Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica

Progetto e sviluppo di un modulo di schedulazione per il framework Quix

Relatore
Prof. Sonia Bergamaschi

Candidato
Michele Gazzetti

Anno accademico 2012/2013

Indice

Indice delle figure

Se cambi il font in arial 12 quando introduci nomi di procedure e classi risulta più evidente la separazione tra termini del linguaggio naturale e nomi di oggetti sw

Introduzione

Tra le varie tipologie di applicazioni presenti nel mercato informatico una grande fetta è costituita dalle applicazioni accessibili per mezzo di un network, che nella maggior parte dei casi è il Web, si parla in questo caso di web application..

Per aziende che si dedicano allo sviluppo di questa tipologia di applicazioni, la tempistica e le risorse sono sempre state fattori cruciali. Per questo motivo, nel corso degli anni, si sono sviluppate strutture logiche di supporto al programmatore tramite le quali un software può essere progettato e realizzato in modo facilitato e con maggiore velocità: queste strutture prendono il nome di framework.

Grazie all'esperienza accumulata nella consulenza e nello sviluppo di applicazioni web il team della azienda Quix S.r.l. di Modena ha sviluppato un proprio framework per la creazione di web application.

Durante il tirocinio presso questa azienda ho avuto modo di apportare il mio contributo a questo progetto, ricordo in particolare:

- L'integrazione di un modulo di amministrazione tramite il quale è possibile visualizzare e gestire i dati di configurazione dell'applicazione creata
- La creazione di un modulo per la visualizzazione e il reperimento dei file di log per applicazioni distribuite
- L'integrazione di un modulo per l'invio automatico di mail
- La creazione di un modulo di gestione di processi schedulati.

In questo elaborato verrà descritta la progettazione e lo sviluppo dell'ultimo modulo sopra citato (gestione di progetti schedulati) prestando particolare attenzione ad un contesto di server distribuiti ed alle problematiche strettamente correlate ad esso. L'elaborato si articola nei seguenti capitoli:

- Il primo capitolo è dedicato allo scheduler: verrà data una sua definizione e verranno descritti i suoi componenti fondamentali
- Nel secondo capitolo verranno descritte le problematiche e gli scenari presi in considerazione
- Il capitolo terzo tratta l'implementazione del sistema progettato
- Il quarto capitolo descrive le interfacce mediante le quali l'utente può

interagire con il sistema

- Il quinto capitolo tratta l'integrazione dello scheduler con il framework esistente
- Nell'appendice sono riportati i framework e gli strumenti significativi utilizzati durante la realizzazione del progetto.

Capitolo 1

Descrizione dello Scheduler

1.1 Lo Scheduler

In informatica lo Scheduler è un software basato su un algoritmo che, dato un insieme di richieste, stabilisce un ordinamento temporale per la loro evasione rispettando determinati parametri o condizioni.

L'attenzione posta su determinati parametri piuttosto che su altri, differenzia la cosiddetta *politica di scheduling***.** **S**olitamente, lo scheduler può soddisfare le richieste in base al loro ordine di arrivo (politica FIFO), oppure dare precedenza a quelle che eseguono il compito assegnato nel minor tempo. Le politiche di scheduling si basano principalmente su principi statistici o sulla predizione, entrambe con l'obiettivo di avvicinarsi il più possibile al risultato ottimale.

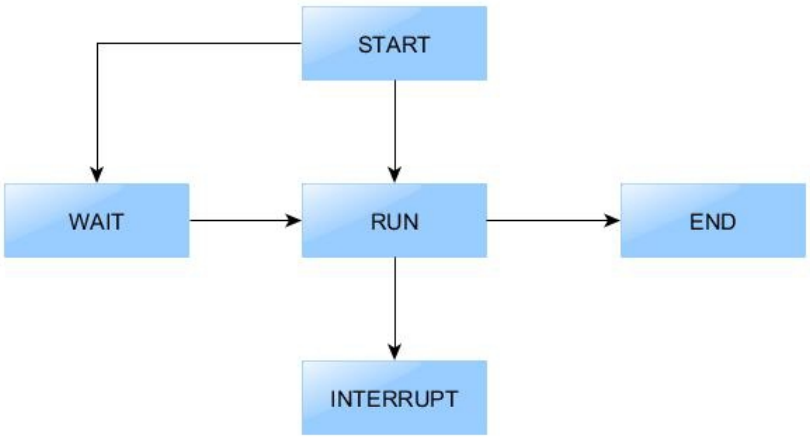

Fig. 1.1: Diagramma degli Stati dello Scheduler

Nel caso preso in esame lo Scheduler è implementato attraverso un'applicazione o una libreria che ha il compito di gestire l'esecuzione automatizzata di procedure. La politica di scheduling definita per il progetto si basa su due fattori:

- *Temporale*: le esecuzioni devono rispettare l'ordine definito dall'utente
- *Comportamentale*: la partenza è condizionata dal rispetto di vincoli comportamentali definiti dall'utente

1.2 Il Job

Il componente primario è il job, detto anche batch process, identifica il processo schedulato, questo ha il compito di eseguire una quantità prefissata di operazioni in backgroud.

È possibile differenziare i job in base alla loro condotta, si dividono quindi in:

- *Job standard*: implementazione standard, non sono definite particolari restrizioni comportamentali, la caratteristica principale è la possibilità di avere più istanze in esecuzione contemporaneamente.
- *Job stateful*: non può esserci più di una sua istanza in esecuzione contemporaneamente: nel caso in cui un processo debba essere avviato, e un istanza del job sia già in esecuzione, questo assume uno stato di wait finché il job già allocato non ha terminato l'esecuzione
- *Job interruptable*: si dà la possibilità all'utente di stoppare l'esecuzione del processo

1.3 Il trigger

Per definire il momento nel quale il job deve essere eseguito si utilizzano i trigger, procedure con il compito di schedulare i job secondo parametri temporali. Il parametro che permette di personalizzare la partenza del processo è una espressione unix-like detta *cron expression*.

Un'espressione *cron* è una stringa specificante 6 campi separati da uno spazio bianco
(es: * * * * * *).

I campi sono descritti come segue in ordine da sinistra verso destra:

Nome campo	Valori Ammessi	Caratteri speciali ammessi
Secondi	0-59	, - * /
Minuti	0-59	, - * /
Ore	0-23	, - * /
Giorno-del-mese	1-31	, - * ? / L W
Mese	1-12 or JAN-DEC	, - * /
Giorno-della-settimana	1-7 or SUN-SAT	, - * ? / L #
Anno (Opzionale)	empty, 1970-2199	, - * /

Spiegazione dei caratteri speciali:

- '*' utilizzato per specificare tutti i valori, ad esempio: se settato nel campo dei secondi viene letto come “ogni secondo”.
- '?' utilizzato quando non si vuole specificare un particolare valore per il campo.
- '-' utilizzato per specificare un intervallo di valori.
- ',' usato per specificare valori addizionali.
- '/' utilizzato per specificare un incremento a partire da una base (base/incremento).
- 'L' utilizzato per indicare l'ultimo valore possibile nell’intervallo.
- 'W' utilizzato per specificare i giorni della settimana vicini al giorno specificato (es: 15W usato nel campo giono-del-mese viene letto come “il giorno della settimana più vicino al 15 del mese).
- '#' utilizzato per il campo giorno-della-settimana, viene utilizzato per specificare l’n-esmio giorno del mese (es: 4#5 viene letto come “il quinto mercoledì del mese”).

Capitolo 2

Descrizione delle problematiche

2.1 Configurazioni

Essendo il servizio di schedulazione utilizzato all'interno di una web application, è necessario porre una particolare attenzione sulla configurazione dei server utilizzati per l'implementazione del sistema.

Si distinguono le seguenti configurazioni:

- *Centralizzata:* il servizio è fornito tramite l'utilizzo di un'unica macchina
- *Distribuita (cluster):* vengono utilizzate più macchine collegate tra la loro tramite un network. Questa scelta porta ad un aumento della complessità di tutti i meccanismi che si occupano del coordinamento dei vari nodi del sistema, tuttavia, la struttura va a beneficiare di una proprietà fondamentale chiamata *resilienza*.

Con il termine resilienza si indica la capacità del sistema di adattarsi alle situazioni d'uso in modo da rendere sempre disponibile un servizio. La configurazione di macchine che beneficia particolarmente di questa proprietà è chiamata *High-Availability clusters (HA clusters)*, dove un gruppo di computer, supportante la stessa server application, rendere minima la probabilità di una indisponibilità del servizio. In mancanza di una configurazione cluster se un server che centralizza su di se l'intera gestione dell'applicazione subisce un blocco improvviso, gli utenti non avranno modo di sfruttare l'applicazione fino a ripristino ultimato. Pensando a servizi bancari, o servizi gestionali utilizzati in fase di produzione la problematica appare critica.

Gli HA clusters hanno la capacità di gestire problemi di affidabilità dirottando il carico di lavoro sulle macchine disponibili evitando in questo modo i *single points of failure*.

Il servizio progettato deve permettere la gestione delle partenze di processi schedulati su server contenenti una o un numero maggiore di applicazioni. Se i server godono di una configurazione in cluster, si deve garantire il servizio anche in caso di errori o problemi su alcuni nodi. La scelta progettuale è stata quella di rendere

disponibili entrambe le configurazioni descritte in precedenza. I possibili scenari presi in considerazione durante lo sviluppo del progetto sono i seguenti:

1. **Unica applicazione all'interno del servlet container**: la funzionalità di schedulazione dei processi è incorporata all'interno dell'applicazione stessa.

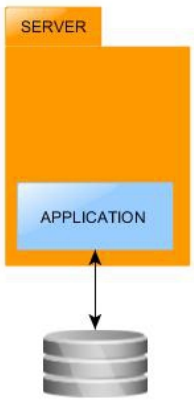

Fig. 2.1: Server contenente una singola applicazione

2. **Più applicazioni all'interno dello stesso servlet container**: vi è l'utilizzo di un unico server ma la necessità di un'applicazione aggiuntiva dedicata alla schedulazione delle procedure appartenenti alle applicazioni situate all'interno del nodo.

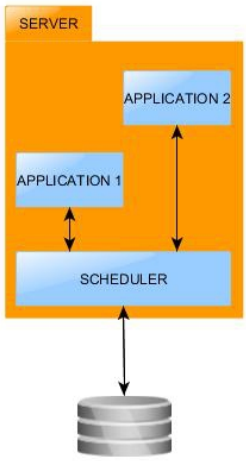

Fig 2.2. Server contenente più applicazioni

3. **Più applicazioni rilasciate utilizzando una configurazione di server in cluster**: in questo caso si necessiterà di una applicazione aggiuntiva per ogni nodo, questa avrà il compito di schedulare i processi per tutte le applicazioni presenti nel sistema, siano queste installate su vari nodi o rilasciate su uno solo di essi.

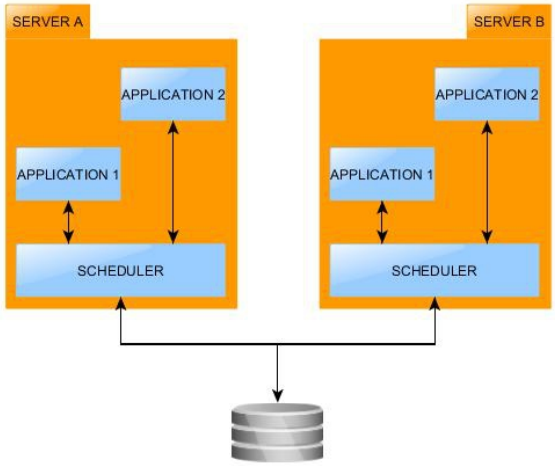

Fig. 2.3: Configurazione di server in cluster

Nel caso di una configurazione distribuita, un vincolo di progettazione è l'utilizzo di un unico database server per tutto il sistema.

Nella progettazione di una soluzione si è cercato di rispettare il più possibile uno dei principi fondamentali nello sviluppo di applicazioni web e cioè la non ripetitività del software, con l'obbiettivo di evitare ridondanza logica in fase di sviluppo. Per il motivo appena citato si è scelto di optare per una implementazione utilizzabile in tutti i contesti, andando a distinguere i vari casi solo in fase di integrazione con il framework esistente.

2.2 Database e Schema Entity-Relationship

Il database risulta essere la risorsa centrale e indispensabile per il coordinamento del sistema, è l'unico strumento tramite il quale ogni nodo riesce a reperire le informazioni utili per l'esecuzione di processi decisionali.

Per ogni applicazione è obbligatorio mantenere i dati relativi al suo contesto, questi comprendono il nome dell'applicazione e l'indirizzo dell'host. Per ogni contesto è possibile definire un numero variabile di processi schedulati. Non è possibile avere job caratterizzati dallo stesso nome appartenenti allo stesso contesto ma, risultano legali due istanze caratterizzate dallo stesso nome e appartenenti a contesti diversi. Per vincolo di progetto i job verranno identificati da un nome univoco per tutti i contesti. Il motivo di tale restrizione è rendere immediata l'identificazione del nome job in presenza di malfunzionamenti.

Le procedure che interagiscono con il sistema possono essere suddivise in due tipologie in base al loro stato di esecuzione:

- *Eseguibili*: procedure pronte ad essere avviate.
- *In esecuzione***:** procedure in fase di interazione con il sistema.
- *Eseguite***:** procedure che hanno terminato di computare e hanno fornito un risultato.

I processi eseguibili sono identificabili univocamente tramite il nome e un contatore utile a specificare la i-esima esecuzione della procedura nel tempo.

Ipotizzando la presenza di una configurazione distribuita, per i processi in esecuzione si vuole mantenere il nome del nodo che ha preso in carico la procedura, dato essenziale per analizzare il numero di processi in esecuzione sui vari nodi. La combinazione di questo ultimo dato e lo stato di esecuzione, permette di avere una visione complessiva del sistema distribuito.

I job in esecuzione possono presentare i seguenti stati:

- *runnable*: indica una procedura pronta ad essere eseguita ma non ancora presa in carico da nessun nodo.
- *running*: procedura presa in carico da un determinato nodo su cui l'applicazione è presente.
- *finished*: procedura terminata e quindi pronto ad essere spostato nella tabella di storico.
- *aborted*: procedura interrotta a causa di un errore o uno spegnimento anomalo del nodo.

Le tabelle derivanti dall'analisi delle problematiche permettono di mantenere informazioni relative all'intero ciclo di vita della procedura, fino all'inserimento del risultato all'interno dello storico.

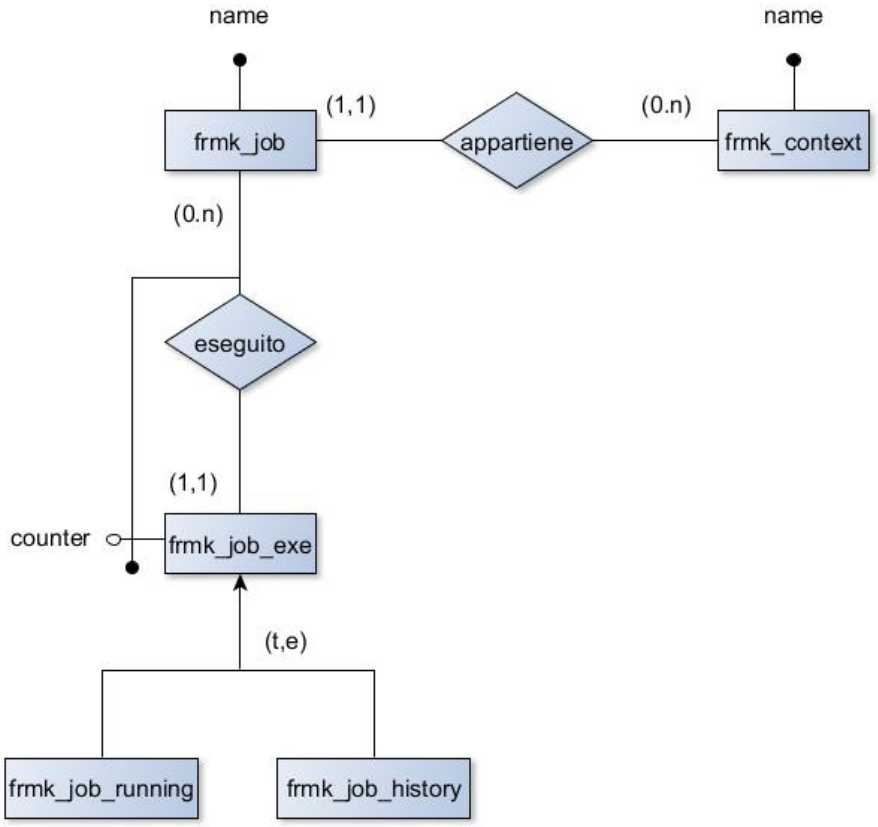

Fig. 2.4: Schema Entity Relationship

Possono quindi essere definite le seguenti entità**:**

CONTEXT (name, url)

JOB (name, context, cronexpression, classname, statefulYN)

FK: context REFERENCES frmk_context

JOB_RUNNING (name, counter, nodename, state)

FK: name REFERENCES frmk_job

JOB_HISTORY (name, counter, nodename, date_start, date_end, duration, result, state, message)

FK: name REFERENCES frmk_job

2.3 Tabelle di progetto

Essendo il database popolato da uno svariato numero di tabelle, tra le quali diverse facenti riferimento alla schedulazione di processi, si è scelto di aggiungere il prefisso FRMK_ per evidenziare l'appartenenza delle tabelle al progetto del framework.

L'entità context viene tradotta nella tabella frmk_job_context

```
CREATE TABLE `frmk_job_context` (
  `NAME` varchar(200) NOT NULL,
  `URL` varchar(200) NOT NULL,
  PRIMARY KEY (`NAME`))
```

L'entità job viene tradotta nella tabella frmk_job

```
CREATE TABLE `frmk_job` (
  `NAME` varchar(200) NOT NULL,
  `CONTEXT` varchar(200) NOT NULL,
  `CRONEXPRESSION` varchar(200) NOT NULL,
  `CLASSNAME` varchar(200) NOT NULL,
  `STATEFULYN` smallint(6) DEFAULT '0',
  PRIMARY KEY (`NAME`),
    CONSTRAINT FOREIGN KEY (`CONTEXT`) REFERENCES `frmk_job_context`
(`NAME`))
```

L'entità job_running viene tradotta nella tabella frmk_job_running

```
CREATE TABLE `frmk_job_running` (
  `NAME` varchar(200) NOT NULL,
  `COUNTER` decimal(19,0) NOT NULL,
  `NODENAME` varchar(200) DEFAULT NULL,
  `STATE` varchar(200) DEFAULT 'RUNNABLE',
  PRIMARY KEY (`NAME`, `COUNTER`),
CONSTRAINT FOREIGN KEY (`NAME`) REFERENCES `frmk_job` (`NAME`))
```

L'entità job_history viene tradotta nella tabella frmk_job_history

```
CREATE TABLE `frmk_job_history` (
  `NAME` varchar(200) NOT NULL,
  `COUNTER` decimal(19,0) NOT NULL,
  `NODENAME` varchar(200) NOT NULL,
  `DATE_START` datetime DEFAULT NULL,
  `DATE_END` datetime DEFAULT NULL,
  `DURATION` bigint(30) DEFAULT NULL,
  `RESULT` varchar(200) NOT NULL,
```

```
`STATE` varchar(200) NOT NULL,
`MESSAGE` text,
PRIMARY KEY (`NAME`, `COUNTER`),
CONSTRAINT FOREIGN KEY (`NAME`) REFERENCES `frmk_job` (`NAME`))
```

Capitolo 3

Implementazione della soluzione

3.1 Strumenti

Il progetto è stato svolto tramite l'utilizzo dei seguenti strumenti:

- Ambiente di sviluppo: Eclipse Java IDE
- Linguaggio di programmazione: Java 1.6
- Librerie esterne: Quartz 1.6

Prima di procedere nella trattazione è necessario descrivere due classi di grande rilevanza e ampiamente utilizzate:

- *FrameworkCoreManager*: implementata con lo scopo principale di fornire metodi di gestione e interconnessione con il database, sfrutta la combinazione di due dei principali Design Patterns utilizzati in ambito di applicazioni Enterprise:
    - *Abstract Factory*: pattern creazionale mediante il quale è possibile la creazione di oggetti dipendenti senza specificare concretamente le classi associate.
    - *Data Access Object (DAO):* oggetto java che fornisce una interfaccia astratta per l'interconnessione con varie tipologie di database o meccanismi di persistenza.
- *SchedulerManager*: classe java implementata con lo scopo di fornire metodi di gestione dello scheduler e dei job istanziati
- *SysConfigHandler*: classe java simile allo SchedulerManager, ha il compito di fornire strumenti di gestione e reperimento dei dati di configurazione.

3.2 Panoramica del sistema

Nella figura 3.1 è rappresentato un sistema caratterizzato da una configurazione distribuita formata da due server, su ognuno di questi sono presenti due web application:

- *Scheduler*: applicazione web che implementa il servizio di schedulazione e le interfacce per la gestione attraverso web browser.
- *Application*: applicazione web che sfrutta lo scheduler per eseguire procedure temporizzate.

Partendo dal presupposto che entrambi i nodi del cluster siano spenti e vengano accesi, non necessariamente contemporaneamente, verrà descritto il funzionamento di un solo nodo seguendo gli step riportati in figura. La trattazione può essere applicata a tutti i server che compongono il sistema distribuito.

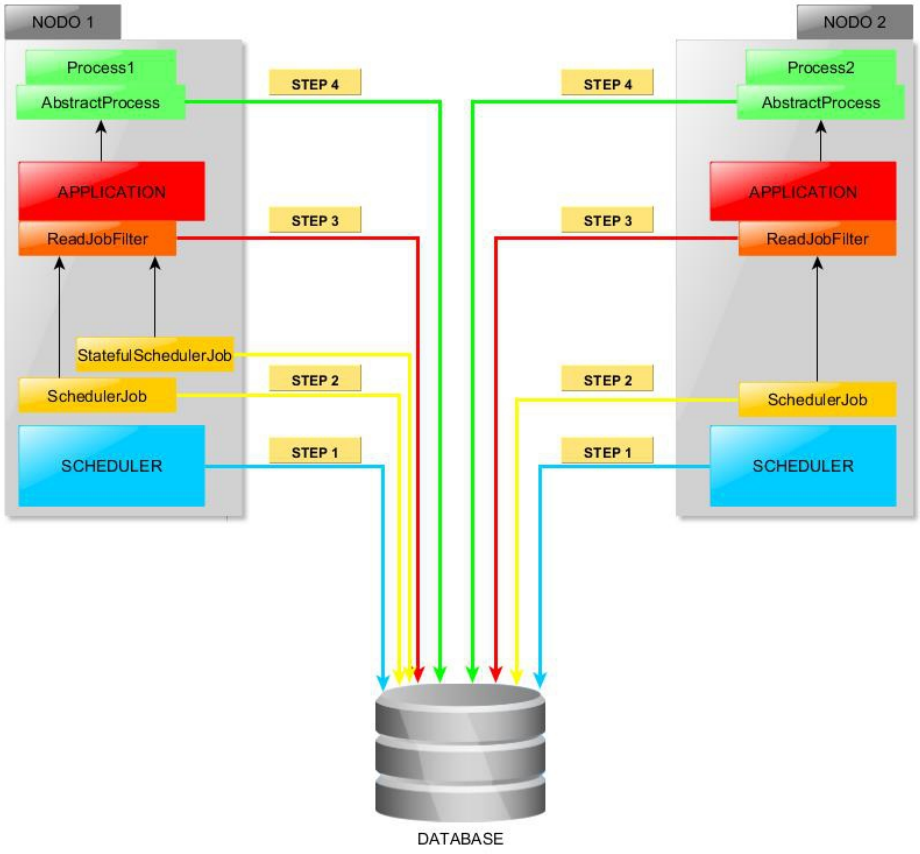

Fig. 3.1: Descrizione del sistema distribuito

3.3 Step 1: Caricamento dei job nello scheduler

Durante la fase di accensione l'applicazione non possiede informazioni riguardanti i nodi che compongono il sistema. Attraverso un'interrogazione delle tabelle viste in precedenza è possibile reperire lo stato di ogni nodo aggiornato all'ultima modifica effettuata. È quindi possibile effettuare un aggiornamento che permetta di allineare il nodo appena avviato con il sistema già in esecuzione.

Nella fase iniziale l'aggiornamento e il caricamento dello scheduler hanno la priorità su qualunque altra operazione, attraverso l'utilizzo di una servlet è possibile definire operazioni eseguibili durante l'accensione del server. Questa interfaccia java impone la definizione di tre metodi, ognuno dei quali descrive una diversa fase del ciclo di vita dell'oggetto associato alla classe.

I metodi in questione risultano essere:

- *init():* gestisce la fase di inizializzazione e rappresenta il metodo all'interno del quale verranno effettuate le interrogazioni al database e il caricamento dello scheduler
- *service():* permette di gestire le richieste effettuate dall'utente e le conseguenti risposte
- *delete():* permette di gestire la fase di eliminazione della servlet

Se configurata all'interno del file web.xml dell'applicazione, la fase di inizializzazione viene effettuata durante l'avviamento del servlet container.

Il servizio di inizializzazione è fornito dallo SchedulerManager attraverso il metodo startScheduler, un valore booleano permette di differire i casi di inizializzazione e di riavvio dello scheduler.

```
schedulerManager.startScheduler(true);
```

È sotto riportato l'activity diagram del metodo startScheduler in un caso di inizializzazione.

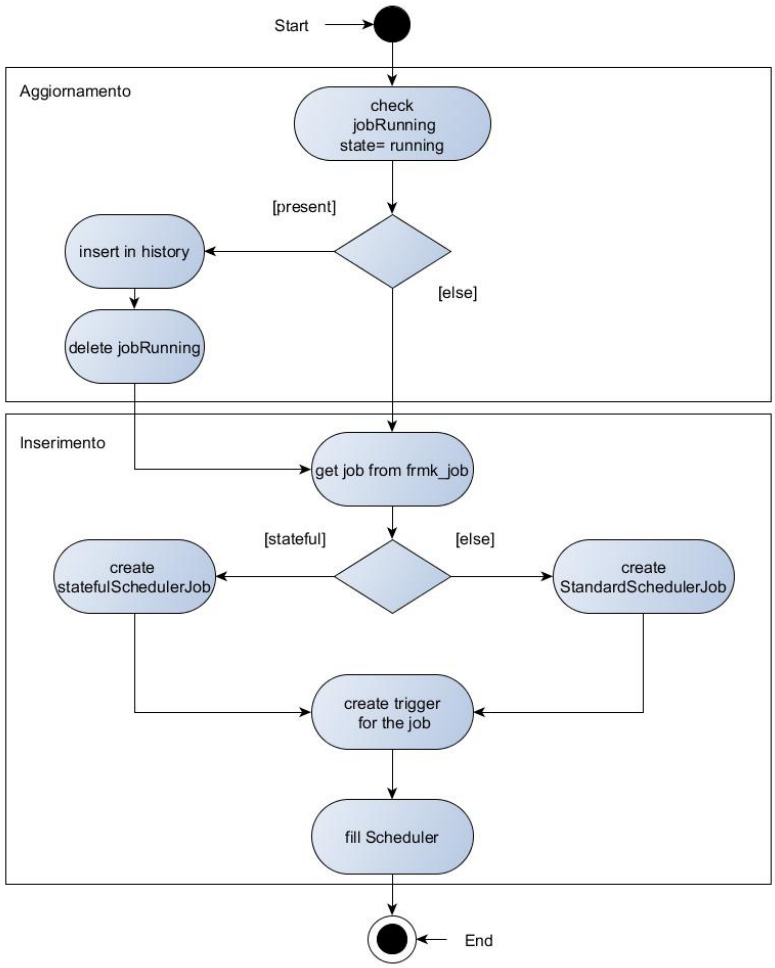

Fig. 3.2: Diagramma di inizializzazione dello scheduler

Aggiornamento delle tabelle

Il primo passo è verificare se durante l'ultimo spegnimento del servlet container erano presenti nel sistema procedure in esecuzione sul nodo. Definendo l'oggetto JobRunningSearch è possibile esplicitare i parametri utilizzati per comporre la query di interrogazione. Il metodo getJobRunningList, fornito dal frameworkCoreManager, riceve in input l'oggetto JobRunningSearch e fornisce la lista di istanze della tabella frmk_job_running che soddisfa i parametri precedentemente definiti.

Richiedendo le istanze della tabella con stato running e nome del nodo coincidente con quello considerato, è possibile ricavare la lista di procedure che non hanno portato a termine le operazioni richieste. Vengono riportati all'interno della tabella di storico frmk_job_history i dati relativi a tali procedure specificando come risultato failed e stato aborted.

Caricamento dello scheduler

Il database risulta essere ora allineato e consistente con lo stato del sistema. Essendo

lo scheduler non ancora configurato, è necessario procedere con il caricamento dei job al suo interno.

Una volta reperiti i processi definiti nella tabella frmk_job, è possibile definire un oggetto JobDetail per ogni job, il suo compito è raggruppare tutti i parametri necessari ad un corretto inserimento della procedura.

I parametri forniti in input al costruttore sono:

- *Name*: nome del job, corrisponde al nome della procedura definita in tabella;
- *Group*: identificatore del nome del gruppo al quale i job appartengono, assume il valore default se definito nullo;
- *jobClass*: nome della classe che implementa il job.

Ad ogni procedura deve corrispondere una classe che implementi i comportamenti ad essa associati, è stato quindi necessario implementare un meccanismo di ereditarietà fra classi che permettesse di gestire i casi presi in esame.

Si distinguono quindi:

- *SchedulerJob:* è la classe padre, implementa la logica di gestione del database e di notifica all'applicazione adibita all'esecuzione della procedura.
- *StatefulSchedulerJob:* è una classe che estende quella precedentemente descritta, eredita attributi e metodi da SchedulerJob e implementa l'interfaccia StatefulJob, che permette il mantenimento dello stato.

```
public void startScheduler(boolean initializeScheduler) throws SchedulerException,
ParseException{
	if(initializeScheduler==false)
		cleanScheduler();
	else{
		JobRunningSearch jobRunningSearch=new JobRunningSearch();
		jobRunningSearch.setState(JobRunningStateEnum.RUNNING.toString());
		jobRunningSearch.setNodeName(System.getProperty("nodeName"));
		List<JobRunning> list = new ArrayList<JobRunning>();
		list = frameworkCoreManager.getJobRunningList(jobRunningSearch);
		Iterator<JobRunning> it = list.iterator();

		while(it.hasNext()){
			JobRunning jobRunning=it.next();

	frameworkCoreManager.updateJobRunningStateFromRunningToAborted(jobRunning);

			JobHistory jobHistory=new
JobHistory(jobRunning.getName(),jobRunning.getCounter()
	,jobRunning.getContext(), jobRunning.getNodeName(),"FAILURE",
				JobRunningStateEnum.ABORTED.toString());

			frameworkCoreManager.saveJobHistory(jobHistory);
		}

	}
```

```
        JobSearch jobSearch=new JobSearch();
        List<Job> jobList=new ArrayList<Job>();

        jobList= frameworkCoreManager.getJobList(jobSearch);

        Iterator<Job> it3=jobList.iterator();
        JobDetail job=null;
        while(it3.hasNext()){
                Job dbJob=it3.next();
                if(dbJob.getStatefulYN()==true)
                        job = new JobDetail(dbJob.getName(), null,
StatefulSchedulerJob.class);
                else
                        job = new JobDetail(dbJob.getName(), null, SchedulerJob.class);

                String triggerName=new String();
                triggerName=dbJob.getName();

                if(log.isInfoEnabled())
                        log.info("creating trigger with name"+triggerName);

                CronTrigger trigger= new
CronTrigger(triggerName,null,job.getName(),null);
                try {
                        trigger.setCronExpression(dbJob.getCronExpression());
                } catch (ParseException e) {
                        if(log.isErrorEnabled()){
                                log.error("unable to create the trigger", e);
                        }
                }
                try {
                        scheduler.scheduleJob(job, trigger);
                } catch (SchedulerException e) {
                        if(log.isErrorEnabled()){
                                log.error("unable to create the job ",e);
                        }
                }
        }
}
```

3.4 Step 2: Esecuzione dei job dello Scheduler

Prima di descrivere il funzionamento dello step 2 è necessario specificare che nel continuo della spiegazione verrà fatta una sostanziale differenza quando ci si riferirà a job e procedura:

- *Job*: classe java i quali compiti sono:
  - Predisporre il sistema (in particolare le tabelle del database) per un funzionamento corretto senza collisioni tra i vari nodi del cluster;
  - Segnalare all'applicazione appartenente allo stesso nodo dello scheduler considerato, che è presente una procedura appartenente al suo contesto in attesa di essere presa in consegna.
- *Procedura*: classe il cui compito è eseguire le operazioni richieste dall'utente, è situata all'interno dell'applicazione.

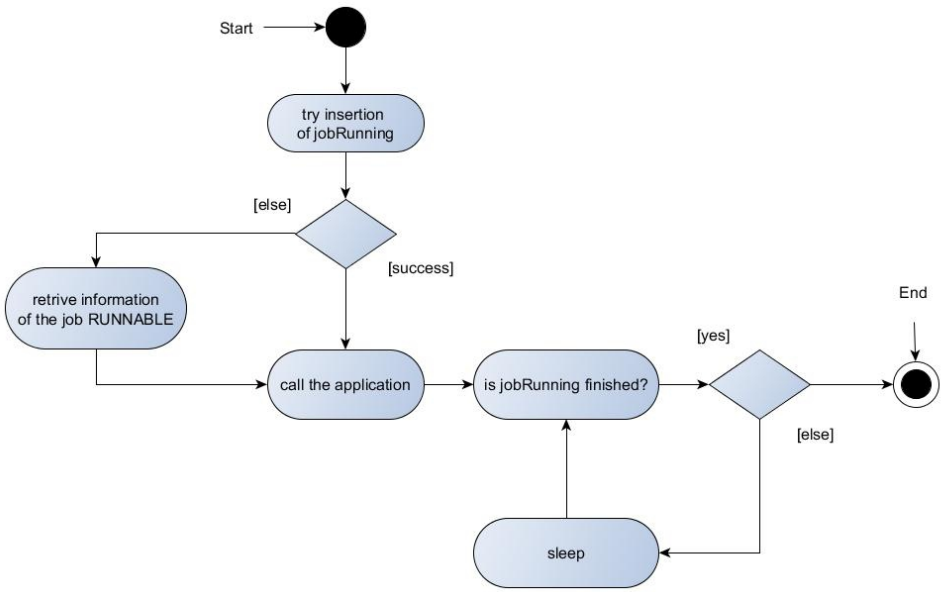

Fig 3.2: Diagramma rappresentante l'esecuzione di un job

Verrà ora descritta la classe SchedulerJob rappresentante il job standard, questa implementa due interfacce:

- *Job*: necessaria per assegnare alla classe i connotati del 'job';
- *InterruptableJob*: permette l'interruzione del job in esecuzione.

Il metodo principale della classe è execute, imposto dalla prima interfaccia, questo riceve come dati di input il JobExecutionContext, oggetto che raccoglie tutte le

informazioni relative all'esecuzione e alla gestione del job.

È possibile suddividere il funzionamento del metodo in due fasi:

Preparazione del database

Tramite il JobExecutionContext è possibile reperire il nome del jobDetail, questo per vincoli di progettazione è in corrispondenza 1:1 con il nome di una procedura definita all'interno della tabella frmk_job.

```
String jobName=jonExecutionContext.getJobDetail().getName();
```

Supponendo la presenza di un job stateless, due istanze di una procedura possono essere in esecuzione contemporaneamente, è quindi necessario reperire il valore del contatore.

Il passo successivo è il tentativo di inserimento di un record identificativo della procedura all'interno della tabella JobRunning con un stato runnable. Per vincoli di progetto non è possibile inserire nella tabella un riferimento ad una procedura con stato runnable se già presente un'istanza di quest'ultima. La query per l'implementazione di questo vincolo utilizza frmk_job_running_support, una tabella di appoggio contenente un solo valore costante, il suo scopo è consentire di specificare una condizione sull'inserimento.

```
insert into jobRunning value(name, context, counter,state)
select 'nameJob','nameContext','n','runnable'
from frmk_job_running_support
where not exist(select *
                from frmk_job_running
                where name=nomejob and state=runnable)
```

In caso di successo è possibile dedurre che tutte le istanze della procedura sono state gestite correttamente. L'insuccesso si traduce in una mancata presa in consegna della procedura precedentemente inserita, questo significa che il sistema potrebbe aver subito un'anomalia, o il numero di procedure definite sono maggiori di quelle che i worker dello scheduler sono in grado di gestire. È compito del nodo, accortosi del problema, tentare di prendere in carico la procedura rimasta in sospeso non andando ad inserire nuove richieste. Questo comporta il reperimento del valore del contatore associato all'istanza presente in tabella.

Una volta noti nome e contatore della procedura caratterizzata dallo stato runnable è possibile registrare all'interno dei file di log informazioni utili all'amministratore di sistema. Quest'ultimo avrà il compito di indagare sulle motivazioni dell'insuccesso nel

lancio di una procedura.

Notifica all'applicazione

Viene ricavato l'url dell'applicazione che implementa la procedura assegnata, questo è normalmente composto da:

- *http://* - identifica il protocollo utilizzato;
- *localhost* - valore di default, permette di chiamare l'applicazione presente sullo stesso nodo dello scheduler, in questo modo è possibile massimizzare la velocità di transizione da scheduler ad applicazione, e diminuire conseguentemente la dilazione temporale nell'esecuzione della procedura;
- *context* - nome dell'applicazione da evocare;
- *readJob* - identifica il filtro dell'applicazione adibito alla gestione della presa in consegna della procedura.

Attraverso una chiamata http è possibile richiamare il filtro dell'applicazione le cui funzionalità verranno descritte nello step 3. Per aumentare il livello di sicurezza si è scelto di non trasmettere informazioni critiche quali nome del job e contatore mediante chiamata http.

Il passo successivo è la chiusura forzata della connessione utilizzata per la comunicazione tra scheduler e applicazione, l'eccezione derivante dal timeout viene gestita riportando l'evento all'interno del file di log.

Attesa del termine della procedura

Una volta effettuata la notifica, il job ha il compito di monitorare l'esecuzione del processo. Non potendo verificare lo stato di avanzamento tramite interrogazione all'oggetto della classe, vengono utilizzati i dati presenti all'interno delle tabelle del database.

Le variabili che permettono il monitoraggio sono:

- *finishedJob*: valore booleano, indica se la procedura è terminata;
- *state*: stringa che esprime lo stato della procedura;
- *timeSleep*: costante di attesa tra due interrogazioni al database.

Per mantenere una corrispondenza tra i job e le procedure in esecuzione è necessario mantenere il worker allocato finchè le operazioni svolte non sono terminate. Vengono quindi effettuate interrogazioni periodiche alla tabella frmk_job_running finché lo stato della procedura non risulta finished o aborted.

Il motivo principale per cui viene mantenuto worker istanziato è non permettere il lancio di un nuovo job. Questo infatti porterebbe creare un disaccoppiamento con i processi in esecuzione e l'accumulo di richieste che il sistema non sarebbe in grado di evadere.

```
public void execute(JobExecutionContext jec) throws JobExecutionException {
	ApplicationContext appCtx = null;
	try {
		appCtx = (ApplicationContext)
		jec.getScheduler().getContext().get("applicationContext");
		if (appCtx == null)
			throw new JobExecutionException("No application context
			available in scheduler context for key= applicationContext");
	} catch (Exception e) {
		throw new SystemException(e);
	}
	FrameworkCoreManager frameworkCoreManager = (
	FrameworkCoreManager)appCtx.getBean("frameworkCoreManager");
	//Fase di preparazione del database
	String jobName=jec.getJobDetail().getName();
	if(log.isInfoEnabled())
		log.info("executing SchedulerJob with jobName= "+jobName);
	String counterName="job".concat(jobName);
	long counter = 0;
	String jobContext=new String();
	try {
		counter=frameworkCoreManager.getCounter(counterName).getValue();
	} catch (ApplicationException e2) {
		String message="error setting the counter for job: " + jobName;
		if(log.isErrorEnabled())
			log.error(message, e2);
	}
	try {
		jobContext = frameworkCoreManager.getJob(jobName).getContext();
	} catch (DAOFinderException e2) {
		String message="error getting the context of the job: "
		.concat(jobName);
		if(log.isErrorEnabled())
			log.error(message, e2);
	}

	try {
		frameworkCoreManager.saveJobRunning(jobName, jobContext, counter);
	} catch (DAOCreateException e1 ) {
```

```
                if(log.isDebugEnabled()){
                        log.debug("there's a jobRunning already runnable ", e1);
                }
                try {

        counter=frameworkCoreManager.getJobRunningRunnable(jobName).getCounter();
                } catch (DAOFinderException e) {
                        if(log.isErrorEnabled()){
                                log.error("unable to get the job with state=RUNNABLE ",
                                e1);
                        }
                }
                if(log.isDebugEnabled()){
                        log.debug("joRunning in state=Runnable" + jobName + " counter=
                        " + counter, e1);
                }
        }
        //Fase di notifica
        String urlString=new String();

        try {
                urlString=frameworkCoreManager.getJobContextUrl(jobName);
        } catch (DAOFinderException e) {
                if(log.isErrorEnabled())
                log.error("error retrieving the url from the context of" + jobName);
        }
        if(urlString!=null){
                HttpClient client = new HttpClient();
                GetMethod method = new GetMethod(urlString);
                try {
                        client.getParams().setSoTimeout(500);
                        int statusCode = client.executeMethod(method);
                } catch (Exception e) {
                        if(log.isDebugEnabled())
                                log.debug("CHIUSURA SOCKET ULTIMATA");
                }
        }

        //Fase di interrogazione
        boolean finishedJob=false;
        String state = new String();
        int timeSleep=1000 * 30;
        while(finishedJob==false){
                try {
                        Thread.sleep(timeSleep);
                } catch (InterruptedException t) {
                        if(log.isErrorEnabled()){
                                log.error("unable send to sleep the job " + jobName,t);
                        }
                }
                try {
                        state =frameworkCoreManager.getJobRunning(jobName,
counter).getState();
                        if(state.equals("FINISHED") || state.equals("ABORTED") ){
                                finishedJob=true;
                        }
                } catch (DAOFinderException e) {
                        if (log.isErrorEnabled()) {
                                log.error("Unable to check the state of the job=
                                "+jobName+" counter= "+ counter,e);
                        }
                }}}
```

3.5 Step 3: Filtro Readjobfilter dell'applicazione web

ReadJobFilter è una classe java che permette all'applicazione di gestire i processi eseguibili, implementa l'interfaccia Filter, questo obbliga lo sviluppatore ad

implementare tre metodi:

- *init*(): come per la servlet descritta nello step 1, questo metodo permette l'inizializzazione dell'oggetto nel momento in cui il servlet container viene avviato;
- *doFilter*(): metodo invocato nel momento in cui avviene scatenato il filtro;
- *destroy*(): metodo invocato per l'eliminazione del filtro;

In aggiunta ai metodi obbligatori è stato necessario fornire un metodo aggiuntivo nominato *invoke*(), il suo compito è quello di gestire la concorrenza evitando collisioni fra le varie richieste effettuate dai thread appartenenti alla web application.

1.INIT()

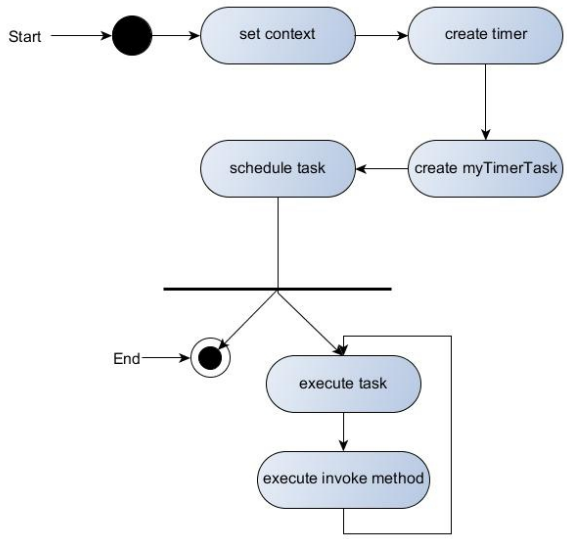

Fig. 3.3: Diagramma di inizializzazione di una procedura

Il filtro non ha accesso ai dati riguardanti l'applicazione a cui appartiene, risulta quindi necessario settare il nome del contesto dell'applicazione web che lo contiene, questo risulterà utile per la lettura delle procedure schedulate all'interno del database.

```
context=config.getServletContext().getContextName();
ApplicationContextUtils.setContext(context);
```

ApplicationContextUtils è una classe java creata appositamente per questo progetto, il suo compito è quello di mantenere i dati relativi all'applicazione, uno dei più importanti è il nome del contesto, essenziale all'interno del frameworkCoreManager per accedere ai dati utilizzando il nome dell'applicazione.

Per definire il sistema resiliente è necessario implementare un meccanismo che permetta di adattarsi e gestire condizioni anomale, offrendo un servizio di

schedulazione il più efficiente possibile. Viene definito un thread, il cui scopo è verificare la presenza di procedure inserite all'interno della tabella frmk_job_running ma non prese in consegna. Il thread necessita di essere eseguito costantemente, viene quindi implementato attraverso la classe di utilità Timer che permette l'iterazione di un processo ad intervalli regolari. Attraverso le metodologie fornite da Spring è possibile reperire il JavaBean dell'oggetto che si occuperà di svolgere le interrogazioni periodiche. L'implementazione del timer avviene attraverso l'utilizzo del metodo schedulerAtFixRate, questo permette di settare:

- Il tempo di ritardo da applicare alla prima esecuzione
- L'intervallo tra due invocazioni.

```
public void init(FilterConfig config){
	if(log.isInfoEnabled())
		log.info("ReadingJobFilter Initialized!");
	String context = new String();
	context=config.getServletContext().getContextPath();
	context=StringUtils.removeStart(context, "/");
	ApplicationContextUtils.setContext(context);
	Timer uploadCheckerTimer = new Timer(true);
	TimerTask timerTask =
	(TimerTask)ApplicationContextUtils.getApplicationContext().
	getBean("myTimerTask");
	uploadCheckerTimer.scheduleAtFixedRate(timerTask, 30 * 1000, 30 * 1000);
}
```

2.DOFILTER()

Invocato una volta ricevuta la richiesta http dal job dello scheduler, ha il compito di effettuare la chiamata al metodo invoke().

3.INVOKE()

Questo metodo permette la presa in consegna della procedura fornita dallo scheduler. Essendo il metodo eseguito da più thread in modo asincrono, la progettazione ha avuto come scopo principale il mantenimento della consistenza dei dati e la gestione dei thread in modo da evitare collisioni.

Il metodo è stato definito statico, questo gli conferisce la particolarità di appartenere alla classe, può essere quindi invocato senza l'obbligo di istanziare un oggetto.

Il metodo è composto da un unico blocco di codice sincronizzato, questo costrutto basa il suo funzionamento su un entità nota come monitor. Il monitor è un attributo anch'esso statico della classe che permette di implementare il lock sulla porzione di codice situata all'interno del blocco synchronized. Il thread in esecuzione per avanzare deve acquisire l'oggetto monitor, essendo questo statico può essere assegnato ad un solo thread alla volta.

Un altro aspetto da considerare è la *fairness*, cioè la capacità del sistema di allocare le risorse in base all'ordine in cui sono state richieste, nel caso preso in esame si traduce in un rispetto dell'ordine con cui viene richiesto il monitor. Si è scelto di non implementare un meccanismo che soddisfacesse questa proprietà per non inficiare le prestazioni del sistema, ricordando che la presenza di un meccanismo di gestione dell'ordine di esecuzione è già fornito dallo scheduler attraverso l'uso dei trigger.

Il primo passo è la definizione del numero di worker disponibili per il nodo considerato, questo dato di configurazione è reperibile attraverso l'utilizzo della classe SysConfigHandler mediante il codice FRMK026_SchedulerWorkerNumber.

Per mantenere il sistema performante è necessario non avviare un numero di procedure maggiore del numero di worker disponibili, questo permette di evitare una esecuzione anomala del thread TimerTask e mantenere il numero di procedure in esecuzione minore o uguale del numero di worker disponibili.

Nel caso in cui il nodo risulti in uno stato consono alla presa in gestione di un nuovo processo, viene invocato un nuovo thread di nome startProcedure, descritto successivamente.

```
public static void invoke(SysConfigHandler sysConfigHandler, FrameworkCoreManager
frameworkCoreManager, ApplicationContext ac){
	synchronized (mutex) {

		Integer nodeNumberJob=null;
		try {
			BigInteger numOfJobExecutorBI =
			sysConfigHandler.getConfigAsInteger ("SCHEDULER", "DEFAULT",
			"FRMK026_SchedulerWorkerNumber");
			nodeNumberJob=numOfJobExecutorBI.intValue();
		} catch (DAOFinderException e) {
			 if(log.isErrorEnabled())
				log.error("unable to find the sysConfig
				FRMK026_SchedulerWorkerNumber",e);
		} catch (Exception e) {
			if(log.isErrorEnabled())
				log.error("error getting the sysConfig
					FRMK026_SchedulerWorkerNumber",e);
		}
		JobRunning jobRunning= new JobRunning();
		jobRunning.setContext(ApplicationContextUtils.getContext());
		jobRunning.setState("RUNNING");
		long nrOfRunningJob = frameworkCoreManager.countJobRunning(jobRunning);
		if(nodeNumberJob<=nrOfRunningJob) {
			if(log.isInfoEnabled()){
				log.info("THREAD NOT AVAIBLE");
				return;
			}
		}

		if(jobPool==null) {
			jobPool=new DefaultThreadPool(5);
		}
		Thread t =(Thread)ac.getBean("startFilterTask");
```

```
                jobPool.invokeLater(t);
        }
}
```

3.6 Step 4: Esecuzione della procedura

L'esecuzione della procedura può essere descritta analizzando gli elementi coinvolti:

1.Thread startProcedure

Il suo compito è quello di garantire una corretta gestione della procedura e la creazione di un ambiente consono alla sua esecuzione.

Quando il sistema interagisce con un oggetto di questa classe, visti i controlli esplicati in precedenza, ci si aspetterebbe che all'interno della tabella frmk_job_running sia presente un'istanza caratterizzata dallo stato runnable, questo non è assicurato. Tramite l'esecuzione asincrona dei thread precedentemente descritti, il risveglio di un timer su un qualsiasi nodo del cluster potrebbe prendere in consegna la procedura. È necessario inserire un ulteriore controllo che permetta la gestione dell'evento anomalo, in modo tale da non creare inconsistenze. Essendo quella appena descritta una problematica appartenente alla gestione del database, la corretta implementazione è propria del gestore del servizio associato. Questo si traduce in un nuovo metodo fornito dal FrameworkCoreManager che maschera la complessità al chiamante.

Il suddetto metodo, chiamato entrustJobRunning, permette di:

- Identificare la presenza di un istanza corretta attraverso il contesto dell'applicazione
- Gestire le eccezioni in maniera consona

Una volta noto il nome della classe associata alla procedura è possibile creare una sua nuova istanza, anche in questo caso non è possibile effettuare una esecuzione non curandosi dei possibili errori derivanti, è necessario che il lancio della procedura avvenga in un ambiente protetto che si preoccupi di gestire i casi di successo e insuccesso. Viene quindi effettuato un *upcasting* dell'oggetto che identifica la procedura nella corrispondente classe astratta AbstractProcedure, questo permette una gestione dei valori di ritorno. Viene successivamente inserito l'oggetto creato all'interno dell' ApplicationContext in modo da poter usufruire dei meccanismi offerti da Spring per l'interazione tra oggetti, quali la dependency injection.

L'ultimo passaggio è il lancio del thread della classe astratta istanziata.

```
public void run() {
```

```
        String context=ApplicationContextUtils.getContext();
        JobRunning jobEntrusted=frameworkCoreManager.entrustJobRunning(context);

        if(jobEntrusted.getName()!=null){

        String className=new String();
        try {

        className=frameworkCoreManager.getJob(jobEntrusted.getName()).getClassName();
        } catch (DAOFinderException e1) {
               String message="Error getting the class associated with the job " +
                      jobEntrusted.getName();
               if(log.isErrorEnabled())
                      log.error(message,e1);
        }

        try {
               job=(AbstractProcedure)Class.forName(className).newInstance();
               job.setJobRunning(jobEntrusted);
               ApplicationContextUtils.getApplicationContext().
               getAutowireCapableBeanFactory().autowireBean(job);
               job.launch();

        } catch (InstantiationException e) {
               if(log.isErrorEnabled())
                      log.error("unable to instantiate the job class", e);
        } catch (IllegalAccessException e) {
               if(log.isErrorEnabled())
                      log.error("illegal access the job class" + className, e);
        } catch (ClassNotFoundException e) {
               if(log.isErrorEnabled())
                      log.error("unable to found the job class" + className, e);
        }
        }
        else
               if(log.isDebugEnabled())
                      log.debug("there is no job ready to start");
}
```

2.AbstractProcedure ed esecuzione della procedura

Il primo elemento da analizzare è AbstractProcedure, questa classe java ha il compito di mantenere dati relativi alla procedura eseguita e gestire le eventuali eccezioni create. I suoi compiti principali sono:

- Mantenere i riferimenti della data di inizio e fine esecuzione
- Invocare la procedura
- Aggiornare lo stato di esecuzione nella tabella frmk_job_running Inserire i dati raccolti e il risultato dell'esecuzione all'interno dello storico
- Allegare all'inserimento un messaggio di errore nel caso di un insuccesso

```
public void launch(){

        Date startTime=new Date();
        if(log.isDebugEnabled()){
               log.debug("startTime="+ startTime);
        }
        boolean result=false;
        long duration=0;
```

```
        String prepareMessage = new String();
        try{
                result=execute();
        }catch (Exception e){
                ByteArrayOutputStream out = new ByteArrayOutputStream();
                PrintStream pt = new PrintStream(out);
                e.printStackTrace(pt);
                pt.close();
                byte[] byteArray = out.toByteArray();
                errorMsg = new String(byteArray);
                if(log.isErrorEnabled()){
                        log.error(errorMsg, e);
                }
                prepareMessage=errorMsg;
        }
        Date endTime=new Date();

        duration=endTime.getTime()-startTime.getTime();
        if(log.isDebugEnabled()){
                log.debug("endTime="+ endTime);

        }
        String resultValue= new String();
        JobHistory job;

        if(result==false){
                resultValue="FAILURE";
                job= new JobHistory(jobRunning.getName(),...,prepareMessage);
                }else{
                        resultValue="SUCCESS";
                        job= new JobHistory(jobRunning.getName()...);
                }
                frameworkCoreManager.updateJobRunningStateFromRunningToFinished(
                jobRunning);
                frameworkCoreManager.saveJobHistory(job);

        }
}
```

Il secondo elemento da definire è la procedura, questa deve obbligatoriamente presentare le seguenti caratteristiche:

- Estendere la classe astratta AbstractProcedure.
- Implementare il metodo execute() con annotazione @Override, questo permetterà l'effettiva esecuzione del metodo appartenente alla procedura.

Viene sotto riportato un esempio di prova, sarà compito dei soggetti che utilizzeranno lo scheduler definire le operazioni da eseguire:

```
public class Procedure1 extends AbstractProcedure {
        @Override
        protected boolean execute() throws SystemException {

                return true;
        }
}
```

Capitolo 4

Interfacce

4.1 Strumenti

Le interfacce sono state realizzate utilizzando i seguenti strumenti

- Ambiente di sviluppo: Eclipse Java IDE
- Linguaggio di programmazione: Java 1.6
- Librerie esterne
    - Spring Framework 2.5.6
    - Struts 2.3.15.1
    - Java Servlet Pages 2.5

Una spiegazione approfondita di alcuni dei framework sopra riportati è disponibile all'interno dell'appendice.

4.2 Descrizione

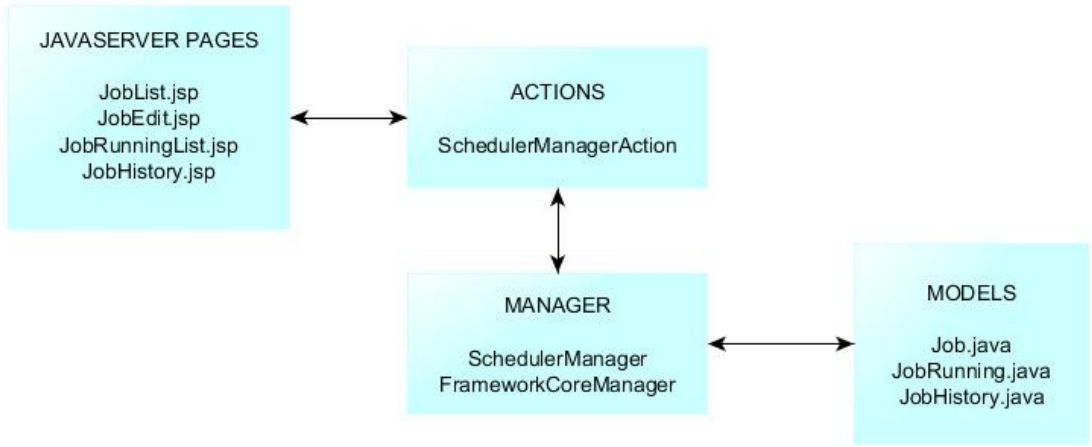

Fig 4.1: Descrizione dei componenti del progetto web

Il progetto delle interfacce si compone di:

- quattro pagine jsp: in corrispondenza 1:1 con le tabelle del database, permettono la visualizzazione dei dati e la fruizione dei servizi implementati;
- SchedulerManagerAction: classe che permette di gestire le richieste effettuate dall'utente interagendo con le pagine jsp;
- SchedulerManager: classe che offre servizi di gestione dello scheduler;
- FrameworkCoreManager: classe che offre servizi di gestione del database;
- Models: rappresentazione java delle tabelle sul database;
- file xml per la configurazione dei framework utilizzati;
- immagini e file css per la cura dell'aspetto grafico delle pagine;

I task della action SchedulerManagerAction sono riportati nella tabella seguente:

Task	**Obiettivo**
listJob	Ha il compito di reperire i job definiti all'interno della tabella frmk_job
listJobRunning	Ha il compito di reperire i job definiti all'interno della tabella frmk_job_running
listJobHistory	Ha il compito di reperire lo storico situato nella tabella frmk_history
Edit	Prepara le variabili utilizzati nel salvataggio di un nuovo job
Save	Richiede il servizio di salvataggio al frameworkCoreManager

Task	**Obiettivo**
resetScheduler	Richiede un servizio di reset allo SchedulerManager che elimina i tutti i job caricati all'interno dello scheduler e li reinserisce previa lettura dalla tabella del

	database
Delete	Permette l'eliminazione di un job, questa operazione va a buon fine solamente nel caso in cui la procedura non sia mai stata eseguita
Stop	Permette l'interruzione del job
forceJobStart	Forza l'esecuzione di nuovo job, l'operazione va a buon fine solamente nel caso in cui sia disponibile un worker

4.3 Rappresentazione delle interfacce

Di seguito sono riportate le rappresentazioni delle quattro interfacce jsp che permettono all'utente di interagire con il sistema.

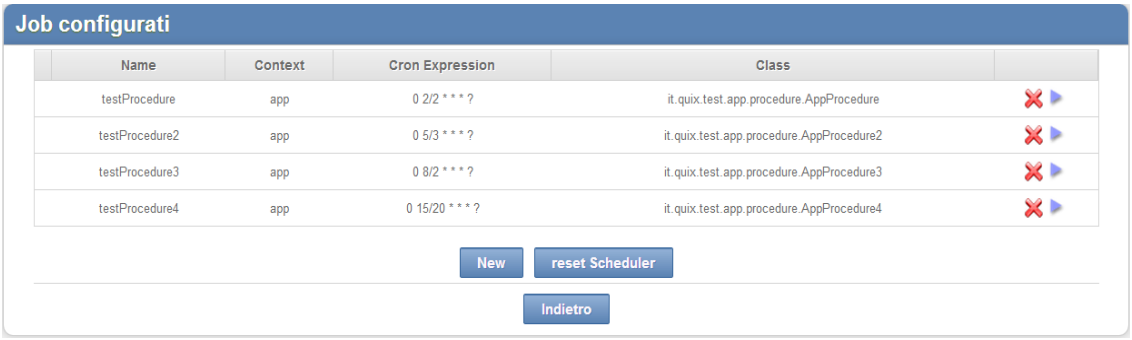

Fig. 4.2: Interfaccia associata alla tabella frmk_job

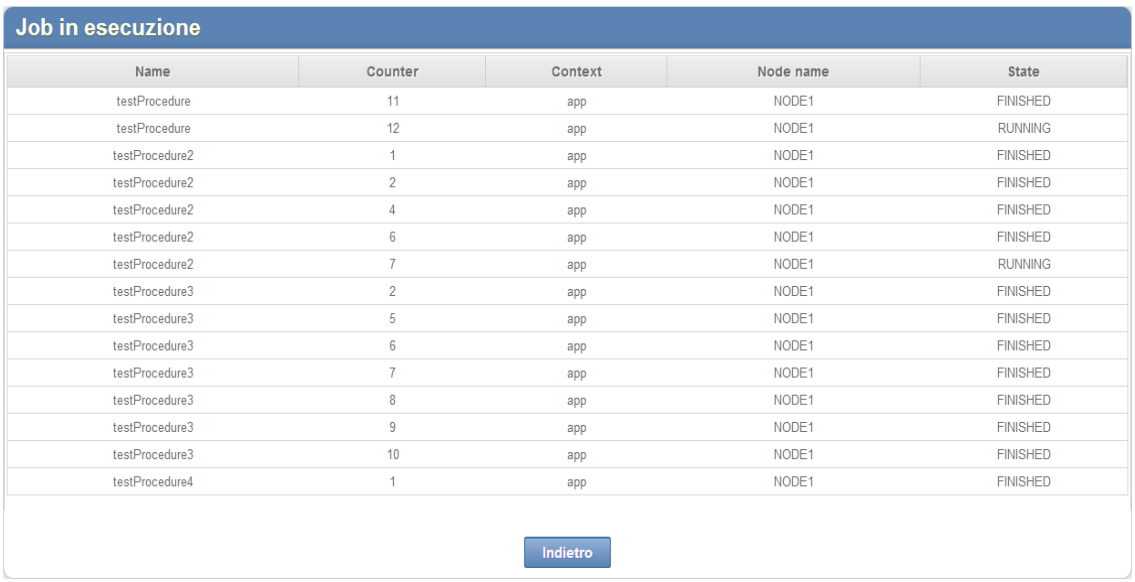

Fig 4.3: Interfaccia associata alla tabella frmk_running

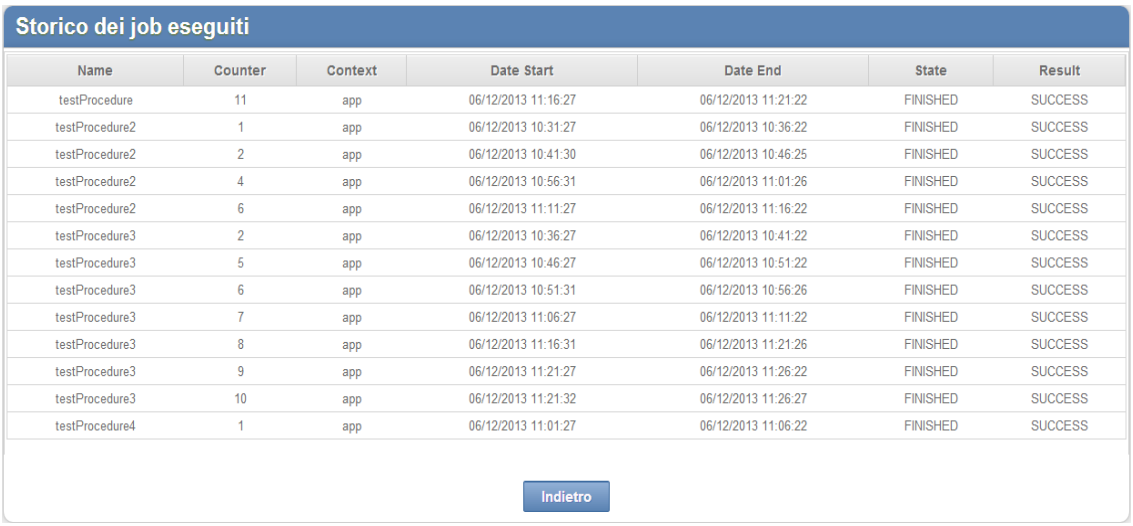

Fig. 4.4: Interfaccia associata alla tabella frmk_job_history

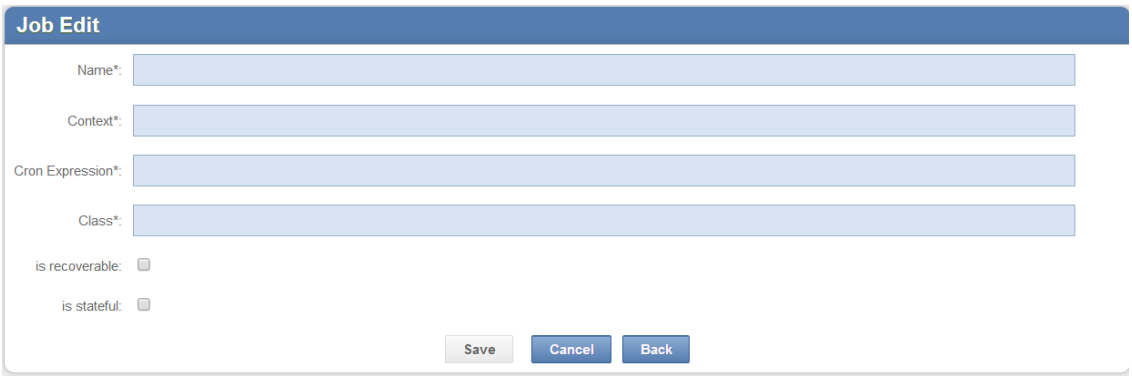

Fig. 4.5: Interfaccia per l'inserimento di un nuovo job

Capitolo 5

Integrazione dello scheduler con il framework

5.1 Creazione di un progetto tramite il framework

La creazione di un progetto attraverso l'utilizzo del framework è divisa in due fasi, entrambe utilizzano come strumento per l'inserimento dei dati di input un file xml. Attraverso la definizione dei dati richiesti è possibile beneficiare delle varie funzionalità fornite dal framework.

Fase1: Generazione dello scheletro del progetto

Il primo file xml permette di definire i dati che identificano univocamente il progetto secondo le regole definite da Maven.

I parametri necessari sono:

- *framework-version*: permette di stabilire quale versione utilizzare per la generazione del progetto, nel caso in cui l'applicazione venga creata con lo scopo di effettuare test verrà scelta la versione chiamata SNAPSHOT, sapendo che questa potrebbe contenere difetti di progettazione. Nel caso in cui si crei una applicazione con lo scopo di implementare un nuovo servizio effettivamente utilizzato, si opterà per l'ultima release stabile. La differenza sostanziale tra i due casi è che la versione snapshot è soggetta ad aggiornamenti.
- *project-name*: nome parlante da dare al progetto;
- *artifactId*: nome utilizzato nel momento della creazione del package relativo all'applicazione;
- *groupId*: identificativo unico del progetto creato;

```
<properties>
      <framework-version>03.00.153-SNAPSHOT</framework-version>
      <project-name>test</project-name>
      <artifactId>scheduler</artifactId>
      <groupId>it.quix.schedulerTest</groupId>
</properties>
```

Il framework estrapola i dati di configurazione dal file xml e crea lo scheletro del progetto secondo le specifiche desiderate. Il progetto così creato contiene solamente i

file standard presenti di default, questi variano da file di risorse a jsp per la visualizzazione delle pagine stardard.

Fase 2: Definizione delle funzionalità del progetto

Il secondo file di configurazione permette la definizione delle funzionalità da utilizzare, la scelta di usufruire di determinati servizi avviene tramite l'utilizzo di valori booleani associati a tag personalizzati.

Alcuni esempi delle funzionalità precedentemente esistenti:

- *useBaseDAO*: informa il sistema che come metodo di gestione del database verranno utilizzati i Data Access Object, verranno di conseguenza predisposti la DAOFactory e il CoreManager all'utilizzo di questo pattern di programmazione;
- *useEqualOnQuery*: permette di scegliere la modalità di interrogazione effettuata con il database, se abilitato permette l'utilizzo del simbolo '=' nelle comparazioni, false permette l'utilizzo di 'like';
- *useWebService*: indica se è richiesta una gestione tramite web service oltre che tramite protocollo http.

Per permettere all'utente l'utilizzo dello scheduler nei casi descritti in fase di progettazione è stato necessario aggiungere due ulteriori tag di configurazione:

- *useScheduler*: indica se per il progetto in questione è previsto l'uso dello scheduler;
- *isDistributed*: indica se l'applicazione creata è distribuita su più server.

È ora possibile descrivere le combinazioni possibili.

useScheduler	isDistributed	Descrizione
True	True	L'applicazione creata beneficerà dell'utilizzo del filtro precedentemente descritto, ma non conterrà al suo interno lo scheduler. Nel caso in esame sarà necessario creare una seconda applicazione implementante il servizio di schedulazione.
True	False	L'applicazione creata conterrà sia il servizio di gestione delle procedure che lo scheduler
False	True	Combinazione non gestita
False	False	Configurazione adatta ad una applicazione che non utilizza procedure schedulate

```
<properties>
      <framework-version>03.00.153-SNAPSHOT</framework-version>
      <useBaseDAO>true</useBaseDAO>
      <useEqualOnQuery>true</useEqualOnQuery>
      <useWebServices>false</useWebServices>
      <useScheduler>true</useScheduler>
      <isDistributed>false</isDistributed>
</properties>
```

5.2 Utilizzo dei tag durante la generazione dell'applicazione

Attraverso l'utilizzo di Maven è possibile configurare l'esecuzione di un task in modo tale che vengano rese disponibili proprietà personalizzate denominate sysproperty. Nel caso in esame viene caricata una sysproperty per ogni proprietà precedentemente definita.

All'interno del tag java è specificata la classe contenente il metodo utilizzato per l'avvio della generazione del progetto.

```
<execution>
   <id>Generate Project</id>
   <phase>generate-sources</phase>
   <configuration>
   <tasks>
   <javac sourcepath="" srcdir="${basedir}/../src/gen/java" destdir="$
{basedir}/../target/classes" >
      <include name="**/*.java"/>
   </javac>
   <java classname="it.quix.framework.codegen.launcher.CodeGeneratorLauncher">
      <sysproperty key="basedir" value="${basedir}"/>
```

```
        <sysproperty key="projectName" value="test"/>
        <sysproperty key="groupId" value="it.quix.schedulerTest"/>
        <sysproperty key="artefactId" value="scheduler"/>
        <sysproperty key="frameworkVersion" value="${framework-version}"/>
        <sysproperty key="useBaseDAO" value="${useBaseDAO}"/>
        <sysproperty key="useWebServices" value="${useWebServices}"/>
        <sysproperty key="useScheduler" value="${useScheduler}"/>
        <sysproperty key="isDistributed" value="${isDistributed}"/>
        .
        .
        .
      </java>
     </tasks>
    </configuration>
    <goals>
       <goal>run</goal>
    </goals>
</execution>
```

Al momento dell'avvio del programma di generazione del progetto la classe java specificata con il nome CodeGeneratorLauncer avrà il compito di reperire le proprietà attraverso il metodo `System.getProperty("keyValue")` e creare una mappa contenente l'insieme dei valori di configurazione utilizzabili durante la creazione dell'applicazione.

5.3 Creazione dinamica di script: Java Emitter Template

Un Java Emitter Template è un motore "model-to-text" che permette la generazione di testo basato su un modello seguente le regole dell'Eclipse Modeling Framework. La definizione di un template attraverso un processo di traduzione crea una classe java utilizzata per la creazione dell'output finale. Questo può essere una jsp, una classe java o un file xml.

La classe che implementa il template può essere inizializzata e attraverso il metodo generate(), creando una stringa contenente il testo del file da creare.

I template JET utilizzano tre differente tipologie di espressioni:

- *Scriptles* ***<% %> :*** permettono l'inserimento di codice java, vengono utilizzate soprattutto per la definizioni di condizioni.
- *Direttive* ***<%@ %>:*** permettono di gestire la traduzione del template.
- *Espressioni* ***<%= %>:*** sono espressioni java che vengono valutate in fase di invocazione del template e il risultato viene aggiunto in append allo StringBuffer contenente l'output.

Utilizzando i valori contenuti all'interno della mappa descritta in precedenza e i template JET è possibile creare dinamicamente file in base ai valori associati ai tag precedentemente descritti.

Viene sotto riportato l'esempio del template per la creazione del file web.xml utilizzato all'avvio del servlet container.

Le prime informazioni da definire sono:

- *Package*: package java che conterrà la classe java creata dal processo di traduzione
- *Class*: nome della classe generata dalla traduzione
- *Imports*: classi java utilizzate all'interno del template
- *Skeleton*: classe che contiene il metodo generate() utilizzato per la creazione dello StringBuffer contenente il corpo del file da creare

```
<%@ jet package="it.quix.framework.codegen.jet.web.xml"
       imports="java.util.*"
       class="WebXmlTemplate"
       skeleton="generator.skeleton"
%>
```

La mappa contenente i dati di configurazione utili alla creazione del file xml è chiamata args.

```
<%
Map<String, Object> args = (Map<String, Object>) argument;
String prjName = (String)args.get("prjName");
boolean useWebServices= (Boolean)args.get("useWebServices");
boolean useScheduler=(Boolean)args.get("useScheduler");
boolean isDistributed=(Boolean)args.get("isDistributed");
%>
```

È ora possibile utilizzare costrutti logici all'interno di scriptlet per gestire il flusso di creazione dello StringBuffer.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="<%=prjName %>" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

       <display-name><%=prjName %></display-name>
       <filter>
              <filter-name>struts</filter-name>
```

```
            <filter-
class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
      </filter>

<% if((useScheduler && !isDistributed)||(useScheduler && isDistributed) ){%>
      <filter>
        <filter-name>ReadJobFilter</filter-name>
        <filter-class>it.quix.framework.web.filter.ReadJobFilter</filter-
class>
    </filter>

      <filter-mapping>
        <filter-name>ReadJobFilter</filter-name>
        <url-pattern>/readJob</url-pattern>
    </filter-mapping>
<%}%>
      <filter-mapping>
            <filter-name>struts</filter-name>
            <url-pattern>/*</url-pattern>
      </filter-mapping>
      <welcome-file-list>
            <welcome-file>index.html</welcome-file>
            <welcome-file>index.htm</welcome-file>
            <welcome-file>index.jsp</welcome-file>
            <welcome-file>default.html</welcome-file>
            <welcome-file>default.htm</welcome-file>
            <welcome-file>default.jsp</welcome-file>
            <welcome-file>Index.jsp</welcome-file>
      </welcome-file-list>
      <listener>
            <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
      </listener>
      <resource-ref>
            <description>Reference to Collection database</description>
            <res-ref-name>jdbc/<%=prjName %></res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
      </resource-ref>
      <servlet>
        <servlet-name>InitServlet</servlet-name>
        <servlet-class>it.quix.framework.web.servlet.InitServlet</servlet-
class>
            <load-on-startup>5</load-on-startup>
    </servlet>
<%if((useScheduler && !isDistributed)){%>
      <servlet>
        <servlet-name>StartUpSchedulerServlet</servlet-name>
```

```
        <servlet-
class>it.quix.framework.web.servlet.StartUpSchedulerServlet</servlet-class>
        <load-on-startup>7</load-on-startup>
    </servlet>
<%}%>
</web-app>
```

Conclusioni

Il lavoro svolto ha portato alla realizzazione con successo di un modulo che permette l'esecuzione di processi schedulati fornendo la possibilità di utilizzare diverse configurazioni. L'applicazione permette all'utente finale di gestire l'inserimento e la manipolazione delle procedure presenti nel sistema attraverso un'interfaccia di gestione.

È possibile apportare miglioramenti al modulo attraverso uno studio prestazionale delle macchine utilizzate nelle varie configurazioni del sistema: ad esempio, in ambito distribuito è possibile assegnare le esecuzioni di differenti procedure su diverse macchine, ma non è possibile prevedere l'impatto prestazionale su di esse. La situazione è resa ancora più complessa dalla moltitudine di differenti operazioni riscontrabili all'interno dei differenti processi e dal tempo necessario per compierle. Un possibile ulteriore miglioramento sarebbe apportato dallo studio del tempo di latenza fra l'esecuzione di due `timerTask`, nell'esempio riportato si considera tale valore costante ma, per migliorare le prestazioni dell'intero sistema sarebbe opportuno applicare un tempo variabile, valutato in base alla durata del processo da eseguire. Questo permetterebbe di liberare in modo più rapido un worker utilizzato per l'esecuzione successiva.

Appendice

Strumenti e pattern utilizzati

Model-View-Controller

Il Model-View-Controller (MVC, Modello-Vista-Controllore) è un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented.
Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- Model: fornisce i metodi per accedere ai dati utili all'applicazione;
- View: visualizza i dati forniti dal model e si occupa di fornire uno strumento di interazione con gli utenti
- Controller riceve i comandi dell'utente e permette la modifica dello stato degli altri due componenti.

Risulta evidente una separazione fra la logica applicativa a carico del controller e del model, e l'interfaccia utente a carico del view.

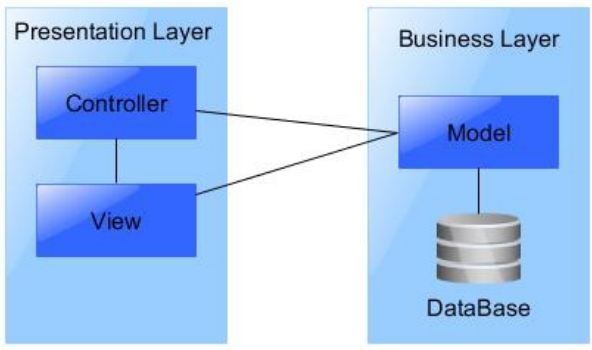

Fig. A.1: Rappresentazione del pattern MVC

Spring

Spring è un framework open source per lo sviluppo di applicazioni su piattaforma Java largamente riconosciuto quale valida alternativa al modello basato su Enterprise JavaBeans (EJB). Rispetto a quest'ultimo, il framework Spring consente una maggiore libertà fornendo allo stesso tempo un'ampia e ben documentata gamma di soluzioni semplici adatte alle problematiche piu comuni.

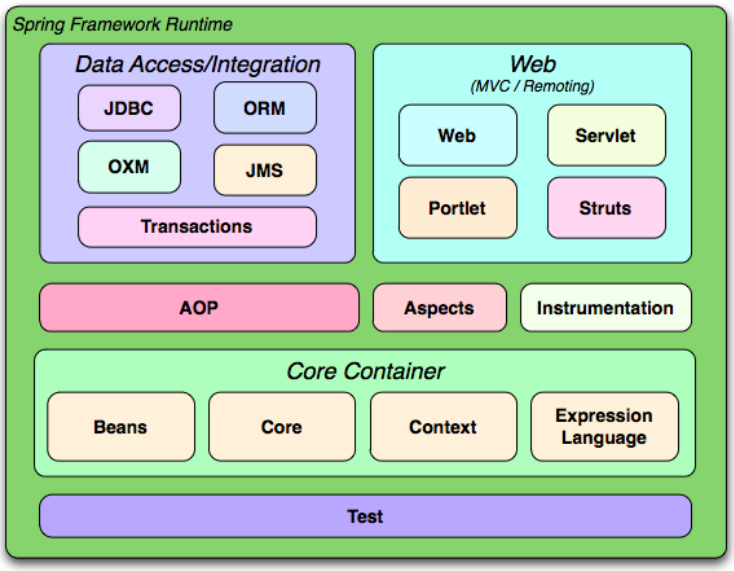

Fig. A.2: Architettura di Spring

L'architettura di Spring rappresentata in fig. A.2 si compone di diversi moduli che forniscono una vasta gamma di servizi:

- Inversion of Control (Core)
- Programmazione orientata agli aspetti (AOP)
- Accesso ai dati (DAO): per interagire con RDBMS in piattaforma Java tramite JDBC.
- Model-View-Controller
- Testing : classi di supporto per la scrittura di unit test

Inversion of Control

L'Inversion of Control (IoC) è un principio architetturale, basato sul concetto di invertire il controllo della programmazione tradizionale.
Nella programmazione tradizionale la logica del controllo di flusso è definita esplicitamente dallo sviluppatore, che si occupa tra le altre cose di tutte le operazioni di creazione, inizializzazione ed invocazione dei metodi degli oggetti. IoC invece inverte il controllo flow facendo sì che non sia lo sviluppatore a doversi preoccupare di questi aspetti, ma sarà compito del framework fornire "dall'esterno" gli elementi utili al funzionamento del programma.

La Dependency Injection è una delle tecniche con le quali si può attuare l'IoC. Essa prende il controllo su tutti gli aspetti di creazione degli oggetti e delle loro dipendenze consentendo di eliminare dal codice applicativo ogni logica di inizializzazione. Senza l'utilizzo di questa tecnica, se un oggetto necessita di usufruire di un particolare servizio, deve possedere un riferimento esplicito all'oggetto utilizzato.

Struts 2

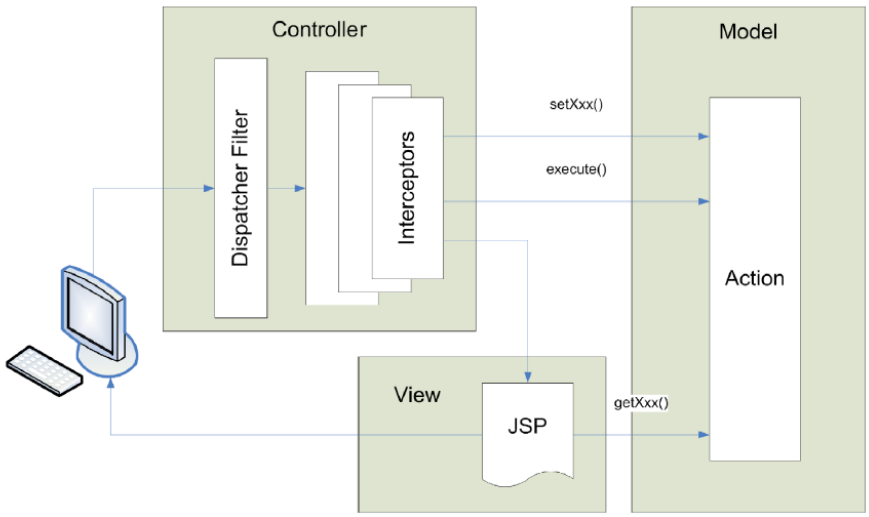

Fig. A.3: Architettura di Struts 2

Struts 2 è un framework utilizzato per la creazione di Applicazioni web java.
Il framework è stato creato per garantire l'intero ciclo di sviluppo: dalla costruzione, all'installazione su server, fino alla manutenzione dell'applicazione nel tempo.

Struts 2 estende le Java Servlet, incoraggiando gli sviluppatori all'utilizzo del pattern

Model-View-Controller:

- Model: implementa la logica applicativa, ed è costituito da un insieme di classi java, tipicamente Java Bean;
- View: insieme di pagine JSP costruite con l'ausilio di tag offerti da Struts 2.
- Controller: Il controllo viene affidato ad un altro (Dispatcher Filter) che gestisce le classi Action ed eventuali classi Helper: in base alle richieste dell'utente il controller decide quale Action eseguire per interagire con il modello. Queste informazioni sono gestibili dal file di configurazione Struts.xml. Sulla base del risultato ritornato dalla Action, il controller decide la modalità tramite la quale gestire la risposta.

Uno dei concetti nuovi di Struts 2, rispetto alla versione precedente, è rappresentato dagli Interceptor, classi stateless invocabili in modo automatico prima e dopo una Action. Di default, Struts 2, prevede un gruppo di interceptor richiamati prima di invocare qualsiasi action. Il cosiddetto stack di default, prevede 17 interceptor con il compito di offrire vari servizi. I principali sono:

- Exception: permette di mappare una particolare eccezione ad una vista;
- Prepare: permette di richiamare un metodo di inizializzazione della Action;
- Debugging: permette di attivare il debug delle viste;
- FileUpload: permette di gestire l'upload dei file;
- Validation: permette di eseguire la validazione dei dati forniti nella form.

Glossario

Application Context: interfaccia java fornita da Spring utile a fornire una configurazione all'applicazione, mentre la BeanFactory offre solamente strumenti basilari per la gestione e manipolazione dei bean; ApplicationContext fornisce funzionalità più consone ad un Framework di programmazione.

Cluster: insieme di computer connessi tra loro tramite un network.

Design pattern: è un concetto che può essere definito " una soluzione progettuale generale a un problema ricorrente". Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software, ancor prima della definizione dell'algoritmo risolutivo della parte computazionale.

High-availabity cluster: un gruppo di computer, supportante la stessa server application, rendere minima la probabilità di una indisponibilità del servizio.

Maven: software usato principalmente per la gestione di progetti Java. Permette l'automatizzazione di un'ampia varietà di compiti che gli sviluppatori software svolgono nelle loro attività quotidiane.

Monitor: è un oggetto utilizzato come blocco di mutua esclusione per i thread.

Server: componente o sottosistema informatico di elaborazione che fornisce, a livello logico e a livello fisico, un qualunque tipo di servizio ad altre componenti (tipicamente chiamate *client*, cioè "cliente") che ne fanno richiesta attraverso una rete di computer, all'interno di un sistema informatico o direttamente in locale su un computer.

Servlet Container: componente del server web che interagisce con le Java Servlet, è responsabile della gestione del ciclo di vita delle servlet, del mapping dell' URL delle servlet e fornisce un controllo sul diritto di accesso.

Thread: è una suddivisione di un processo in due o più filoni o sottoprocessi, che vengono eseguiti concorrentemente.

Trigger: procedura che viene eseguita in maniera automatica in coincidenza di un determinato evento da un DBMS.

Worker: nome specifico del thread adibito all'esecuzione di un job.

Jdbc (Java DataBase Connectivity): è un connettore per database che consente l'accesso alle basi di dati da qualsiasi programma scritto con il linguaggio di programmazione Java, indipendentemente dal tipo di DBMS utilizzato

Jsp (Java Servlet Pages): è una tecnologia di programmazione Web in Java per lo sviluppo di applicazioni Web che forniscono contenuti dinamici in formato HTML o XML. Si basa su un insieme di speciali tag con cui possono essere invocate funzioni predefinite o codice Java.

Servlet: oggetto Java che operante all'interno di un server web potenziandone le funzionalità

XML (eXtensible Markup Language): è un linguaggio di markup composto da un insieme di regole di codifica di documenti in formato leggibile dalla macchina.

Bean Factory: è un contenitore che permette di permette di configurare, instanziare e gestire Java Bean. Questi bean collaborano tra di loro e spesso risultano dipendenti gli uni dagli altri.

Java Bean: sono classi scritte secondo una particolare convenzione. Sono utilizzate per incapsulare più oggetti in un oggetto singolo (il bean), cosicché tali oggetti possano essere passati come un singolo oggetto bean invece che come multipli oggetti individuali.

Bibliografia

1. "Beginning Spring Framework 2", Thomas Van de Velde, Bruce Snyder, Christian Dupuis, Sing Li, Anne Horton, Naveen Balani 2007
2. "Struts 2 in Action", Donald J. Brown, Chad Michael Davis, Scott Stanlick, 2008
3. http://quartz-scheduler.org/documentation/quartz-1.x/quick-start
4. http://en.wikipedia.org/wiki/Computer_cluster
5. http://en.wikipedia.org/wiki/High-availability_cluster
6. http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html
7. http://it.wikipedia.org/wiki/Design_pattern