INFN Istituto Nazionale di Fisica Nucleare

LABORATORI NAZIONALI DI FRASCATI
SIS – Pubblicazioni

LNF–09/16(NT)
18 Dicembre, 2009

JAVA REF Framework

(Rapid Enterprise Framework)

Claudio Bisegni

INFN-Laboratori Nazionali di Frascati Via E. Fermi 40, 00044 Frascati, Italy

Abstract

Gli applicativi di Gestione Ospiti e GOVA realizzati in ambito nazionale si basano su un framework custom per lo sviluppo di applicativi java a tre livelli. E' composto da tre parti: Client, Common e Server. Integra un sistema di comunicazione in HTTP basato sulla libreria Apache HTTPClient (http://hc.apache.org/httpclient-3.x) e usa una servlet come entry-poiny per il server layer. Lo scopo del framework è quello di velocizzare lo sviluppo di sistemi software a tre livelli e di conseguenza lo sviluppo di business logic (processi applicativi) nel middle-tier e renderne veloce la pubblicazione e l'esecuzione tramite protocollo http.

PACS.: 01.50.hv
Key words: e.g. PACS

1 – INTRODUZIONE

Gli applicativi enterprise offrono un notevole vantaggio: sicurezza, scalabilità, estendibilità e fruizione di servizi ad altri applicativi. Gli applicativi Nazionali dell'INFN, Gestione Visitatori e GOVA (Gestione Visitatori Ospiti e Associati) si basano su un Framework Java mirato alla realizzazione di applicativi enterprise.

FIG. 1 – Applicazione a tre livelli

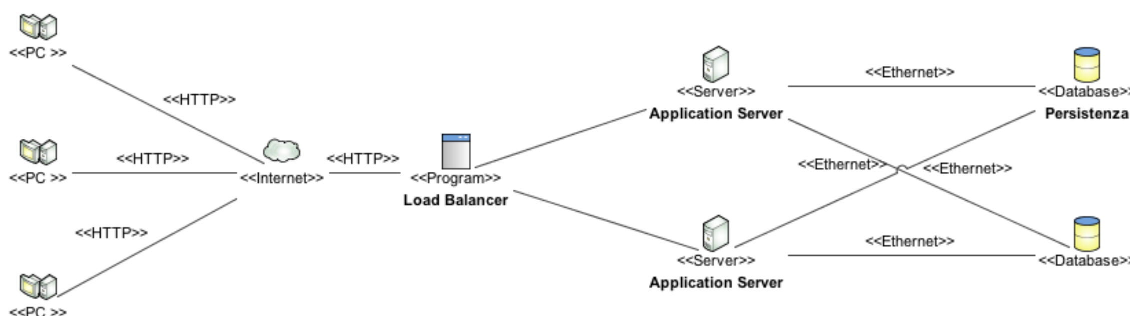

FIG. 2 – Applicazione a tre livelli in alta affidabilità
con cluster di Application Server e Database

Le figure 1 e 2 indicano due tipologie di applicativi a tre livelli, una semplice e una in High Availability(HA) usando un cluster di application server e un cluster di database.

In questo tipo di sistemi software il desktop client esegue la sola visualizzazione dei dati, permettendo le varie operazioni di ricerca, modifica e inserimento dei dati mentre la business logic, cioè le regole con cui vengono effettuate le operazioni sui dati, è gestita dal middle-tier insieme alle regole di persistenza per la memorizzazione dei dati nel database. Un disegno applicativo che separi la business logic dalla visualizzazione permette di avere un enorme flessibilità. Lo scopo principale è quindi quello di rendere veloce lo sviluppo di codice applicativo nel middle-tier codificato nei **processi applicativi**, renderlo fruibile all'esterno mediante specifici protocolli e semplificarne il deploy negli application server. In aggiunta possono essere creati Web Service[1] per rendere disponibili alcune funzionalità della business logic ad altri sistemi.

Il framework non implementa nessun tipo di autenticazione ed autorizzazione che spetta

[1] Interfaccia software per l'interoperabilità di sistemi attraverso il protocollo HTTP codificando i dati in SOAP o REST(XML Messaging)

invece all'implementazione dell'applicazione. Perciò mette a disposizione dei plug-ins per estendere le procedure di allocazione dei processi ed esecuzione delle azioni, e ne disciplina l'uso. GOVA implementa questi plug-in per autenticare ed autorizzare i propri utenti seguendo le regole della INFN AAI.

2 – Protocollo di comunicazione

Il framework usa un protocollo di comunicazione custom molto semplice. La classe **DataTransporter**, contenuta nel common layer, permette di codificare tutte le operazioni che il middle-tier e in grado di eseguire. Una volta valorizzata viene serializzata dal client e spedita via http/s alla servlet entry-point del server layer dove viene de-serializzata. Il risultato dell'operazione eseguita o l'eccezione nel caso di errore, vengono inviati in risposta al client con lo stesso sistema della chiamata.

3 – Software Layer

Il framework è diviso in tre layer, ognuno di questi mette a disposizione delle classi per la realizzazione di diverse parti di un applicazione enterprise:

- **Server Layer** gestisce la logica di creazione di processi business sviluppati per eseguire operazioni su domini di dati incapsulati in classi business. L'entry point e' costituito da una servlet che esegue le operazioni codificate tramite il **DataTransporter**;
- **Common Layer** racchiude classi di utilità per la realizzazione delle classi business, ed è usato sia dalla parte client sia da quella server;
- **Client Layer** racchiude classi per la realizzazioni delle UI[2] e per la comunicazione con il middle-tier;

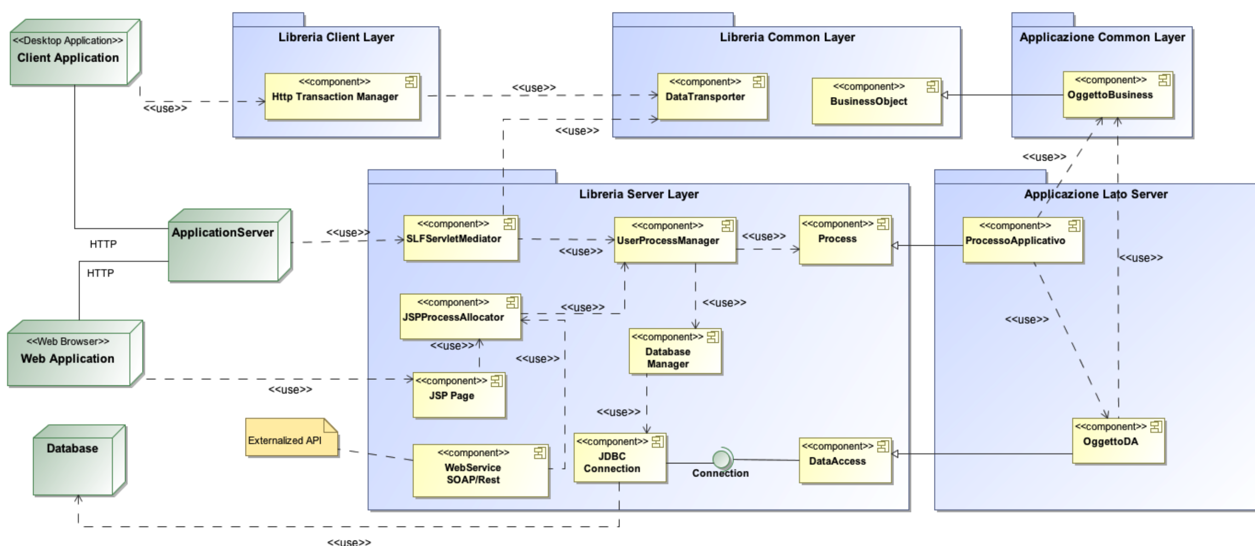

FIG. 3 – Schema generale del framework.

[2] User Interface

La figura 3 mostra le varie parti di un'applicazione con l'uso del framework, i vari layers della libreria e delle applicazioni e le classi principali indicandone il loro uso.

3.1 – Client Layer

Il client layer costituisce un insieme di classi per la realizzazioni della UI di un applicazione desktop e per la comunicazione con il lato server. La parte di comunicazione, come precedentemente descritto, implementa un protocollo proprietario di controllo per il middle-tier. La classe **ProcessProxyManager**, implementa le seguenti funzioni per il controllo del middle-tier:

- `public void setConnectionListener(ConnectionListener cl)`, permette di impostare il listener per poter vere le notifiche di avvenuta connessione o disconnessione dal server;
- `public void setCompression(boolean compression)`, imposta l'uso della compressione per il payload nelle transazioni http;
- `public void setHttpParameter(String endPointUrl)`, permette di impostare l'url della servlet principale del framework;
- `public void initHeartBeat(int hbTime)` e `public void deinitHeartBeat()`, controllano lo stato dell'heartbeat per mantenere aperta la sessione con il lato server anche se l'utente non esegue nessuna transazione;
- `public Object openProcess(String processName) throws Throwable`, avvia nel middle-tier il processo identificato dalla stringa in input;
- `public void closeProcess(String processName) throws Throwable`, esegue la chiusura del processo identificato dalla stringa in input;
- `public Object execProcessAction(Object... data, String processName, String actionName, boolean stateless) throws Throwable`, esegue il metodo di un processo precedentemente aperto. L'attributo state lese se impostato a "true" esegue una chiamata stateless.
- `public Object queueTProcessRequest(Object data, String processName, String action, ThreadedProcessStateListener listener) throws Throwable`, avvia l'esecuzione di un metodo di un processo applicativo all'interno di un thread il risultato di questa chiamata sarà l'identificativo della richiesta, necessaria per il controllo dello stato di esecuzione del metodo.
- `public Object checkTProcessRequest(String processKey) throws Throwable`, ritorna il risultato, se disponibile, del metodo eseguito in un thread, identificato dalla chiave in input.

Le api sopra descritte modificano opportunamente la classe **DataTransporter**, che viene poi serializzata e trasmessa via http/s alla servlet di entry-point, permettendo così di controllare il middle-tier.

3.2 – Common Layer

Il common layer, mette a disposizioni classi di utilità. Nelle applicazioni a tre livelli c'è necessità di scambiare dati tra client e middle-tier, il pacchetto ***org.ref.common.business*** contiene tutto il necessario per la creazione e gestione delle classi business. Ogni classe usata per mappare i dati da elaborare deve estendere la classe **BusinessObject**. Per la mappatura di una classe business ad una tabella di un database, il

framework mette a disposizione delle "*annotation*" Java, che permettono di codificare staticamente la definizione di una classe come tabella di un database:

- **DBTable** descrive una classe Java definendola una tabella del Database;
- **DBColumn** mappa i field di una classe usando i seguenti attributi:
  - *Name* imposta il nome della colonna associata al field;
  - *isComplex* imposta il field come classe relazionata ad un'altra tabella;
  - *maxDimension* imposta la dimensione massima del numero di caratteri ammessi dalla colonna del database;
  - *isDefault* imposta il field come default di una classe;
  - *isNullable* imposta un field di una classe in modo che possa accettare valori null.
- **DBPrimaryKey** mappa più field di una classe come primary key.
- **DBRelationClass** mappa le foreign-key della classe corrente con le primary-key della classe associata al field definito come "*isComplex*"

Queste "*annotation*" sono usate dai **DataAccess**, cioè oggetti usati per la gestione della persistenza sui back-end definiti nel middle-tier e per il caricamento e il salvataggio automatico delle classi **BusinessObject**.

3.3 – Server Layer

Il server layer gestisce l'intero middle-tier e svolge le operazioni necessarie per processare i dati inviati dal client nei vari processi applicativi. L'uso delle servlet per l'entry-point http, permette l'uso di ogni application server conforme allo standard j2ee. Dalla ricezione della POST HTTP all'esecuzione di un processo le classi principali coinvolte sono:

- **ServletMediator** servlet entry-point che esegue le istruzioni contenute nella classe **DataTransporter** la cui forma serializzata e contenuta nel Body della POST HTTP;
- **UserProcessManager** gestore dei processi applicativi legati alla sessione corrente. Ad ogni sessione http è associato un solo process manager che mantiene lo stato dei processi applicativi tra una post http e la successiva.
- **REFProcess** superclasse di tutti i processi applicativi(ApplicationProcesses) codificati dall'applicazione. Contiene la connessione JDBC al database corrente impostato nel file di configurazione del server layer.

Al ricevimento di una post HTTP il **ServletMediator** ne controlla l'header e verifica la versione della libreria usata dal client, tenta infine di deserializzare il contenuto del BODY della POST. Se la de-serializzazione ha successo, la servlet ha quindi accesso all'oggetto di classe **DataTransporter(DT)**. Il **ServletMediator** controllando il field *internalOperation* del **DT**, è in grado di eseguire le operazioni richieste dal client, le quali sono definite dalle seguenti costanti:

```
public static final int  PROCESS_START          = 0;
public static final int  PROCESS_STOP           = 1;
public static final int  NORMAL_OPERATION  = 2;
public static final int  HEART_BEAT             = 3;
public static final int  STATELESS              = 4;
```

I seguenti diagrammi UML descrivono la sequenza delle chiamante tra le classi precedentemente descritte.

3.3.1 – *Avvio di una nuova Sessione HTTP*

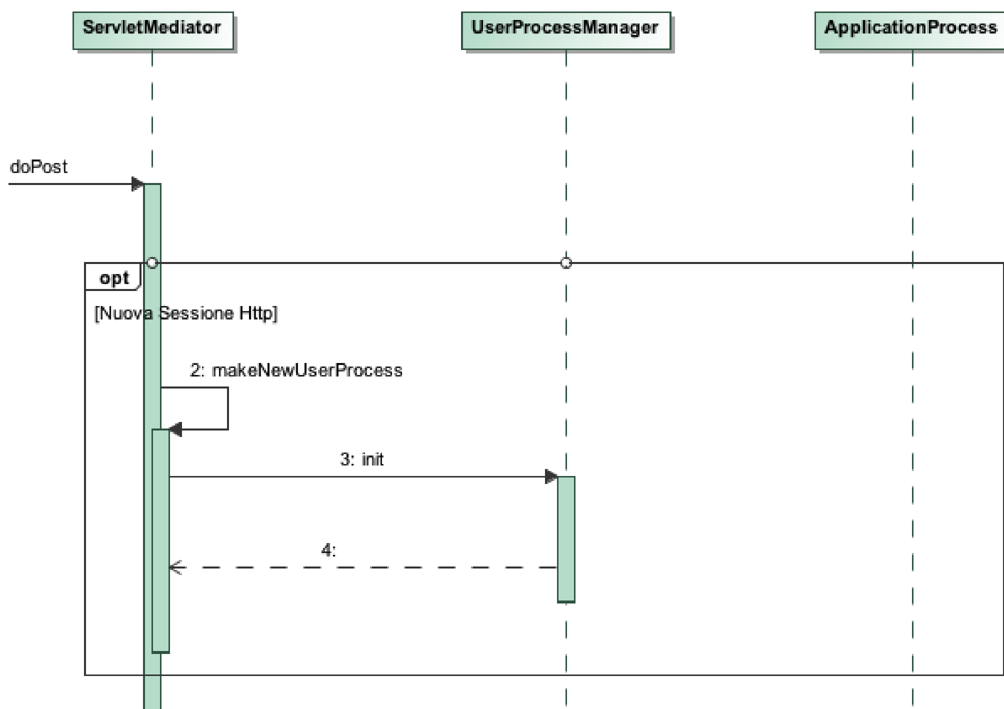

FIG. 4 – Avvio di una nuova transazione http nell'application server

La prima volta che il client esegue una chiamata al server layer, la servlet **ServletMediator** crea una nuova istanza della classe **UserProcessManager(UPM)** e la inizializza. Tale istanza viene registrata nella sessione http per essere ripresa ogni volta che verrà eseguita una nuova transazione di tale sessione(*la gestione dei processi applicativi è demandata all'istanza della classe **UPM**, il solo compito della servlet è quello di verificare l'header http, deserializzare l'istanza dell'oggetto **DT** e invocare i metodi del process manager per eseguire i compiti richiesti*).

3.3.2 – *Creazione di un processo*

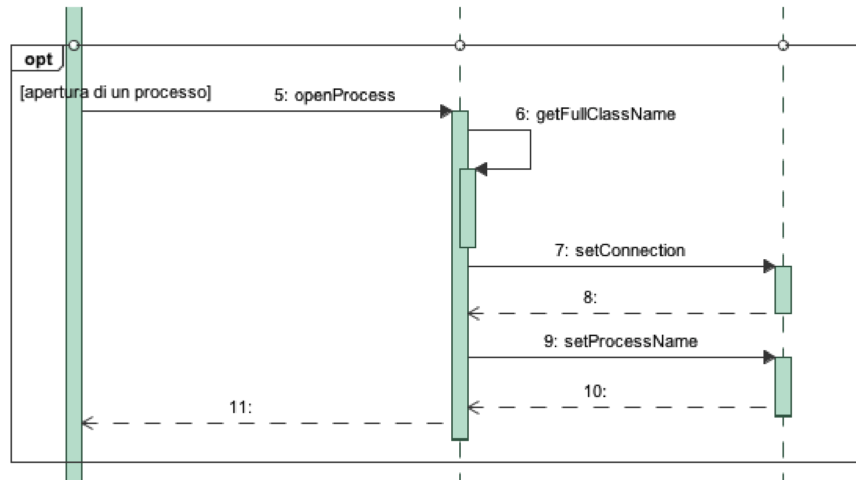

FIG. 5 – Creazione di un processo.

La creazione (o apertura) di un processo applicativo è definita dalla costante della **DT** PROCESS_START. Alla sua ricezione la servlet chiama il metodo *openProcess* dell'**UPM** (contenuto nella sessione http corrente) passando come parametro l'oggetto **DT**. Il metodo openProcess crea il processo mediante i seguenti passi:

- Recupera la classe associata al nome del processo dalle configurazioni del server layer e ne crea una nuova istanza;
- Crea una nuova connessione JDBC associandola all'istanza del processo creato;
- Imposta il nuovo nome del processo usando l'UUID generato al momento della creazione. Questo è ritornato al client valorizzando il field *processName* della **DT**. Ad ogni successiva chiamata indirizzata a tale processo, il client dovrà usare l'UUID.

3.3.3 – *Esecuzione di un'azione di un processo*

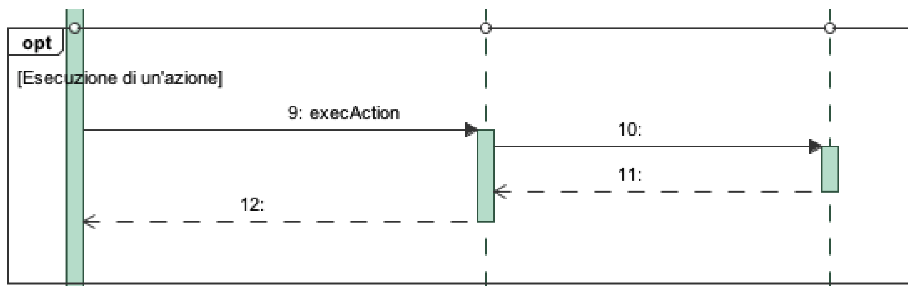

FIG. 6 – Esecuzione di un'azione di un processo applicativo

Una volta che il processo è stato instanziato nel middle-tier il client può cominciare ad usare le sue funzioni e può eseguire qualsiasi metodo del processo che in fase di scrittura del codice è stato dichiarato "pubblicabile". Il client, mediante il metodo

execProcessAction della classe **ProcessProxyManager(PPM),** è in grado di usare l'UUID assegnato dal server layer al processo, il nome e parametri di un azione in modo da eseguirla nel middle-tier ed ottenere l'oggetto che ne risulta. Dal punto di vista del server layer l'esecuzione di un'azione è identificata dal valore NORMAL_OPERATION del field "internalOperation" del **DT**. A questo punto viene eseguito il metodo *execAtion* dell'**UserProcessManager(UPM)** dal **ServletMediator(SM).** Questo metodo usa il field *processName* della **DT(**contenente l'UUID) per riprendere il processo precedentemente istanziato.

3.3.4 – *Chiusura di un processo*

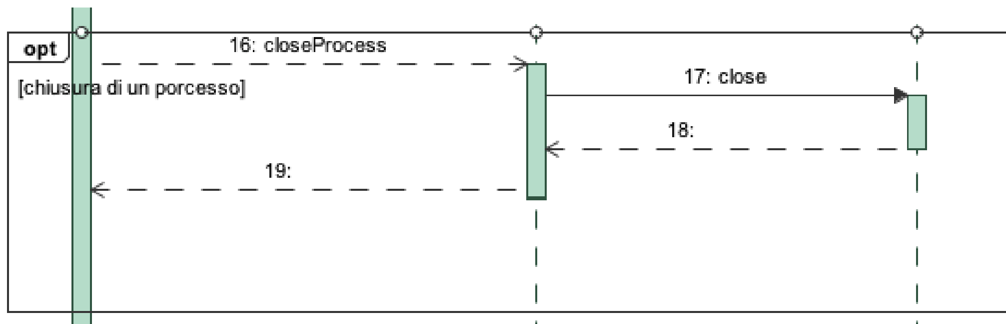

FIG. 7 – Chiusura di un processo.

Quando il client non ha più bisogno dei servizi di un determinato processo, può provvedere alla sua chiusura mediante il metodo *closeProcess* della **PPM** con l'UUID del processo come parametro. Nel server layer questa operazione viene identificata dal valore PROCESS_STOP del field *internalOperation* della DT. La **ServletMediator** a questo punto, invoca il metodo *closeProcess* del **UPM,** che a sua volta chiamerà il metodo *close* dell'istanza del processo che si occuperà di chiudere la connessione JDBC ad esso associata. La sottoclasse di **REFProcess** che definisce il processo applicativo, può ridefinire il metodo *close* per deallocare le proprie risorse.

3.3.5 – *Evento di HeartBeat*

L'evento di HeartBeat viene generato dal client layer abilitando il suo uso con la funzione del **ProcessProxyManager** *initHeartBeat* e interrompendolo con la funzione *deinitHeartBeat*. Il server layer identifica questa richiesta dal valore HEART_BEAT del field *internalOperation* della classe **DataTransporter**, ma alla sua ricezione non esegue nessun operazione. In definitiva l'evento di HeartBeat esegue il reset del tempo di timeout della sessione http.

3.3.6 – *Chiamata Stateless*

Di solito il client apre un processo, esegue una serie di azioni di quel processo e ne richiede la chiusura. Tra l'esecuzione di un azione e l'altra, l'istanza rimane persistente nel middle-tier. Nel caso il client debba chiamare una sola volta un'azione di un processo, può eseguire direttamente la chiamata *execProcessAction* della classe **ProcessProxyManager**

impostando a “true” l’attributo “stateless”. In questo modo il server layer esegue i seguenti passi:

- Apertura del processo;
- Esecuzione dell’azione;
- Chiusura del processo e de allocazione delle risorse.

4 – Pubblicazione delle azioni di un processo.

Un processo applicativo è definito creando una sottoclasse della **REFProcess** ed usando due tipi di *annotation* per descriverlo:

- **ProcessDescription** descrive una processo definendone l’alias pubblico con cui deve essere invocato dal client. Se l’alias è impostato al valore di default il server layer lo valorizzerà con il nome della classe;
- **ProcessActionDescription** descrive una azione del processo definendone l’alias con cui deve essere invocata. Anche in questo caso se l’alias è uguale al valore di default il framework lo valorizzerà con il nome del metodo.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ProcessDescription {
//Alias associato all'azione
public String publicAlias() default "";
}
```

Segmento Java 1

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ProcessActionDescription {
//Alias associato all'azione
public String publicAlias() default "";
}
```

Segmento Java 2

Una volta definiti i processi applicativi, bisogna registrarli in modo che il server layer, all’avvio dell’application server, li possa configurare e pubblicare al client layer. Per configurare il server layer occorre definire il file web.xml che definisce la **ServletMediator** con i seguenti tag xml:

```
<init-param>
<param-name>CONFIG_PATH</param-name>
<param-value>/process.xml</param-value>
</init-param>
```

Il parametro CONFIG_PATH configura il **ServletMediator** con il path del file xml dove sono configurati i processi, in modo da poterli caricare all’avvio dell’application server. Tale file è definito dal seguente frammento DTD che deve essere incluso nel file stesso:

```
<!DOCTYPE processes [
  <!ELEMENT processes (process*)>
  <!ATTLIST process class CDATA #REQUIRED>
]>
```

L’xml quindi dovrà contenere il tag di root <*processes*> con un insieme di tag del tipo <*process* class='nome.pacchetto.java.NomeClasse'/>. Tutte le classi cosi definite saranno registrate come processi applicativi e associate con gli alias definiti dalle *annotation* sopra descritte.

5 – Plug-in

Il framework permette di poter interagire con la sua parte "core" di inizializzazione infrastruttura e gestione delle chiamate ai processi applicativi, mediate l'implementazione di due interfacce Java:

- **REFInitPlugin**
- **REFUserProcessManagerPlugin**

5.1 - *REFInitPlugin*

```
public void custoInit();
```

Il plug-in mette a disposizione un unico metodo per permettere all'applicazione eseguire delle inizializzazioni nel momento in cui viene avviata l'intera infrastruttura.

5.1 – *REFUserProcessManagerPlugin*

```
/**
 * Inizializza il plug-in
 *
 * @throws Throwable
 */
abstract public void init() throws Throwable;

/**
 * Metodo chiamato prima della creazione del processo, il lancio dell'eccezione blocca la
 * creazione del processo
 *
 * @param ProcessName
 * @throws Throwable
 */
abstract public void processWillBeCreated(String ProcessName) throws Throwable;

/**
 * Metodo chiamato alla fine della creazione del processo
 *
 * @param ProcessName
 * @throws Throwable
 */
abstract public void processDidCreated(RefProcess processClass);

/**
 * Metodo chiamato alla chiusura del processo
 *
 * @param processName
 */
abstract public void processWillBeClosed(String processName) throws Throwable;

/**
 * Metodo chiamato alla riuscita della chiusura del processo
 *
 * @param processName
 */
abstract public void processDidClosed(String processName);

/**
 * Metodo chiamato all'inizio dell'esecuzione dell'azione, il lancio di un'eccezione blocca
 * l'azione stessa
 *
 * @param actionName
 * @param processName
 * @throws Throwable
 */
abstract public void actionWillBeExecuted(String processName, String actionName) throws Throwable;

/**
 * Metodo chiamato alla fine dell'esecuzione dell'azione
 *
 * @param actionName
 * @param processName
 */
```

```
    abstract public void actionHasBeenExecuted(String processName, String actionName);

    /**
     * Notifica l'avvenuta chiusura della sessione utente
     */
    abstract public void sessionWillBeClosed();

    /**
     * Evento di idle del client
     */
    abstract public void idleEvent();
```

Il *REFUserProcessManagerPlugin* permette all’applicazione di intervenire durante gli eventi di creazione/distruzione dei processi e durante la chiamata delle azioni di un processo. GOVA utilizza questo plug-in per poter eseguire le autorizzazioni sulle esecuzioni di processi e azioni. I metodi *processWillBeCreated* e *actionWillBeExecuted* posso bloccare l’esecuzione dell’operazione associata, facendo il “trow” di una qualsiasi eccezione che estende Throwable. GOVA usa questi due metodi per controllare se negli entitelment (presi da LDAP) dell’utente correntemente loggato, siano presenti quelli che autorizzano l’esecuzione del processo o dell’azione a cui tali metodi fanno riferimento.

7 – Configurazione del Framework

Il middle-tier del framework viene gestito totalmente dalla servlet di entry-point **ServletMediator**, quindi la sua configurazione avviene mediante la valorizzazione dei parametri di input della servlet di entry-point nel file web.xml.

7.1 – *Configurazione della servlet di entry-point*

Per poter permettere lo startup del middle-tier del framework bisogna registrare nel file web.xml la servlet di entry-poit:

```
<servlet>
    <description>Descrizione servlet</description>
    <display-name>REFServletMediator</display-name>
    <servlet-name>REFServletMediator</servlet-name>
    <servlet-class>org.ref.server.slfservletadaptor.REFServletMediator</servlet-class>
…
</servlet>
```

7.1 – *Configurazione dei plug-ins*

```
<!--  Configurazione per il plugin dell'initi di gova -->
<init-param>
    <param-name>ref.init.plugin</param-name>
    <param-value>org.infn.gestioneospiti.server.reffplugin.GOVAInitPlugin</param-value>
</init-param>
<!--  Configurazione per il plugin del processmanager -->
<init-param>
    <param-name>ref.process.manager.plugin</param-name>
    <param-value>org.infn.gestioneospiti.server.refplugin.AAIUserProcessManagerPlugin</param-
value>
</init-param>
```

Le classi *GOVAInitPlugin* e *AAIUserProcessManagerPlugin* sono i plugin che estendono rispettivamente le interfacce *REFInitPlugin* e *REFUserProcessManagerPlugin.*

7.2 – *Configurazione del file di descrizione dei processi*

```
<!--  Configurazione dei processi -->
<init-param>
    <param-name>ref.process.config.path</param-name>
    <param-value>/process.xml</param-value>
</init-param>
```

7.3 – *Configurazione dei log*

```
<!--  Configurazione per i log -->
<init-param>
    <param-name>log.file.path</param-name>
```

```
      <param-value>/var/log/go/ospiti%g.log</param-value>
</init-param>
```

La configurazione dei log consiste nel valorizzare il parametro per determinare il path in cui verranno scritti i log del middle-tier.

7.3 – *Configurazione database*

La gestione delle connessione al database può essere gestita dal framework secondo due modalità:

- Usando un connection pool JDBC configurato sull'application server
- Creando un connection pool JDBC direttamente gestito dal framework

7.3.1 – *Uso del connection pool dell'application server*

```
<init-param>
      <param-name>db.appserver.datasource</param-name>
      <param-value>jdbc/gova</param-value>
</init-param>
```

Per usare il connection pool definito dall'application server, bisogna valorizzare il parametro *db.appserver.datasource* con il nome della risorsa JNDI Java esportata.

7.3.2 – *Uso del connection pool del framework*

```
<init-param>
      <param-name>db.jdbc.driver</param-name>
      <param-value>oracle.jdbc.driver.OracleDriver</param-value>
</init-param >
<init-param>
      <param-name>db.url</param-name>
      <param-value>jdbc:oracle:thin:@gosrv.lnf.infn.it:1521:sis10</param-value>
</init-param>
<init-param>
      <param-name>db.username</param-name>
      <param-value>ospiti</param-value>
</init-param>
<init-param>
      <param-name>db.password</param-name>
      <param-value>adminospiti34</param-value>
</init-param>
```

Per usare il connection pool gestito direttamente dal framework vanno valorizzati i parametri per la configurazione del tipo di: driver, URL, user e password per il database target da usare per il connection pool JDBC.

8 – Conclusioni

Il deploy di un'applicazione sviluppata con questo framework è molto semplice e veloce. Di fatto l'intera applicazione rappresenta una web-application j2ee in cui una servlet si comporta da entry-point per i messaggi inviati dal client. Le applicazioni Gestioni Ospiti e GOVA sono distribuite ognuna in un singolo file WAR. La parte client è contenuta nel file WAR e può essere eseguita sia come Applet che come applicazione Java WebStart. Il deploy quindi risulta veramente veloce, si tratta infatti di inviare il solo file war all'application server e tutto è disponibile per l'uso.