettista consegna lo scheletro degli oggetti da sviiluppare, la configurazione da utilizzare, e la descrizione in linguaggio naturale di tutte le entità da sviluppare.
Questa modalità garantisce una metodologia di sviluppo trasversale a tutti gli oggetti che permette di aumentare la qualità del software e la sua manutenibilità.

1.3 Multicanalità

Per multicanalità si intende la capacità di un'applicazione di ricevere le richieste attraverso più canali (ad esempio via browser web, telefono cellulare WAP, etc) e di presentare il contenuto della risposta in modo coerente con il canale attraverso cui è stata effettuata la richiesta.
Ad esempio una richiesta effettuata attraverso un browser web, riceverà in generale una risposta molto più ricca (se non in termini di contenuti, almeno in termini di veste grafica) rispetto alla stessa richiesta effettuata attraverso un browser WAP.
L'architettura è aperta e permette di aggiungere nuovi canali, oltre a quelli già implementati, che sono i seguenti:

- **HTTP/WAP**: permette di effettuare le richieste al framework tramite il protocollo HTTP, e può essere utilizzato sia da browser web che browser WAP.
- **SOAP**: permette di effettuare richieste di servizi tramite il protocollo SOAP su HTTP.
- **EJB**: permette di invocare richieste di servizi esposti come EJB (Enterprise Java Bean).
- **JMS**: permette al framework di ricevere richieste di servizio attraverso un bus JMS.



L'aggiunta di nuovi adapter è possibile tramite un meccanismo di plug-in.

La multicanalità è ottenuta tramite l'uso di appositi wrapper che nascondono i dettagli del singolo canale:per il canale HTTP, ad esempio, non si lavora mai con le classi *HttpSession* o *HttpServletRequest*, in quanto tutte le informazioni disponibili da tali classi sono incapsulate in apposite classi del framework.

2 Servizi

Vediamo quali sono i servizi offerti dal framework.

2.1 SourceBean

Alla base di tutto il framework c'è l'XML: tutti i sottosistemi del framework comunicano tra loro attraverso flussi XML.
Cosa si intende esattamente per flussi XML ?
Non si intende l'oggetto *String* che contiene XML, ma un oggetto apposito che si chiama **SourceBean** che ha la capacità di costruirsi come *DOM* (Document Object Model), ha capacità di navigazione, si può costruire da uno stream XML, da una stringa,etc.
Perchè è stato costruito il SourceBean, visto che esiste già un oggetto standard con caratteristiche simili, che è *org.w3c.dom.Document* ?
Perchè il SourceBean aggiunge al DOM dei servizi di navigazione piu semplici: ha capacità navigazionali con notazioni puntate, estremamente semplice e potente.
Considerate ad esempio il seguente frammento XML:

```
<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>
```

Una volta letto il frammento XML in un SourceBean, è possibile accedere a tutte le buste con chiave *CONDITION* (quelle evidenziate) tramite lo statement:

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

Non è necessario specificare la busta principale ***EVENT_CONDITIONS***. Il risultato è un vettore di SourceBean, sui quali è ovviamente possibile navigare con la notazione puntata per accedere al contenuto delle buste.

Da questo esempio si può notare che il SourceBean è un contenitore multi-value: una busta XML può contenere più elementi in sequenza con lo stesso nome.
Tutti i sottosistemi del framework comunicano attraverso oggetti di tipo SourceBean: sono di questo tipo il configuratore, la richiesta di servizio, la sessione, la risposta, etc.
Il tracer è in grado di registrare i SourceBean: ad esempio si possono tracciare tutte le richieste (pratica estremamente pesante e per questo sconsigliata se non per motivi di debugging).

E' possibile richiedere ad un SourceBean la sua rappresentazione in XML: poichè il SourceBean può contenere qualunque tipo di oggetto, deve avere un meccanismo per sapere come trasformare in XML il suo contenuto.
Ci sono due casi:

1. L'oggetto contenuto nel SourceBean implementa l'interfaccia *it.eng.spago.base.XMLObject*, pertanto è in grado di rappresentarsi in XML.
2. L'oggetto non implementa l'interfaccia *XMLObject*: viene usato il metodo *toString* dell'oggetto.
   Questo meccanismo viene usato se, ad esempio, nel SourceBean si inserisce una Hashtable.

E' necessario implementare l'interfaccia *XMLObject* quando i contenuti servono in un contesto multicanale ovvero quando a fronte degli stessi contenuti XML è necessario produrre rappresentazioni diverse per i vari canali, ad esempio con fogli di stile XSL. Se si utilizzano pagine JSP non è necessario implementare l'interfaccia *XMLObject* in quanto i dati sono comunque accessibili tramite il response container, session container o application container.
Un'alternativa all'implementare l'interfaccia *XMLObject* (che contiene 5 metodi) consiste nell'estendere la classe del framework *AbstractXMLObject* e implementare il metodo *toElement*(*Document*).

2.1.1 Lettere accentate

Quando si genera uno stream XML che deve essere pubblicato, tale stream deve contenere le definizioni doc-entity per tutti i caratteri non ASCII 127 (ad esempio tutte le lettere accentate).
Tra i file di configurazione utilizzati dal framework c'è un file chiamato *xhtml-lat1.ent* che contiene tutti i caratteri non ASCII 127, tale file si usa per produrre un doc-entity che viene appeso ai file XML prodotti dal metodo *toXML* del SourceBean.
Questa descrizione doc-entity è molto grossa, per cui il framework genera tale stream solo quando lo deve passare ad un transcoder.
Questo automatismo è disponibile nel SourceBean, che implementa l'interfaccia *XMLObject*. Se si implementano oggetti propri, che implementano l'interfaccia *XMLObject*, va prevista la possibilità di dover effettuare la stessa attività svolta dal SourceBean per gestire le lettere accentate.

2.2 Configurazione

Il framework si basa su una serie di file di configurazione il cui caricamento è controllato da un file principale di configurazione chiamato *master.xml*. All'interno di questo file sono memorizzati tutti i percorsi relativi dei file da caricare. Ecco un esempio di questo 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>
```

Questo file, e tutti i file in esso indicati, vengono caricati da un oggetto che si chiama *ConfigSingleton* : è un singleton che permette di accedere a tutte le informazioni presenti nei file, in quanto è un SourceBean.
Pertanto si può accedere alle informazioni presenti nei file di configurazione tramite la notazione puntata delle buste.
Il fatto che le informazioni siano suddivise in diversi file di configurazione, è trasparente all'utente in quanto tutta la configurazione è accessibile tramite un SourceBean.
Ad esempio se i file *actions.xml* e *modules.xml* contengono il seguente testo:

actions.xml

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

modules.xml

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

tramite lo stesso configuratore è possibile accedere ad entrambe le informazioni:

Accesso al file *actions.xml*

Accesso al file *modules.xml*

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

Nel caso in cui serva un nuovo file di configurazione, è sufficiente registrarlo nel file *master.xml* per avere a disposizione le informazioni tramite il *ConfigSingleton*. Il nome del file *master.xml* è un nome di default configurato nel file *web.xml* tramite il parametro AF_CONFIG_FILE che indica il path relativo alla web root.
In web.xml, il parametro AF_ROOT_PATH se non impostato assume il valore della context root. In questo caso per recuperare i file di configurazione viene utilizzato il metodo *getResourceAsStream* di *ServletContext*. Si suppone che il server duranate l'avvio non esploda il *.war* nella directory corrispondente.

2.2.1 Reset della configurazione

E' possibile forzare la rilettura della configurazione chiamando il metodo statico *ConfigSingleton.**release**()*. La configurazione verrà riletta al primo accesso ad una qualunque informazione della configurazione.

2.2.2 Organizzazione della configurazione

Nel caso di applicazioni complesse è possibile dividere i file di configurazione in più file. In questo modo il file *actions.xml*, ad esempio, diventa il folder *actions* il quale contiene i file *actions1.xml* e *actions2.xml*.
In questo modo è possibile organizzare i file di configurazione per aree logiche dell'applicazione.

Riportiamo un esempio riguardante il file *actions.xml*. Il file *master.xml* rimane invariato:

```
<?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>
```

Il contenuto del file *actions.xml*, però, è diverso dalla struttura classica: in questo caso assume la stessa sintassi del file *master.xml*, come si vede dal seguente esempio:

```
<?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>
```

Nella cartella *actions* saranno presenti i file *actions1.xml* e *actions2.xml* con l'usuale struttura.
Ecco ad esempio *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>
```

2.3 Tracing

Il servizio di logging del framework viene configurato mediante l'impostazione dei parametri desiderati nel file *tracing.xml*, che ha la seguente struttura:

```
<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>
```

dove ogni busta <LOGGER> definisce un logger, ossia una classe che implementa l'interfaccia IFaceLogger e che può accedere alla busta <CONFIG> se implementa l'interfaccia InitializerIFace.
Descriviamo qui di seguito il significato degli attributi del tag LOGGER:

- name: nome dell'applicazione che utilizza il logger. E' possibile registrare un logger di default DEFAULT_LOGGER per il tracciamento di sottosistemi che non sono mappati nel file
- class: nome completo della classe che implementa il logger.

La busta CONFIG definisce la configurazione caratteristica di ciascun logger ed è caratterizzata dagli attributi:

- trace_min_log_severity: severity minima di tracciamento (valori compresi tra 0 – 4 inclusi).
- debug: può assumere il valore "true", se viene abilitato il livello di DEBUG o "false" in caso contrario.
- trace_path: individua il percorso dell'eventuale file di log
- trace_name: indica il nome dell'eventuale file di log.
- trace_thread_name (solo per il DefaultLogger): può assumere il valore "true" o "false". Abilita o meno la scrittura del nome del thread nel file di log. In precedenza nel file di log veniva scritto il numero di thread del gruppo del thread corrente, informazione che non era molto utile.

La classe TracerSingleton fornisce i metodi statici di log che verranno utilizzati dalla applicazione ed è in grado di richiedere un'istanza alla classe FactoryLogger e abilitare tutti i logger censiti nel file *tracing.xml*.
Il framework attualmente mette a disposizione due logger:

- DefaultLogger: gestisce i tracciamenti su file di log.
- Log4JLogger: utilizzando le librerie di Log4J è in grado di gestire i tracciamenti su tutti gli Appender definiti nel file log4j.properties inerpellato da Log4J.

E' importante sottolineare che mediante la definizione nel file di configurazione della coppia *name / class* , ovvero applicazione / logger, si possono differenziare le modalità di tracciamento per ogni applicazione che interagisce col framework, oltre che separare i messaggi di log specifici del framework da quelli dell'applicazione. Nell'esempio appena visto, il secondo logger verrà utilizzato per i messaggi relativi al run-time del framework.
La distinzione tra le varie modalità adottate include la definizione di diversi livelli di tracciamento per ciascun logger.

2.3.1 DefaultLogger

Il framework fornisce un sistema per il logging e il tracing di informazioni, warning, ed errori.
Questo sistema ha la caratteristica che a mezzanotte effettua lo switch su un altro file di log per permettere l'adozione di politiche di backup e compressione dei file di log.
Un esempio di utilizzo è il seguente:

TracerSingleton.log("nome applicazione", TracerSingleton.DEBUG, "messaggio");

Ad ogni chiamata al tracer bisogna specificare il tipo di messaggio da tracciare, le tipologie sono le seguenti:

- INFORMATION: andrebbero catalogati con questa tipologia i messaggi informativi.
- WARNING
- MINOR
- MAJOR

- CRITICAL: andrebbero catalogati con questa tipologia i messaggi corrispondenti ad errori di tipo bloccante.
- DEBUG

E' possibile specificare il livello minimo delle informazioni da tracciare: ad esempio se viene specificato MAJOR verranno tracciati tutti i messaggi con livello MAJOR e superiori, ossia MAJOR, CRITICAL e DEBUG.
Il tracer è in grado di tracciare sia stringhe che interi flussi XML.

2.3.2 Log4JLogger

Questa classe si occupa di gestire la modalità di logging che utilizza le librerie di Log4J. Il framework delega a Log4J il recupero del file *log4j.properties*, in cui è possibile definire tutti i tipi di Appender utili ai fini del progetto e configurare le caratteristiche di Log4J.
La severity minima da tracciare e l'abilitazione della modalità di debug viene definita nel file *tracing.xml*, così come avviene per AFLogger.

2.4 Gestione errori

Il framework fornisce uno stack di errori su cui si possono fare varie operazioni.
Ogni errore è caratterizzato da una severity e da una descrizione.
Esistono i seguenti tipi di errori:

- *Errori generici o eccezioni (it.eng.spago.error.**EMFInternalError**)*: vengono utilizzato per mappare errori di terze parti, ad esempio SQL exception. Se non si vuole rimappare questa tipologia di errori con un errore applicativo, va generato un *EMFInternalError* che oltre al messaggio d'errore può contenere anche un'eccezione nativa.
  Tra i vari costruttori ne ha anche uno che prende come parametro un Object:

  *public EMFInternalError(String severity, **Object additionalInfo**)*

  additionalInfo è una proprietà che posso associare all'errore: la presentazione può reperire questo oggetto e completare così la presentazione dell'errore. L'unico vincolo è che l'oggetto sia serializzabile (in quanto verrà trasformato in XML per essere passato tra i vari sottosistemi).
  Tipicamente questo tipo di errori non vengono pubblicati sull'interfaccia utente.

- *Errori utente (it.eng.spago.error.**EMFUserError**)*: sono errori di tipo applicativo, o messaggi di natura informativa, contraddistinti da un codice identificativo. Il codice identificativo serve per la catalogazione multilingua della messaggistica di errore (spiegata in Gestione multilingua).

  public EMFUserError(String severity, String code)

  Va generato un errore di questo tipo se l'errore è riconosciuto e intercettato: la descrizione dell'errore verrà reperita dalle risorse multilingua.
  I messaggi possono avere dei placeholder che verranno riempiti con un elenco di parametri. Questa funzionalità permette di dinamicizzare i messaggi d'errore.

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

  In questo costruttore, *params* è un vettore di parametri contenente i valori che verranno sostituiti ai placeholder presenti nel messaggio. I placeholder vengono definiti come %1,%2, etc.
- *Errori sui campi (it.eng.spago.error.**EMFFieldUserError**)*: sono prodotti in relazione a campi presenti nei form.

- *Errori di validazione (it.eng.spago.error.**EMFValidationError**)*: relativi a errori di validazione. Questi errori sono normalmente prodotti dal motore di validazione o da validatori custom.

Una volta generato un errore (applicativo o meno), va passato al gestore degli errori che è *it.eng.spago.error.**EMFErrorHandler**.* Tale gestore si ottiene, dalle action o dai moduli, tramite il metodo *getErrorHandler()*.
Il gestore degli errori mantiene la lista di tutti gli errori generati per la richiesta di servizio: questi errori possono poi essere utilizzati o meno dalla logica di presentazione.
Ad esempio per accedere all'ErrorHandler da una pagina JSP, supponendo che la pagina derivi da *it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage,* utilizzo il seguente codice:

```
<%@ page extends="it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage"%>

<% EMFErrorHandler errorHandler = getErrorHandler(request);
   Collection errors = errorHandler.getErrors();
  // do something %>
```

Il framework fornisce una tag library, tra i cui tag ve ne è uno che accede allo stack degli errori, presentandoli all'utente. Questo tag è *<spago:error/>* ed è spiegato nella sezione Generazione automatica lista/dettaglio-Visualizzazione.

2.4.1 Categorie di errore

Ogni errore, oltre alla severity, contiene anche l'informazione della categoria. Le seguenti categorie sono predefinite nella classe *it.eng.spago.error.**EMFErrorCategory***:

- *INTERNAL_ERROR*
- *USER_ERROR*
- *VALIDATION_ERROR*

L'utente può definire nuove categorie secondo le necessità.
Nella classe *EMFErrorHandler* c'è un metodo che permette di verificare la presenza di errori con una determinata categoria.
L'interfaccia del metodo è la seguente:

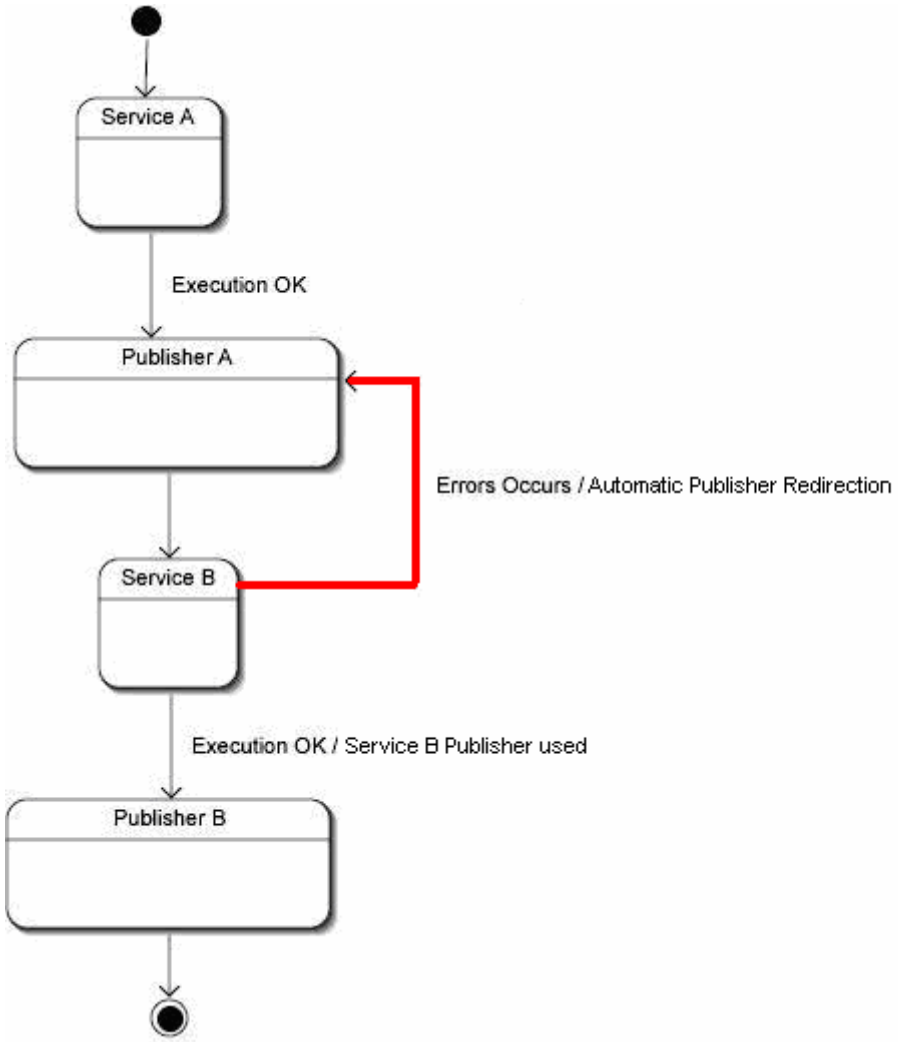
```
public boolean isOKByCategory(String categoryList);
```

2.4.2 Redirezione automatica del publisher

E' possibile avere una redirezione automatica alla pagina corrente in caso d'errore. Per attivare questa funzionalità va specificato l'attributo *AUTOMATIC_PUBLISHER_REDIRECTION* nel file *common.xml*, con il valore TRUE:

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

Con tale configurazione, in caso d'errore (qualunque tipo d'errore), la pubblicazione è automaticamente effettuata sull'ultima pagina utilizzata.



Per configurare quale tipo di errori devono essere considerati da questa gestione, va utilizzato l'attributo *AUTOMATIC_PUBLISHER_REDIRECTION_ON_ERRORS* in cui specificare le categorie di errori (separate da virgola ","). La suddetta configurazione, pertanto, è equivalente a:

<COMMON

....

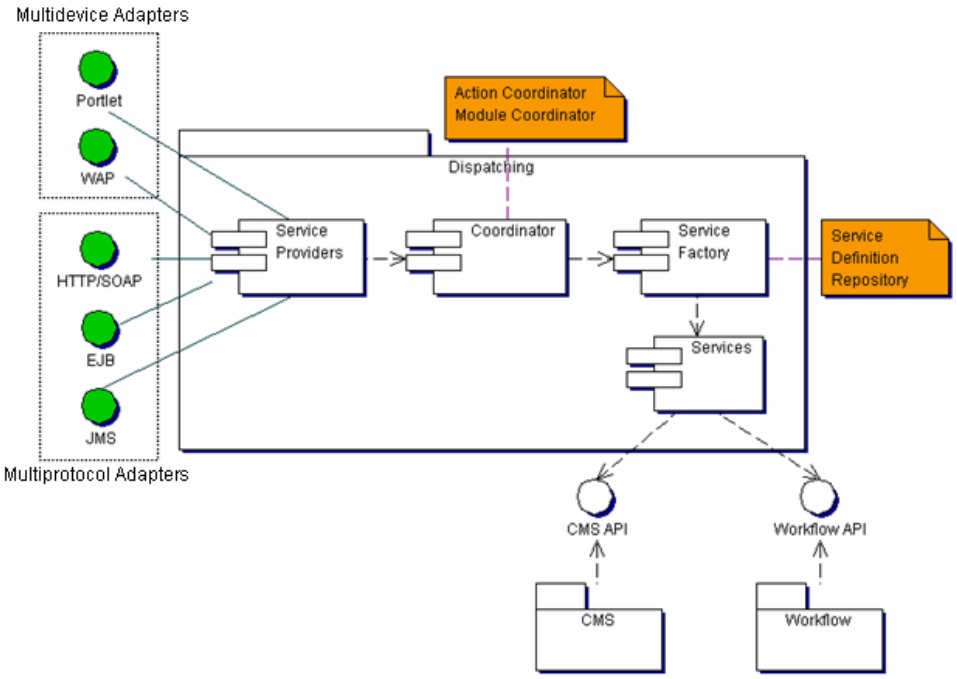
AUTOMATIC_PUBLISHER_REDIRECTION*="TRUE"*

AUTOMATIC_PUBLISHER_REDIRECTION_ON_ERRORS*="INTERNAL_ERROR,USER_ERROR, VALIDATION_ERROR">*

</COMMON>

2.5 Dispatching

Il framework fornisce il supporto alla multicanalità: essendo però le informazioni provenienti da ciascun canale peculiari del canale, vengono forniti un certo numero di adapter che sono oggetti in ascolto delle richieste di servizi: questi adapter traducono le richieste nel formato interno del framework (ossia XML) e attivano il servizio.



A causa del loro legame con il canale, i vari adapter vengono istanziati in Virtual Machine diverse, ossia:

- Gli adapter HTTP/WAP, SOAP vivono nel Web Container: sono delle servlet (si trovano in *spago-web.jar* e *spago-axis.jar*)
- Gli adapter EJB risiedono nell'EJB Container (si trovano in *spago-ejb.jar*) e consistono di:
  - *it.eng.spago.dispatching.ejbchannel.**AdapterEJBBean***: è un session bean statefull con interfaccia remota di tipo CMP (container managed persistence) e BMT (bean managed transaction).
  - *it.eng.spago.dispatching.ejbchannel.**SessionFacadeEJBBean***: è un session bean stateless con interfaccia locale. Essendo stateless ci sono i meccanismi di pooling dell'EJB container per il load balancing, in più non avendo stato non viene serializzato su database tra una richiesta e l'altra.
  - *it.eng.spago.dispatching.ejbchannel.**RemoteSessionFacadeEJBBean***: è un session bean stateless con interfaccia remota.
- L'adapter JMS risiede nell'EJB Container ed è un EJB di tipo Message Driven Bean, implementato dalla classe *it.eng.spago.dispatching.jmschannel.**JmsAdapterMDB*** (si trova in *spago-ejb.jar*).

Il framework fornisce due modalità di dispatching della logica di business: ad action o a moduli.

2.5.1 Action

Analizziamo in dettaglio la modalità più semplice di dispatching offerta dal framework: quella ad Action.
In questa modalità ogni richiesta di servizio viene espletata da un oggetto di business, ovvero da un oggetto che implementa l'interfaccia *ActionIFace* (per brevità, d'ora in poi diremo semplicemente da un'action).
Per chi conosce il framework open-source "Struts", questa modalità è la stessa offerta da tale prodotto.

2.5.1.1 Censimento

Le action vengono censite in un file di configurazione chiamato *actions.xml*. Ecco un esempio di tale file:

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

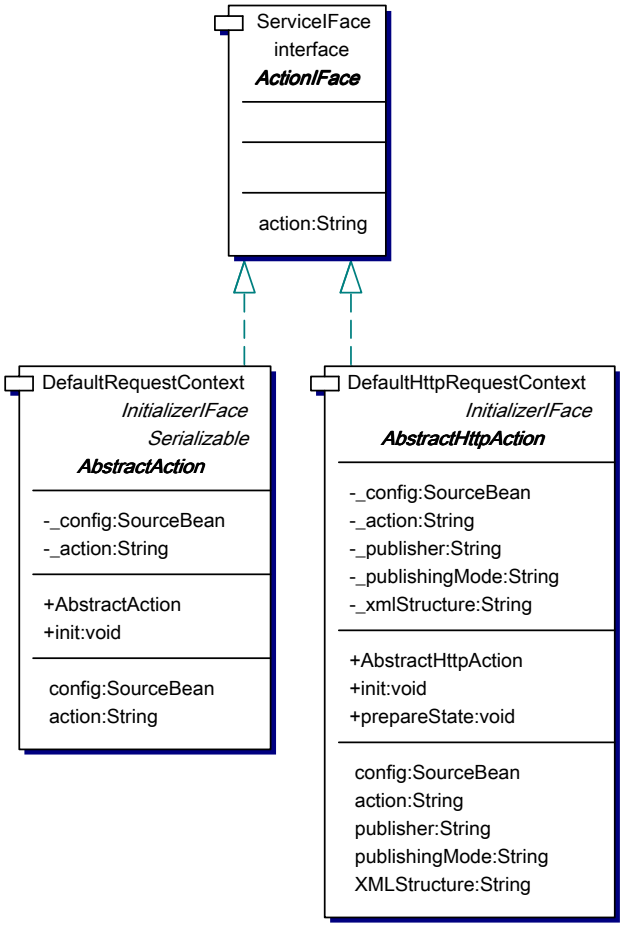
Per ogni action vanno specificate le seguenti informazioni:

- Nome logico: le richieste di servizio devono specificare il nome logico dell'action da invocare.
- Classe: nome della classe che implementa l'action.
- Scope: scope della classe indicata nel parametro precedente. E' possibile specificare i seguenti scope:
  - REQUEST: l'action viene istanziata ad ogni richiesta di servizio.
  - SESSION: l'action viene istanziata una sola volta nell'ambito della sessione.
  - APPLICATION: l'action viene istanziata una sola volta nell'ambito dell'applicazione: più precisamente in questo caso l'action ha lo scope del class loader. Questo significa che se l'applicazione gira su una singola Virtual Machine, allora la classe viene effettivamente istanziata una sola volta, ma se l'applicazione è rilasciata su un cluster, in cui ci sono più web container, allora la classe verrà istanziata più volte, una volta per ciascun web container.

  La modalità consigliata è REQUEST: l'unica controindicazione nell'usare questa modalità è quando nel costruttore dell'action vengano svolte attività computazionalmente pesanti, per cui istanziare ogni volta la classe porterebbe ad un degrado delle performance.

2.5.1.2 Implementazione

Esistono 2 modi per implementare un'action:

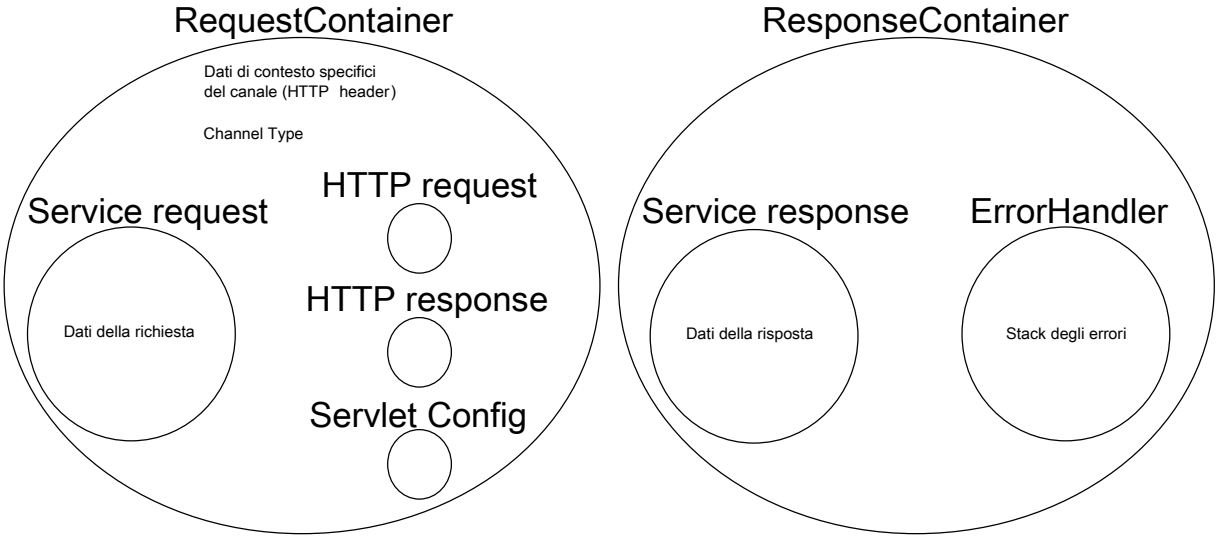


- Implementare l'interfaccia *ActionIFace*.
- Estendere la classe *AbstractAction* o *AbstractHTTPAction*: è il modo più semplice.

In entrambi i casi il metodo principale da implementare è *service*:

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

Questo metodo riceve come parametri la richiesta (sotto forma di SourceBean) e la risposta (sempre come SourceBean): in questo modo, indipendentemente dal canale, l'action può accedere ai parametri della richiesta e valorizzare la risposta con un flusso XML, che verrà poi utilizzato nella parte di presentazione (se presente).
In questo modo la logica di business è indipendente dal canale. La richiesta oltre ad avere accesso ai parametri della richiesta, può anche accedere ai parametri specifici del canale (ad esempio i parametri presenti nell'HTTP header) tramite un oggetto chiamato *RequestContainer*, con il quale si può poi accedere ai parametri presenti in sessione tramite un oggetto chiamato *SessionContainer* .



Supponendo, ad esempio, di aver memorizzato in sessione l'identificativo dell'utente corrente, esso si può recuperare come di seguito:

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

I container a cui ha accesso l'action (request container e response container) sono contenitori single-value (a differenza del SourceBean che è un contenitore multi-value).

Nel Request container la busta più esterna contiene informazioni legate al canale: contiene ad esempio tutto l'header del canale HTTP, le informazioni relative all'utente corrente, il tipo di canale, e per il solo canale HTTP contiene la HTTP request, HTTP response e la ServletConfig.

Il contenitore più interno è la *Service request* che è la richiesta effettiva di servizio, ed è l'oggetto che viene passato all'oggetto di business (l'action o il modulo).

2.5.1.3 Inizializzazione

In fase di inizializzazione di un'action possono essere utilizzate informazioni di configurazione definite nella busta *CONFIG*, nella definizione dell'action. Ecco un esempio:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
        <ACTION name="LISTA_UTENTI_ACTION" class="com.engiweb.demo.action.ListaUtentiAction" scope="REQUEST">
            <CONFIG pool="afdemo" title="Lista Utenti" rows="2">
              <QUERIES>
                <SELECT_QUERY statement="LISTA_UTENTI"/>
                <DELETE_QUERY statement="DELETE_UTENTE"/>
              </QUERIES>
            </CONFIG>
    </ACTION>
```

```
</ACTIONS>
```

Queste informazioni vengono passate al metodo *init* ogni volta che viene istanziata l'action.

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

Tali informazioni sono accessibili sotto forma di *SourceBean*.

2.5.1.4 Specificità del canale

Il framework incapsula i dati provenienti dai vari canali, fornendo la richiesta e la risposta sotto forma di SourceBean. In alcuni casi, però, è necessario poter accedere agli oggetti nativi del canale: ad esempio per implementare un action che effettui l'upload di un file.
In questi casi si può accedere alle informazioni del canale, ma il servizio non sarà multicanale.
Nel caso del canale HTTP si può accedere alle informazioni del canale estendendo la classe *AbstractHTTPAction* invece di *AbstractAction*. In questi casi bisogna fare attenzione a non entrare in conflitto con il meccanismo di presentazione del framework : il framework va informato se l'action si prende anche la responsabilità di produrre la risposta, tramite il metodo *freezeHttpResponse()*.
Al momento questo è l'unico caso che è stato individuato in cui si è presentata l'esigenza di accedere direttamente alle informazioni del canale: il framework non fornisce un meccanismo equivalente per accedere all'EJB context dell'EJB container.

2.5.2 Moduli

La modalità alternativa alle action per gestire le richieste di servizio è quella dei moduli. Questa modalità è più complessa rispetto alla precedente perchè prevede che la costruzione della risposta ad una richiesta di servizio venga evasa da più moduli cooperanti tra loro, organizzati in un'unità logica chiamata PAGE.
D'ora in poi, parlando dei moduli, useremo il termine pagina per indicare questa unità logica, che nulla ha a che vedere con le pagine HTML/JSP.

2.5.2.1 Censimento moduli

I moduli vengono censiti in un file di configurazione chiamato *modules.xml*. Ecco un esempio di tale file:

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

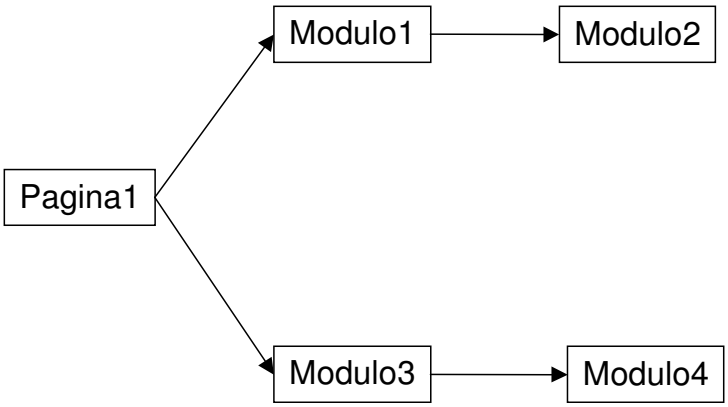
Per ogni modulo vanno specificate le seguenti informazioni:

- Nome logico: le pagine devono specificare il nome logico dei moduli da invocare.
- Classe: nome della classe che implementa il modulo.

Facendo l'analogia con le action, si nota che non è possibile specificare lo scope: questo perchè il modulo eredita lo scope della pagina che lo invoca.

2.5.2.2 Pagina

Una pagina è una composizione logica di moduli: ovvero è un grafo che specifica in che sequenza vanno invocati i moduli e con che parametri.

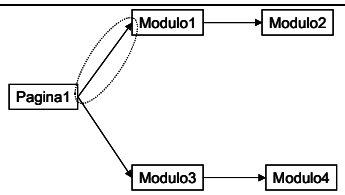


Supponiamo di aver definito un grafo come in figura: il punto di inizio valutazione del grafo è la pagina. L'attività di valutazione del grafo è ricorsiva, e la navigazione del grafo è in profondità: questo significa che i moduli vengono invocati nel seguente ordine: Modulo1, Modulo2, Modulo3, Modulo4.
La risposta all'invocazione del servizio è la concatenazione delle varie risposte dei moduli.

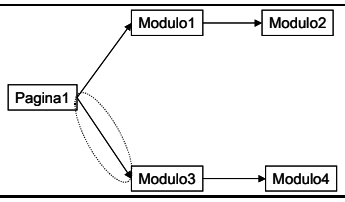
2.5.2.3 Censimento pagine

La definizione della struttura delle pagine viene censita in un file che si chiama *pages.xml*, ad esempio, per il grafo precedente il file sarebbe così composto:

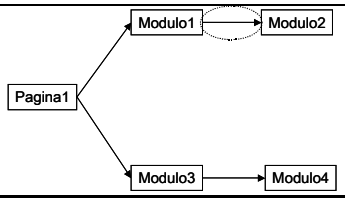
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PAGES>
  <PAGE name="Pagina1" scope="SESSION">
    <MODULES>
      <MODULE name="Modulo1"/>
      <MODULE name="Modulo2"/>
      <MODULE name="Modulo3"/>
      <MODULE name="Modulo4"/>
    </MODULES>
    <DEPENDENCIES>
      <DEPENDENCE source="Pagina1" target="Modulo1">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```



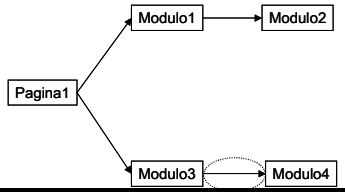

```
      <DEPENDENCE source="Pagina1" target="Modulo3">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```




```
      <DEPENDENCE source="Modulo1" target="Modulo2">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```



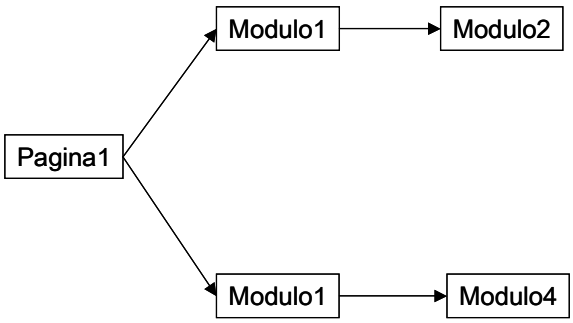

```
      <DEPENDENCE source="Modulo3" target="Modulo4">
        <CONDITIONS>
        </CONDITIONS>
        <CONSEQUENCES/>
      </DEPENDENCE>
```




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

Ogni arco del grafo è una busta *DEPENDENCE* che specifica nell'attributo *source* il nodo di partenza e nell'attributo *target* il nodo di arrivo.

Un modulo può partecipare a stati diversi del grafo. Il seguente grafo, ad esempio, dove il modulo *Modulo1* è presente in più nodi, è perfettamente lecito.



In questi casi, se un modulo viene eseguito più volte, solo l'ultima risposta del modulo viene messa nella risposta della pagina.
Se più moduli devono cooperare tra loro, ad esempio se *Modulo1* deve passare dei dati a *Modulo2*, hanno a disposizione un'area dati comune chiamata *shared data* che è condivisa tra tutti i moduli.

Abbiamo detto in precedenza che il modulo eredita lo scope della pagina a cui appartiene. Questo comportamento può essere bypassato aggiungendo un parametro aggiuntivo chiamato *keep_instance* quando si definisce la pagina.

```
<MODULES>
  <MODULE name="Modulo1" keep_instance="true"/>
  <MODULE name="Modulo2"/>
  <MODULE name="Modulo3"/>
  <MODULE name="Modulo4"/>
</MODULES>
```

La presenza di questo parametro valorizzato a *false* fa in modo che *Modulo1* venga istanziato secondo lo scope della pagina. Questo è il comportamento di default, nel caso in cui il parametro venga omesso.
Se il parametro viene valorizzato a *true*, il modulo viene istanziato a livello di sessione, indipendentemente dallo scope configurato per la pagina.

Un altro parametro che si può specificare a livello di configurazione del modulo nella pagina è *keep_response*: se valorizzato a *true*, permette di riottenere la risposta di un modulo prodotta alla prima invocazione della pagina, nel caso in cui il modulo non venga attivato nelle invocazioni successive della pagina. Il valore di default è *false*. Lo scope della pagina deve essere *SESSION*.

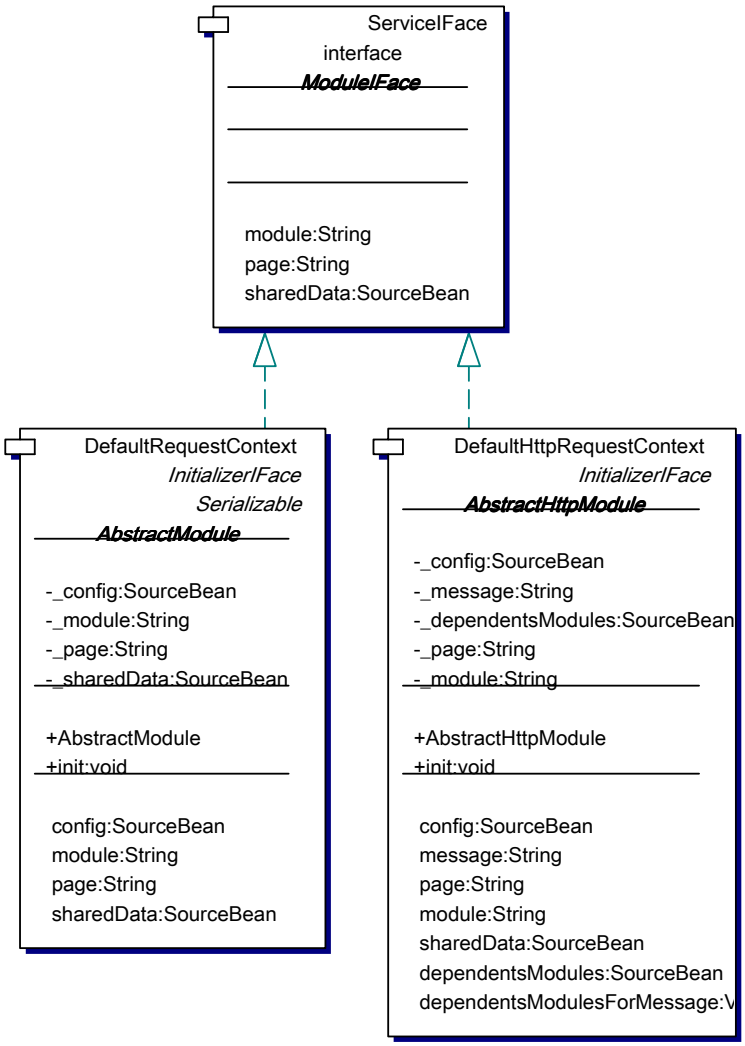
```
<MODULES>
  <MODULE name="Modulo1" keep_response="true"/>
  <MODULE name="Modulo2"/>
</MODULES>
```

Se un modulo viene usato da più pagine, tramite il metodo *getPage()* può sapere il nome logico della pagina in cui sta operando, per eventualmente differenziare il suo operato. Inoltre una stessa classe può essere usata per censire più moduli logici, per cui un modulo ha anche modo di sapere qual'è il suo nome logico.

E' anche possibile censire *DEPENDECIES* che hanno gli stessi nodi come source e target.

2.5.2.4 Implementazione

Esistono 2 modi per implementare un modulo:



- Implementare l'interfaccia *ModuleIFace*.
- Estendere la classe *AbstractModule* o *AbstractHttpModule*: è il modo più semplice.

Come per le action, in entrambi i casi il metodo da implementare è *service*:

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

Anche in questo caso c'è la possibilità di accedere direttamente alle informazioni del canale (per il solo canale HTTP), estendendo il modulo da *AbstractHTTPModule* invece che da *AbstractModule*.

2.5.2.5 Condizioni

Finora abbiamo considerato solo il caso in cui tutti i moduli del grafo vengono invocati, indipendentemente dai parametri presenti nella richiesta, o dai risultati dei moduli precedenti.
Questo comportamento si ottiene mantenendo vuota la busta *CONDITIONS* associata al ramo del grafo che stiamo considerando.

```
<DEPENDENCE source="Pagina1" target="Modulo1">
  <CONDITIONS>
  </CONDITIONS>
  <CONSEQUENCES/>
</DEPENDENCE>
```

Generalmente però l'invocazione dei moduli deve essere effettuata solo se sono verificate determinate condizioni, come ad esempio la presenza di certi parametri nella richiesta, o l'assenza di certi parametri, o la presenza di certi parametri con determinati valori, e così via.
La busta CONDITIONS è una lista di condizioni booleane, che devono essere TUTTE verificate, che assumono la seguente forma:

```
<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>
```

Relativamente allo scope del parametro, il significato dei possibili valori è il seguente:

- USER: permette di verificare le proprietà dell'utente reperite dal modulo di autenticazione (che vedremo successivamente), come ad esempio l'identificativo dell'utente loggato.
- ADAPTER_REQUEST: permette di verificare i parametri presenti nel contesto conversazionale della richiesta, ad esempio nell'HTTP header (per il canale HTTP).
- SERVICE_REQUEST: permette di verificare i parametri presenti nella richiesta di servizio.
- SESSION: permette di verificare i parametri presenti nel session container.
- APPLICATION: permette di verificare i parametri presenti nell'application container.
- ADAPTER_RESPONSE : permette di verificare i parametri presenti nel response container. Attualmente il framework non memorizza nulla in questo container, per cui al momento questo scope non serve.
- SERVICE_RESPONSE : permette di verificare i parametri presenti nella risposta al servizio, valorizzata dai moduli precedenti.
- ERROR : consente di verificare la presenza di errori di tipo Internal o User gestiti dall' Error handler.

Per quanto riguarda il valore il significato è:

- AF_DEFINED: parametro presente nello scope specificato.
- AF_NOT_DEFINED: parametro non presente nello scope specificato.
- ALTRO: parametro presente nello scope specificato con il valore specificato (valore non consentito nel caso di scope ="ERROR").

Una indicazione sul valore dell'attributo "name" relativamente al caso di scope = "ERROR" :

- AF_INFORMATION : tutti gli errori con severity "INFORMATION"
- AF_WARNING : tutti gli errori con severity "WARNING"
- AF_ERROR : tutti gli errori con severity "ERROR"
- AF_BLOCKING : tutti gli errori con severity "BLOCKING"
- ALTRO : se ALTRO è una stringa, questa deve indicare un codice di errore presente tra gli errori di tipo User, ed indica la richiesta di verificare la presenza o meno di errori di tipo User con quel codice; se invece il campo "name" è vuoto, interessano tutti gli errori, sia di tipo User che Internal con qualsiasi severity.

Per lo scope di tipo ERROR quindi valgono le seguenti considerazioni:

<PARAMETER name="" scope="ERROR" value="AF_DEFINED" />
vera se il gestore degli errori contiene almeno un messaggio

<PARAMETER name="" scope="ERROR" value="AF_NOT_DEFINED" />
vera se il gestore degli errori non contiene messaggi

<PARAMETER name="<severity>" scope="ERROR" value="AF_DEFINED" />
vera se il gestore degli errori contiene almeno un messaggio di severity <severity> scelto fra AF_INFORMATION, AF_WARNING, AF_ERROR, AF_BLOCKING

<PARAMETER name="<severity>" scope="ERROR" value="AF_NOT_DEFINED" />
vera se il gestore degli errori non contiene messaggi di severity <severity>

<PARAMETER name="<code>" scope="ERROR" value="AF_DEFINED" />
vera se il gestore degli errori contiene almeno un messaggio di tipo "EMFUserError" con codice <code>

<PARAMETER name="<code>" scope="ERROR" value="AF_NOT_DEFINED" />
vera se il gestore degli errori non contiene messaggi di tipo "EMFUserError" con codice <code>

Ad esempio per verificare nella richiesta la presenza del parametro *module* e la presenza del parametro *user* valorizzato a "ciccio pelliccio" si usano le seguenti condizioni:

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

Per verificare invece la presenza del parametro *sample*, inserito dal modulo *Modulo1* precedente nel grafo di invocazione:

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

2.5.2.6 Condizioni espresse tramite regole

Esiste la possibilità di esprimere le condizioni in modo differente da come descritto nel paragrafo precedente, facendo uso di regole espresse in un qualunque linguaggio.

Tale possibilità è offerta dall'integrazione di Spago con il motore di regole Drools: tale motore permette di esprimere regole arbitrariamente complesse in svariati linguaggi tra cui Java, Groovy, Python, etc.
L'integrazione con Drools è implementata tramite una libreria di Spago aggiuntiva (*spago-rules.jar*), alcune informazioni di configurazione aggiuntive e i file contenenti le regole applicative.
Descriviamo di seguito come utilizzare questa modalità.

1. Censire in un nuovo file di configurazione, chiamato *rules.xml*, i file applicativi delle regole, quelli con estensione .drl come nel seguente esempio:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<RULES>
        <RULE name="SAMPLE_RULE" file="drl/sample.java.drl"/>
</RULES>
```

   A ciascun file di regole va associato un nome logico univoco.

2. Aggiungere al file *master.xml* il nuovo file di configurazione:

```
<CONFIGURATOR path="/WEB-INF/conf/spago/rules.xml" />
```

3. Aggiungere al file *initializer.xml* la seguente sezione:

```
<INITIALIZER class="it.eng.spago.rules.RulesInitializer" config="" />
```

   Tale inizializzatore ha lo scopo di caricare allo startup dell'applicazione tutti i file di regole in modo che possano essere utilizzati successivamente.
4. Aggiungere *spago-rules.jar* e tutte le librerie dipendenti all'applicazione.

Vediamo ora come configurare una pagina in modo da utilizzare le regole. Le parti che variano rispetto alla modalità standard sono:

- La classe corrispondente alla pagina deve essere *it.eng.spago.rules.dispatching.**RulesPage.***
- La busta CONDITIONS, all'interno di ciascuna DEPENDENCE, è sostituita dalla busta RULES in cui è presente l'attributo GROUP_NAME che contiene l'identificatore del file di regole da verificare, come censite nel file *rules.xml*.

Ecco un esempio:

```
<PAGE name="TEST_RULES" scope="SESSION"
                        class="it.eng.spago.rules.dispatching.RulesPage">
        <MODULES>
                <MODULE name="MODULE1" />
                <MODULE name="MODULE2" />
        </MODULES>
        <DEPENDENCIES>
                <DEPENDENCE source="TEST_RULES" target="MODULE1">
                        <CONSEQUENCES />
                </DEPENDENCE>
                <DEPENDENCE source="MODULE1" target="MODULE2">
                        <RULES GROUP_NAME="SAMPLE_RULE"/>
                        <CONSEQUENCES />
                </DEPENDENCE>
        </DEPENDENCIES>
</PAGE>
```

Nel nostro esempio, il file delle regole (*sample.java.drl*) è definito come segue:

```
<?xml version="1.0"?>

<rule-set name="SampleRuleSet"
        xmlns="http://drools.org/rules"
        xmlns:java="http://drools.org/semantics/java"
        xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:schemaLocation="http://drools.org/rules rules.xsd
                http://drools.org/semantics/java java.xsd">

  <import>it.eng.spago.base.RequestContainer</import>
  <import>it.eng.spago.base.ResponseContainer</import>

  <rule name="Check parameter in session">
    <parameter identifier="requestContainer">
      <class>RequestContainer</class>
    </parameter>
    <parameter identifier="responseContainer">
      <class>ResponseContainer</class>
    </parameter>

    <java:condition>requestContainer.getSessionContainer().getAttribute("param1") !=
null</java:condition>
    <java:consequence/>
  </rule>

  <rule name="Check parameter in request">
    <parameter identifier="requestContainer">
      <class>RequestContainer</class>
    </parameter>
    <parameter identifier="responseContainer">
      <class>ResponseContainer</class>
    </parameter>

    <java:condition>requestContainer.getServiceRequest().getAttribute("param2") != null</java:condition>
    <java:consequence/>
  </rule>

</rule-set>
```

La prima parte del file contiene le direttive:

```
<import>it.eng.spago.base.RequestContainer</import>
<import>it.eng.spago.base.ResponseContainer</import>
```

Questo perché Spago mette a disposizione del motore di regole gli oggetti RequestContainer e ResponseContainer. Pertanto tali direttive vanno incluse sempre in tutti i file di regole.

Questo permette, nella prima regola di esempio, di verificare la presenza di un attributo in sessione:

```
<rule name="Check parameter in session">
  <parameter identifier="requestContainer">
    <class>RequestContainer</class>
  </parameter>
  <parameter identifier="responseContainer">
    <class>ResponseContainer</class>
  </parameter>

  <java:condition>requestContainer.getSessionContainer().getAttribute("param1") != null</java:condition>
  <java:consequence/>
</rule>
```

La sezione relativa ai parametri *requestContainer* e *responseContainer* è fissa e deve essere riportata in tutte le regole

```
<parameter identifier="requestContainer">
  <class>RequestContainer</class>
</parameter>
```

```
<parameter identifier="responseContainer">
  <class>ResponseContainer</class>
</parameter>
```

Analogamente, la seconda regola di esempio verifica la presenza di un attributo in request:

```
<java:condition>requestContainer.getServiceRequest().getAttribute("param2") != null</java:condition>
```

In ogni regola la busta delle consequence deve essere vuota:

```
<java:consequence/>
```

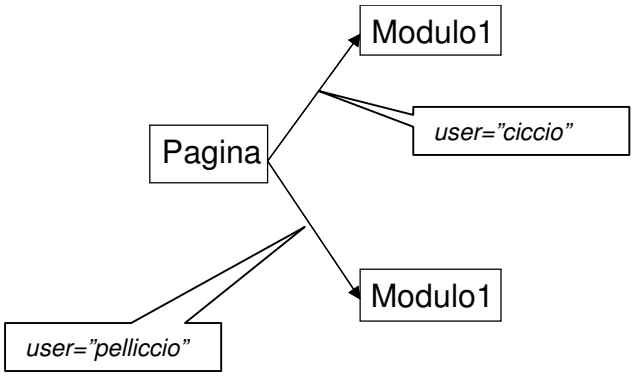
Drools richiede che la busta sia presente, ma Spago non richiede che sia specificata alcuna consequence, in quanto la consequence è l'esecuzione di un modulo, che viene gestita da Spago e non da Drools.

L'esempio qui descritto utilizza regole espresse in Java, ma nulla vieta di utilizzare qualunque altro linguaggio supportato da Drools.

2.5.2.7 Condizioni in OR

Le *DEPENDENCIES* valutano le condizioni in AND, ovvero tutte le condizioni devono essere verificate. Se si vogliono definire condizioni in OR, vanno inserite nuove transizioni.

Ad esempio se voglio verificare che il parametro *user* valga "*ciccio*" o *"pelliccio"* devo inserire due rami nel grafo, in cui censisco entrambe le condizioni.



Ecco come si presenterebbe l'XML di configurazione di questa pagina.

```
<PAGE name="Pagina" scope="SESSION">
  <MODULES>
    <MODULE name="Modulo1"/>
  </MODULES>
  <DEPENDENCIES>
    <DEPENDENCE source="Pagina" target="Modulo1">
      <CONDITIONS>
          <PARAMETER name="user" scope="SERVICE_REQUEST" value="ciccio"/>
```

```
            </CONDITIONS>
            <CONSEQUENCES/>
          </DEPENDENCE>
```

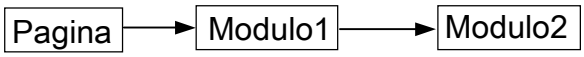
```
          <DEPENDENCE source="Pagina" target="Modulo1">
            <CONDITIONS>
                <PARAMETER name="user" scope="SERVICE_REQUEST" value="pelliccio"/>
            </CONDITIONS>
            <CONSEQUENCES/>
          </DEPENDENCE>
```

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

2.5.2.8 Risposta dei moduli

I parametri impostati nella risposta dai moduli vengono inseriti all'interno di una busta XML che ha lo stesso nome logico del modulo.

Se ad esempio viene eseguita una pagina configurata come segue:



ed entrambi i moduli impostano nella risposta un attributo, come nel seguente esempio:

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

la risposta prodotta dalla pagina ha una struttura del tipo:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE RESPONSE (View Source for full doctype...)>
<RESPONSE>
        <SERVICE_RESPONSE>
                <MODULO1 test=" Stringa di test"/>
                <MODULO2 test=" Stringa di test"/>
        </SERVICE_RESPONSE>
<ERRORS/>
</RESPONSE>
```

2.5.2.9 Consequences

Una volta stabilito quali sono i rami del grafo e quali sono le condizioni associate ai rami, è possibile stabilire quali sono i parametri aggiuntivi da passare ai vari moduli target, nel caso in cui la transizione venga attivata, oltre a quelli già presenti nella richiesta. Posso inserire parametri con valori fissi, o parametri con valori presi da diversi scope.

Questa attività si effettua valorizzando opportunamente la busta *CONSEQUENCES* all'interno della busta *DEPENDENCE*.

```
< CONSEQUENCES >
 <PARAMETER
    name="nome parametro"
    type="ABSOLUTE | RELATIVE"
    scope=" USER | ADAPTER_REQUEST | SERVICE_REQUEST | SESSION  | APPLICATION | ADAPTER_RESPONSE |
                SERVICE_RESPONSE"
    value="valore parametro"/>
</ CONSEQUENCES >
```

I possibili valori dell'attributo *type* significano:

- ABSOLUTE: indica che il parametro specificato assumerà il valore definito nell'attributo *value*. Solo in questo caso l'attributo *value* deve essere specificato. In questo caso non va specificato il parametro *scope*. Ad esempio per passare il parametro *user* impostato al valore *"ciccio pelliccio"* si utilizzerebbe una consequence del tipo:

```
< CONSEQUENCES >
 <PARAMETER name="user" type="ABSOLUTE" value="ciccio pelliccio"/>
</ CONSEQUENCES >
```

- RELATIVE: indica che il parametro il cui nome è specificato da *name* assumerà il valore del parametro specificato da *value* nello scope indicato da *scope*.
  Ad esempio per passare il parametro *utente* impostato al valore del parametro *user* reperito dalla sessione si utilizzerebbe una consequence del tipo:

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

I parametri definiti nelle consequences sono accessibili dal modulo tramite il SourceBean della richiesta, ad esempio per accedere ad un parametro così definito:

```
< CONSEQUENCES >
 <PARAMETER name="user" type="ABSOLUTE" value="ciccio pelliccio"/>
</ CONSEQUENCES >
```

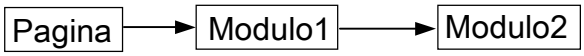
il modulo utilizza un'istruzione del tipo:

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

Nella richiesta sono presenti pertanto, oltre a tutti i parametri definiti nell'URL di invocazione, anche tutti i parametri definiti nelle consequences.

2.5.2.10 Cooperazione tra moduli

Lo scopo principale delle consequences è quello di permettere lo scambio di informazioni tra i moduli.
Supponiamo, ad esempio, di aver configurato una pagina nel seguente modo:



Se il *Modulo2* ha necessità di accedere ad informazioni inserite nella risposta dal *Modulo1* è necessario introdurre una consequence relativa all'arco tra *Modulo1* e *Modulo2*:

```
<CONSEQUENCES>
 <PARAMETER name="parametroModulo1" type="RELATIVE" scope="SERVICE_RESPONSE" value="Modulo1.esempio"/>
</CONSEQUENCES>
```

in questo modo, se *Modulo1* ha inserito nella risposta un parametro chiamato *esempio* con un'istruzione del tipo:

```
void service(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception {
        .....
         serviceResponse.setAttribute("esempio", "Stringa di esempio");
        ....
}
```

il Modulo2 ha accesso a tale parametro nel seguente modo:

```
void service(SourceBean serviceRequest, SourceBean serviceResponse) throws Exception {
        .....
        String param = serviceRequest.getAttribute("parametroModulo1");
        ....
}
```

Nel caso le consequence non siano sufficienti a far cooperare tra loro moduli diversi, è possibile utilizzare un'area di memoria condivisa tra tutti i moduli, chiamata *shared data*.

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

Tramite il metodo *getSharedData* è possibile accedere ad un SourceBean in cui i moduli possono memorizzare informazioni di qualunque tipo. Quest'area di memoria viene ricreata ad ogni richiesta di servizio.
Ovviamente l'uso di quest'area di memoria aumenta l'accoppiamento tra i moduli (ossia ne diminuisce la possibilità di riuso), per cui quando è possibile, è preferibile utilizzare il meccanismo delle consequence per scambiare informazioni tra i moduli.

2.5.2.11 Grafi

E' possibile definire nel file di configurazione *graphs.xml* **,** dei template di grafi di elaborazione delle unità logiche "page", nel file *pages.xml* verrà censita la specializzazione del grafo. Vediamo un esempio:

```
<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>
```

Il template del grafo ha identificativo "1", è composto di due nodi (moduli) e di 3 rami (dipendenze); le dipendenze "source" - "target" sono espresse in questo caso in base agli identificativi (alias) dei moduli; la configurazione di una "page" che utilizza questo template potrebbe essere:

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

in tal caso la configurazione del grafo della pagina è basata sul template "1", in cui il modulo di identificativo "1" è "BasicListaUtenti", quello di identificativo "2" è "DettaglioUtente", e la dipendenza con identificativo "2" è stata ridefinita come ramo dal nodo "2" al nodo "1"; questo meccanismo ricorda quello di " overriding" dei servizi di un oggetto in cui è possibile

ridefinire in un oggetto i servizi ereditati da un altro oggetto; in questo caso le proprietà ridefinibili sono i nodi (moduli) e i rami (dipendenze) del grafo modello

2.5.2.12 Gestione delle eccezioni

Se il metodo *service* di un modulo genera un'eccezione la valutazione della pagina viene interrotta, ossia i moduli successivi non vengono invocati.
Se non si vuole che in questi casi la valutazione del grafo della pagina venga interrotta, le eccezioni vanno gestite dal modulo ed eventualmente trasformate in *EMFUserError* o *EMFInternalError* e aggiunte all'*ErrorHandler* in modo da farli pervenire (eventualmente) fino alla presentazione.

2.5.2.13 Considerazioni

La modalità di dispatching a moduli è estremamente più complessa di quella ad action, però è estremamente flessibile e potente. Va utilizzata quando in fase di analisi si riescono ad identificare delle componenti riutilizzabili in più servizi.
In questi casi, una volta scritte le componenti di base, il lavoro consiste nel comporre i grafi corrispondenti alle pagine.

2.6 Eventi sui Servizi Applicativi

Quando viene eseguito un servizio applicativo all'interno di Spago vengono sollevati degli eventi legati all'esecuzione dei servizi. E' possibile (anche se non obbligatorio) istruire le applicazioni sviluppate con Spago per gestire gli eventi lanciati da Spago attraverso la configurazione di opportuni listener.

Il meccanismo di notifica degli eventi di Spago prevede che:

1. Le classi che rappresentano gli eventi implementino l'interfaccia *it.eng.spago.events.**ISpagoEvent*** che definisce il metodo public *Object getSource()* per ottenere informazioni sulla causa dell'evento.
2. Gli eventi una volta instanziati siano notificati ai listener applicativi attraverso degli oggetti responsabili per la notifica degli eventi. Questi oggetti devono implementare l'interfaccia *it.eng.spago.events.**IEventNotifier*** che definisce il metodo:

   *public void **notifyEvent**(ISpagoEvent event, RequestContainer requestContainer, ResponseContainer responseContainer);*

3. Gli eventi siano gestiti da classi applicative denominate Event Listener che devono implementare l'interfaccia *it.eng.spago.listeners.**ISpagoListener*** che definisce l'unico metodo:

   *public void **onEvent**(ISpagoEvent event, RequestContainer requestContainer, ResponseContainer responseContainer);*

Gli eventi attivati da spago durante il dispatching dei servizi sono i seguenti:

- ***ActionStartEvent(it.eng.spago.event.ActionStartEvent):*** Questo evento viene lanciato da Spago, quando inizia l'esecuzione di un servzio applicativo di tipo ACTION.

- ***ActionEndEvent(it.eng.spago.event.ActionEndEvent):*** Questo evento viene lanciato da Spago, quando l'esecuzione di un servzio applicativo di tipo ACTION termina con successo.
- ***ActionExceptionEvent(it.eng.spago.event.ActionExceptionEvent):*** Questo evento viene lanciato da Spago quando l'esecuzione di un servzio applicativo di tipo ACTION produce delle eccezioni.
- ***PageStartEvent(it.eng.spago.event.PageStartEvent):*** Questo evento viene lanciato da Spago, quando inizia l'esecuzione di un servzio applicativo di tipo PAGE.
- ***PageEndEvent(it.eng.spago.event.PageEndEvent):*** Questo evento viene lanciato da Spago, quando l'esecuzione di un servzio applicativo di tipo PAGE termina con successo.
- ***PageExceptionEvent(it.eng.spago.event.PageExceptionEvent):*** Questo evento viene lanciato da Spago quando l'esecuzione di un servzio applicativo di tipo PAGE produce delle eccezioni.
- ***ModuleStartEvent(it.eng.spago.event.ModuleStartEvent):*** Questo evento viene lanciato da Spago, quando inizia l'esecuzione di un servzio applicativo di tipo MODULE.
- ***ModuleEndEvent(it.eng.spago.event.ModuleEndEvent):*** Questo evento viene lanciato da Spago, quando l'esecuzione di un servzio applicativo di tipo MODULE termina con successo.
- ***ModuleExceptionEvent(it.eng.spago.event.ModuleExceptionEvent):*** Questo evento viene lanciato da Spago quando l'esecuzione di un servzio applicativo di tipo MODULE produce delle eccezioni.

La configurazione per la gestione degli eventi avviene attraverso la busta EVENTS nel file events.xml di cui vediamo un esempio:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<EVENTS>
        <EVENTS-NOTIFICATION-ENABLED>true</EVENTS-NOTIFICATION-ENABLED>

        <EVENTS-NOTIFIER notifierClass="it.eng.spago.event.SynchronousEventNotifier"/>

        <EVENTS-LISTENER listenerClass="it.eng.spago.sample.SimpleActionListener"/>

</EVENTS>
```

La busta EVENTS contiene:

- Un elemento *EVENTS-NOTIFICATION-ENABLED che* deve essere valorizzato con **true** o **false** per indicare a spago se si vuole gestire la notifica degli eventi ai listener applicativi.
  Nel caso la busta EVENTS non sia presente o per qualche motivo questo attributo non sia disponibile, la notifica viene disabilitata.

- Un elemento *EVENTS-NOTIFIER* che contiene nell'attributo *notifierClass* il nome di una classe di spago che si occupa di notificare i listener registrati alla verifica di un evento.

- Una lista di uno o più elementi *EVENTS-LISTENER* che identificano attraverso l'attributo *listenerClass* le classi che vengono notificate quando spago lancia un evento.

Per implementare un listener applicativo in spago è quindi sufficiente scrivere una classe che implementi l'interfaccia *ISpagoListener.* Un semplice esempio di listener è essere il seguente:

```
package it.eng.spago.sample;
```

```
import it.eng.spago.base.RequestContainer;
import it.eng.spago.base.ResponseContainer;
import it.eng.spago.dispatching.action.ActionIFace;
import it.eng.spago.event.ActionEndEvent;
import it.eng.spago.event.ActionEvent;
import it.eng.spago.event.ActionExceptionEvent;
import it.eng.spago.event.ActionStartEvent;
import it.eng.spago.event.IExceptionEvent;
import it.eng.spago.event.ISpagoEvent;
import it.eng.spago.listeners.ISpagoListener;

public class SimpleActionListener implements ISpagoListener {

        public void onEvent(ISpagoEvent event,
    RequestContainer requestContainer, ResponseContainer responseContainer) {

                if (event instanceof ActionEvent) {
                        handleActionEvent(event,requestContainer, responseContainer);
                }else{
                        System.out.println("The event raised is not an Action event");
                }

        }

        public void handleActionEvent(ISpagoEvent event,
RequestContainer requestContainer,
 ResponseContainer responseContainer){
                if (event instanceof ActionStartEvent) {
                        System.out.println("**************** Action "
+ ((ActionIFace)event.getSource()).getActionName() +
" EXECUTION STARTED");
                }if (event instanceof ActionEndEvent) {
                        System.out.println("**************** Action "
+ ((ActionIFace)event.getSource()).getActionName() + " EXECUTION FINISHED");
                }if (event instanceof ActionExceptionEvent) {
                        System.out.println("**************** Action "
+ ((ActionIFace)event.getSource()).getActionName()
+ " EXCEPTION OCCURED DURING EXECUTION OF");
                        System.out.println("**************** Exception is ************"
+ ((IExceptionEvent)event).getException() );
                }
        }

}
```

2.7 Validazione dei dati

La validazione dei dati introdotti dall'utente è uno dei problemi più comuni da risolvere nello sviluppo di un'applicazione Web. Per facilitare tale compito è stato sviluppato un servizio che permette di automatizzare parzialmente le operazioni di validazione dei dati.

L'utilizzo del componente è opzionale e configurabile: si possono avere pagine che utilizzano la validazione e pagine che non ne fanno uso. La validazione può essere definita bloccante o meno; in questo secondo caso la logica di business viene eseguita ugualmente.

Il componente effettua le validazioni lato server: questo significa che al momento bisogna scrivere del Javascript ad hoc se si vogliono effettuare delle validazioni lato client in modo da ridurre la comunicazione con il server.

Il componente fornisce due tipi di validazione:

1. Validazione automatica dei tipi di dati più comuni.
2. Validazioni effettuate da classi Java scritte ad hoc.

2.7.1 Configurazione

In spago la configurazione della validazione prevede la creazione di due file di configurazione:

- Il file *field-validators.xml* per la configurazione dei Field Validator elementari.
- Il file *validation.xml* contenente le regole di validazione associate ai servizi applicativi

2.7.1.1 Configurazione dei Field Validator

Un Field Validator è una classe Java che ha la responsabilità di:

- Effettuare la validazione di un singolo campo della richiesta applicativa.
- Effettuare opzionalmente una conversione di tipo del valore della richiesta se la validazione ha successo
- Salvare opzionalmente in una sottobusta (TYPED_SERVICE_REQUEST) della richiesta, il valore originale o convertito con un alias.

I Field Validator sono associati al campo da validare attraverso l'attributo TYPE della busta FIELD nel file di configurazione validation.xml (come verrà spiegato nella sezione successiva).

Per definire un nuovo Field Validator è necessario definire una classe Java che derivi dalla classe *it.eng.spago.validation.fieldvalidators.**AbstractFieldValidator,*** e associare la classe implementata nel file di configurazione *fieldvalidators.xml* ad un tipo.

Spago mette a disposizione una serie di validatori già implementati associati ai tipi che è possibile specificare di default nelle buste FIELD presenti nel file *validation.xml.*

Il file fieldvalidators.xml con la definizione, dei validatori standard di spago ha la struttura seguente:

```
<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>
```

Come si può notare dalla struttura della busta *FIELD-VALIDATORS* ogni validatore può utilizzare una sezione di configurazione *CONFIG* che può essere recuperata attraverso il metodo: ***public SourceBean FieldValidatorIFace.getConfig().***

I Tipi e i Field Validator già implemenati in spago sono i seguenti:

- **1 o GENERIC (it.eng.spago.validation.fieldvalidators.GenericFieldValidator)**
  Questo validatore non effettua controlli sintattici e non esegue nessuna conversione di tipo. Questo validatore viene utilizzato qualora ad esempio su di un campo si voglia fare solo un controllo sul massimo numero di caratteri. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **2 o FISCALCODE: (it.eng.spago.validation.fieldvalidators.FiscalCodeValidator)**
  Questo validatore verifica se il campo è un codice fiscale o una partita iva secondo le leggi italiane. Non viene effettauata nessuna conversione di tipo. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **3 o EMAIL: email (it.eng.spago.validation.fieldvalidators.EMailValidator)**
  Verifica che il campo sia una email valida. Non viene effettuata nessuna conversione di tipo. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **4 o URL: indirizzo sito internet. (it.eng.spago.validation.fieldvalidators.URLValidator)**
  Verifica che il campo sia un URL valido. Se la validazione ha successo viene effettuata la conversione al tipo *java.net.URL.* Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **5 o DATE: data (it.eng.spago.validation.fieldvalidators.DateValidator)**
  Verifica che un campo sia una data valida secondo il formato specificato nell'attributo *DATEFORMAT.dateFormat* della busta CONFIG associata al validatore. Se la busta *config* non viene trovata il validatore effettua il controllo

utilizzando il formato italiano *dd/MM/yyyy*. Se la validazione ha successo viene effettuata la conversione al tipo *java.util.Date*. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.

- **6 o NUMERIC: campo numerico (it.eng.spago.validation.fieldvalidators.NumericValidator)**
  Verifica che il campo sia un campo numerico valido, secondo il formato numerico dipendente dal paese e dal linguaggio definiti nella busta *CONFIG* attraverso gli attributi *COUNTRY.country* e *LANGUAGE.language*. Se la busta config non viene trovata vengono usate di default le impostazioni italiane. E' accettato il segno del numero. Se la validazione ha successo viene effettuata la conversione al tipo *java.lang.Double.* Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **7 o ALFANUMERIC: campo alfanumerico (it.eng.spago.validation.fieldvalidators.AlphaNumericValidator)**
  Verifica che il campo sia composto solo da caratteri che rappresentino lettere alfabetiche o numeri. Non viene effettuata nessuna conversione di tipo. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **8 o LETTERSTRING: campo letterale (it.eng.spago.validation.fieldvalidators.LetterStringValidator)**
  Verifica che il campo sia composto solo da caratteri che rappresentino lettere alfabetiche. Non viene effettuata nessuna conversione di tipo. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- **10 o REGEXP: campo regular-expression (it.eng.spago.validation.fieldvalidators.RegExpValidator)**
  il campo viene validato tramite la regular expression specificata nell'attributo "REGEXP" della busta FIELD. Non viene effettuata nessuna conversione di tipo. Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.
- ***11 o DECIMAL* (it.eng.spago.validation.fieldvalidators.DecimalValidator)**
  Il campo viene validato come campo importo con numero di cifre decimali pari a quelle specificate nell'attributo "DECIMALS", anche in questo caso , il formato numerico è dipendente dal paese e dal linguaggio definiti nella busta *CONFIG* attraverso gli attributi *COUNTRY.country* e *LANGUAGE.language* e se la busta config non viene trovata vengono usate di default le impostazioni italiane. E' accettato il segno del numero. Se la validazione ha successo viene effettuata la conversione al tipo *java.lang.Double.* Il campo viene salvato nella sottobusta TYPED_SERVICE_REQUEST eventualmente con un alias.

E' importante comunque sottolineare che la definizione dei validatori attraverso classi Java e l'associazione ai campi attraverso l'attributo TYPE rende il meccanismo di validazione di spago potente e flessibile in quanto è possibile:

- Definire nuovi validatori e o nuovi tipi, implementando semplici classi Java.
- Cambiare l'implementazione di un validatore associata ad un tipo.
- Cambiare a livello di configurazione la logica di validazione associata ad un tipo.

2.7.1.2 Configurazione della Validazione per i servizi applicativi

Per attivare la validazione dei dati presenti in una pagina bisogna creare una nuova busta di configurazione, nel file *validation.xml*.

Per ciascuna pagina va creata una busta VALIDATION contenente le indicazioni sui campi da validare:

```
<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="Nome" />
                        <FIELD name="age" maxLength="5" mandatory="false" type="6" />
                        <FIELD name="department" maxLength="20" mandatory="false" type="7"
                                aliasAfterValidation ="Dipartimento" />
                    </FIELDS>
                </VALIDATION>
        </SERVICE>
</VALIDATIONS>
```

La busta SERVICE ha i seguenti attributi:

- NAME: nome logico della action o page a cui applicare le validazioni.
- TYPE: assume i seguenti valori: ACTION o PAGE. Indica la modalità con cui viene eseguito il servizio

All'interno della busta SERVICE deve essere configurata la busta VALIDATION con gli attributi:

- VALIDATORS: lista dei nomi logici dei validatori Java da applicare alla pagina. I validatori Java verranno spiegati nel prossimo paragrafo. E' un parametro opzionale.
- BLOCKING: indica se eseguire o meno il servizio in caso di errori durante la fase di validazione

All'interno della busta VALIDATION devono essere configurate le buste CONDITION: le buste VALIDATION verranno valutate in sequenza, e solo quelle per cui le condizioni specificate nella busta CONDITION sono soddisfatte, saranno applicate. Se la busta CONDITION viene omessa, le condizioni si suppongono automaticamente soddisfatte.
Si possono così avere validazioni effettuati solo in determinate condizioni, come nel seguente caso:

```
        <SERVICE name="insertEmployee" type="ACTION">
                <VALIDATION blocking="true" validators="">
                    <CONDITIONS>
                        <PARAMETER name="TEST" scope="SERVICE_REQUEST" value="1" />
                    </CONDITIONS>
                    <FIELDS>
                        <FIELD name="A" maxLength="10" mandatory="true" type="8" />
                        <FIELD name="B" maxLength="5" mandatory="false" type="6" />
                        <FIELD name="C" maxLength="20" mandatory="false" type="7"  />
                    </FIELDS>
                </VALIDATION>
                <VALIDATION blocking="true" validators="">
                    <CONDITIONS>
                        <PARAMETER name="TEST" scope="SERVICE_REQUEST" value="2" />
                    </CONDITIONS>
                    <FIELDS>
                        <FIELD name="D" maxLength="10" mandatory="true" type="8" />
                        <FIELD name="E" maxLength="5" mandatory="false" type="6" />
                        <FIELD name="F" maxLength="20" mandatory="false" type="7" />
                    </FIELDS>
                </VALIDATION>
        </SERVICE>
```

In questo caso se il parametro TEST vale "1" vengono validati i campi A, B, e C, mentre se il campo vale "2" vengono validati i campi D, E e F.

La busta FIELDS contiene le dichiarazioni dei campi che si intendono verificare.
Ogni busta FIELD contiene i seguenti attributi:

- NAME: nome del campo come impostato nel form della pagina.
- TRIM: "true" | "false", se impostato a TRUE elimina gli spazi iniziali e finali del campo.
- TOUPPERCASE: "true" | "false", se impostato a TRUE trasforma il campo in maiuscolo.

- TYPE: tipo del campo. Il valore di questo attributo determina il Field Validator elementare da applicare al campo. Viene cercato nella busta *FIELD-VALIDATORS* l'elemento *FIELD-VALIDATOR* che ha come attributo *fieldType* il valore uguale a quello specificato. Se non viene specificato il valore di default è "GENERIC".
- MANDATORY: "true" | "false", se valorizzato a TRUE indica che il campo è obbligatorio. La validazione viene effettuata solo se il valore è presente in request.
- STRICTMANDATORY: "true" | "false" se valorizzato a TRUE indica che il campo è obbligatorio. La validazione viene effettuata anche se il valore non è presente in request.
- MAXLENGTH: "numero intero" indica l'eventuale limite sul numero di caratteri di cui è composto il campo.
- REGEXP: espressione regolare.
- Per i campi di tipo DECIMAL:
  - DECIMALS: "numero intero", numero di decimali utilizzato.
- Per i campi di tipo NUMERIC, DECIMAL e DATE:
  - MIN_VALUE: valore minimo del campo, espresso secondo il formato della lingua che si aspetta il validatore.
  - MAX_VALUE: valore massimo del campo, espresso secondo il formato della lingua che si aspetta il validatore.
- ALIASAFTERVALIDATION: è il nome attraverso il quale il campo è salvato (eventualmente dopo una conversione di tipo) nella sottobusta TYPED_SERVICE_REQUEST.

Gli attributi di validazione possono essere specificati anche a livello di configurazione di field validator (*fieldvalidators.xml*), permettendo così di creare nuovi "sotto-tipi" basati sui tipi base, come nel seguente esempio:

```
<FIELD-VALIDATOR fieldType="DATE" fieldValidatorClass="it.eng.spago.validation.fieldvalidators.DateValidator">
    <CONFIG maxLength="30">
        <DATE-FORMAT dateFormat="dd/MM/yyyy"/>
    </CONFIG>
</FIELD-VALIDATOR>
```

Come potete notare, l'attributo *maxLength* è specificato direttamente nella configurazione del validatore di campo, in modo che si possa omettere tale informazione nella configurazione del singolo campo, come nel seguente esempio:

```
<SERVICE name="TEST_VALIDATION" type="ACTION">
    <VALIDATION blocking="true" validators="">
        <FIELDS>
            ......
            <FIELD name="DATESAMPLE" type="DATE"/>
            ......
        </FIELDS>
    </VALIDATION>
</SERVICE>
```

E' possible definire gli attributi di configurazione anche a livello di campo, come nel seguente esempio:

```
<SERVICE name="TEST_VALIDATION" type="ACTION">
    <VALIDATION blocking="true" validators="">
```

```
            <FIELDS>
                ......
                <FIELD name="DATESAMPLE" maxLength="30" type="DATE"/>
                ......
            </FIELDS>
        </VALIDATION>
    </SERVICE>
```

Se un campo non supera una validazione, viene inserito il corrispondente errore nell'error handler con l'indicazione del campo che ha prodotto l'errore. In particolare nell'error handler viene inserito un oggetto di tipo *it.eng.spago.validation* .***EMFValidationError.***
Attraverso questa classe è possibile richiedere il nome del campo che ha generato l'errore attraverso il metodo: *EMFValidationError.getFieldName().*

Il range riservato agli errori di validazione nel file dei messaggi è da 10100 a 20000.

2.7.1.3 Valori di default

E' possibile definire dei valori di default dei campi, se l'utente non specifica alcun valore.
Il valore di default va specificato tramite l'attributo *DEFAULT_VALUE* che deve essere espresso con la sintassi prevista dal validatore. Ecco un esempio di tale configurazione:

```
<FIELD name="DATEBIRTH" maxLength="10" type="DATE" DEFAULT_VALUE="18/08/2006"/>
```

I valori di default possono anche essere definiti a livello di configurazione del validatore di campo, come nel seguente esempio:

```
<FIELD-VALIDATOR fieldType="DATE" fieldValidatorClass="it.eng.spago.validation.fieldvalidators.DateValidator">
  <CONFIG DEFAULT_VALUE="30/12/1968">
    <DATE-FORMAT dateFormat="dd/MM/yyyy"/>
  </CONFIG>
</FIELD-VALIDATOR>
```

2.7.1.4 Messaggi degli errori di validazione

Spago fornisce dei messaggi di default per gli errori di validazione prodotti dai validatori built-in. Tali messaggi possono essere personalizzati in due modi:

1. Il modo più semplice è quello di cambiare i file di properties contenenti i messaggi.
2. Una modalità più flessibile è quella di definire un nuovo codice d'errore per il messaggio da modificare.
   Per esempio, nel validatore decimale possono essere modificati le seguenti tipologie di errore:
   ERROR_CODE, ERROR_CODE_MAXVALUE, ERROR_CODE_MINVALUE, ERROR_CODE_MAX_DECIMALS, ERROR_CODE_REQUIRED_DECIMALS. Ecco una configurazione di esempio:

```
<FIELD NAME="NUMERIC" maxLength="30" type="NUMERIC"
    MIN_VALUE="5" MAX_VALUE="30"
    ERROR_CODE_MAXVALUE="CUSTOM_ERROR_CODE"/>
```

La seguente tabella riassume gli identificatori degli errori prodotti dai validatori di Spago, e gli shortcut disponibili per tali messaggi (gli shortcut utilizzati sono evidenziati in grassetto).

Validatore	Identificatore errore	Codice	Shortcut
ALFANUMERIC	ERROR_CODE	10107	%0 = Nome campo %1 = **Etichetta campo**
DATE	ERROR_CODE	10105	%0 = Nome campo %1 = **Etichetta campo**
DECIMAL	ERROR_CODE	10106	%0 = Nome campo %1 = **Etichetta campo**
	ERROR_CODE_MAXVALUE	10118	%0 = Nome campo %1 = **Etichetta campo** **%2 = Valore massimo**
	ERROR_CODE_MINVALUE	10119	%0 = Nome campo %1 = **Etichetta campo** **%2 = Valore minimo**
	ERROR_CODE_MAX_DECIMALS	10114	%0 = Nome campo %1 = **Etichetta campo** **%2 = decimal number**
	ERROR_CODE_REQUIRED_DECIMALS	10111	%0 = Nome campo %1 = **Etichetta campo** **%2 = Numero decimali previsto**
	ERROR_CODE_MIN_MAXVALUE	10120	%0 = Nome campo %1 = **Etichetta campo** **%2 = Valore minimo** **%3 = Valore massimo**
EMAIL	ERROR_CODE	10103	%0 = Nome campo %1 = **Etichetta campo**
FISCALCODE	ERROR_CF_CODE	10101	%0 = Nome campo %1 = **Etichetta campo**
	ERROR_PI_CODE	10102	
GENERIC	Nessuna validazione effettuata.		
IMPORTO	Vedere DECIMAL		

LETTERSTRING	ERROR_CODE	10108	%0 = Nome campo %1 = **Etichetta campo**
NAME	ERROR_CODE	10113	%0 = Nome campo %1 = **Etichetta campo** **%2 = caratteri ammessi**
NUMERIC	ERROR_CODE	10106	%0 = Nome campo %1 = **Etichetta campo**
	ERROR_CODE_MAXVALUE	10118	%0 = Nome campo %1 = **Etichetta campo** **%2 = Valore massimo**
	ERROR_CODE_MINVALUE	10119	%0 = Nome campo %1 = **Etichetta campo** **%2 = Valore minimo**
REGEXP	ERROR_CODE	10110	%0 = Nome campo %1 = **Etichetta campo** **%2 = regular expression**
RE	Vedere REGEXP		
URL	ERROR_CODE	10104	%0 = Nome campo %1 = **Etichetta campo**

I codici d'errore possono essere personalizzati sia sul file *fieldvalidators.xml* che sul file *validation.xml* (con risultati diversi). Segue un esempio di entrambe le configurazioni:

[fieldvalidators.xml]

```
<FIELD-VALIDATOR fieldType="FISCALCODE"
        fieldValidatorClass="it.eng.spago.validation.fieldvalidators.FiscalCodeValidator">
        <CONFIG ERROR_CF_CODE="50001"/>
</FIELD-VALIDATOR>
```

[validation.xml]

```
<SERVICE name="TEST_VALIDATION" type="ACTION">
        <VALIDATION blocking="true" validators="">
                <FIELDS>
                        ……
                        <FIELD name="FISCALCODE" maxLength="30" type="FISCALCODE"
```

***ERROR_CF_CODE**="50000" />*

……

</FIELDS>

</VALIDATION>

</SERVICE>

2.7.1.4.1 Etichette dei campi

Quando viene prodotto un errore di validazione relativo al campo, per esempio "TEST_FIELD", il messaggio è qualcosa del tipo "Il campo TEST_FIELD non ha il formato corretto".

Per personalizzare il messaggio con indicazioni più specifiche, ad esempio "Il campo Nome Utente non ha il formato corretto", ci sono due modi:

1. In uno dei file di properties definire un'etichetta con la sintassi *ACTION/PAGE.<NOME_SERVIZIO>.<NOME_CAMPO>=<ETICHETTA>*
   Per esempio, se il servizio invocato è un'action il cui nome è TEST_VALIDATION e il nome del campo è TEST_FIELD, nel file di properties *messages.properties* definire un'etichetta come segue:
   ACTION.TEST_VALIDATION.TEST_FIELD=Nome Utente

2. Le etichette possono anche essere definite in file di properties specifici per ogni servizio. Questi file non necessitano di essere censiti in *messages.xml*.
   Nell'esempio precedente, lo stesso risultato si può ottenere con un file di properties di nome *action/test_validation.properties* (dove "action" è la cartella contenente il file) e una proprietà definita come segue:
   TEST_FIELD=Nome Utente

2.7.2 Validazione Java

Nel caso in cui i meccanismi di validazione automatica non siano sufficienti, ad esempio perchè bisogna effettuare controlli incrociati sui dati, o controlli la cui logica sia particolarmente complessa, è possibile agganciare alle pagine delle apposite classi di validazione.

Una classe di validazione, che d'ora in poi chiameremo "validatore", è una classe che deriva da *it.eng.spago.validation.**AbstractValidator*** o che implementa l'interfaccia *it.eng.spago.validation.**ValidatorIFace*** e che implementa i seguenti metodi:

- *public void **init**(final SourceBean config)*: memorizza la configurazione (opzionale) del validatore. La configurazione consiste della busta "CONFIG" presente nella configurazione del validatore, come nel seguente esempio:

  <VALIDATOR name="FIRST_VALIDATOR" class="validators.FirstValidator">
  *<**CONFIG**>*
  <PUT SOME INFORMATIONS HERE.....>
  *</**CONFIG**>*
  </VALIDATOR>

- *public boolean **check**(SourceBean request, EMFErrorHandler errorHandler);*

Effettua i controlli formali sui dati del form. Normalmente si implementano in questo metodo i controlli di validità sintattica dei campi.
Deve restituire true se la verifica deve proseguire con gli altri validatori, false altrimenti.

- *public boolean **validate**(SourceBean request, EMFErrorHandler errorHandler);*

Effettua i controlli di dominio sui dati della sezione. Normalmente si implementano in questo metodo i controlli incrociati sui campi.
Questo metodo viene invocato solo se i controlli formali non hanno prodotto errori.
Deve restituire true se la verifica deve proseguire con gli altri validatori, false altrimenti.

Dal momento che ad una pagina possono essere associati più validatori, a seconda della tipologia dei controlli che devono essere effettuati, vanno implementati nel metodo check o nel metodo validate tenendo conto delle seguenti regole:

1. Vengono prima invocati i metodi **check** di tutti i validatori della pagina. Se una chiamata restituisce false, la catena di invocazione viene interrotta.
2. Se nessun metodo check restituisce false, vengono invocati i metodi **validate** di tutti i validatori della pagina. Se una chiamata restituisce false, la catena di invocazione viene interrotta.
3. Al termine della validazione saranno presenti nell'error handler tutti gli errori prodotti dalla validazione.

Le istanze dei validatori vengono create dinamicamente ad ogni richiesta di validazione.
I validatori devono prevedere un costruttore pubblico senza argomenti.

I validatori vengono definiti in un file *validators.xml* che ha la seguente struttura:

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

Ogni busta VALIDATOR contiene due attributi:

- NAME: nome logico del validatore, che verrà referenziato nel file *validation.xml*.
- CLASS: nome completo della classe che effettua la validazione.

Nulla vieta di utilizzare un validatore in più pagine, o di comporre le regole di validazione utilizzando più validatori elementari.

2.7.3 Sostituzione del motore di validazione

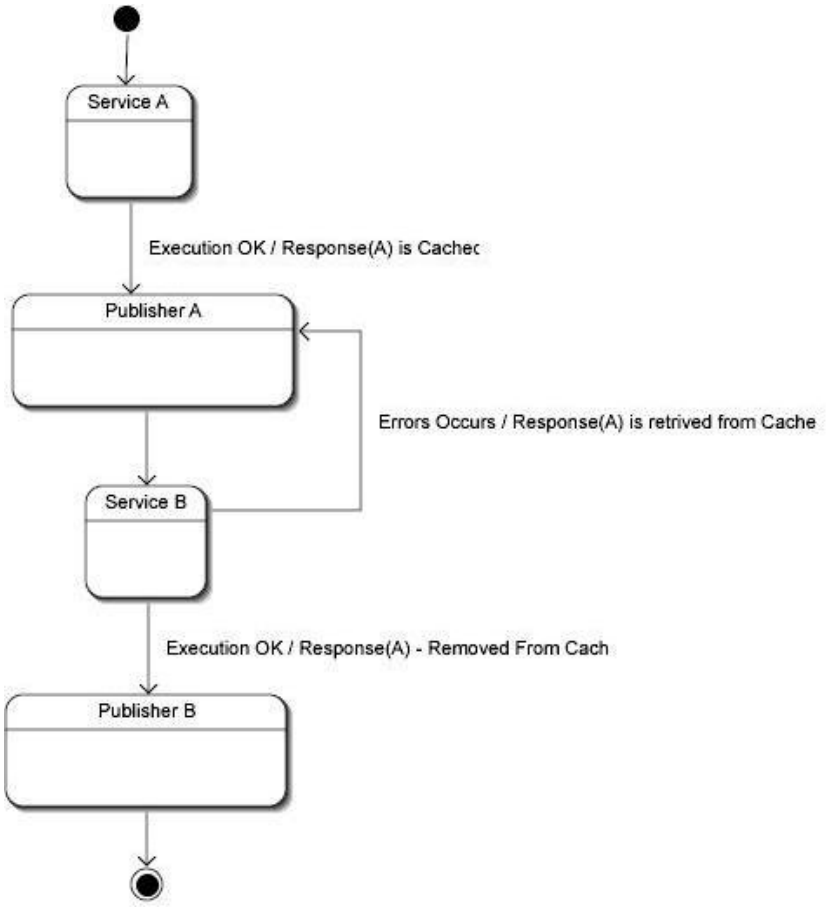
E' possibile sostituire l'implementazione del motore di validazione: nel file *validators.xml* c'è un attributo ***ENGINE*** che contiene il nome della classe che implementa la validazione. Questa classe implementa l'interfaccia *it.eng.spago.validation.* ***ValidationEngineIFace***. Segue un esempio del file validators.xml:

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

Se l'attributo *ENGINE* non è presente nel file, la validazione non viene applicata.

2.8 Caching delle risposte sui servizi applicativi

Per alcune tipologie di applicazioni, può essere utile fornire un meccanismo automatico di caching delle risposte fornite dai servizi applicativi, al fine che queste possano essere poi recuperate in caso di errore nell'esecuzione dei servizi successivi.
Un tipico caso d'uso è quello descritto dal seguente diagramma:



1. Il servizio A viene eseguito con successo e produce la risposta Response(A). La risposta di A viene mantenuta in cache.
2. La risposta di A viene presentata sul publisher Publisher A.
3. Viene eseguito il servzio B. A questo punto distinguiamo due casi:
   3.1. Il servizio B produce errori applicativi. In questo caso, la risposta di A in cache viene recuperata e viene di nuovo effettuato il rendering sul publisher di A, sul publisher di A vengono inoltre pubblicati gli eventuali errori applicativi, e anche i campi della richiesta che ha prodotto gli errori.
   3.2. Il servizio B viene eseguito con successo, In questo caso la risposta di A viene rimossa dalla cache e l'esecuzione continua in modo normale.

Per poter gestire questo tipo di casi di uso si ha in generale la necessità di:

1. Configurare **per quali servizi è necessario effettuare il cache delle risposte**. In Spago questo avviene aggiungendo la sottobusta REACH-SERVICES alle buste ACTION o PAGE a seconda del tipo di servizio che si sta configurando. Nella busta REACH-SERVICES sono elencati tutti i servizi raggiungibili dal servizio sorgente con la richiesta successiva. Per il caso d'uso precedente avremo una configurazione del tipo.

```
<ACTION name="SERVICE_A"
        class="it.eng.spago.sample.cacheresponse.ServiceA"
        scope="SESSION">
```

```
        <CONFIG></CONFIG>
        <REACH-SERVICES>
                <TARGET-SERVICE>SERVICE_B</TARGET-SERVICE>
        </REACH-SERVICES>
</ACTION>

…
<ACTION name="SERVICE_B"
        class="it.eng.spago.sample.cacheresponse.ServiceB"
        scope="SESSION">
        <CONFIG></CONFIG>
</ACTION>
```

2. Fornire dei meccanismi affinché quando un servzio applicativo produce degli errori applicativi, sia ripristinata la risposta del servzio precedente. Spago fornisce due differenti meccanismi (alternativi) per questa funzionalità, equivalenti nel risultato, ma che differiscono per la configurazione:
   a. Aggiungere il seguente listener al file *events.xml*:

      *<EVENTS-LISTENER listenerClass="it.eng.spago.listeners.**CacheResponseListener**"/>*

      Per verificare se una risposta proviene dalla cache è possible esaminare se contiene l'attributo *AF_CACHED_RESPONSE* (CacheResponseHandler.CACHED_RESPONSE_ATTRIBUTE) valorizzato a "TRUE".
   b. Utilizzare il Java publisher ***it.eng.spago.cache.DynamicCacheResponsePublisher***. Mappando il servizio che ha bisogno di ripristinare la risposta a questo publisher, si attiva la funzionalità di caching.
      La semantica di questo publisher Java è la seguente: se il servizio è eseguito con successo viene effettuato il rendering sul publisher indicato nella busta TARGET-PUBLISHER. Se invece viene prodotto qualche errore applicativo viene recuperata dalla cache la risposta del servizio precedente e viene effettuato di nuovo il rendering sul publisher del servzio precedente.
      Nel caso d'uso precedente si avrà la seguente configurazione nella busta *PUBLISHER.XML:*

```
<PUBLISHER name="ServiceB_Java_Publisher">
    <RENDERING channel="HTTP" type="JAVA" mode="">
       <RESOURCES>
            <ITEM prog="0" resource="it.eng.spago.cache.DynamicCacheResponsePublisher">
                  <CONFIG>
                        <TARGET_PUBLISHER>ServiceBJSPPublisher</TARGET_PUBLISHER>
                  </CONFIG>
            </ITEM>
       </RESOURCES>
    </RENDERING>
</PUBLISHER>

<PUBLISHER name="ServiceBJSPPublisher">
    <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
            <RESOURCES>
                  <ITEM prog="0" resource="/pageServiceB.jsp" />
            </RESOURCES>
    </RENDERING>
</PUBLISHER>
```

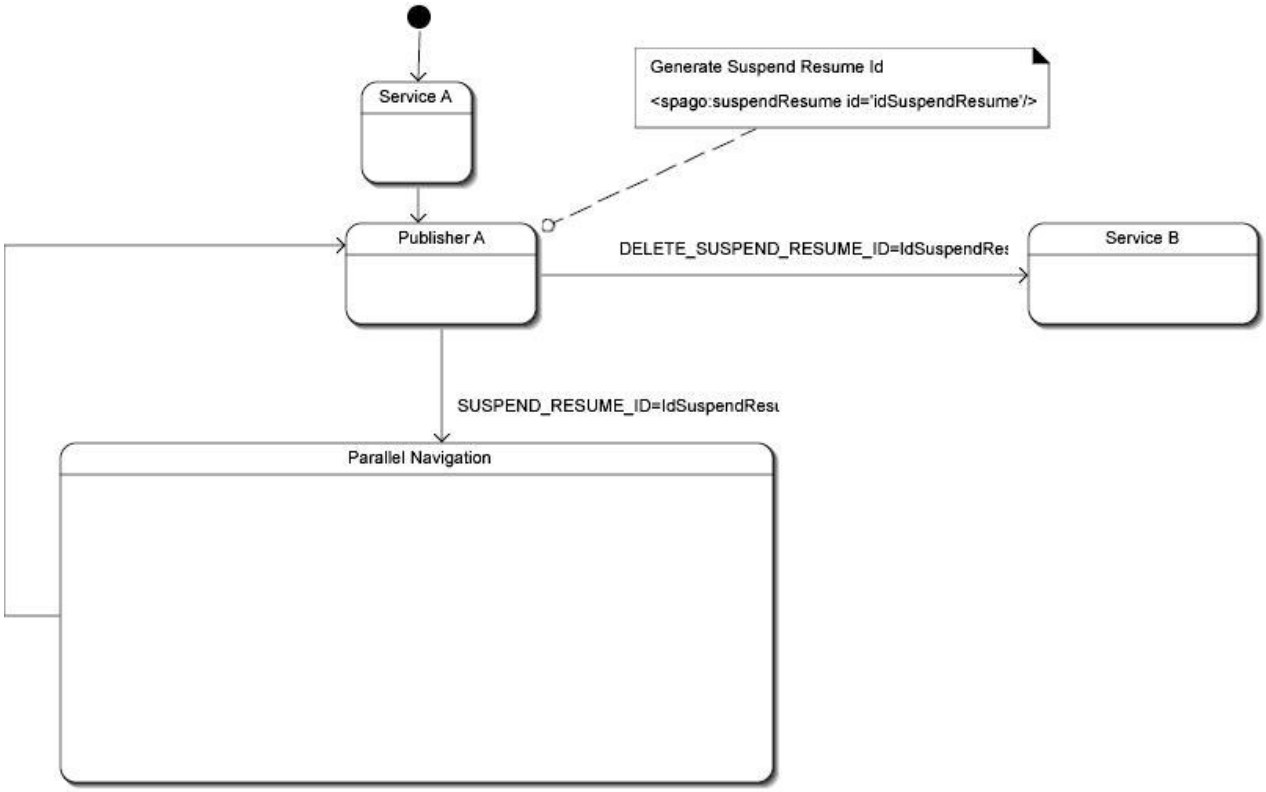
      e nella busta PRESENTATION il servizio B è associato al Java publisher:

```
<MAPPING business_type="ACTION"
         business_name="SERVICE_B"
         publisher_name="ServiceB_Java_Publisher"/>
```

2.9 Automatismo di Suspend/Resume

Per alcune tipologie di applicazioni, possono presentarsi dei casi d'uso in cui durante la navigazione è necessario ad un certo punto effettuare un salvataggio della situazione esistente, iniziare un percorso di navigazione "parallelo", e poi tornare alla situazione salvata fornendo dei meccanismi per recuperare sia i dati "salvati" sia quelli raccolti nel nuovo percorso di navigazione.

Una tipica situazione di questo tipo si ha per esempio nelle applicazioni che hanno bisogno di reperire dei dati su tabelle di lookup in presenza di vincoli sull'accessibilità per persone diversamente abili. In questo tipo di applicazioni Web non è possibile infatti aprire finestre di popup per selezionare i dati, è quindi necessario, "salvare la situazione" prima di iniziare una selezione di lookup, eseguire uno o più servizi di business per la selezione dei dati necessari e poi tornare alla pagina di partenza sia con i dati presenti precedentemente sia con i nuovi dati selezionati durante la lookup.



Per far fronte a situazioni di questo tipo in Spago viene fornito l'automatismo di ***Suspend/Resume***. Questo automatismo non può essere implementato in modo completamente automatico in Spago, per ottenere una "situazione istantanea" prima di iniziare una navigazione parallela è necessario, se ci si trova ad esempio nel publisher A:

- Salvare la ***risposta del servzio (in entrata)*** che ha come rendering il publisher A
- Salvare la ***richiesta in uscita dal publisher A*** (si pensi ad una pagina con un form complesso di 20 campi se l'utente riempie la maggior parte di questi e poi inizia una navigazione in lookup, quando ritorna alla pagina si aspetta di trovare i campi precedentemente riempiti e il campo di lookup valorizzato con il nuovo valore)

In spago il meccanismo di Suspend/Resume è legato al publisher e viene implementato attraverso i seguenti meccanismi:

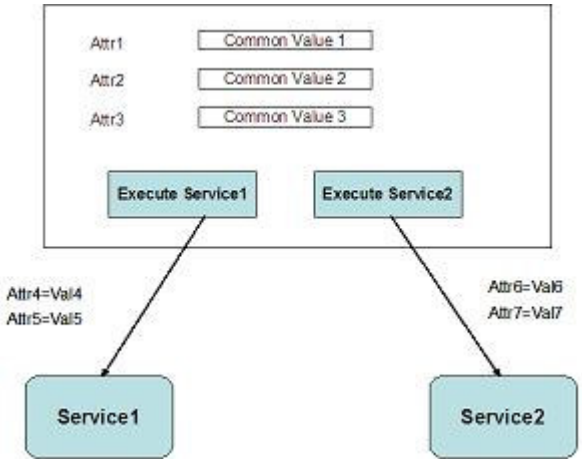
1. All'interno di un publisher che partecipa a una navigazione di tipo Suspend/Resume, cioè è punto di partenza e di arrivo per una navigazione parallela è necessario generare un identificativo di Suspend/Resume attraverso l'utilizzo della classe di utilità ***it.eng.spago.tags.SuspendResumeIdGenerator*** utilizzando il metodo statico: ***public static String getSuspendResumeId(RequestContainer requestContainer)***.
2. Sempre all'interno del publisher utilizzare il tag ***<spago:suspendResume id="idSupendResume"/>*** e passare come input l'identificativo ottenuto al punto 1. Questo tag verifica se esiste una situazione salvata con l'identificativo passato come input. Se esiste un salvataggio, viene fatto un merge tra i dati presenti nei contenitori "sospesi" in precedenza e i dati presenti nei contenitori attuali. In ogni caso sia che venga effettuato un merge, sia che non venga effettuato il contenitore con la risposta *<SERVICE-RESPONSE>* viene salvato ed associato all'id passato come input al tag.
3. Nel publisher è necessario che le azioni (link su URL, form) che iniziano la navigazione parallela contengano il parametro *SUSPEND_RESUME_ID="idSuspendResume"*. Questo serve per effettuare il salvataggio della richiesta in uscita.
4. Per effettuare la rimozione di una situazione di Suspend/Resume salvata con un determinato ID e necessario effettuare una richiesta passando a spago il comando *DELETE_SUSPEND_RESUME_ID="idSuspendResume"*.

2.10 Form Accessibili con spago

Uno dei principali problemi nello sviluppo, quando è necessario realizzare applicazioni che rispettino i vincoli sull'accessibiltà per le persone diversamente abili, riguarda lo sviluppo di pagine web senza utilizzare la tecnologia Javascript. Questo crea notevoli problemi nella gestione dei submit e delle logiche di navigazione dell'applicazione.

Il problema più grosso si ha quando è necessario realizzare delle pagine con dei form html con diversi pulsanti e ogni pulsante avvia l'esecuzione di servizi di business diversi (in spago vengono invocati ACTION o PAGE diversi) ed è inoltre necessario avere una serie di campi comuni a tutti i servizi, e dei campi che invece devono essere in mutua esclusione.

Il diagramma seguente illustra questo use case:



In Spago questo problema è stato risolto nel seguente modo:

1. I pulsanti del form devono essere dichiarati tutti di tipo submit (<input type="submit"/>)
2. L'attributo name del pulsante di submit può contenere un valore con una sintassi particolare che indica a spago di gestire questo attributo come una query string. La sintassi è la seguente:

   <input type="submit" value="Execute Service 1"
   name="QUERY_STRING{ACTION_NAME=SER1_ACTION, ATTR4=VAL4, ATTR5=VAL5}"/>

   <input type="submit" value="Execute Service 2"
   name="QUERY_STRING{ACTION_NAME=SER2_ACTION, ATTR6=VAL6, ATTR7=VAL7}"/>

   Con questo meccanismo, nella richiesta HTTP compare il nome del bottone premuto e l'adapter HTTP provvede a gestire i campi che come nome riportano la sintassi QUERY_STRING{ ATTR=VAL,…} come un insieme di attributi di richiesta.

Questo meccanismo permette di gestire form di medio/alta complessità senza utilizzare Javascript.

2.11 Gestione degli upload

Spago utilizza la libreria *commons-fileupload* per gestire i form multipart. Nel file di configurazione *upload.xml* sono censiti i gestori dei campi di tipo file, in modo che sia possibile scegliere l'implementazione più adatta al tipo di applicazione.
L'attributo *UPLOAD-MANAGER* contiene il nome dell'implementazione utilizzata.
Ecco la struttura del 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>
```

Esistono due gestori preconfigurati che sono:

- *SAVE_FILE_AS_ATTRIBUTE*: salva i file presenti nell'HTTP request, come attributi all'interno della service request. Per ciascun file vengono salvati due attributi con i seguenti nomi:
  - *<NOME CAMPO DI INPUT>_FILENAME*: dove viene memorizzato il nome del file. <NOME CAMPO DI INPUT> è il nome del campo così come definito nel form.

- *<NOME CAMPO DI INPUT>_FILEDATA*: è l'array di byte (byte[]) dove viene memorizzato il contenuto del file.

- *SAVE_FILE_ON_DISK*: salva i file presenti nell'HTTP request, su disco, in una directory definita a livello di configurazione (nell'attributo PATH della busta CONFIG). Per ciascun file, oltre al salvataggio su disco, viene salvato un attributo con il seguente nome:
  - <NOME CAMPO DI INPUT>_FILENAME: dove viene memorizzato il nome del file. <NOME CAMPO DI INPUT> è il nome del campo così come definito nel form.

Il gestore di default è quello che salva su disco.
E' possibile implementare un gestore custom implementando l'interfaccia *it.eng.spago.dispatching.httpchannel.upload.**IUploadHandler*** definita come segue:

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

FileItem è l'oggetto restituito da *commons-fileupload*. Il gestore va censito in *upload.xml* e definito come gestore attivo indicandolo nell'attributo *UPLOAD-MANAGER*.

2.12 Prevent Resubmit

Spago fornisce un'implementazione del pattern PreventResubmit.
Ad ogni richiesta il *Coordinator* genera un id che viene memorizzato nel *PermanentContainer*. Il tag *GenerateToken* genera nel form il campo hidden *SPAGO_TOKEN* contente lo stesso id che viene confrontato al submit con quello memorizzato nel *PermanentContainer*. Se i valori non coincidono viene generato un *EMFInternalError* contenete il messaggio "Navigazione non permessa"
E' possibilire disattivare a run time questo meccanismo impostando nell'url l'attributo SPAGO_GENERATETOKEN=FALSE.
Nel caso di chiamate alle look up bisogna disabilitare la funzionalità.
Le liste ed i dettagli generati automaticamente hanno impostato di default il meccanismo di prevent resubmit per i bottoni di chiarati in `<DELETE_CAPTION>` (per la lista) e `<SUBMIT_BUTTON` > (per il dettaglio).

2.13 Pubblicazione

Il sottosistema di presentazione fornisce varie alternative per effettuare la pubblicazione delle risposte:

- XML/XSLT : è possibile transcodificare il flusso XML di risposta con un foglio di stile XSL, o una serie di fogli di stile XSL in cascata.
- JSP: la presentazione del risultato di un servizio può essere delegata ad una pagina JSP. La pagina ha accesso sia alla busta della risposta che agli eventuali errori.
- Servlet: la presentazione del risultato di un servizio può essere delegata ad una servlet che di solito reperisce i dati da visualizzare dalla sessione (valorizzata dalla action o dai moduli precedenti).
- JAVA : la logica per determinare la modalità di presentazione è delegata ad una classe JAVA.
- LOOP (solo per il canale HTTP): la presentazione viene delegata all' AdapterHTTP invocando un nuova servizio

- Nessuna: non tutti i servizi necessitano della parte di presentazione: un servizio SOAP, ad esempio, potrebbe presentare nella risposta direttamente lo stream XML fornito dalla action o dalla pagina. Questa modalità è utile in fase di debug.

Ovviamente non tutte queste modalità sono adatte a gestire la pubblicazione per tutti i canali: ad esempio le modalità Servlet/JSP non può essere utilizzata per il canale SOAP o EJB.

2.13.1 Configurazione

La prima attività da fare per pubblicare il risultato di una richiesta di servizio è quella di associare l'action, o la pagina, al corrispondente oggetto di pubblicazione. Tale associazione viene effettuata nel file *presentation.xml*, di cui viene riportato un esempio:

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

Per ogni action, o pagina, viene specificato il nome logico dell'oggetto di pubblicazione da utilizzare. In questo modo, un pubblicatore può essere utilizzato da più action, o pagine.
Gli oggetti di pubblicazione sono definiti nel file *publishers.xml*, di cui riportiamo un esempio:

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

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

```
    </RENDERING>
  </PUBLISHER>
</PUBLISHERS>
```

Per ogni pubblicatore è definita una busta *PUBLISHER* nel file di configurazione.
Come si può notare, ogni publisher è contraddistinto da un nome logico:

*<**PUBLISHER** name="nome logico publisher">*

Il sottosistema di presentazione deve entrare nel merito di qual'è il canale dove deve essere presentata la risposta, per questo motivo all'interno della busta *PUBLISHER* va definita una busta *RENDERING* per ciascuno dei canali su cui si intende effettuare la pubblicazione.

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

I canali possibili sono:

- HTTP
- WAP
- SOAP
- EJB

Stabilito il canale, va indicata la modalità di presentazione per quel canale. E' possibile specificare i seguenti valori:

- NOTHING: non viene effettuata nessuna transcodifica, il flusso XML del risultato viene presentato direttamente sul canale di uscita. Utile anche per scopi di debug.
- SERVLET
- JSP: la presentazione viene delegata ad una pagina JSP, che ha accesso al flusso XML di risposta.
- XSL: il flusso XML di risposta viene transcodificato con uno o più fogli di stile XSL, in cascata.
- JAVA : la logica per determinare la modalità di presentazione è delegata ad una classe JAVA.
- LOOP (solo per il canale http e SOAP): la presentazione viene delegata all' AdapterHTTP invocando un nuova servizio

Non tutte le modalità sono valide per tutti i canali, ad esempio non ha senso la modalità servlet sul canale EJB.La seguente tabella riporta le varie possibili scelte.

Channel	type	mode	prog	resource
HTTP	NOTHING		0	
	SERVLET	FORWARD	0	Nome della servlet
		SENDREDIRECT	0	Nome della servlet
	JSP	FORWARD	0	Nome della pagina JSP
		SENDREDIRECT	0	Nome della pagina JSP
	XSL		"0, 1, 2, …"	Nomi degli XSL da applicare
	LOOP		0	Parametri della nuova richiesta
WAP	NOTHING		0	
	SERVLET	FORWARD	0	Nome della servlet

		SENDREDIRECT	0	Nome della servlet
	JSP	FORWARD	0	Nome della pagina JSP
		SENDREDIRECT	0	Nome della pagina JSP
	XSL		"0, 1, 2, …"	Nomi degli XSL da applicare
SOAP	NOTHING		0	
	XSL		"0, 1, 2, …"	Nomi degli XSL da applicare
	LOOP		0	Parametri della nuova richiesta
EJB	NOTHING		0	
	XSL		"0, 1, 2, …"	Nomi degli XSL da applicare
JMS	NOTHING		0	
	XSL		"0, 1, 2, …"	Nomi degli XSL da applicare

2.13.2 Configurazione semplificata

Per facilitare l'attività di configurazione è stato fornito un meccanismo semplificato di associazione tra i servizi (action/page) e i corrispondenti publisher.

Nella modalità standard (descritta nel paragrafo precedente), Spago necessita delle informazioni presenti nel file *presentation.xml*: questo nell'ottica di una netta separazione tra la business logic e la presentazione.

Nella configurazione semplificata è possibile censire l'informazione del publisher direttamente nei file *actions.xml* o *pages.xml*: se presente viene utilizzato l'attributo *PUBLISHER_NAME* (case insensitive). Se tale attributo non è presente, viene utilizzato il meccanismo standard.

Segue una configurazione di esempio per una action:

```
<ACTION name="TEST_SERVLET_PUBLISHING" class="it.eng.spago.sample.TestValidation"
                                        scope="SESSION"
                                        PUBLISHER_NAME="SERVLET_PUB">
    <CONFIG></CONFIG>
</ACTION>
```

La sintassi per le page è analoga:

```
<PAGE name="PAGE_TEST_PUBLISHER" scope="SESSION"
                    PUBLISHER_NAME ="AGENT_DEPENDENT_PUBLISHER">
```

2.13.3 Pubblicazione Condizionata dall'agent

Alcune applicazioni potrebbero necessitare di associare risorse di pubblicazione (jsp, fogli xsl etc..) diverse a seconda del valore dello user-agent che accompagna una richiesta.

Per le esigenze di questo tipo in Spago è possibile, opzionalmente, avere all'interno delle buste *PUBLISHER* più sezioni di *RENDERING* per lo stesso canale, ma che riportano valori diversi nell'attributo *user-agent.*

Il valore dell'attributo *user-agent* identifica uno ***user agent logico*** all'interno di spago. Il mapping tra le stringhe che arrivano nella richiesta e user agent logici utilizzati per condizionare la pubblicazione avviene nella busta *USER-AGENTS* del file di configurazione *useragents.xml*.

Ad esempio la busta seguente individua due user agents logici denominati IE6 ed 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>
```

che possono essere utilizzati ad esempio nella busta publisher nel seguente modo:

```
<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 questo caso, il servizio applicativo di spago (ACTION, o PAGE) che nel file *presentation.xml* viene associato al publisher denominato *AGENT_DEPENDENT_PUBLISHER* produrrà come risultato la pagina */testAgentIE.jsp* nel caso la richiesta sia pervenuta da Internet Explorer o la pagina */testAgentFirefox.jsp* nel caso la richiesta provenga da Firefox.

Nel caso la stringa nella richiesta non identifichi univocamente uno user-agent logico o venga determinato uno user-agent logico per la quale non esiste una busta rendering specifica viene utilizzata la prima busta rendering tra quelle disponibile per il canale specificato.

2.13.4 XML/XSLT

Con questa modalità possono essere censiti più fogli di stile da applicare.
La transcodifica XML/XSL è onerosa se l'XSL viene compilato ogni volta: per questo motivo il framework precompila gli XSL all'avvio dell'applicazione e li mantiene in cache. Inoltre in caso di modifica degli XSL, il framework li ricompila dinamicamente senza necessità di riavviare l'applicazione.
Il risultato del rendering con XSL non è limitato all'HTML: può essere prodotto qualunque tipo di output. Sono state effettuate delle prove con Crystal Clear e Jreport per produrre l'output in formato PDF.
Sono state effettuate anche delle prove con FOP (Formatting Objects Processor) ma è un meccanismo abbastanza complesso da gestire.

Esiste un esempio di output prodotto in PDF nel portale interno Engiweb@Work: se si cerca un nominativo nella rubrica e si effettua la stampa, il risultato è in PDF.
Attualmente il framework utilizza Xalan come transcoder.

2.13.4.1 Sostituzione del transcoder XSL

Una factory si occupa di istanziare il transcoder XSL. E' possibile definire la classe da utilizzare per l'attività di transcoding modificando il file *transcoder.xml*:

```
<TRANSCODERS>
    <TRANSCODER class="it.eng.spago.transcoding.Transcoder"/>
</TRANSCODERS>
```

Il transcoder deve implementare l'interfaccia *it.eng.spago.transcoding.**TranscoderIFace*** che definisce il seguente metodo:

*void **perform**(SourceBean toRender, Templates xslTemplate, Result out) throws EMFInternalError;*

2.13.5 Servlet/JSP

Le JSP di progetto, che estendono la classe it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage, hanno direttamente a disposizione i seguenti servizi:

- 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);

2.13.6 JAVA

La modalità JAVA consente di gestire una logica più complessa di presentazione, cui è preposta una classe Java. Tale classe viene instanziata runtime ed ha la possibilità di estendere la classe astratta AbstractPublisherDispatcher, che le permette di avere a disposizione la sua configurazione. La sua configurazione è definita nella busta CONFIG del file *publishers.xml*. di cui forniamo un esempio:

```
<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>
```

E' stato implementato inoltre, un publisher di tipo Java per descrivere a livello di configurazione semplici logiche di selezione della vista;la classe da registrare come publisher è:

it.eng.spago.presentation.DefaultPublisherDispatcher

implementata nel framework, estendendo la classe astratta AbstractPublisherDispatcher, in grado di tradurre questa grammatica:

```
<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>
```

dove i possibili valori dell'attributo scope sono:

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

I possibili valori dell'attributo value sono:

- *AF_DEFINED*
- *AF_NOT_DEFINED*
- *il valore del parametro*

La busta CHECKS contiene i controlli che devono essere effettuati dalla classe Java per individuare quale publisher (anch'esso censito nel file *publishers.xml*) si occuperà della presentazione.
Se la condizione contenuta nella busta CONDITION risulta verificata, allora il publisher designato sarà quello indicato nell'attributo "target" della rispettiva busta CHECK.

2.13.7 LOOP

La modalità LOOP, implementata per il canale HTTP e SOAP, permette di eseguire un loop_back. Lo scenario diventa il seguente:

- il browser invoca un servizio attraverso l'AdapterHTTP
- la logica di business del servizio viene eseguita
- la presentazione viene delegata all'AdapterHTTP invocando un nuovo servizio
- la logica di business del nuovo servizio viene eseguita
- la presentazione viene delegata al publisher

- la risposta del servizio arriva al browser

I parametri della nuova serviceRequest vengono definiti nelle buste <PARAMETER> con la solita grammatica di dichiarazione dei parametri dinamici.

Vediamo un esempio di configurazione per il canale HTTP:

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

Esiste la possibilità di generare run-time la nuova richiesta; in tal caso la configurazione del publisher di tipo LOOP sarà ad esempio:

```
<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>
```

2.13.8 Un esempio

Vediamo ora, tramite un esempio, come è possibile associare diversi publisher ad un'action, per i vari canali.

La seguente action effettua una lettura da database di test, recuperando le informazioni di alcuni utenti.

```
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("afdemo");
            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
```

Senza entrare nel dettaglio di come funziona questa action (sarà oggetto di uno dei prossimi paragrafi), l'importante è sapere che questa action valorizza la risposta con un flusso XML di questo tipo:

```
<?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>
```

Applicheremo a questa risposta 3 delle 5 tipologie possibili di publisher per il canale HTTP, ovvero:

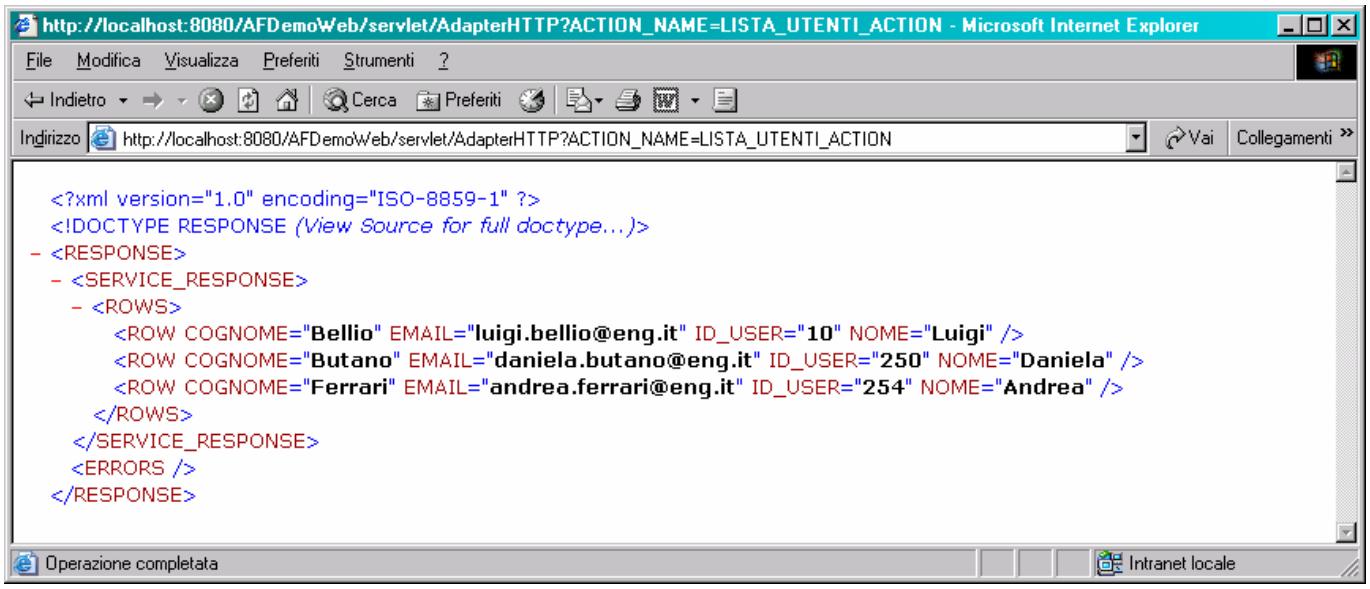
- Nessuna trasformazione
- XSL
- JSP

2.13.8.1 Canale HTTP e nessuna trasformazione

Con questa modalità il flusso XML prodotto dall'action viene inviato senza alcuna modifica al browser: attualmente Internet Explorer versione 5.5 e superiori, è in grado di leggere un file XML e di presentarlo.
Per ottenere questo risultato è sufficiente NON indicare un publisher associato all'action prescelta, nel file *presentation.xml*.
Nel nostro esempio abbiamo commentato la sezione:

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

Il risultato dell'invocazione dell'action, da Internet Explorer, è il seguente:



2.13.8.2 Canale HTTP e XSL

Utilizziamo ora l'XSL per effettuare il transcoding del flusso XML visto nel passo precedente. Per far ciò, associamo un publisher all'action *ListaUtentiAction*, scommentando la sezione *MAPPING* nel file *presentation.xml* , vista nel paragrafo precedente.

A questo punto, quando con il browser web effettuiamo la richiesta di servizio tramite un URL del tipo:

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

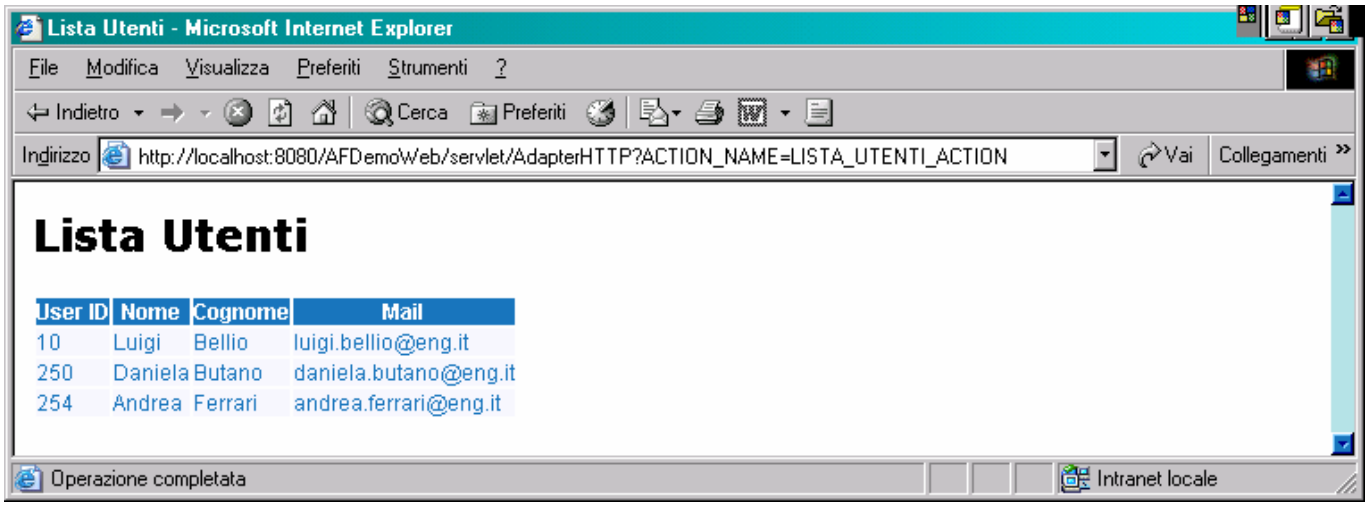
il framework verificherà che tipo di rendering effettuare per il publisher specificato, per il canale HTTP.

Abbiamo configurato il file *publishers.xml* come segue:

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

Possono essere specificati più fogli di stile da applicare, ciascuno contraddistinto da un numero progressivo (attributo *prog*) che ne determina l'ordine di applicazione.

Per il canale HTTP, l'action *ListaUtentiAction* effettuerà il rendering del risultato tramite transcoding della risposta con l'XSL *HTTPListaUtentiAction.xsl*.



Ecco il file XSL con cui si è ottenuto questo risultato:

```
<?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>
```

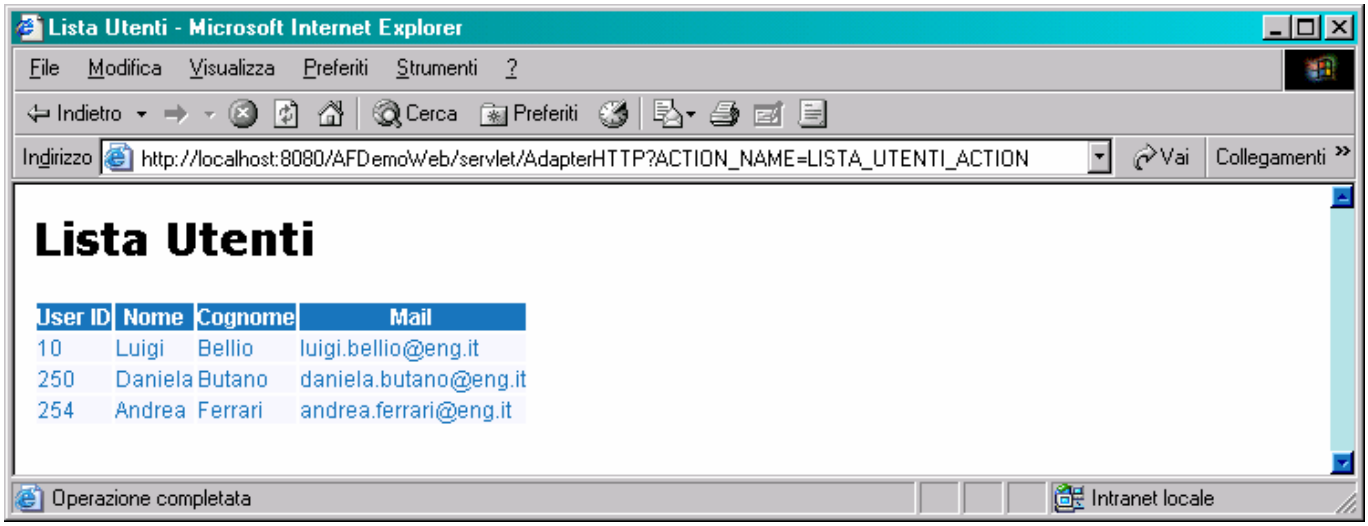
Il foglio di stile si limita a scandire la busta delle righe ottenute da database (*ROWS*) e a produrre una colonna per ciascun dato.

2.13.8.3 Canale HTTP e JSP

Verifichiamo adesso come lo stesso risultato può essere ottenuto tramite una pagina JSP.
Manteniamo invariata l'associazione tra l'action e il publisher corrispondente, ovvero il file *presentation.xml*, e modifichiamo il tipo di rendering da applicare, nel file *publishers.xml*, specificando che il publisher deve essere una pagina JSP:

```
<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>
```

La pagina JSP che utilizzeremo produce il seguente risultato:



La pagina JSP che effettua il rendering è la seguente:

```
<%@ 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.getAttributeAsList("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>
```

Come si può vedere la pagina JSP ha accesso al response container, e tramite questo reperisce i dati della risposta.

2.13.8.4 Canale WAP e XSL

Abbiamo accennato al fatto che il pubblicatore deve entrare nel merito del canale su cui verrà effettuata la pubblicazione: questo perchè a seconda del canale, le informazioni verranno visualizzate in modo diverso, o addirittura potrebbero essere presentate informazioni diverse.

Vediamo ora un esempio di come la stessa action utilizzata nell'esempio precedente possa essere attivata da un browser WAP: il risultato verrà prodotto tramite transcoding con foglio di stile XSL, ovviamente diverso da quello utilizzato per il canale HTTP.

Nel file *publishers.xml* dovremo aggiungere una busta *RENDERING* per il canale WAP:

```
<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>
```

Richiamando l'action con un emulatore WAP, otterremo il seguente risultato:

Da notare che poichè stiamo utilizzando un emulatore WAP, e non un vero browser WAP, non stiamo effettuando la richiesta attraverso un gateway WAP: il framework pertanto non ha modo di stabilire che la richiesta proviene dal canale WAP. Per fornire questa informazione al framework, aggiungiamo all'URL della richiesta il parametro *CHANNEL_TYPE*:

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

L'XSL utilizzato ha la stessa struttura di quello utilizzato per il canale HTTP, semplicemente fa il rendering di meno informazioni.

```
<?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>
```

2.14 Tag Libraries

Spago fornisce l'implementazione delle seguenti tag libraries:

- **list**: utilizzata nella generazione automatica di lista-dettaglio per eseguire il rendering di una lista. Recupera le informazioni inerenti il layout dalla busta *<CONFIG>* censita dentro la busta *<ACTION>* in *actions.xml* (o *<MODULE>* in *modules.xml)* corrispondente alla action (o modulo) passata in input. Recupera le informazioni da visualizzare dal *serviceResponse* prodotto dalle action (o moduli) di lista fornite da Spago
- **detail**: utilizzata nella generazione automatica di lista-dettaglio per esegue il rendering di un dettaglio. Recupera le informazioni inerenti il layout dalla busta *<CONFIG>* censita dentro la busta *<ACTION>* in *actions.xml* (o *<MODULE>* in *modules.xml)* corrispondente alla action (o modulo) passata in input. Recupera le informazioni da visualizzare dal *serviceResponse* prodotto dalla action (o modulo) di dettaglio fornita da Spago
- **error**: implementa la funzione java script *checkError()* che visualizza tramite un pop-up lo stack degli errori contenuti in ErrorHandler
- **lookup** e **lookupClient**: Spago implementa la lookup tramite queste due tag library. La pagina JSP contente il campo di tipo lookup utilizzerà la tag **lookupClient**. Al verificarsi dell'evento onClick() sulla lookup, bisogna chiamare la funzione *lookup_valoreidLookup* definita dal tag che visualizzerà in una nuova finestra la lista di lookup . Questa seconda pagina utilizzerà **lookup** per la creazione della lista di lookup. Lookup e lookupClient utilizzano i parametri di configurazione censiti nel file *lookup.xml.*

```
<LOOKUPS>
    <LOOKUP idlookup="ListaUtenti">
            <PARAMETERS pageName="ListaUtentiLookupPage" moduleName="ListaUtentiLookupModule"
            formName="StartLookup">
                    <FIELDS>
                            <FIELD lookupTableField="userid" formField="userid" likeField="true" />
                    </FIELDS>
            </PARAMETERS>
    </LOOKUP>
</LOOKUPS>
```

Gli attributi della busta *<PARAMETERS>* servono al tag lookupClient per invocare il modulo che esegue la lista; le buste *<FIELD>* servono per associare il campo definito nel form con il campo della lista.

- **navigationToolbar**: visualizza una barra di navigazione dalla quale è possibile selezionare uno dei servizi già eseguiti durante la conversazione. Per maggiori dettagli vedi toolbar di navigazione
- **comboNavigationToolbar**: visualizza una combo di navigazione dalla quale è possibile selezionare uno dei servizi già eseguiti durante la conversazione. Per maggiori dettagli vedi toolbar di navigazione
- **generateToken**: genera il campo hidden *SPAGO_TOKEN* contente l'ultimo id generato dal server per la gestione del prevent resubmit
- **message**: visualizza il messaggio contenuto nel file di properties *message_languagecode_countrycode.properties* corrispondente al codice passato in input
- **reloadField**: recupera dalla *serviceRequest* il valore dell'attributo passato in input. Da utilizzare quando, in caso di errore, si vogliano visualizzare i dati appena inseriti.
- **writeString**: visualizza il valore della stringa passata in input o il valore di nullValue (passato in input il cui valore di default è blank) se la stringa e' nulla

2.15 Gestione anomalie

Il framework gestisce le anomalie contestuali alla richiesta di un servizio nel modo seguente:

- SE il client richiede un servizio e la sessione è scaduta ALLORA il framework cerca una action registrata con il nome "**SESSION_EXPIRED_ACTION**" (actions.xml)
- SE il client richiede un servizio e la "action" o "module" lancia una "exception" ALLORA il framework cerca un publisher registrato con il nome "**SERVICE_ERROR_PUBLISHER**" (publishers.xml)
- SE il client richiede un servizio e la JSP di pubblicazione lancia una "exception" ALLORA il framework cerca il publisher JSP **"/jsp/spago/jspError.jsp**" (jspError.jsp)

2.16 Gestione multilingua

Il catalogo multilingua degli errori è gestito con il meccanismo standard di java, ovvero con file di properties che si trovano nel classpath. Il file contenente i messaggi d'errore si chiama *messages.properties* e ha una struttura del tipo:

10001 = La lista non contiene righe
10002 = Cancellazione eseguita correttamente
10003 = Errore durante la cancellazione

ogni messaggio è contraddistinto da un codice, che è quello che va specificato nel costruttore della classe d'errore:

public EMFUserError(String severity, String code)

I file contenenti i messaggi nelle varie lingue dovranno trovarsi nello stesso path del file *message.properties* e dovranno chiamarsi *message_languagecode_countrycode.properties*, dove *languagecode* e *countrycode* sono rispettivamente i codici della lingua e del paese (i codici principali sono reperibili da apposite costanti nella classe *java.util.Locale*). Ad esempio il file per la lingua italiana dovrà chiamarsi *messages_it_IT.properties*, quello per l'americano *messages_en_US.properties,* etc.
Per utilizzare una determinata lingua, vanno impostate nel *permanent container* due stringhe corrispondenti al codice della lingua e del paese, come nel seguente esempio che imposta la lingua italiana:

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

Una volta impostati tali valori, la lingua rimarrà in uso per tutta la sessione.
Normalmente questi valori vengono impostati in sessione in fase di login dell'utente, in base alle preferenze memorizzate per quest'ultimo.

2.16.1 Pubblicazione multilingua

Il meccanismo fornito dal framework permette di gestire i messaggi d'errore in modalità multilingua.
Tale meccanismo è disponibile per qualunque messaggio presente nei file di properties (*messages_xx_yy.properties*), tramite l'oggetto *MessageBundle*. Questo oggetto permette di reperire un messaggio, specificato il suo codice:

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

Un meccanismo analogo permette di effettuare anche la pubblicazione dei contenuti in modalità multilingua.
Consulare il tag **message** per maggiori dettagli.
Per i canali SOAP ed EJB il problema della lingua non dovrebbe porsi in quanto i servizi tramite questi canali dovrebbero fornire solo dati, senza alcuna logica di presentazione.

2.16.2 Separazione dei messaggi in più file

Per separare i messaggi in più file di properties vanno censiti in *messages.xml*. Ecco un esempio di tale configurazione:

```
<MESSAGES>
    <BUNDLE name="properties.messages"/>
    <BUNDLE name="properties.validation_messages"/>
</MESSAGES>
```

Per utilizzare i file di properties all'interno di cartelle, va utilizzata la dot-notation. Nel precedente esempio i file di properties sono all'interno di un folder chiamato "properties".
Infine, nel file *initializer.xml*, va indicato il seguente initializer:

```
<INITIALIZER class="it.eng.spago.message.MessageBundleInitializer" />
```

L'implementazione di MessageBundle cercherà i messaggi in tutti i bundle definiti in *messages.xml*, a meno che non sia fornito esplicitamente il nome del bundle in cui cercare.

2.17 Distribuzione della logica di business

Una delle caratteristiche più interessanti del framework è la possibilità di poter scegliere se eseguire la logica di business (action e moduli) su web container o EJB container, utilizzando un interfaccia locale o remota, agendo a livello di file di configurazione.

Per quanto riguarda le implicazioni che le diverse modalità comportano, si rimanda al paragrafo Gestione delle transazioni in quanto la differenza di comportamento riguarda essenzialmente la gestione del database.

La distribuzione dell'esecuzione dei servizi è controllata dal file *distribution.xml* che ha una struttura del tipo:

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

In questo file vanno censite tutte le action e le pagine che vanno eseguite su EJB container.

Per quanto riguarda le pagine, tutti i moduli che fanno parte della pagina verranno eseguiti su EJB container: non è possibile avere un'esecuzione di un servizio ibrida, ovvero in parte su web container e in parte su EJB container.

Utilizzando le interfacce locali (ovvero il DistributeCoordinator), non si ha una penalizzazione delle performance (non troppo almeno). In questo modo se qualche operazione fallisce nella generazione di una pagina, viene effettuato il rollback di tutti i moduli appartenenti alla pagina (dato che il flag è associato alla pagina e non al modulo).

2.18 Accesso ai dati

Il Data Abstract Layer è il modulo che virtualizza l'accesso ai dati: aggiunge delle funzionalità a quelle già presenti in JDBC (che è già un layer che virtualizza l'accesso ai dati).
L'esigenza che si prefigge di risolvere il modulo di accesso ai dati è quella di minimizzare lo sforzo richiesto per migrare un applicativo sviluppato facendo uso di un database, su un altro database.
Utilizzando JDBC questa migrazione richiederebbe molto lavoro; il Data Abstract Layer, invece, astrae da JDBC e indipendentemnente dal database utilizzato fornisce una serie di funzionalità come ad esempio gli scrollable result set, la gestione dei pool di connessione, etc.
E' comunque possibile utilizzare i tipi nativi del vendor (ad esempio il tipo CLOB di Oracle) che ovviamente se utilizzati rendono l'applicativo non portabile su altri database.
Questo sottosistema offre le potenzialità di JDBC2 su qualunque tipo di database, permettendo la migrazione delle applicazioni su vari database, con uno sforzo minimo.
Il modulo di accesso ai dati è utilizzabile sia da action che da moduli: la business logic riceve i dati sotto forma di SourceBean, abbiamo visto un esempio sul paragrafo relativo alla pubblicazione dei dati, in cui dopo aver effettuato una SELECT su database non era necessario scorrere il result set, ma il risultato era disponibile sotto forma di XML.

2.18.1 Requisiti

I requisiti che si prefigge di soddisfare il modulo di accesso ai dati sono:

1. Aderenza allo standard SQL '92: deve quindi essere possibile eseguire comandi di SELECT, INSERT, UPDATE, DELETE e l'esecuzione di Stored Procedure.
2. Possibilità per la parte di business di utilizzare ed eseguire comandi su database diversi.
3. Indipendenza dal vendor e assenza di codice proprietario.
4. Indipendenza dalla versione del driver JDBC da utilizzare.
5. Gestione di pool di connessioni, in particolare il modulo dovrà essere in grado di gestire sia pool di connessioni nativi (aderenti alle specifiche JDBC 2.0) sia pool di connessione managed (definiti all'interno di un application server e recuperabili attraverso un contesto e un nome JNDI)
6. Gestione delle transazioni come definite da JDBC 2.0. Di default se una transazione non viene aperta, tutte le operazioni verranno considerate con modalità autocommit.
7. Il risultato dell'esecuzione di una query o di una istruzione deve essere conforme ad un solo tipo "autodescrivente"
8. Il formato di date o stringhe rappresentanti date o timestamp devono essere conforme ad un formato definito nel file di configurazione del modulo di accesso ai dati che sarà utilizzato dagli oggetti di business per leggere o scrivere dati di questo tipo.
9. Tutte le operazioni che nel modulo di accesso ai dati causino problemi di tipo applicativo o di sistema solleveranno eccezioni di tipo *EngInternalError*.

2.18.2 Principi generali di progettazione

Nella progettazione del modulo di accesso ai dati, visti anche i requisiti sopra esposti si è fatto molto uso di **Design Pattern,** in particolare:

- L'oggetto *DataConnectionManager* è stato implementato come un Singleton.
- Utilizzo dei pattern **AbstractFactory** e **FactoryMethod** per la creazione di tutti quegli oggetti che debbano avere un'implementazione differente pur mantenendo una stessa interfaccia.
- Gli oggetti di tipo *DataField*, *ScrollableDataResult*, *SQLCommand* sono esempi di oggetti per la quale la creazione passa attraverso un AbstractFactory o un FactoryMethod.
- Nel caso degli oggetti di tipo *SQLCommand* l'oggetto che funge da AbstractFactory configura degli oggetti conformi al pattern **Command.**
- Gli oggetti di tipo **ConnectionPoolInterface** gestiti dall'oggetto *DataConnectionManager* vengono creati attraverso una factory che lavora con l'ausilio dell'introspezione (questo per evitare in fase di compilazione di dover includere i jar di tutti i vendor).

2.18.3 Gestione delle connessioni e dei pool

L'oggetto fondamentale del modulo di accesso ai dati è costituito da *DataConnectionManager*, le quali funzionalità sono sostanzialmente due:

1. Creare e configurare pool conformi all'interfaccia *ConnectionPoolInterface*.
2. Fornire Connessioni (ossia oggetti *DataConnection*) sui vari pool di connessioni gestiti.

I pool di connessioni sono descritti in un file XML (*data_access.xml*) nel quale viene anche specificata la classe che si occupa della creazione di quel particolare pool.
Attraverso l'utilizzo di questo oggetto il programmatore degli oggetti di business non deve preoccuparsi del codice inerente alla creazione del pool e nemmeno preoccuparsi del fatto che sta utilizzando un pool.

Al programmatore dei business object sarà semplicemente necessario scrivere:

```
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()
```

I pool vengono censiti nel file di configurazione *data_access.xml* di cui riportiamo un esempio:

```
<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"/>
    <CONNECTION-POOL-PARAMETER parameterName="jdbcDriver" parameterValue="oracle.jdbc.OracleDriver"/>
    <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>
```

Ogni pool viene definito tramite una busta *CONNECTION-POOL* che ha i seguenti attributi:

- *connectionPoolName*: nome del pool. Per utilizzare un pool, nel metodo *DataConnectionManager.getConnection* va specificato il nome del pool da utilizzare, o nessun nome nel caso si voglia accedere al pool di default, che è il primo della lista.
- *connectionPoolFactoryClass*: factory che istanzia la classe di gestione del pool. Il framework fornisce le implementazioni:
  - *it.eng.spago.dbaccess.factory.**AppServerManagedConnectionPoolFactory*** che istanzia *it.eng.spago.dbaccess.pool.**AppServerManagedConnectionPool***, un pool gestito dall'application container tramite un nome JNDI. In questo caso è stata resa opzionale la presenza dei parametri "user" e "userPassword" in quanto alcuni application server prevedono che tali parametri vengano specificati direttamente in fase di configurazione del data source
  - *it.eng.spago.dbaccess.factory. **OracleConnectionPoolDataSourceFactory*** che istanzia un pool di connessioni di tipo JDBC 2.0 nativo oracle.
- *connectionString*: nome del pool (solo per il *native).*
- *jndiName*: nome jndi con cui si è definito il pool sull'Application Server (solo per il *managed*).
- *jdbcDriver*: driver JDBC da utilizzare per accedere al database, ad esempio *com.mysql.jdbc.Driver* (solo per il *native).*
- *driverVersion*: versione del driver
- *user*: utente con cui aprire la connessione.
- *userPassword*: password dell'utente specificato al punto precedente.
- *poolMinLimit*: numero minimo di connessioni da mantenere sempre aperte nel pool (solo per il *native*).
- *poolMaxLimit*: numero massimo di connessioni da mantenere nel pool (solo per il *native*).

- *sqlMapperClass*: classe da utilizzare per le conversioni delle date e dei timestamp da/a stringa. Questa classe si rende necessaria perchè alcuni tipi standard hanno una rappresentazione diversa a seconda del database: i mapper si occupano di ridurre ad un formato comune queste informazioni. Il framework fornisce i seguenti mapper:
  - *it.eng.spago.dbaccess.sql.mappers.**DB2SQLMapper***: mapper per DB2.
  - *it.eng.spago.dbaccess.sql.mappers.**MySQLMapper***: mapper per MySQL.
  - *it.eng.spago.dbaccess.sql.mappers.**OracleSQLMapper***: mapper per Oracle.

  E' possibile aggiungere nuovi mapper implementando l'interfaccia `SQLMapper`.

Per il **pool di connessioni nativo di Oracle** sono stati introdotti anche **:**

- *cacheTimeToLiveTimeout*: opzionale
- *cacheInactivityTimeout*: opzionale
- *cacheScheme*:

Con il pool nativo di Oracle esiste la possibilità di cifrare la password presente nel file di configurazione. La classe *PasswordProvider* fornisce il servizio di decifratura ed è già presente nella distribuzione del framework. Va impostata nell'attributo del file di configurazione:

- *algorithm*

L'algoritmo che utilizza è un DES/ECB/PKCS5Padding presente nelle SUN JCE, la chiave è inserita all'interno della classe *Key* presente nel jar del framework.
Per generare nuove chiavi e nuove password è stata realizzata una semplice interfaccia grafica che è possibile lanciare utilizzando l'interprete java a riga di comando: *it.eng.spago.dbaccess.encript.gui.CriptGui*
L'algoritmo utilizzato è incluso nella JDK1.4, per utilizzare questa caratteristica con la JDK 1.3/1.2 è necessario inserire nel classpath il seguente jar: jce1_2_2.jar allegato alla distribuzione del framework .

2.18.4 Esecuzione di Comandi SQL

Gli oggetti di tipo *DataConnection* rappresentano essenzialmente un wrapper sull'oggetto *java.sql.Connection*. Nel sottosistema di accesso ai dati questa classe è molto importante in quanto è a questo livello che viene gestita la creazione dei comandi e quindi la dipendenza a livello di implementazione con il driver. Per ottenere quindi dei comandi da eseguire è quindi necessario utilizzare i factory method esposti dalla classe *DataConnection*.

```
//
// 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);
```

2.18.5 Censimento degli statement

C'è la possibilità di gestire tutti gli statement in un repository XML apposito e identificare gli statement tramite nomi logici.

```
String statement = SQLStatements.getStatement("LISTA_UTENTI");
```

Il file che contiene gli statements è *statements.xml*, di cui riportiamo un esempio:

```
<?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>
```

Non è necessario accedere al catalogo degli statement: è possibile eseguire degli statement passandoli direttamente al metodo *getStatement*, ad esempio se lo statement viene costruito dinamicamente.

2.18.6 Ottenere i Risultati di un comando SQL

Come evidenziato dall'analisi dei requisiti è necessario che l'esecuzione di un comando SQL produca sempre un risultato conforme allo stesso tipo per tutti i comandi: per questo è stato studiato l'oggetto *DataResult* che non è altro che un contenitore che contiene un oggetto conforme all'interfaccia *DataResultInterface* e sa esattamente di che tipo è l'oggetto in esso contenuto.

Al solito evidenziamo il tutto con un esempio:

```
//
// 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);
i (dr.getDataResultType().equals(DataResultInterface.SCROLLABLE_DATA_RESULT)){
      ScrollableDataResult sdr = (ScrollableDataResult)dr.getDataObject();
}
```

Gli oggetti di tipo *DataResultInterface* sono sostanzialmente 4:

1. **ScrollableDataResult**: è l'oggetto più importante perché è essenzialmente un wrapper su un *java.sql.ResultSet*. I servizi offerti dall'oggetto *ScrollableDataResult* sono essenzialmente i servizi di un ResultSet di tipo standard più i servizi di navigazione diretta/inversa e di posizionamento assoluto.
   Non è fornito supporto alle operazione di update diretto sui dati del resultset per ragioni di compatibilità.

L'oggetto *ScrollableDataResult* rappresenta quindi una via di mezzo tra il tradizionale oggetto ResultSet d JDBC 1.0 e l'oggetto ResultSet di JDBC 2.0.

2. **InformationDataResult**: Come suggerito dal nome questo oggetto viene restituito come risultato di un comando di insert, update, delete per la quale ci interessa sapere se il comando è andato a buon fine e quante righe sono state inserite, cancellate, modificate.
3. **PunctualDataresult**: Questo oggetto rappresenta un tipo SQL (che non sia un result Set) resituito come valore di ritorno da una stored procedure e contiene essenzialmente un datafield.
4. **CompositeDataResult:** L'implementazione di questo oggetto si è resa necessaria in quanto è possibile definire StoredProcedure che abbiano valori di ritorno multipli tra cui vi siano ResultSet. In questo caso se una stored procedure ha come valore di ritorno un int e come parametro di output un resultSet il risultato del comando sarà un oggetto di tipo *CompositeDataResult* che conterrà a sua volta un *PunctualDataResult* e uno *ScrollableDataResult*.

```
// Apro la connessione e creo l'oggetto per
// l'invocazione della procedura PL/SQL
DataConnection conn = DataConnectionManager.getConnection();
String storeProcedureString = "{ call " + getPackageName() + ".get_all(?,?,?,?) }";
SQLCommand command = conn.createStoredProcedureCommand(storeProcedureString);

int paramIndex = 0;

// Preparo i parametri di input impostati a null
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++);

// Preparo il parametro in cui viene restituito il numero di record letti
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++);

// Chiamo la stored procedure
DataResult result = command.execute(parameters);

// Reperisco il CLOB del risultato
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());
}
```

2.18.7 Gestione delle transazioni

La gestione delle transazioni avviene in modo diverso a seconda che si esegua la logica di business su web container o EJB container.

2.18.7.1 Logica su Web container

In questa modalità tutte le operazioni effettuate su database vengono considerate autocommittanti, di default. Questa è l'impostazione che normalmente hanno le connessioni, quando prelevate dal pool di connessioni, ed è la modalità di default definita da JDBC 2.0.
Se invece si voglio gestire le transazioni in modalità esplicita vanno utilizzati i metodi ***initTransaction***, ***commitTransaction*** e ***rollBackTransaction*** della classe *it.eng.spago.dbaccess.sql.**DataConnection*** corrispondente alle connessioni prelevate dal pool.
Nel caso in cui il servizio genera un'eccezione, nessuna operazione di commit o rollback viene effettuata dal framework sulle connessioni utilizzate.
L'eventuale chiamata a servizi esterni, da parte delle action o dei moduli, rientra nell'ambito della stessa transazione solo se viene condivisa la stessa connessione.

2.18.7.2 Logica su EJB container

In questa modalità i metodi *initTransaction*, *commitTransaction* e *rollBackTransaction* non hanno alcun effetto sulla gestione delle transazioni, in quanto le transazioni vengono gestite dal Transaction Manager dell'EJB container.
Il framework attiva una nuova transazione all'inizio di ogni richiesta di servizio e ne effettua il commit quando la richiesta è stata servita. Tutti i servizi invocati dall'action o dai moduli coinvolti rientrano nell'ambito della stessa transazione.
La commit/rollback invocata dal framework ha effetto su tutte le connessioni utilizzate dal servizio, in quanto si tratta di un'unica transazione distribuita.
Nel caso in cui il servizio genera un'eccezione, viene effettuato il rollback della transazione attiva.
In effetti, un EJB container è in grado di gestire il two phase commit tramite oggetti che implementano l'interfaccia XA.Il framework quando riceve un'eccezione contatta il transaction manager e fa il rollback della user transaction, prendendosi la resposabilità di gestire le transazioni. Il rollback viene effettuato solo se il datasource implementa l'interfaccia XA. Attualmente c'è solo WebSphere.

2.18.8 Rilascio delle risorse

La politica di rilascio delle risorse rimane quella standard JDBC: ovvero *si consiglia* di rilasciare le risorse nell'ordine inverso rispetto al quale sono state allocate. Se ad esempio è stata fatta una SELECT, vanno rilasciati in ordine il cursore, lo statement e la connessione.
Abbiamo usato la frase "*si consiglia*" perchè le specifiche a riguardo non sono precise su come vadano rilasciate le risorse. Ad esempio, si potrebbe pensare che rilasciando la connessione vengano rilasciati anche tutti gli oggetti ad essa collegati: non è così.
Per semplificare la gestione delle risorse è stato messo a disposizione nel framework un metodo per rilasciare le risorse in modo automatico, nel corretto ordine:

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

2.19 Hibernate

Oltre al layer di accesso ai dati precedentemente descritto, è stato aggiunta in un secondo momento la possibilità di usare Hibernate come modulo di persistenza object/relational (http://hibernate.sourceforge.net/). L'integrazione di Hibernate ha comportato fondamentalmente la conciliazione di due aspetti:

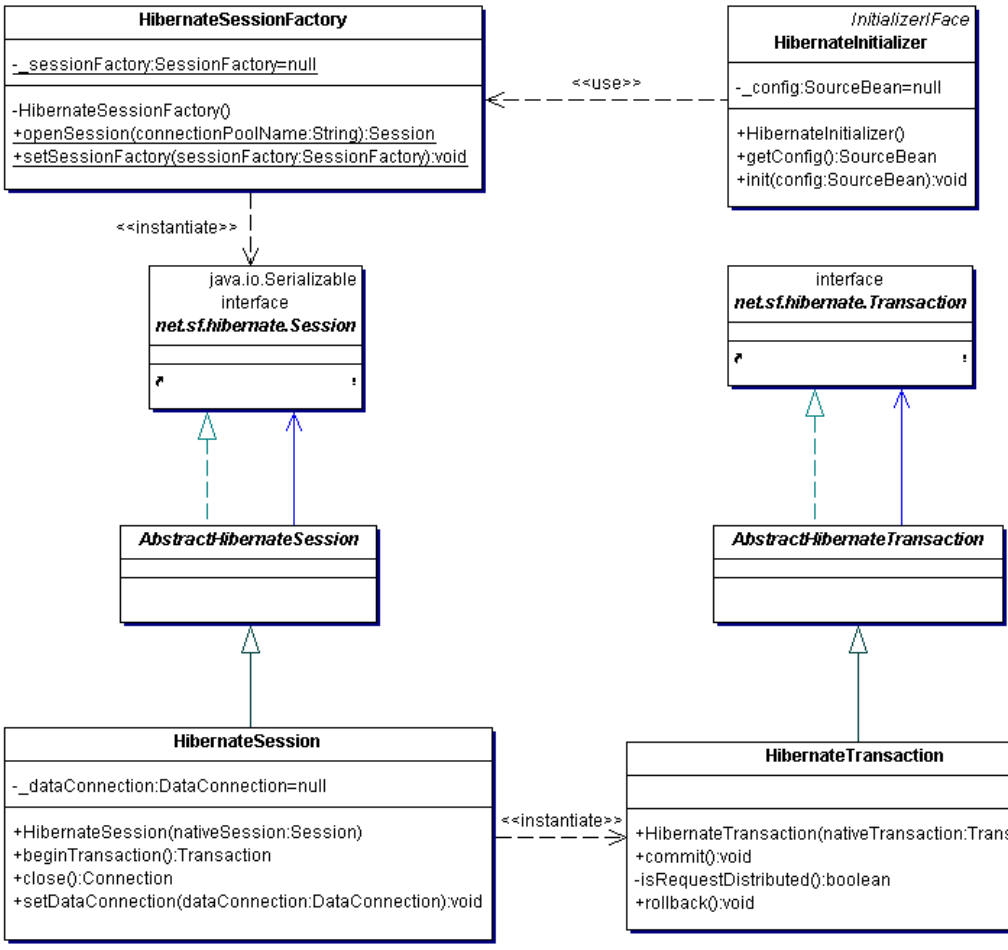
- la definizione dei pool di connessioni verso i database
- la gestione delle transazioni JDBC e JTA

Al fine di ridurre l'impegno necessario a censire tutte le regole di mapping fra gli oggetti Java e le tabelle del database, può essere utile utilizzare alcuni strumenti open source che consentono data la definizione del DB di generare automaticamente i file XML di configurazione e le classi java. L'installazione e l'utilizzo di Hibernate e dei tool esula dagli obiettivi di questo documento. Con Spago non viene rilasciato Hibernate scaricabile dal sito http://www.hibernate.org/.

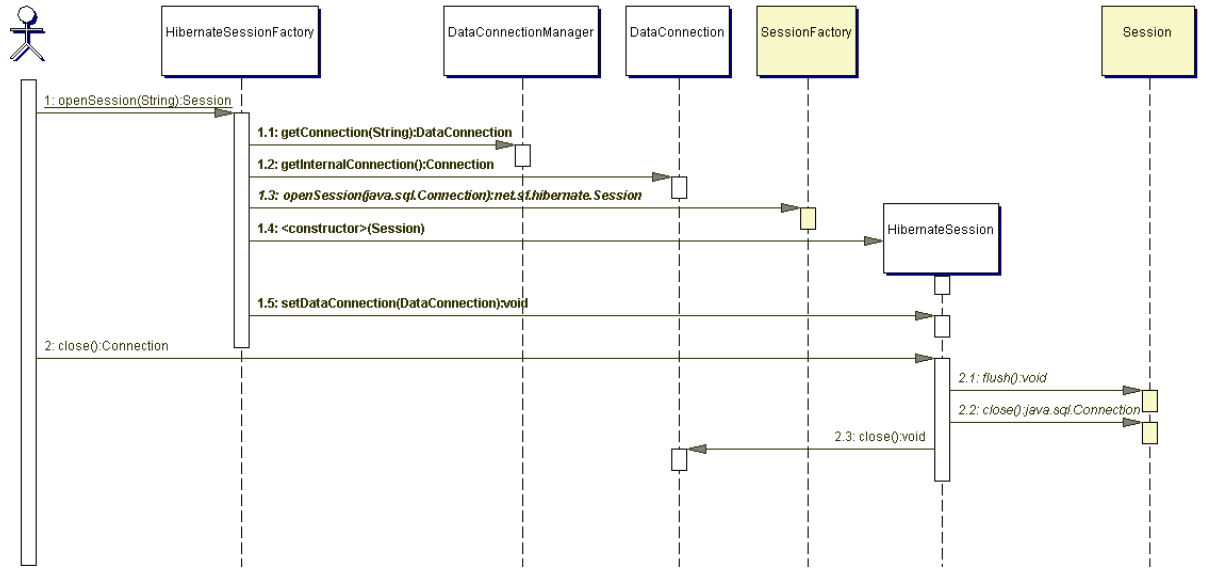
2.19.1 Integrazione Hibernate

Il punto di partenza per usare il layer di persistenza è ottenere una sessione Hibernate dato il nome di un pool di connessioni censito nel file di configurazione del framework *data_access.xml*. La classe che ha tale responsabilità è *it.eng.spago.dbaccess.hibernate.**HibernateSessionFactory***; il metodo *public static Session openSession(String connectionPoolName)* permette di ottenere un'istanza della classe *it.eng.spago.dbaccess.ibernate.**HibernateSession*** implementata come decorator della classe Session di Hibernate.

La classe *it.eng.spago.dbaccess.hibernate.**HibernateSessionFactory*** imposta la session factory di Hibernate nel metodo *init* di *it.eng.spago.dbaccess.ibernate.**HibernateInitializer*** che deve essere censita in *initializer.xml*. Una volta ottenuta la sessione Hibernate questa può essere usata come da documentazione del prodotto senza nessuna limitazione.



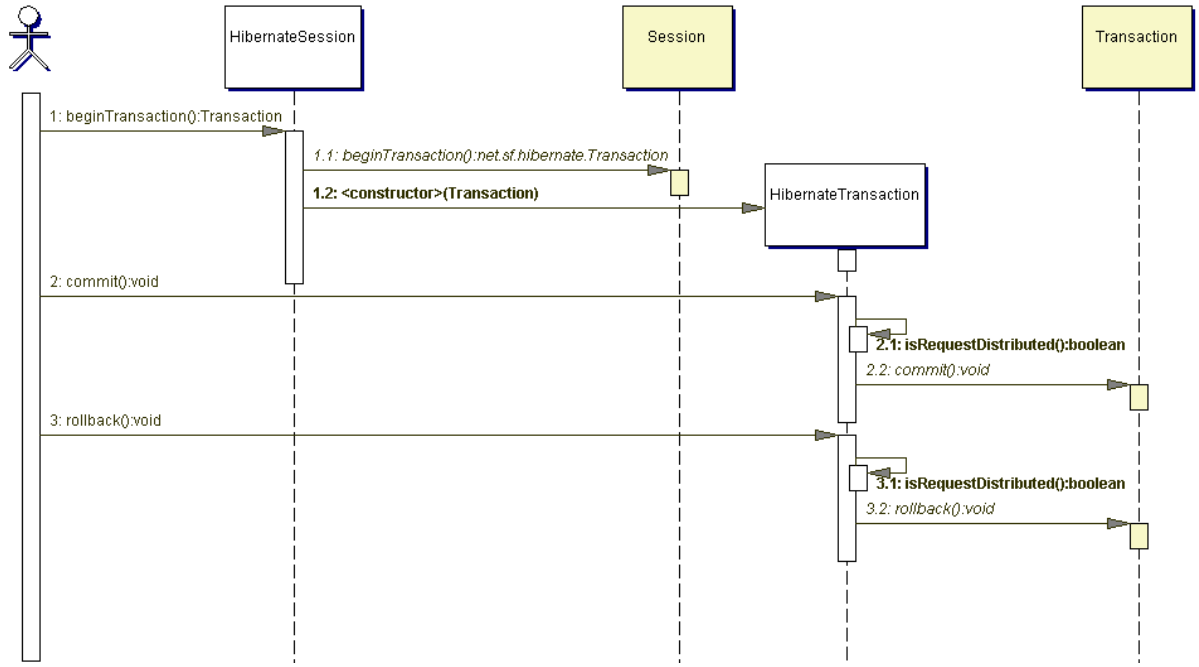
Lo scenario che descrive i servizi di apertura e chiusura della sessione è il seguente:



dove gli oggetti ***SessionFactory*** e ***Session*** sono due oggetti Hibernate.

Il diagramma delle classi mette inoltre in evidenza la realizzazione di due *decorator*, uno per l'oggetto ***Session*** e l'altro per l'oggetto ***Transaction***. Il motivo per cui sono stati introdotti riguarda la necessità di intercettare le richieste di *beginTransaction*, *commit* e *rollback* delle transazioni al fine di conciliare la gestione JTA eseguita da parte del framework con quella eseguita da parte di Hibernate.

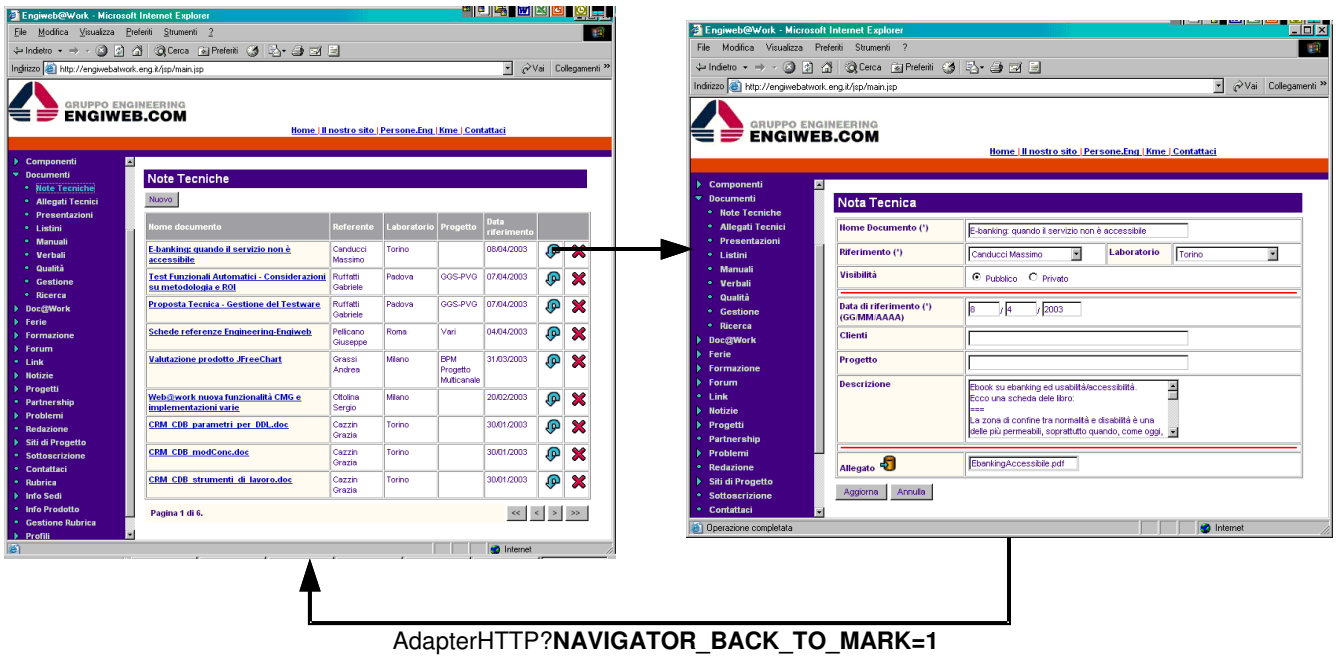
In particolare, nel caso di logica applicativa distribuita nell'EJB container, la demarcazione della transazione globale è a carico del framework e qualsiasi altro tentativo di gestire l'interfaccia JTA da parte di Hibernate deve essere ignorato. Solo nel caso in cui non sia aperta una transazione globale i servizi di *commit* e *rollback* saranno delegati all'oggetto *Transaction* di Hibernate.



2.20 Gestione navigazione

Tra i vari servizi trasversali messi a disposizione dal framework c'è anche il navigatore: è un oggetto che memorizza in uno stack le richieste ed è in grado di rispondere alla richiesta di un servizio di tipo back.
Questa funzionalità viene utilizzata ad esempio da Engiweb@Work per gestire il ritorno alle liste dopo un'attività di modifica di un documento.

Navigator.**signService**(getRequestContainer());



In questo esempio l'action che produce la lista chiede al navigatore di contrassegnare l'action corrente, tramite il metodo *signService*. In seguito, dopo aver presentato la pagina di modifica di un elemento della lista, quando l'utente conferma la modifica, per ripresentare la lista viene chiesto al navigatore di tornare alla prima action contrassegnata in precedenza, tramite il comando *NAVIGATOR_BACK_TO_MARK=1.*

Il navigatore non solo risottomette solo la richiesta, ma porta anche indietro lo stato della sessione.
Il navigatore viene attivato impostando nell'URL della richiesta determinate variabili che si chiamano *NAVIGATOR_XXX* dove *XXX* è il comando che si vuole eseguire:

- NAVIGATOR_RELOAD: si chiede al framework di rieseguire l'ultimo servizio.
- NAVIGATOR_BACK=?: permette di specificare di quante pagine a ritroso devo andare.
- NAVIGATOR_FORCEBACK=?: permette di tornare indietro considerando anche le richieste disabilitate.
- NAVIGATOR_BACK_TO=?: permette di specificare il nome del servizio a cui voglio tornare. Si tornerà all'ultima invocazione del servizio richiesto.
- NAVIGATOR_BACK_TO_MARK=?: permette di rieseguire il servizio che è stato contrassegnato dalla logica applicativa.
- NAVIGATOR_RESET: resetta tutta la storia della navigazione. Un problema di questo comando è che resetta anche i dati presenti in sessione, effetto collaterale indesiderato in alcuni casi. Il framework mette allora a

disposizione anche un container immune dal reset della storia di navigazione, accessibile tramite il session container:

public synchronized SessionContainer ***getPermanentContainer****();*

- NAVIGATOR_DISABLED: fa in modo che il comando corrente non venga considerato ai fini della navigazione. Va utilizzato quando, ad esempio, si lavora su un sito multiframe e alla pressione di un tasto più frame vengono rinfrescanti contemporaneamente. In questo caso la concorrenza lato client è anche concorrenza lato server: sul server più thread servono la richiesta e non è possibile sapere quale richiesta è stata servita per prima. Può capitare perciò che facendo il back su una richiesta venga visualizzato il frame sbagliato.
- NAVIGATOR_FREEZE="TRUE": una richiesta che porta questo comando di navigazione viene evasa senza creare un ulteriore elemento nello stack delle richieste memorizzate, ma la richiesta corrente sovrascrive la richiesta precedente.
- NAVIGATOR_BACK_TO_SERVICE_LABEL="LABEL": viene richiesta l'esecuzione dell'ultimo servizio firmato con l'etichetta LABEL attraverso il servizio della classe “Navigator”

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

E’ possibile gestire gli errori dei comandi di navigazione applicativa, registrando una action con label NAVIGATION_ERROR_ACTION.

L’uso del navigatore ha un effetto collaterale: il mantenimento dello stack delle richieste e delle sessioni è un’attività pesante. Le richieste e la sessione sono flussi XML: questi dati vengono mantenuti nell’HTTP session, per il canale HTTP, mentre per i canali SOAP e EJB sono attributi di istanza del server statefull.

Con il navigatore attivo, tutti i dati presenti in sessione vengono serializzati per poter essere registrati su database (se il session management è attivo) in modo da permettere l’uso di meccanismi di load balancing, ed è un’attività pesante.

Nella filosofia di IBM la sessione HTTP non dovrebbe contenere più di 4kbyte da serializzare/deserializzare. In sessione dovrebbero esserci solo gli identificativi degli oggetti da recuperare.

In questi casi il navigatore applicativo può essere disabilitato, per contenere la dimensione della sessione. L’uso o meno del navigatore si configura tramite il file *common.xml* impostando la variabile *navigator_enabled*.

```
<?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>
```

Nella classe AdapterHTTP è stata aggiunta la possibilità di calcolare la dimensione dei dati presenti in sessione, verificando allo stesso tempo se contiene oggetti non serializzabili, nel qual caso viene generata un'eccezione.

La dimensione della sessione viene scritta nel file di log e l'attributo che comporta tale comportamento si trova nel file *common.xml* :

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

Il default è FALSE. Si consiglia di disattivare questa opzione negli ambienti di produzione, in quanto la serializzazione della sessione è un'attività pesante.
Questa opzione può essere utile nel caso di ambienti in cui la sessione viene persistita su database, per verificare preventivamente che il contenuto della sessione sia serializzabile.

2.20.1 Toolbar di navigazione

Sono stati implementati due custom tag:

it.eng.spago.tags.DefaultNavigationToolbarTag
it.eng.spago.tags.ComboNavigationToolbarTag

per il rendering (con due modalità diverse) di una toolbar di navigazione da cui sia possibile selezionare uno dei servizi già eseguiti durante la conversazione.
I servizi selezionabili vengono etichettati con la chiamata:

public static void signService(RequestContainer request, String serviceLabel)

della classe "Navigator", dove “serviceLabel” è l'etichetta con cui il servizio sarà visualizzato nella toolbar.

2.21 Paginazione

Il paginatore è un oggetto che risponde all’esigenza di usufruire di un servizio di paginazione, senza voler utilizzare i moduli del framework per gestire liste e dettaglio, perchè ad esempio i dati non risiedono su database ma su LDAP.
Il paginatore ha la responsabilità di paginare collezioni di oggetti, lasciando al programmatore la responsabilità di recuperare gli oggetti.
Esistono due implementazioni del paginatore:

- *One-shot*: è l’implementazione più semplice e che richiede meno lavoro al programmatore. Prevede che la collezione venga letta per intero: la paginazione è solo a livello di presentazione, non a livello di recupero dei dati. Se la lista è grande possono esserci problemi di tempi di attesa (per il recupero iniziale dei dati) e elevata occupazione di memoria. Se non si possono applicare criteri di filtro questo paginatore può non essere utilizzabile.
- *Con cache*: è una modalità più evoluta della precedente, che prevede che la paginazione venga effettuata anche a livello di recupero dei dati, gestendo una “finestra” sui dati che vengono visualizzati.

2.21.1 One-shot

Il sottosistema di paginazione consiste in alcune interfacce di base e alcune implementazioni di default.

Le interfacce coinvolte, e le relative responsabilità, sono:

- *it.eng.spago.paginator.basic.**PaginatorIFace***: è l'interfaccia che definisce le funzionalità di un paginatore. Comprende tutti i servizi necessari a gestire una lista di pagine.
  - *void addRow(Object row)*: aggiunge una riga all'elenco degli oggetti gestiti dal paginatore.
  - *int getPageSize()*: restituisce la dimensione della pagina.
  - *void setPageSize(int size)*: imposta la dimensione della pagina.
  - *int rows()*: restituisce il numero di righe della lista.
  - *int pages()* : restituisce il numero di pagine.
  - *SourceBean getPage(int page)* : restituisce una pagina.
  - *SourceBean getAll()*: restituisce tutta la lista.

  Il framework fornisce un'implementazione di questa interfaccia nella classe *it.eng.spago.paginator.basic.impl.**GenericPaginator***. Alle istanze di questa classe va impostata la lista da paginare, usando il metodo *addRow* per aggiungere tutti gli elementi della lista.

- *it.eng.spago.paginator.basic. **ListIFace**:* è l' interfaccia che definisce tutti i metodi necessari per gestire una lista paginata, con servizi che restituiscono il numero di pagine, i contenuti della pagina corrente, permettono di avanzare sulle pagine, e così via.

  Il framework fornisce un'implementazione di questa interfaccia nella classe *it.eng.spago.paginator.basic.impl.**GenericList***. L'oggetto *GenericList* utilizza un'istanza di *GenericPaginator* per gestire la paginazione. All'interno di *GenericList* sono contenute anche informazioni statiche come le etichette delle colonne.

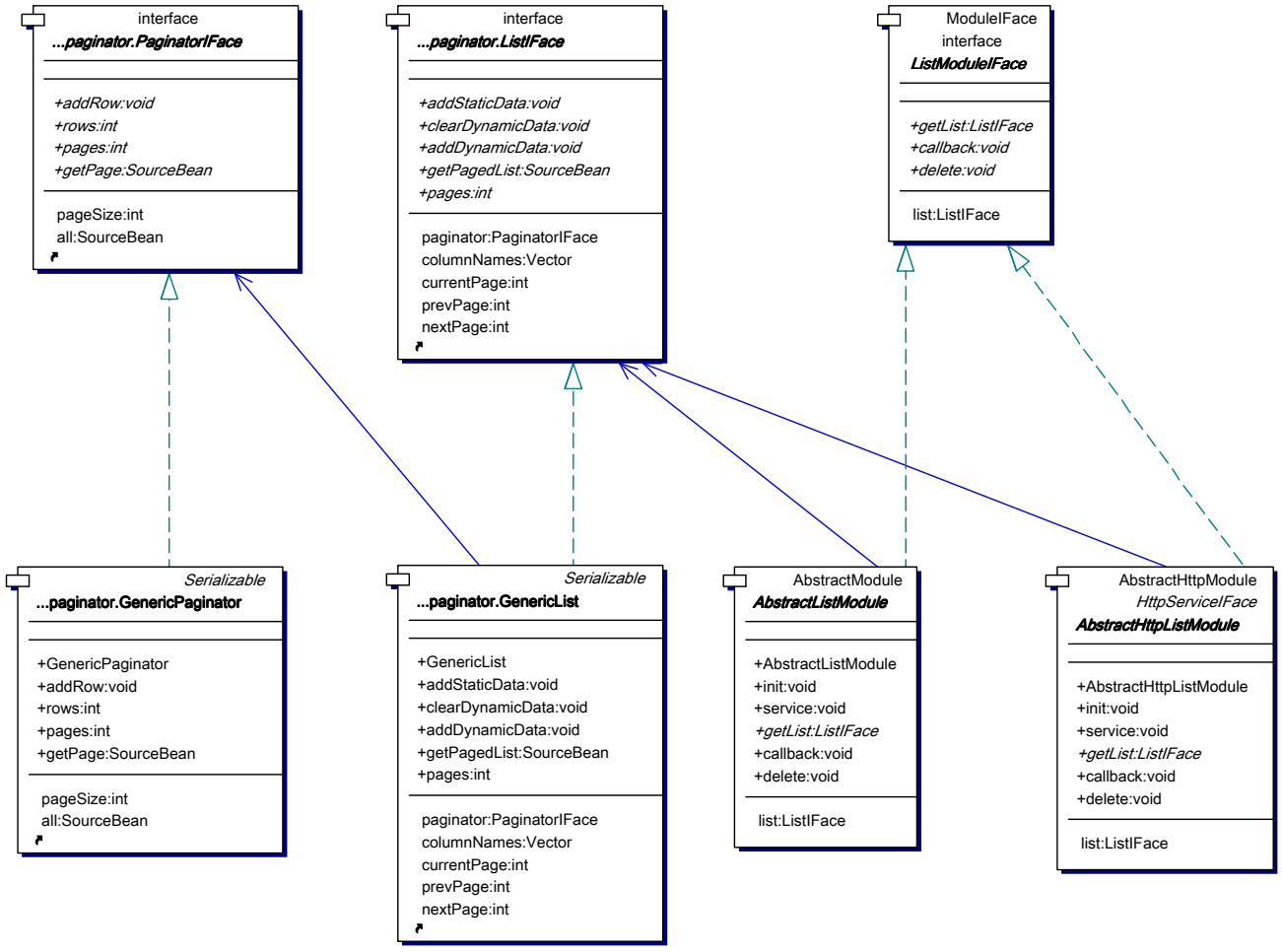
  Per arricchire la risposta di ulteriori contenuti si hanno a disposizione i metodi *addStaticData* e *addDynamicData*:
  - *addStaticData*: permette di inserire nella risposta delle buste XML che arrivano alla parte di presentazione, in una busta STATIC_DATA. Va usata per aggiungere alla risposta tutti quei contenuti che sono costanti nella navigazione della lista.
  - *addDynamicData*: analogo al metodo *addStaticData*, ma le informazioni aggiunte vengono perse una volta che il servizio è stato evaso.

- *it.eng.spago.dispatching.service.list.basic.IFaceBasicListService*: è l'interfaccia che definisce i servizi di un modulo che gestisce una lista paginata. Definisce i seguenti metodi:
  - *public ListIFace **getList()***: è il metodo che recupera l'intera lista (che ha capacità di paginazione, definite dall'interfaccia ListIFace).
  - *public void **callback(SourceBean request, ListIFace list, int page)***: viene invocato dal framework ogni qualvolta, durante la navigazione dell'utente, viene attivata la paginazione, ovvero ad ogni cambio pagina. Permette di aggiungere alla risposta del modulo delle informazioni che servono all'attività di rendering.
  - *public void **delete**(SourceBean request)*: viene invocato quando viene cancellato un elemento dalla lista.

Il framework fornisce fornisce un'implementazione di questa interfaccia nelle classi *it.eng.spago.dispatching.module.list.basic.**AbstractBasicListModule*** e *it.eng.spago.dispatching.module.**AbstractHttpListModule***.

Queste implementazioni sono comunque classi astratte in quanto definiscono solo i metodi *callback* e *delete* (vuoti), lasciando allo sviluppatore la responsabilità di implementare il metodo *getList.*

La classe *AbstractHttpListModule* è la variante di *AbstractListModule* che permette di accedere al contesto HTTP del canale.



Questo paginatore (*GenericPaginator)* richiama il metodo *getList* una sola volta (o tutte le volte che viene chiesto il refresh della lista, con un comando esplicito), ma tramite il servizio *callback* possiamo comunque essere informati quando l'utente chiede di navigare sulle pagine.

Non c'è modo di modificare i dati, una volta recuperati.

Vista la struttura del sottosistema di paginazione one-shot, come si usa ?

Ci sono due possibilità:

1. La più semplice è quella di utilizzare la funzionalità del framework di generazione automatica dei form di liste e dettaglio, spiegata nel paragrafo Generazione automatica lista/dettaglio.
   Tale funzionalità fornisce un automatismo che permette di generare automaticamente pagine di liste e dettaglio di dati recuperati da database.
2. L'altra modalità consiste nel creare un nuovo modulo, estendendo la classe *AbstractListModule* o *AbstractHttpListModule*. Questa classe dovrà definire il metodo:
   public ListIFace getList(SourceBean request)

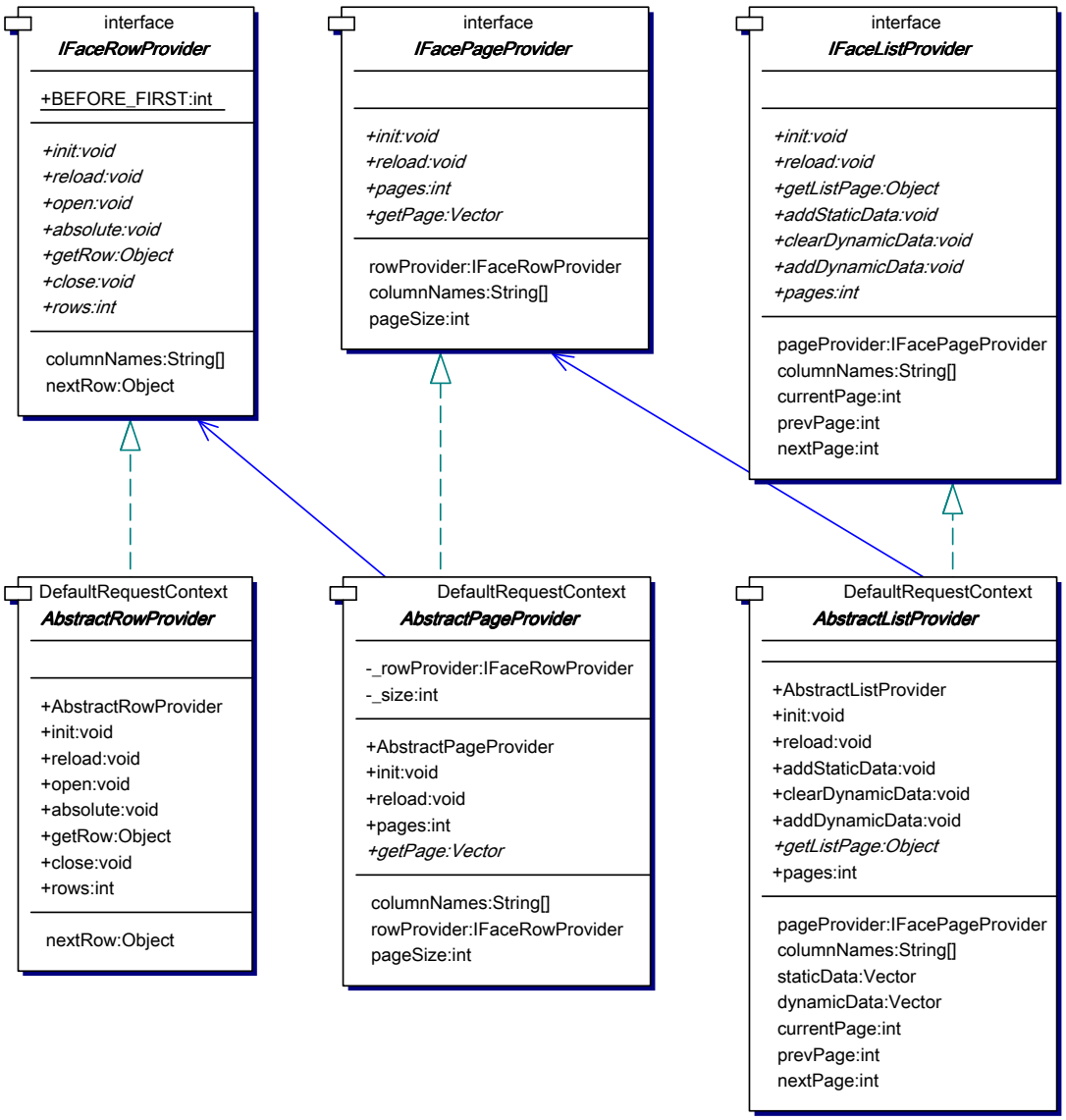
   per far ciò, dovrà istanziare un *GenericPaginator* al quale aggiungere le righe della lista, e una *GenericList* a cui associare il paginatore.
   Un esempio di tale attività è presente nel modulo *it.eng.spago.dispatching.module.list.basic.impl.**DefaultBasicListModule*** implementato per la funzionalità di Generazione automatica lista/dettaglio.

```
// Istanziazione del paginatore
PaginatorIFace paginator = new GenericPaginator();
paginator.setPageSize(<numero di righe per pagina>);

// Aggiungo TUTTE le righe al paginatore
for (int i = 0; i < <numero di righe>; i++)
   paginator.addRow(<riga i-esima>);

// Istanziazione della lista
ListIFace list = new GenericList();
// Associazione lista-paginatore
list.setPaginator(paginator);
list.setColumnNames(<vettore contenente i nomi delle colonne>);
return list;
```

3. Una volta definito il modulo, andrà definita una pagina JSP che fa uso di questo modulo. La pagina farà uso di apposite tag library del framework per il rendering. Queste tag library sono spiegate in dettaglio nella sezione Generazione automatica lista/dettaglio-Visualizzazione.

2.21.2 Con cache

Il paginatore visto al paragrafo precedente ha il grosso limite che carica la lista completamente in memoria: serve un meccanismo di paginazione più efficiente.Per questo è stato introdotto un nuovo paginatore.
Questo nuovo paginatore cerca anche di risolvere un altro problema: fornire un automatismo per paginare dati che possono provenire da un qualunque datasource. Si vuole poter recuperare dati da sistemi LDAP, da database, etc, semplicemente lavorando su parametri di configurazione.
Il sottosistema di paginazione definisce alcune interfacce che identificano le responsabilità di una serie di oggetti coinvolti nella funzionalità di paginazione.



Le 3 interfacce definiscono le seguenti responsabilità:

- *IFaceRowProvider*: definisce principalmente il metodo per il recupero di una riga (*getRow*).
- *IFacePageProvider*: definisce le funzioni necessarie alla gestione di una pagina, che è un sottoinsieme della collezione. Si appoggia all'interfaccia *IFaceRowProvider* per reperire le righe. E' a questo livello che si possono introdurre algoritmi di caching più intelligenti del caching totale.
- *IFaceListProvider*: definisce le funzionalità per la gestione di una lista. Ha la responsabilità di completare le informazioni del sottoelenco della pagina con le informazioni delle etichette delle colonne,etc.
  Chi implementa questa interfaccia riceve le callback per popolare, eventualmente informazioni diverse dalla collezione da paginare, che guidano il rendering dei contenuti.

L'implementazione del framework effettua una paginazione effettiva, a differenza del paginatore one-shot: più precisamente la SELECT per recuperare i dati da database estrae tutta la lista, ma il fetch del risultato viene fatto solo sui dati che

interessano. In questo modo non viene eliminato il tempo necessario per effettuare l'operazione di SELECT su database, ma viene eliminato quello di trasferimento dei dati.

Il framework fornisce un'implementazione delle suddette interfacce che sono rispettivamente:

- *DBRowProvider*: permette di reperire dati da database, mediante l'interpretazione della grammatica della busta <ROW_PROVIDER> del file *actions.xml*. Esistono due modalità per reperire lo statement da eseguire:
  - Recuperandolo dal file *statements.xml* mediante il nome logico definito nell'attributo "statement" delle buste <*_QUERY> all'interno del tag <CONFIG> come nel seguente esempio:

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

  - Delegando ad una apposita classe, che implementa l'interfaccia IfaceQueryProvider, la costruzione dinamica dello statement. Il nome della classe che svolge tale compito è definito nell'attributo "class", del tag <*_QUERY>, qualora l'attributo "statement" non sia presente o abbia valore di stringa vuota. Qui di seguito un esempio di grammatica:

```
<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>
```

- *CacheablePageProvider*: questo paginatore mantiene un certo numero di pagine laterali a quella corrente, nell'ipotesi che l'utente navighi intorno alla pagina corrente (è un meccanismo di logica predittiva banale). Si può configurare quante pagine laterali a quella corrente mantenere in memoria.
  Attualmente il caricamento delle pagine è contestuale al click dell'utente. Si sta cercando di verificare se è possibile effettuare il caricamento in maniera ascincrona accorgendosi dell'approssimarsi dell'avvicinamento ai limiti della finestra in memoria.
- *DefaultListPRovider*: ha un suo file di configurazione dove legge le informazioni necessarie per costruire le etichette delle colonne.

Il limite principale di questa implementazione è che attualmente è in grado di reperire le informazioni solo dal database.
Un esempio di utilizzo di questa implementazione si trova nella sezione Generazione automatica lista/dettaglio-con cache.

2.22 Generazione automatica lista/dettaglio-one shot

Questa funzionalità è un esempio di sviluppo, secondo la modalità a moduli, di una lista paginata **con il paginatore one-shot**.

E' un'esigenza piuttosto comune quella di gestire dati su tabelle: liste paginate, visualizzazione dettaglio, poter decidere quante righe visualizzare per pagina, definire quali campi sono modificabili, il tutto configurato da file XML.

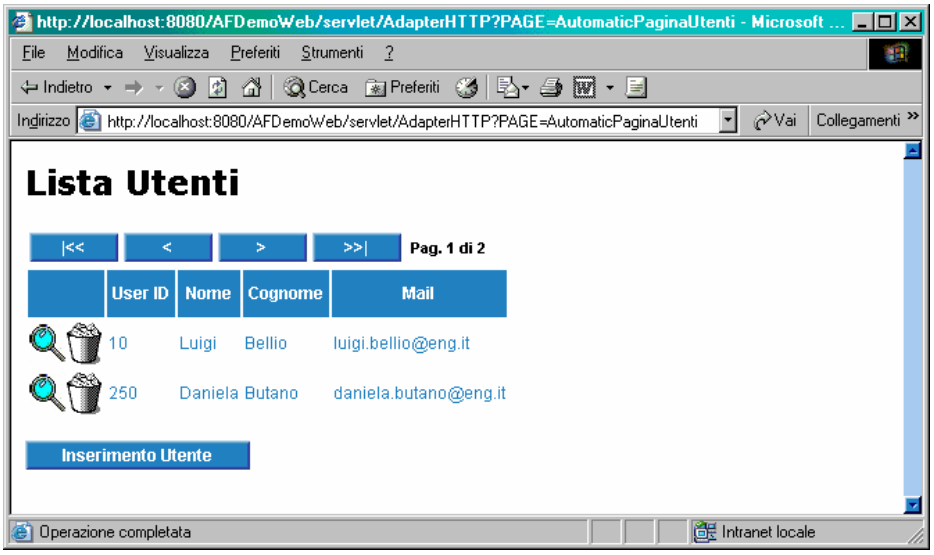
Questo esempio usa logica di business (moduli) e tag library.

E' comunque possibile estendere il meccanismo con nuove tag library.

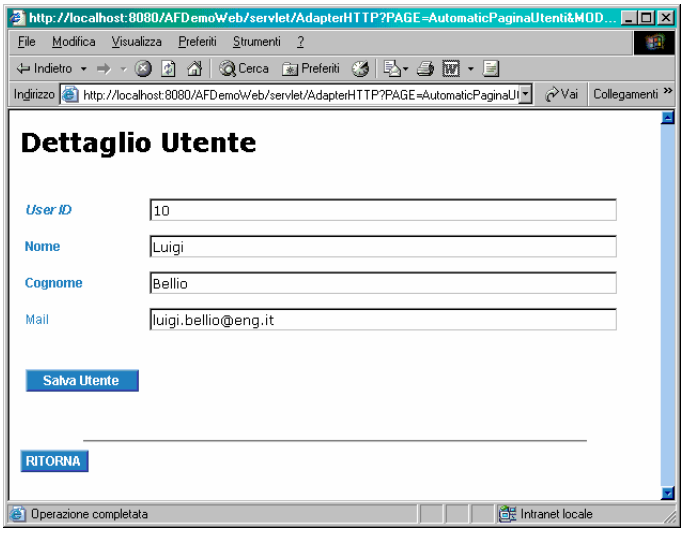
Questa funzionalità fa uso dei moduli *it.eng.spago.dispatching.module.list.basic.impl.**DefaultBasicListModule***, per la generazione della lista, e *it.eng.spago.dispatching.module.detail.impl.**DefaultDetailModule*** per la generazione del dettaglio.

La classe DefaultListModule fa uso delle funzionalità del paginatore one-shot, e come illustrato nel paragrafo One-shot, estende il modulo astratto *AbstractListModule*.

Fornendo alcune informazioni di configurazione, questo modulo con l'utilizzo di tag library fornite dal framework è in grado di produrre pagine contenenti liste di dati recuperate da database.



La lista può avere funzionalità di visualizzazione dettaglio, cancellazione elementi, inserimento elementi e navigazione tra le pagine.



2.22.1 Lista

Supponiamo di voler accedere ad una tabella strutturata come segue:

Vediamo innanzitutto la configurazione del modulo per la visualizzazione della lista, nel 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="LISTA_UTENTI"/>
      <DELETE_QUERY statement="DELETE_UTENTE">
        <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="Dettaglio Utente" confirm="FALSE">
        <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPaginaUtenti" scope=""/>
        <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDettaglioUtente" scope=""/>
```

```
        <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
      </SELECT_CAPTION>
      <DELETE_CAPTION image="" label="Elimina Utente" confirm="TRUE">
        <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
      </DELETE_CAPTION>
    </CAPTIONS>
```

```
    <BUTTONS>
      <INSERT_BUTTON image="" label="Inserimento Utente" confirm="FALSE">
        <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPaginaUtenti" scope=""/>
        <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDettaglioUtente" scope=""/>
      </INSERT_BUTTON>
    </BUTTONS>
  </CONFIG>
</MODULE>
```

Il modulo contiene una sezione di configurazione *CONFIG* in cui sono definiti i seguenti attributi:

- *pool*: contiene il nome del connection pool da utilizzare per effettuare la connessione al database. Per la spiegazione di come configurare i connection pool si veda la sezione *Accesso ai Dati*
- *title*: titolo della lista.

- *rows*: numero di righe di ciascuna pagina.

Oltre ai suddetti attributi, la sezione *CONFIG*, contiene le seguenti buste XML di configurazione:

- QUERIES: contiene i riferimenti agli statement (censiti nel file *statements.xml*) per il recupero dell'intera lista (*SELECT_QUERY*), e per la cancellazione di un elemento della lista (*DELETE_QUERY)*.

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

In questo caso le query sono definite come segue:

```
<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 = ?"/>
```

Nel caso in cui le query prevedano dei placeholder, come ad esempio la query di cancellazione, è possibile recuperare i parametri dalla richiesta, dalla sessione, o dall'applicazione, con la stessa grammatica utilizzata per definire le condizioni nei moduli (vedi Condizioni).
Vanno specificate tutte le query relative alle caption (i bottoni che compaiono all'inizio di ogni riga) che si specificano nella busta *CAPTIONS*.

- COLUMNS: contiene la definizione delle colonne da visualizzare nella lista. Per ciascuna colonna è specificato il nome della colonna su database (*name*) e l'etichetta da visualizzare sulla pagina (*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>
```

La query di recupero dei dati può anche reperire più dati di quelli che vengono visualizzati nella lista.

- CAPTIONS: contiene la configurazione dei bottoni che permettono di visualizzare il dettaglio di una riga (*SELECT_CAPTION*) e per la cancellazione di una riga (*DELETE_CAPTION*).

```
<CAPTIONS>
  <SELECT_CAPTION image="" label="Dettaglio Utente" confirm="FALSE">
    <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPaginaUtenti" scope=""/>
    <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDettaglioUtente" scope=""/>
    <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
  </SELECT_CAPTION>
  <DELETE_CAPTION image="" label="Elimina Utente" confirm="TRUE">
    <PARAMETER name="id_user" type="RELATIVE" value="id_user" scope="LOCAL"/>
  </DELETE_CAPTION>
</CAPTIONS>
```

Le caption possibili sono:

- SELECT_CAPTION: bottone di visualizzazione dettaglio. Utilizza la query *SELECT_QUERY* per recuperare il dettaglio.Va indicata la pagina di visualizzazione del dettaglio, specificando il parametro *PAGE* puntando alla pagina che contiene il modulo di visualizzazione dettaglio. Va definito il parametro che sostituirà il placeholder nella query di selezione.
- DELETE_CAPTION: bottone di cancellazione di una riga. Viene utilizzata la query *DELETE_QUERY.* Anche in questo caso vanno definiti i parametri per la sostituzione dei placeholder. Dopo aver effettuato la cancellazione della riga si torna sempre alla lista.

  Per questo bottone, viene applicato il meccanismo di prevent resubmit, disattivabile impostare l'attributo prevent_resubmit="FALSE"

E' comunque possibile aggiungere una caption customizzata utilizzando la seguente sintassi (all'interno della busta *CAPTIONS*):

```
<CAPTION image="URL IMMAGINE" label="ETICHETTA" confirm="TRUE | FALSE">
  <PARAMETER name="PAGE" type="ABSOLUTE" value="NOME PAGINA" scope=""/>
  <PARAMETER name="MODULE" type="ABSOLUTE" value="NOME MODULO" scope=""/>
  LISTA PARAMETRI AGGIUNTIVI
</CAPTION>
```

Va specificata l'URL dell'immagine da utilizzare per la caption (*image*), l'etichetta (*label*), e un flag che indica se deve essere richiesta conferma dell'operazione tramite un pop-up generato da javascript (*confirm*).
Inoltre va specificata la pagina destinazione della caption, specificando il parametro *PAGE* ed eventualmente il parametro *MODULE*. Vanno inoltre specificati tutti i parametri aggiuntivi che servono alla pagina di destinazione.

- BUTTONS: contiene la configurazione dei bottoni da visulizzare sotto alla lista.

```
<BUTTONS>
  <INSERT_BUTTON image="" label="Inserimento Utente" confirm="FALSE">
    <PARAMETER name="PAGE" type="ABSOLUTE" value="AutomaticPaginaUtenti" scope=""/>
    <PARAMETER name="MODULE" type="ABSOLUTE" value="AutomaticDettaglioUtente" scope=""/>
  </INSERT_BUTTON>
</BUTTONS>
```

I bottoni possibili sono:

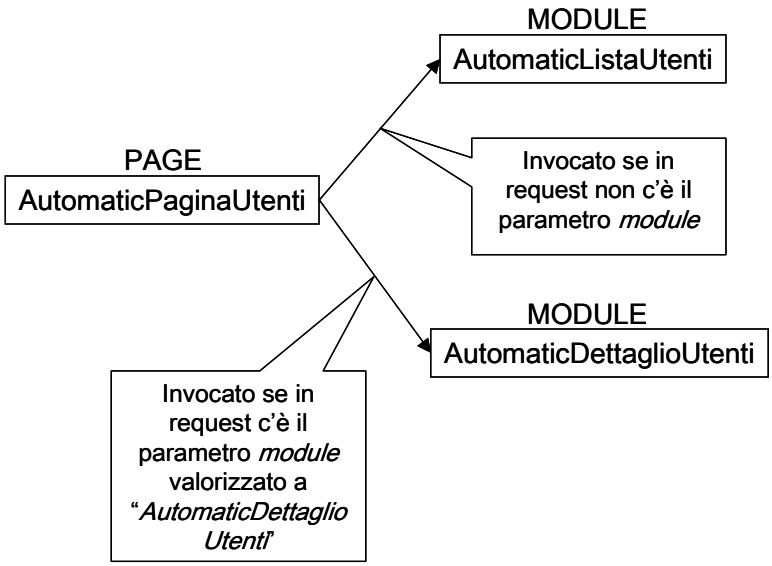
- INSERT_BUTTON: bottone per l'inserimento di una nuova riga. Va indicata la pagina che contiene il modulo di visualizzazione dettaglio, tramite il parametro *PAGE*.

E' comunque possibile aggiungere bottoni customizzati utilizzando la seguente sintassi (all'interno della busta *BUTTONS*):

```
< BUTTON image="URL IMMAGINE" label="ETICHETTA" confirm="TRUE | FALSE">
  <PARAMETER name="PAGE" type="ABSOLUTE" value="NOME PAGINA" scope=""/>
  <PARAMETER name="MODULE" type="ABSOLUTE" value="NOME MODULO" scope=""/>
  LISTA PARAMETRI AGGIUNTIVI
</BUTTON>
```

Va specificata l'URL dell'immagine da utilizzare per la caption (*image*), l'etichetta (*label*), e un flag che indica se deve essere richiesta conferma dell'operazione tramite un pop-up generato da javascript (*confirm*).
Inoltre va specificata la pagina destinazione del bottone, specificando il parametro *PAGE* ed eventualmente il parametro *MODULE*. Vanno inoltre specificati tutti i parametri aggiuntivi che servono alla pagina di destinazione.

Nel suddetto esempio, è stata utilizzata un'unica pagina per la visualizzazione della lista e del dettaglio, configurata come segue:



2.22.2 Comandi della Lista

E' possibile inviare alcuni messaggi al modulo di lista, specificando alcuni parametri nella richiesta.
Ecco i messaggi che è possibile inviare:

- Pagina precedente: va inserito nella richiesta il parametro ***MESSAGE=LIST_PREV***.
- Pagina successiva: va inserito nella richiesta il parametro ***MESSAGE=LIST_NEXT***.
- Prima pagina: va inserito nella richiesta il parametro ***MESSAGE=LIST_FIRST***.
- Ultima pagina: va inserito nella richiesta il parametro ***MESSAGE=LIST_LAST***.
- Pagina i-esima: va inserito nella richiesta il parametro ***MESSAGE=LIST_PAGE&LIST_PAGE=i*** dove *i* è l'indice della pagina da visualizzare.
- Ricaricare la lista: va inserito nella richiesta il parametro ***LIST_NOCACHE=TRUE***. Questo parametro va specificato, ad esempio, quando la ricerca è legata ad un form contenente i parametri di ricerca: al cambiamento

dei parametri di ricerca la query deve essere rieffettuata, mentre la navigazione tramite la toolbar non deve ricaricare la lista.

I primi 4 messaggi vengono utilizzati per gestire la toolbar di navigazione.

2.22.3 Dettaglio

Analizziamo ora la configurazione del modulo di dettaglio *AutomaticDettaglioUtente* dal file *modules.xml*:

```
<MODULE
  name="AutomaticDettaglioUtente"
  class="it.eng.spago.dispatching.module.detail.impl.DefaultDetailModule">
  <CONFIG pool="afdemo" title="Dettaglio Utente">
    <QUERIES>
      <INSERT_QUERY statement="INSERT_UTENTE">
        <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_UTENTE">
        <PARAMETER type="RELATIVE" value="id_user" scope="SERVICE_REQUEST"/>
      </SELECT_QUERY>
      <UPDATE_QUERY statement="UPDATE_UTENTE">
        <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>
```

Come per il modulo di lista sono presenti i seguenti attributi:

- *pool*: contiene il nome del connection pool da utilizzare per effettuare la connessione al database. Per la spiegazione di come configurare i connection pool si veda la sezione *Accesso ai Dati*
- *title*: titolo del dettaglio.

Inoltre la sezione *CONFIG* contiene le seguenti buste XML di configurazione:

- QUERIES: contiene i riferimenti agli statement (censiti nel file *statements.xml*) per l'inserimento su database di una riga, lettura e aggiornamento:
  - INSERT_QUERY
  - SELECT_QUERY
  - UPDATE_QUERY
- FIELDS: contiene la definizione di tutti i campi che devono essere presenti sulla pagina di dettaglio, sia essa di visualizzazione, inserimento o aggiornamento. Per ogni campo dovrà essere specificata una busta *FIELD* con i seguenti attributi:
  - *name*: nome della colonna su database:
  - *label*: etichetta del campo.
  - *widget*: widget da utilizzare per la rappresentazione. Attualmente non va valorizzato perchè è supportato solo il widget corrispondente all'area di testo.
  - *size*: dimensione del campo.
  - *is_readonly*: indica se il campo è editabile.
  - *is_mandatory*: indica se il campo è obbligatorio. I campi obbligatori vengono rappresentati con le etichette in grassetto. Inoltre per i campi obbligatori viene anche generato il codice Javascript di controllo lato client.
  - *is_visible*: indica se il campo è visibile.
- BUTTONS: contiene la configurazione dei bottoni da visulizzare sotto alla lista. I valori possibili sono:
  - SUBMIT_BUTTON: bottone di submit del form di inserimento/aggiornamento. Se l'attributo *confirm* è valorizzato a *true* viene generato il codice Javascript che chiede conferma dell'operazione.

Per questo bottone, viene applicato il meccanismo di prevent resubmit, disattivabile impostare l'attributo prevent_resubmit="FALSE"

Come nel caso della lista è possibile aggiungere nuovi bottoni aggiungendo delle buste *BUTTON*.

2.22.4 Visualizzazione

Abbiamo visto la configurazione dei moduli di lista e dettaglio: ma come viene gestita la presentazione ?
Nel suddetto esempio abbiamo utilizzato la stessa pagina per la visualizzazione della lista e del dettaglio, discriminando le due visualizzazioni in base alla presenza nella richiesta del parametro *module*.
Per la presentazione è stata utilizzata una pagina JSP, configurando il file *presentation.xml* come segue:

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

```
<PRESENTATION>
  <MAPPING
    business_name="AutomaticPaginaUtenti"
    business_type="PAGE"
    publisher_name="AutomaticPaginaUtenti"/>
</PRESENTATION>
```

e il file publishers.xml come segue:

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

La pagina JSP utilizza una tag library del framework per effettuare il rendering:

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

Questa tag library definisce i seguenti tag:

- *<spago:**error**/>*: accede allo stack degli errori (*EMFErrorHandler*) generando una funzione javascript che si chiama *checkError()*. Tale funzione deve essere richiamata al caricamento della pagina JSP, con una sintassi del tipo:

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

  nell'apertura del tag *body*. La funzione javascript produrrà un alert che riporta la lista di tutti gli errori contenuti nello stack degli errori. Ecco un esempio del risultato e del codice che lo produce.

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

- *<spago:**list** moduleName="nome del modulo" />*: effettua il rendering della lista.
  Sia nella lista che nel dettaglio è possibile configurare i moduli in modo che venga richiesta conferma delle operazioni da effettuare, funzione solitamente utilizzata nella cancellazione delle righe. Perchè questa funzionalità sia disponibile, il tag della lista genera una funzione javascript che si chiama *goConfirm(url,alertFlag)* che viene richiamata in caso di richiesta di conferma operazione. Riportiamo di seguito la funzione generata, e un pop-up di esempio da questa prodotto:

  ```
  <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>
```

- *<spago:**detail** moduleName="nome del modulo"/>*: effettua il rendering del dettaglio.

Ed ecco come si presenta la pagina JSP:

```
<%@ 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="AutomaticListaUtenti" />
<% } else { %>
   <spago:detail moduleName="AutomaticDettaglioUtente" />
   <hr width="80%" size="2">
   <table>
      <tr><td>
         <input class="ListButtonChangePage" type="button" value="RITORNA"
            onclick="window.location='AdapterHTTP?NAVIGATOR_BACK=1&LIST_NOCACHE=TRUE'" />
      </tr></td>
   </table>
<% } %>
</body>
</html>
```

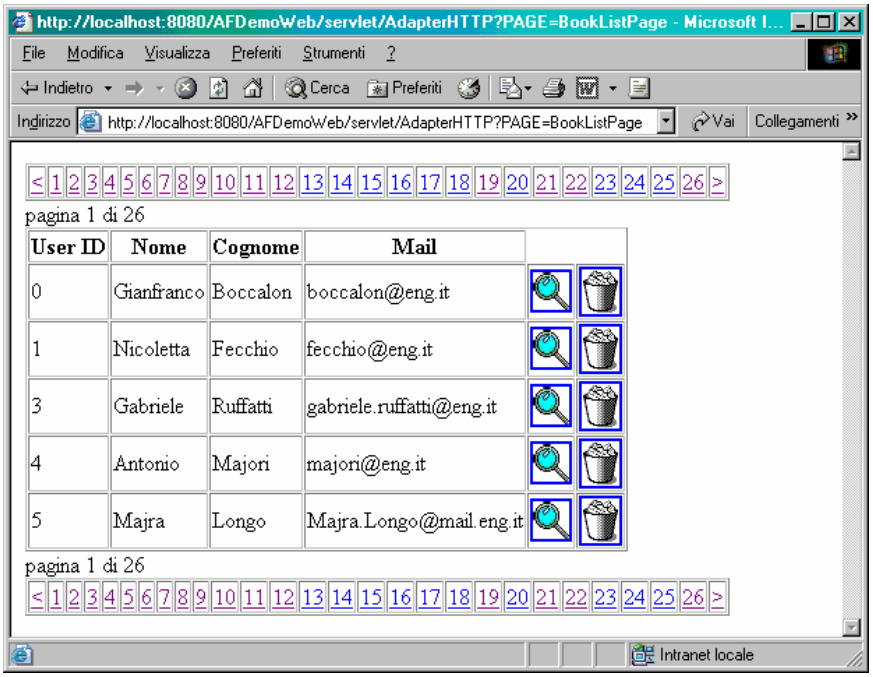
nel caso in cui si voglia modificare il layout delle pagine prodotte va modificata la tag library del framework o ne va prodotta una di nuova.

2.23 Generazione automatica lista/dettaglio-con cache

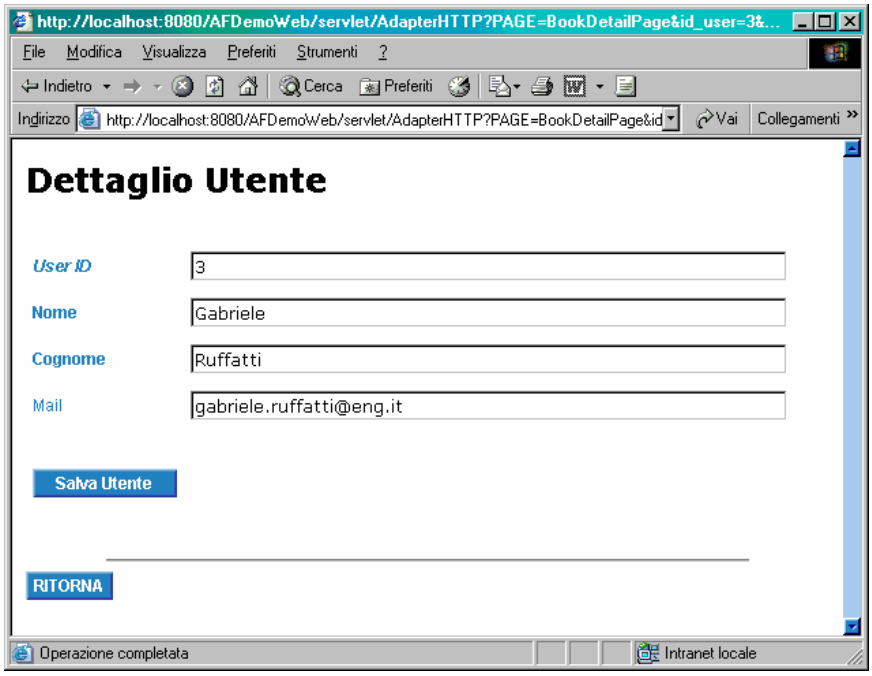
Questa funzionalità fa uso dei moduli *it.eng.spago.dispatching.module.list.smart.impl.**DefaultSmartListModule***, per la generazione della lista, e *it.eng.spago.dispatching.module.impl.**DefaultDetailModule*** per la generazione del dettaglio (ovvero lo stesso modulo utilizzato al paragrafo precedente).

La classe DefaultSmartListModule fa uso delle funzionalità del paginatore con cache, e come illustrato nel paragrafo Con cache, estende il modulo astratto *AbstractSmartListModule*.

Fornendo alcune informazioni di configurazione, questo modulo con l'utilizzo di tag library fornite dal framework è in grado di produrre pagine contenenti liste di dati recuperate da database.

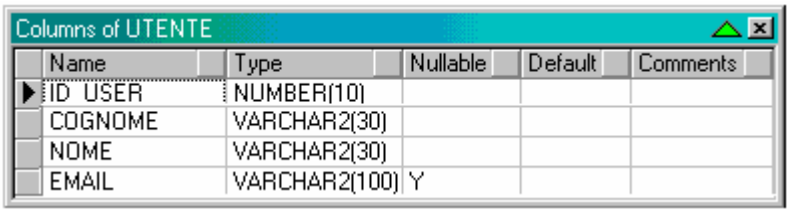


La lista può avere funzionalità di visualizzazione dettaglio, cancellazione elementi, inserimento elementi e navigazione tra le pagine.



2.23.1 Lista

Supponiamo di voler accedere ad una tabella strutturata come segue:



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

Vediamo innanzitutto la configurazione del modulo per la visualizzazione della lista, nel file *modules.xml*:

```
<MODULE name="SmartListaUtenti" class="it.eng.spago.dispatching.module.list.smart.impl.DefaultSmartListModule">
    <CONFIG title="Smart Lista Utenti Module">
        <ROW_PROVIDER class="it.eng.spago.paginator.smart.impl.DBRowHandler">
            <CONFIG pool="afdemo">
                <LIST_QUERY statement="LISTA_UTENTI"></LIST_QUERY>
                <DELETE_QUERY statement="DELETE_UTENTE">
                    <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="Dettaglio Utente" confirm="FALSE">
                                <PARAMETER name="PAGE" type="ABSOLUTE" value="SmartPaginaUtenti" scope=""
                                />
                                <PARAMETER name="MODULE" type="ABSOLUTE" value="DettaglioUtente" 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="SmartPaginaUtenti" scope=""
                                />
                                <PARAMETER name="MODULE" type="ABSOLUTE" value="DettaglioUtente" scope=""
                                />
                        </INSERT_BUTTON>
                </BUTTONS>
        </CONFIG>
</MODULE>


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

Il modulo contiene una sezione di configurazione *CONFIG* in cui sono definiti i seguenti attributi:

- *ROW_PROVIDER*: l'attributo *class* contiene il nome logico del row handler da utilizzare.

  Nella busta di configurazione *CONFIG* viene definito il nome del pool da utizzare per accedere al database (*pool*). Vengono inoltre forniti i nomi logici delle query da utilizzare per le operazioni di recupero lista (*LIST_QUERY*), e cancellazione elemento (*DELETE_QUERY*). Queste query devono essere censite nel file *statements.xml*. Ecco come sono state censite le query per questo esempio:

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

- *PAGE_PROVIDER*: l'attributo *class* contiene il nome logico del page provider da utilizzare

  Nella busta di configurazione *CONFIG* viene definito il numero di righe per pagina (*page_size*) e il numero di pagine da mantenere in cache (*side_pages*).

- *LIST_PROVIDER*: l'attributo *class* contiene il nome logico del list provider da utilizzare.

2.23.2 Comandi della Lista

E' possibile inviare alcuni messaggi al modulo di lista, specificando alcuni parametri nella richiesta.

Ecco i messaggi che è possibile inviare:

- Pagina precedente: va inserito nella richiesta il parametro ***MESSAGE=LIST_PREV***.
- Pagina successiva: va inserito nella richiesta il parametro ***MESSAGE=LIST_NEXT***.
- Prima pagina: va inserito nella richiesta il parametro ***MESSAGE=LIST_FIRST***.
- Ultima pagina: va inserito nella richiesta il parametro ***MESSAGE=LIST_LAST***.
- Pagina i-esima: va inserito nella richiesta il parametro ***MESSAGE=LIST_PAGE&LIST_PAGE=i*** dove *i* è l'indice della pagina da visualizzare.
- Ricaricare la lista: va inserito nella richiesta il parametro ***LIST_NOCACHE=TRUE***. Questo parametro va specificato, ad esempio, quando la ricerca è legata ad un form contenente i parametri di ricerca: al cambiamento dei parametri di ricerca la query deve essere rieffettuata, mentre la navigazione tramite la toolbar non deve ricaricare la lista.

2.23.3 Dettaglio

Il modulo di dettaglio è configurato esattamente come il modulo di dettaglio dell'esempio precedente.

2.23.4 Visualizzazione

In questo esempio è stata utilizzata una pagina per la produzione della lista e per la produzione del dettaglio: entrambe utilizzano la stessa pagina JSP.

Ecco come è stato configurato il file *pages.xml* per la pagina di lista:

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

dove nel file di configurazione *graphs.xml* abbiamo definito il grafo con id che vale 2 nel seguente modo:

```
<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>
```

A questa pagina è stato associato il corrispondente publisher, nel file *presentation.xml*:

```
<MAPPING business_type="PAGE" business_name="SmartPaginaUtenti" publisher_name="SmartPaginaUtentiModule" />
```

e il publisher di tipo JSP configurato nel file *publishers.xml*:

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

2.24 Profilatura

Il sottosistema di sicurezza è composto solo da interfacce: i vari sistemi di autenticazione sono integrati con wrapper specifici. L'interfaccia del framework da implementare per gestire la profilatura è *it.eng.spago.security.**IEngUserProfile***: definisce una serie di metodi per ottenere le credenziali dell'utente corrente, i suoi ruoli, le funzioni a cui è abilitato, etc.
Spago fornisce anche la classe astratta *it.eng.spago.security.**AbstractEngUserProfile***, che oltre ad implementare l'interfaccia *IEngUserProfile*, implementa anche l'interfaccia *DefaultRequestContext*, in modo che nei metodi di controllo *isAbleToExecuteXXX* sia possibile accedere anche al contesto di esecuzione della richiesta (request, response).
Nel caso in cui l'interfaccia *IEngUserProfile* sia implementata senza estendere *AbstractEngUserProfile*, ricordarsi di implementare l'interfaccia *DefaultRequestContext* se si necessita di accedere al contesto di esecuzione.

E' compito dello sviluppatore, all'atto del login, inserire un oggetto che implementa questa interfaccia nel permanent container (non influenzato dalla logica di navigazione applicativa), come attributo con nome *IEngUserProfile.ENG_USER_PROFILE*:

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

Ad ogni richiesta di servizio, sia essa servita da una action che da una pagina, viene verificato se l'utente corrente ha l'autorizzazione a richiedere il servizio e solo se c'è l'autorizzazione il servizio viene eseguito.
In caso di assenza di autorizzazioni:

- se si utilizza la modalità Action è possibile registrare un publisher con nome "SECURITY_ERROR_PUBLISHER"
- se si utilizza la modalità Moduli non viene generata alcuna eccezione: questo permette, ad esempio, nell'esecuzione di una pagina di servire correttamente la richiesta eseguendo solo i moduli per cui l'utente ha l'autorizzazione.

Nel framework non ci sono file di configurazione in cui censire i privilegi di sicurezza, in quanto tale attività è sempre delegata ad un sistema esterno.

2.24.1 Integrazione con sistemi esterni

Implementare l'interfaccia IEngUserProfile è la modalità più semplice di integrazione con sistemi esterni di profilatura. Ne esiste anche un'altra che permette di separare le funzionalità di autenticazione da quelle di autorizzazione.
Tale integrazione si effettua tramite l'uso di due interfacce:

- Autenticazione: *it.eng.spago.security.IAuthenticationHandler*. Ha la responsabilità di autenticare l'utente, reperendo le informazioni dalla richiesta, tramite il metodo:

  *public Object **authenticate**(SourceBean request, SourceBean response)*
  throws EMFInternalError;

- Autorizzazione: *it.eng.spago.security.IAuthorizationHandler*. Ha la responsabilita di memorizzare nel contesto (ovvero nel session container) le informazioni di profilatura dell'utente corrente (ovvero l'oggetto che implementa l'interfaccia *IEngUserProfile*), tramite il metodo:

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

Queste due interfacce vengono utilizzate dalla classe *GenericLoginHandler* che viene configurata tramite il file *security.xml*, di cui riportiamo un esempio*:*

```
<?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>
```

La classe *GenericLoginHandler* ha un unico metodo *service(RequestContainer,SourceBean,SourceBean)* che va invocato quando le informazioni di autenticazione sono state inserite nella request: tale metodo istanzierà le classi di autenticazione e autorizzazione, le quali provvederanno ad impostare nel session container la classe che implementa l'interfaccia *IEngUserProfile*.

Nel framework esiste un'implementazione di esempio che reperisce le informazioni degli utenti da un file XML: sono state implementate le interfacce di autenticazione (dalla classe *it.eng.spago.security.xmlauthorizations.XMLAuthenticationHandler*) e autorizzazione (dalla classe *it.eng.spago.security.xmlauthorizations.XMLAuthorizationHandler).* Questa implementazione di esempio di trova nel file *spago-profile.jar*.

2.24.2 Implementazione di esempio su file XML

Il framework include un'implementazione di esempio di un sistema di autenticazione/autorizzazione basato su file XML. Questa implementazione si trova nel package *it.eng.spago.security.xmlauthorizations* presente nel file *spago-profile.jar*.
Per utilizzare questa implementazione il file *security.xml* deve essere configurato come segue:

```
<?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>
```

Questa implementazione richiede una busta XML di configurazione denominata *XMLPM* che contenga il path assoluto al file che contiene le informazioni relative agli utenti.

Anche in questo caso vengono profilati gli use-cases, pertanto viene utilizzato il file *business_map.xml* per gestire l'associazione tra le action/moduli e le corrispondenti funzioni.

Riportiamo un esempio del file delle autorizzazioni *authorizations.xml* :

```
<?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>
```

Il file contiene le seguenti buste :

- *ENTITIES*: contiene gli utenti, i ruoli e le funzioni rispettivamente nelle buste *USERS*, *ROLES* e *FUNCTIONALITIES*.
- *RELATIONS*: contiene l'associazione ruoli - utenti nella busta *BEHAVIOURS*, e l'associazione delle funzioni con i ruoli nella busta *PRIVILEDGES*.

Le password degli utenti sono memorizzate codificate con una funzione di tipo one-way hash (sh1) ovvero codificate con una lunghezza fissa in modo che non sia possibile risalire al valore originale della password.

2.25 Inizializzazione

Il framework fornisce un servizio che consente di inizializzare al boot dell'applicazione gli oggetti applicativi: per usufruire di questo servizio bisogna implementare l'interfaccia *it.eng.spago.init.InitializerIFace*. Questa interfaccia contiene il metodo:

```
void init(SourceBean config);
```

che verrà invocato dal framework all'avvio dell'applicazione. Il parametro *config* contiene la busta di configurazione, come definita nel file *initializers.xml*, di cui riportiamo un esempio:

```
<?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>
```

In questo esempio all'oggetto *EventConditionInitializer* verrà passata la busta di configurazione *EVENT_CONDITIONS* (definita altrove), mentre a *EventHandlerInitializer* verrà passata la busta *EVENT_HANDLERS*.

2.26 Monitoring

Il prodotto JAMon disponibile al sito http://www.jamonapi.com/ è stato integrato all'interno del framework per eseguire il profiling delle attività di business, di acceso ai dati e di presentazione.
L'uso o meno di JAMon si configura tramite il file *common.xml* impostando la variabile *monitor_enabled* .
L'interfaccia di amministrazione viene rilasciata sotto la cartella "web/root/jamon" del build.

3 Rilascio su cluster

Riportiamo di seguito alcune considerazioni relative allo sviluppo di applicazioni, sviluppate con il framework, che verranno rilasciate su sistemi in cluster.
In questo contesto bisogna tenere conto, durante lo sviluppo dell'applicazione, che a seconda di come l'applicazione verrà installata, i processi potrebbero girare su Virtual Machine diverse e potenzialmente su macchine diverse.
Vediamo quali sono i possibili problemi in un contesto di questo tipo:

- Application Container: teoricamente è un container a livello applicativo, condiviso da action e moduli. Tecnicamente è implementato come un singleton: ciò significa che vive all'interno del class loader in cui viene invocato, per cui se ci sono più application server (come nel caso di un cluster) saranno istanziati più application container. Ad esempio, in una configurazione classica, ce ne potrebbe essere un'istanza nel Web Container, e una nell'EJB Container.
  Per questo motivo l'application container, in questi contesti, si può utilizzare solo come repository per dati memorizzati in sola lettura: non si può utilizzarlo per passare dati tra i business object (action e moduli).
  Se si ha la necessità di implementare un vero singleton, le linee guida SUN consigliano di usare un entity bean o un servizio remoto, ad esempio un server CORBA. Nella soluzione del server CORBA, il problema è che tale server non rientrerebbe nel meccanismo di failover dell'application server, risultando un potenziale punto debole del sistema. La soluzione migliore è l'entity bean.
- Session Container: un elemento critico è la dimensione della sessione HTTP. IBM, ad esempio, nei capacity plan indica in 4K la dimensione massima che dovrebbe avere la sessione; questo perchè con il session management attivo questi dati vengono serializzati e deserializzati. Per questo motivo la filosofia IBM prevede di avere in sessione solo gli identificativi degli oggetti da recuperare: in quest'ottica il caching delle pagine esulerebbe da questa specifica.
  Si sta rivedendo il framework per rendere configurabili tutti i parametri che permettono di controllare quanti dati mettere in sessione.
  Minimizzando la quantità di dati da mettere in sessione si possono perdere dei servizi: ad esempio la navigazione.