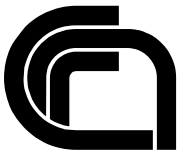


Consiglio Nazionale delle Ricerche

Http Request Scheduler

M. Abrate, C. Bacciu, F. Ronzano

IIT B4-01/2010

Nota Interna

maggio 2010

Istituto di Informatica e Telematica

Indice

Capitolo 1

Introduzione

Negli ultimi anni, la Rete si è ridisegnata attorno al concetto di offrire e richiedere *servizi web*, creando un grande sistema distribuito di cui questi servizi sono il mattone principale. Essi non solo si occupano di fornire documenti agli utenti come nel concetto tradizionale di web, ma svolgono le funzioni più diverse, dalla prenotazione dei voli all'automatizzazione e organizzazione dei processi delle aziende. Attraverso l'uso di formati e protocolli standard quali ad esempio XML o HTTP, queste unità, scritte in diversi linguaggi e operanti sulle piattaforme più disparate, sono in grado di collaborare scambiandosi richieste e risposte in un linguaggio comune.

Questa ricerca ha l'intento di calare in questo scenario un particolare mattone, un servizio web di *schedulazione delle richieste*, che può essere cioè istruito per interrogare o comandare altri servizi ad istanti di tempo predefiniti.

L'idea è nata pensando ad un modo per far sì che un database che offre un'interfaccia web service venisse aggiornato ad intervalli regolari tramite richieste HTTP.

Le capacità di cron e degli altri schedulatori desktop possono essere sfruttate nel campo web per offrire nuovi servizi quali dare la possibilità di salvare delle *history* con le prime pagine di cronaca di un quotidiano, salvare diversi fotogrammi presi a intervalli regolari da un sito che gestisce una webcam, svolgere attività di tracciamento di risorse, poter attivare e disattivare dei servizi a determinate ore.

Nel prossimo capitolo descriveremo sommariamente le principali idee e tecnologie dietro ai servizi web: il linguaggio XML, i protocolli HTTP e SOAP e lo stile REST. L'ultima parte del capitolo tratterà invece i concetti legati alla schedulazione di compiti (task) e presenterà alcuni esempi di schedulatori.

Nel capitolo 3 definiremo meglio l'idea di questo servizio web, arrivando a delineare l'architettura di un HTTP Request Scheduler (HRS), uno strumento in grado di effettuare richieste HTTP in momenti determinati dall'utente. Nel capitolo 4 confronteremo le funzionalità di tre schedulatori con quelle previste per HRS. Verrà inoltre progettato un apposito linguaggio XML per istruire il componente, descritto nel capitolo 5.

Il capitolo 6 descrive l'implementazione in Java del nostro prototipo di HRS, completo di servlet REST e SOAP e vari test ed esempi di uso attraverso interfacce.

Capitolo 2

Background

2.1 Web services

Un *web service* è un elemento funzionale disponibile in un punto qualsiasi di Internet ed accessibile tramite protocolli comuni come HTTP o SMTP. [1] Più in particolare è un sistema software progettato per consentire l'interazione tra varie macchine facenti parte di una rete [2]. Una caratteristica importante dei web service è l'interoperabilità, garantita da protocolli *platform-independent* e dall'uso di standard aperti. Secondo [1], infatti, ciò che rende i web service diversi da tecnologie analoghe come CORBA e CGI, è l'uso dell'XML al posto di linguaggi binari proprietari.

2.2 XML

XML[1] è una tecnologia per la descrizione di dati strutturati. Darne una definizione univoca sarebbe riduttivo, in quanto comprende aspetti diversi.

L'XML (eXtensible Markup Language) è un linguaggio di marcatura che deriva dall'SGML (Standard Generalized Markup Language, un metalinguaggio usato per progetti di documentazione tecnica), ma è più regolare e facile da usare. Le sue specifiche forniscono un insieme di regole per formulare testi per rappresentare dati strutturati e fa sì che la generazione, la lettura e il controllo di questa struttura siano fattibili sia tramite programmi che, eventualmente, anche da chiunque abbia a disposizione un editor di testo. Come l'HTML, fa uso di tag ed attributi, però, diversamente da come avviene in esso, il significato dei tag non è prestabilito, e l'interpretazione dipende dall'applicazione che legge il documento.

Un esempio di frammento XML può essere questo:

```
<nota per="Claudia" da="mamma">
  <titolo>Spesa</titolo>
```

[1] Vedi anche [3].

```
  <testo>Ricordati di comprare le patate.</testo>
</nota>
```

La specifica XML ha reso possibile lo sviluppo di un insieme di linguaggi modulari, combinabili tra loro ed utilizzabili per definirne di nuovi. Della famiglia XML fanno parte ad esempio XHTML (eXtensible Hyper Text Markup Language), una riformulazione dell'HTML secondo le regole XML, SVG (Scalable Vector Graphic), che serve per la descrizione di grafica a due dimensioni, XUL (XML User-interface Language), per descrivere inerfacce utente, XSL (Extensible Stylesheet Language), per scrivere i fogli di stile, XSLT (XSL Transformations), un linguaggio di trasformazione usato per riordinare, aggiungere e cancellare tag e attributi, XML Schema che aiuta a definire le strutture dei formati XML. Ci sono poi degli strumenti come XLink (XML Linking Language), che descrive una modalità standard per aggiungere collegamenti ipertestuali ad un file XML, XPointer (XML Pointer Language), una sintassi per puntare parti di un documento XML ed XQuery (XML Query Language) un linguaggio per query che usa la struttura dell'XML.

Le caratteristiche principali dell'XML sono dunque la semplicità (in contrasto con l'SGML), la leggibilità, la comprimibilità dei documenti e il fatto che possano essere trasportati su protocolli per testo, l'indipendenza dalla piattaforma, l'estendibilità (al contrario dell'HTML) e l'interoperabilità tra i linguaggi della famiglia.

2.3 HTTP

HTTP (HyperText Transfer Protocol) [4] è un protocollo dello strato di applicazione[2], ed è il principale sistema per la trasmissione di informazioni usato dalla Rete. Un grande numero di servizi web si basa su questo protocollo. È lui che regola l'interazione tra i client e i server web che gestiscono ed inviano ipertesti (principalmente HTML), e in generale dati (in particolar modo in formato binario o XML).

Le risorse accessibili via HTTP sono identificate da URI (Uniform Resource Identifier) o URL (Uniform Resource Locator) [6], cioè stringhe corte che vengono loro assegnate, e che le rendono disponibili ed indirizzabili in modo semplice in diversi schemi di nomi e metodi di accesso (FTP ed Internet Mail, oltre ad HTTP).

È il client HTTP che dà inizio ad una sessione stabilendo una connessione TCP[3] su una porta di un host remoto. Un server HTTP in ascolto su quella porta aspetta che il client invii un messaggio di richiesta; dopo aver ricevuto la richiesta, il server risponde con un messaggio di stato ed eventualmente un proprio messaggio che costituisce il corpo della risposta, che può essere un file richiesto o un'altra informazione. Il server non immagazzina nessuna informazione di stato relativa al client: il protocollo è cioè *stateless*.

[2] Vedi [5].

[3] TCP è infatti il protocollo di trasporto su cui si basa l'HTTP. Vedi [5].

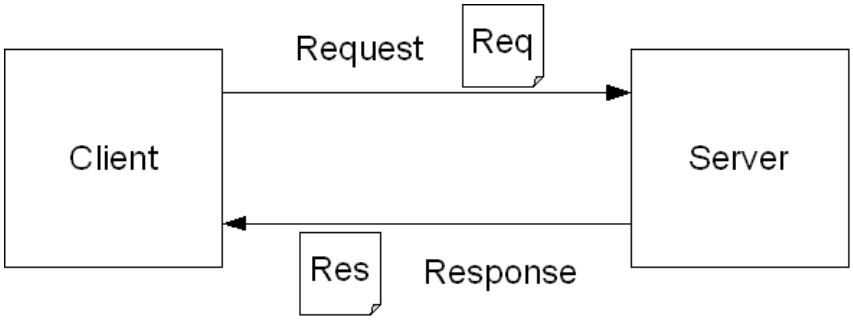


Figura 2.1: HTTP: request-response

Una *richiesta* mandata dal client al server contiene un'informazione sul *metodo*, cioè un'indicazione dello scopo della richiesta. In HTTP 1.1 esistono otto metodi: GET, PUT, POST, DELETE, HEAD, TRACE, OPTIONS, CONNECT. I più importanti ed utilizzati nel nostro contesto sono i primi 4:

- GET: ha lo scopo di ottenere una qualsiasi informazione (sotto forma di entità) sia identificata dall'URI della richiesta;
- PUT: è usato per richiedere che l'entità inclusa nella richiesta sia salvata nel punto specificato dall'URI;
- POST: è usato per richiedere che l'entità inclusa nella richiesta venga processata dalla risorsa identificata dall'URI;
- DELETE: è usato per richiedere che la risorsa identificata dall'URI venga cancellata.

Una *risposta* proveniente dal server ha una struttura simile alla richiesta, ma al posto del metodo comunica un *codice di stato* ed eventualmente un contenuto. Il codice di stato è un numero di tre cifre che indica l'esito della richiesta, ad esempio 200 per OK, 404 per NOT FOUND, 500 per INTERNAL SERVER ERROR.

La generazione e l'interpretazione dei codici di stato e dei contenuti è demandata ai protocolli di livello superiore.

2.4 SOAP

SOAP è definito dal W3C come un protocollo per lo scambio di informazioni strutturate in un ambiente decentralizzato e distribuito. Utilizza tecnologie XML per definire un framework che fornisce un costrutto per messaggi che possono essere scambiati indipendentemente dalla scelta del protocollo di strato inferiore. [7] In pratica fornisce una struttura per incapsulare e trasportare i

documenti attraverso protocolli come HTTP, FTP e SMTP (anche se in teoria è possibile utilizzare altri protocolli, come RMI o JMS).

Ci sono delle importanti regole di sintassi da usare in un messaggio SOAP:

- deve essere codificato usando l'XML;
- deve usare il namespace SOAP Envelope;
- deve usare il namespace SAOP encoding;
- non può contenere un riferimento a un DTD;
- non può contenere *processing instructions* XML.

Un esempio di messaggio SOAP è il seguente:

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
    <m:GetPrice xmlns:m="http://namespaces.server-solutions.com">
      <manufacturer>Sun</manufacturer>
      <model>Server Sun Fire T1000</model>
    </m:GetPrice>
  </soap:Body>
</soap:Envelope>
```

L'elemento `Envelope` è obbligatorio, è la radice di un messaggio SOAP e lo identifica come tale. Il suo namespace è indicato dall'attributo `xmlns:soap`. L'attributo `soap:encodingStyle` viene invece usato per definire i tipi di dato presenti nel documento, in quanto un messaggio SOAP non ha una codifica di default. Può apparire in qualsiasi elemento SOAP e si applicherà al contenuto di quell'elemento e di tutti gli elementi figli.

Anche l'elemento `Body` è obbligatorio e contiene l'effettivo messaggio che deve essere consegnato. Nell'esempio sopra viene richiesto il prezzo di un server. L'elemento `m:GetPrice` (il cui namespace è indicato dall'attributo `xmlns:m`) ed i suoi figli sono specifici per l'applicazione.

SOAP viene spesso usato per chiamate di procedura remote (RPC). Nell'esempio `GetPrice` è il metodo da chiamare e i figli di `m:GetPrice` sono i parametri del metodo. Anche se in origine SOAP era nato per facilitare le RPC sincrone, con le specifiche più recenti due punti si possono scambiare dati attraverso messaggi asincroni.

Il client e il server coinvolti in una comunicazione SOAP non hanno bisogno di essere strettamente legati tra loro, ma basta che siano in grado di costruire e decodificare gli eventuali allegati binari in base alle regole di codifica MIME, costruire e decodificare i documenti XML in base alle regole di *enveloping* e di codifica di SOAP, eseguire l'operazione di richiesta specificata nel documento SOAP [1].

I maggiori vantaggi di SOAP sono a qusto punto abbastanza chiari: è *platform-independent*, indipendente dal linguaggio e dal protocollo di trasporto. Ha però anche degli svantaggi come la mancanza di specifiche sulla sicurezza dei metodi per la *garbage collection*.

2.5 REST

REST(Representational State Transfer) è un termine coniato da Roy Thomas Fielding nella sua dissertazione per il Dottorato di Ricerca [8], in cui descrive uno stile architetturale per sistemi distribuiti.

L'acronimo REST deriva da questa sequenza di eventi, propria di un sistema che segue questo stile [9]:

1. un client richiede una risorsa ad un servizio fornendo un identificatore;
2. viene restituita la *rappresentazione* della risorsa come sequenza di byte, insieme a dei metadati che la descrivono. I metadati sono nella forma coppia nome-valore e possono essere rappresentati da header HTTP;
3. l'aver ottenuto la rappresentazione fa sì che il client *trasformi* il suo *stato*.

REST risulta essere *stateless*, pertanto deve essere sempre il client a presentare le richieste al server. L'XML è una buona scelta come linguaggio per i messaggi di richiesta e risposta. La localizzazione delle risorse si basa sull'uso di URL.

Le risorse possono essere viste in termini di HTTP come oggetti su cui si possono invocare i metodi 4 metodi fondamentali:

- GET per ottenere informazioni su una risorsa;
- PUT per aggiungere o modifica una risorsa;
- POST per inviare inforamzioni ad un processo;
- DELETE cancellare una risorsa.

In REST una risorsa è un'entità specifica e rintracciabile, non un concetto astratto, e nella progettazione di un'applicazione in stile REST è buona cosa rendere più entità possibili rintracciabili tramite GET HTTP.

Un'applicazione di tipo REST si discosta profondamente da un'altra modellata secondo lo stile RPC per la scelta delle risorse e delle operazioni. In un sistema RPC le risorse saranno gli stessi programmi o servizi, e le operazioni su di essi verranno definite appunto come invocazioni di procedure remote. Un esempio può essere questo:

```
risorsa: http://www.example.org/servizioUtenti

operazioni:
addUser();
getUser();
```

```
removeUser();
addGroup();
getGroup();
removeGroup();
```

La stessa applicazione vista secondo lo stile REST sarebbe invece organizzata così:

```
risorse:
group (http://www.example.org/servizioUtenti/nomeGruppo);
user (http://www.example.org/servizioUtenti/nomeGruppo/nomeUtente);

operazioni:
PUT
GET
DELETE
```

2.6 Scheduler

Nel mondo dell'informatica con la parola *scheduler* si identificano due componenti distinti:

- un mezzo che bilancia il carico di un sistema decidendo come distribuire le risorse a diversi richiedenti (ad esempio gli scheduler dei sistemi operativi)
- uno strumento attraverso il quale è possibile associare comandi o sequenze di comandi ad istanti di tempo, per consentire che determinate azioni siano svolte in momenti prestabiliti (ad esempio lo scheduler cron).

Dato il nostro intento, in questa relazione il termine *scheduler* sarà usato nella seconda accezione.

2.6.1 Terminologia

Esiste una terminologia abbastanza comune e condivisa per ciò che riguarda la descrizione degli scheduler e del loro funzionamento. Verranno perciò utilizzati alcuni termini considerandone il significato che risulta maggiormente compatibile con la documentazione reperibile in Rete.

Task

Un *task* è il compito che deve essere eseguito, dice cioè cosa deve essere fatto e quando. Alcuni schedulatori lo considerano un insieme di due entità, il job e il trigger.

Job

Un *job* rappresenta solo l'azione da eseguire: può essere ad esempio un comando, uno script o un programma, che lo scheduler deve lanciare.

Trigger

Un *trigger* è invece la descrizione della condizione temporale che caratterizza l'istante (o gli istanti) in cui un job deve essere lanciato. Una volta verificata la condizione, il trigger "scatta" mandando in esecuzione il job.

Dividere il concetto di job da quello di trigger dà la possibilità di lanciare job diversi con lo stesso trigger, e viceversa, che un job sia lanciato da trigger differenti.

2.6.2 Tipi di trigger

In generale (come specificato anche dallo scheduler open-source Quartz [10]) un trigger potrebbe essere creato tramite l'opportuna combinazione di direttive come le seguenti:

- scattare ad una certa ora di un determinato giorno;
- scattare ripetutamente un certo numero di volte;
- scattare ripetutamente fino ad una specifca data od ora;
- scattare ripetutamente indefinitamente;
- scattare ripetutamente con un intervallo di ritardo;
- scattare in certi giorni della settimana, del mese o dell'anno;
- non scattare in alcuni giorni elencati in un calendario (ad esempio giorni festivi)

Quindi si possono identificare quattro tipi di trigger:

1. Il "one-shot trigger", che scatta una volta sola;
2. Il trigger ripetuto ad intervalli regolari;
3. Il trigger basato su espressioni di tipo "crontab";
4. Il trigger basato sulla definizione di calendari.

2.6.3 Esempi di scheduler

Scheduled Tasks di Windows XP

La funzione Schedued Tasks di Windows XP ("Operazioni PianificatÈ' nella versione italiana) è un semplice esempio di schedulatore che permette di pianificare l'esecuzione di un programma o uno script ad un'ora prestabilita. Si basa su un *wizard* che consente la selezione del programma e dei parametri temporali necessari.

Cron

Cron [11] è un demone presente nei sistemi *UNIX-like* (Linux, BSD, OSX, ecc.) per l'esecuzione di comandi o script a tempi prefissati. La specifica del comando da eseguire e del momento in cui eseguirlo fatta in tabelle di tipo *crontab*, scritte con una sintassi particolare, relative ad ogni utente. Ogni minuto il demone esamina tutte le tabelle crontab e controlla se ci sono comandi da eseguire in quel minuto. Durante l'esecuzione l'output del comando è spedito all'utente proprietario della tabella o ad un altro specificato in essa.

Cron è forse il sistema di scheduling più famoso, ed è per questo che la sintassi per la specifica di tabelle crontab è supportata da molti altri schedulatori.

Capitolo 3

HTTP Request Scheduler (HRS)

In questo capitolo descriveremo l'architettura di un HTTP Request Scheduler (HRS), uno strumento in grado di effettuare richieste HTTP in momenti determinati dall'utente.

Il funzionamento di base di HRS è quello di tipo *request/response* dell'HTTP, diviso però in tre fasi successive, senza che sia noto a priori quanto tempo trascorre tra una fase e l'altra.

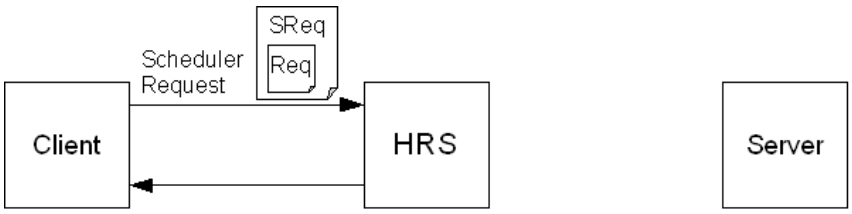


Figura 3.1: Fase di configurazione delle richieste

Nella prima fase (configurazione) il client invia ad HRS una richiesta (SReq) espressa in un linguaggio XML di definizione di task, il quale incapsula la rappresentazione della richiesta HTTP (Req) da inviare al server di destinazione, i parametri (uri del server e metodo HTTP con cui contattarlo) e i trigger. HRS processa la richiesta (SReq), aggiunge tra quelli presenti il task ricevuto, dopo di che invia al client una risposta (SRes) che lo avvisa dell'avvenuta schedulazione o di eventuali errori.

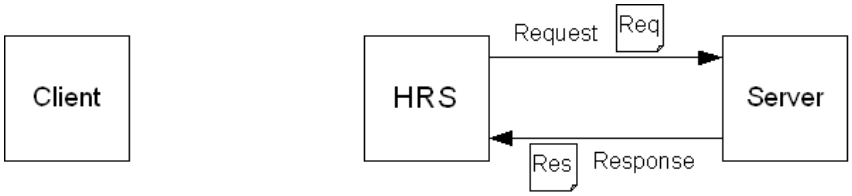


Figura 3.2: Fase di esecuzione delle richieste

È HRS che dà inizio alla seconda fase, quella di esecuzione delle richieste: allo scattare di un trigger, invia al server di destinazione la richiesta HTTP associata (Req) e riceve la risposta (Resp). Questa fase ha luogo indipendentemente dal client, e si ripete seguendo le direttive temporali imposte dalla richiesta ottenuta nella prima fase. Ogni risposta sarà immagazzinata in un database, in modo da poter successivamente accedere a tutta la sequenza di messaggi ricevuti nei diversi istanti.

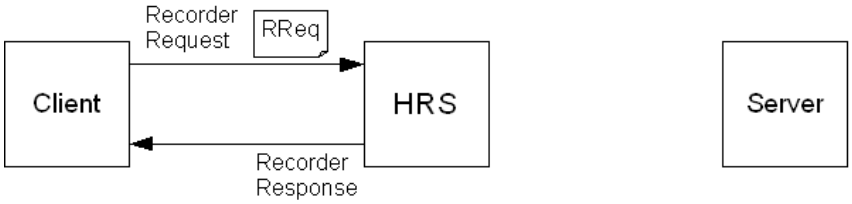


Figura 3.3: Fase di recupero degli esiti delle richieste

Infine Il client, dopo un intervallo di tempo indefinito, interroga HRS (RReq) sull'esito delle schedulazioni (fase di recupero degli esiti delle richieste) e riceve una risposta (RRes) che incapsula la lista delle risposte (Res) ottenute dal server, indicando i link da seguire per accedere al database contenente l'history.

La fase di configurazione può ripetersi ogni volta che il client vuole modificare le direttive dello schedulatore. Sarà dunque messo in grado di:

- creare una lista di task, ciascuno con un job e una lista di trigger;
- visualizzare i task definiti;
- aggiornare un task, aggiungendo o rimuovendo trigger o sostituendo il job;
- cancellare un task;

3.1 Possibili casi d'uso

Un servizio di questo genere può trovare applicazione in contesti quali architetture distribuite in cui vi sia bisogno di aggiornare delle risorse a tempi prestabiliti o in maniera costante.

Alcuni casi d'uso che si possono immaginare sono questi:

- un gioco on-line in cui l'ambiente deve evolversi indipendentemente dal fatto che i giocatori siano collegati o meno: lo scheduler aggiornerà costantemente il database con i dati sull'ambiente;
- un client interessato a tenere traccia dello stato di una risorsa: lo scheduler si occuperà di richiedere le informazioni per conto del client, con la frequenza da lui desiderata, rendendole disponibili in un secondo momento;
- un server vuole presentare una pagina dinamica aggiornata costantemente: lo scheduler apporterà volta per volta le modifiche e il client, magari tramite Ajax[1], si occuperà di fare *polling* mostrando all'utente una pagina in continua evoluzione;
- una risorsa deve essere disponibile solo per un determinato periodo di tempo e poi deve essere cancellata: lo scheduler potrà sia renderla disponibile sia eliminarla in un determinato momento. Questo può essere ad esempio il caso in cui su una *message-board* su Internet vengano pubblicati messaggi privati; un utente potrebbe volere, magari per questioni di sicurezza, che i suoi messaggi siano cancellati dopo alcuni minuti dalla loro lettura.

[1] Vedi [12]

Capitolo 4

Stato dell'arte

Tra gli scheduler non commerciali di cui si possono facilmente trovare informazioni in Rete ci siamo soffermati su tre in particolare, analizzandoli e cercando di capire se si adattavano alle nostre necessità: Quartz, JobScheduler e Jcrontab.

4.1 Quartz

Quartz[1] è un framework per job-scheduling scritto in Java. Include caratteristiche come un supporto per database, per espressioni di tipo cron, job precostruiti per EJB, e altre. Il compito da eseguire deve essere contenuto in una classe Java che implementa l'interfaccia `Job`. La logica vera e propria sarà contenuta in un metodo `execute()` del job. Quando lo Scheduler determina che è giunto il momento, chiama il metodo `execute()` sulla classe `Job`. I job sono considerati divisi dai trigger. Una volta che i job sono stati schedulati, la loro gestione è fatta attraverso un `JobStore`. Quartz fornisce due tipi principali di JobStore. Il primo tipo è il RAMJobStore, che utilizza la RAM per mantenere le informazioni di scheduling. Questo tipo è il più facile da configuare e far partire, però, dato che le informazioni sono mantenute nella memoria assegnata alla Java Virtual Machine, quando l'applicazione viene fermata tutte le informazioni vengono perse. Il secondo tipo è il JDBC Jobstore, che sfrutta un driver JDBC per salvare le informazioni di scheduling in un database relazionale. Oltre che tramite l'interfaccia `Scheduler`, i job possono essere schedulati anche in maniera dichiarativa. Il framework contiene infatti un plugin che all'avvio di un'applicazione Quartz legge un file XML che contiene informazioni sui job da svolgere e sui trigger. Tutti i job descritti nel file sono aggiunti allo Scheduler, assieme ai trigger loro associati. Un'altra caratteristica di Quartz sono i Listener, cioè delle classi (che possono contenere una qualsiasi logica) che vengono richiamate dal framework quando si verifica un determinato evento. Sono inclusi Listener per lo Scheduler, per i Job, per i Trigger.

[1] Link: `www.opensymphony.com/quartz`

4.2 Job scheduler

Job Scheduler[2] è un programma batch che opera come demone e può essere controllato attraverso l'interfaccia grafica del web server integrato. Usa un file XML per la configurazione di file eseguibili, shell script e per la scelta dei tempi e della frequenza del lancio dei job. Può processare compiti in maniera sequenziale o parallela. I job possono essere organizzati in catene per assicurare l'esecuzione sequenziale e, per motivi di scalabilità, possono essere eseguiti in istanze multiple.

Un job può essere lanciato in diversi modi: attraverso il monitoraggio di directory, cioè quando avvengono cambiamenti in una directory come l'aggiunta o la cancellazione di file o sottodirectory; quando arriva un determinato momento: ora del giorno, settimana, giorno del mese, fine di un periodo; a controllo di programma, usando le API (per Java, Javascript, VBScript, Perl); attraverso notifiche in formato TCP e UDP; in modo manuale usando la GUI; in modo manuale da linea di comando. Alcune aratteristiche importanti di Job Scheduler sono queste[3]:

- organizzazione di job, task e catene di job;
- possibilità di assegnare una priorità ai job;
- esecuzione di procedure su database;
- schedulazione di job per MySQL;
- Graphical user interface;
- operazioni a riga di comando;

4.3 Jcrontab

Jcrontab[4] è uno scheduler scritto in Java che: Fornisce un sistema per eseguire classi, thread, EJB, metodi, programmi in un determinato momento. Immagazzina e legge tabelle di tipo crontab da file e database e può essere esteso tramite l'interfaccia `DataSource` per leggere file crontab da qualsiasi luogo. Può essere integrato in *application server*. Può essere usato anche come applicazione *stand-alone* al posto di Cron.

4.4 Conclusioni

Job Scheduler sembra essere molto potente, ma per i nostri scopi non è sembrato adatto a causa della sua complessità e per il fatto che la versione open-source è rilasciata sotto licenza GPL, ma offre supporto tecnico solo a pagamento.

[2]Link: `www.sos-berlin.com/modules/cjaycontent/index.php?d=62&page=osource_scheduler_introduction_en.htm`

[3]Sul sito è presente una documentazione piuttosto ricca, sia in formato HTML che PDF.

[4]Link: `www.jcrontab.org/index.shtml`

Jcrontab fondamentalmente è l'equivalente Java di cron, per cui è risultato molto limitato.

Quartz si è rivelato più un framework che una applicazione stand-alone.

Dall'analisi è risultato quindi che nessuno dei 3 schedulatori si comporta esattamente come previsto per un HTTP Request Scheduler. Il framework Quartz però risulta essere un ottimo punto di partenza per sviluppare un prototipo. Infatti:

- fornisce un'interfaccia molto versatile e potente per lo scheduling;
- viene distribuito sotto licenza Apache [13];
- supporta la divisione tra job e trigger;
- ha una servlet[5] di inizializzazione, utile per lavorare in ambiente J2EE.

Nel capitolo 5 illustreremo un apposito linguaggio XML per istruire l'HRS, mentre nel capitolo 6 useremo Quartz per implementare un HTTP Request Scheduler in Java, e lo renderemo disponibile come servizio creando delle servlet per SOAP e REST.

[5] Vedi [14]

Capitolo 5

HRS Markup Language

Il linguaggio HRSML si propone di rappresentare per mezzo dell'XML i tre concetti fondamentali della schedulazione: task, job e trigger. Attraverso di esso sarà possibile istruire lo schedulatore o visualizzare il suo stato.

5.1 Presentazione

5.1.1 `<task>`

Consideriamo subito un semplice esempio:

```
<task id="task1">
  <job uri="http://www.web.ing.unipi.it" http-method="GET" />
  <triggers>
    <trigger name="tuno">
      <at>
        <relative to="now">10000</relative>
      </at>
    </trigger>
  </triggers>
</task>
```

Configurando lo scheduler per eseguire questo task, esso eseguirà una richiesta HTTP GET all'indirizzo `http://www.web.ing.unipi.it` 10 secondi dopo l'avvenuta configurazione. L'elemento `<job>` descrive quindi una richiesta HTTP, specificando tra i suoi attributi l'uri da invocare e il metodo. Il tag `<triggers>` ospita invece le definizioni degli istanti temporali in cui il job dovrà essere eseguito, ognuna contenuta in un elemento `<trigger>` dotato dell'attributo `name`. In questo esempio è presente un unico trigger, di nome '`tuno`'. L'attributo `id` del task consente di riferirsi ad esso all'interno dello schedulatore.

5.1.2 <job>

Come già osservato, un job ha due parametri, 'uri' e 'http-method' (che può valere GET, POST, PUT o DELETE). A seconda del metodo scelto risulta però necessario specificare un contenuto per la richiesta HTTP (è il caso dei metodi POST e PUT). Questo corpo può essere inserito all'interno dell'elemento <job> [1], come in questo esempio in cui viene fatta una POST di una direttiva xupdate [15]:

```
<job uri="http://localhost:8080/exist/servlet/db/universe/
  universe.xml" http-method="POST"> <xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:update select="//settlement/resources/@solid">
      <xupdate:value-of select="//settlement/resources/@solid+10" />
    </xupdate:update>
  </xupdate:modifications>
</job>
```

5.1.3 <trigger>

L'attributo name è necessario per poter manipolare dei trigger già presenti in un task. Ad esempio se si volesse cancellare il secondo trigger da questo task di esempio, basterebbe mandare un comando di cancellazione allo schedulatore indicando il task 'task1' e il trigger 'tdue'.

```
<task id="task1">
  <job uri="http://www.web.ing.unipi.it" http-method="GET" />
  <triggers>
    <trigger name="tuno">
      <at>
        <relative to="now">10000</relative>
      </at>
    </trigger>
    <trigger name="tdue">
      <at>
        <relative to="now">30000</relative>
      </at>
    </trigger>
  </triggers>
</task>
```

I nomi dei trigger seguono una precisa sintassi: devono cominciare con una lettera e possono continuare con zero o più caratteri alfanumerici o underscore.

[1] HRSML supporta solo contenuti di tipo XML

Tipi di trigger: `<at>` e `<repeat>`

In questa prima stesura, l'HRSML supporta due tipi di trigger: 'at' e 'repeat'.

Il primo tipo è un one-shot trigger [2] che scatta nel momento indicato tramite il tag `<relative>` o il tag `<absolute>`:

```
<trigger name="tuno">
  <at>
    <relative to="now">10000</relative>
  </at>
</trigger>

<trigger name="tdue">
  <at>
    <absolute>2001-12-31T12:00:00</absolute>
  </at>
</trigger>
```

Attraverso l'uso di `<relative>` è possibile specificare i millisecondi di ritardo con cui lo scheduler dovrà eseguire il job dal momento in cui riceverà la specifica del trigger. Il valore '`now`' dell'attributo `to` è obbligatorio per i trigger di tipo 'at'. `<absolute>` invece specifica una data e un'ora esatte in formato `xsd:dateTime` [3], nell'esempio il 31 Dicembre 2001, a mezzogiorno.

Il trigger 'repeat' è utile invece per ripetere l'esecuzione di job a intervalli regolari:

```
<trigger name="tuno">
  <repeat>
    <begin>
      <absolute>2001-12-31T12:00:00</absolute>
    </begin>
    <every>1000</every>
    <count>4</count>
    <end>
      <relative to="begin">10000</relative>
    </end>
  </repeat>
</trigger>
```

Gli elementi `<begin>` e `<end>` hanno una struttura analoga a quella di `<at>`, e identificano rispettivamente l'istante da cui cominciare a ripetere il job e l'istante nel quale cancellare il trigger. L'unica differenza con `<at>` sta nel fatto che nell'elemento `<end>` l'attributo `to` di `<relative>` può assumere oltre al valore '`now`' anche '`begin`'. In questo secondo caso, come nell'esempio, l'istante finale sarà calcolato sommando il tempo specificato da `<end>` a quello iniziale. Il

[2] si veda il paragrafo 2.6.2

[3] xsd:dateTime è un tipo predefinito dell'XML Schema, si veda [16]

trigger comincerà a scattare il 31 Dicembre 2001 a mezzogiorno. La ripetizione avverrà ogni secondo, come indicato da `<every>`, e il trigger sarà cancellato dopo essere scattato 4 volte (direttiva `<count>`) o al raggiungimento dell'istante finale. Siccome vale la condizione più restrittiva, il job associato verrà eseguito 4 volte.

Ognuno dei quattro campi può essere omesso, a patto che i restanti diano sufficienti informazioni. Omettere `<begin>` significa far cominciare a scattare il trigger appena possibile; la mancanza di `<end>` non pone nessun limite di tempo alla vita del trigger; l'assenza di `<count>` non limita gli scatti ad un numero preciso di volte. `<every>` può essere omesso solo nel caso in cui siano definiti almeno `<count>` e `<end>`, in modo tale da poter facilmente calcolare il periodo di ripetizione dividendo il tempo di vita del trigger in parti uguali.

5.1.4 `<tasks>`

È stata prevista anche la possibilità di descrivere una lista di task con il tag `<tasks>`, che viene così ad essere la classica radice di un documento HRSML (anche se altre possibili radici sono `<task>`, `<job>`, `<triggers>` e `<trigger>`):

```
<?xml version="1.0" encoding="UTF-8"?>
<tasks xmlns="http://www.example.org">
  <task id="task1">
    <job uri="http://www.web.ing.unipi.it" http-method="GET" />
    <triggers>
      <trigger name="tuno">
        <at>
          <relative to="now">10000</relative>
        </at>
      </trigger>
      <trigger name="tdue">
        <repeat>
          <begin>
            <absolute>2001-12-31T12:00:00</absolute>
          </begin>
          <every>1000</every>
          <count>4</count>
          <end>
            <relative to="begin">10000</relative>
          </end>
        </repeat>
      </trigger>
    </triggers>
  </task>
  <task id="task2">
    <job uri="http://www.somesite.org" http-method="POST">
  ...
```

```
    </job>
    <triggers>
      <trigger name="tuno">
        <repeat>
          <every>10000</every>
        </repeat>
      </trigger>
    </triggers>
  </task>
</tasks>
```

5.2 Esempi

5.2.1 Job

Descrivere un job che esegue una GET:

```
<job uri="http://www.anysite.com" http-method="GET" />
```

Descrivere un job che esegue una PUT di un documento XHTML:

```
<job uri="http://www.anysite.com" http-method="PUT">
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>title</title></head>
    <body>
      <p>
        Text.
      </p>
    </body>
  </html>
</job>
```

5.2.2 Trigger

Definire un trigger che scatta una volta, tra 10 secondi:

```
<trigger name="tuno">
  <at>
    <relative to="now">10000</relative>
  </at>
</trigger>
```

Definire un trigger che scatta immediatamente e si ripete ogni 60 secondi, per sempre:

```
<trigger name="tdue">
  <repeat>
```

```
    <every>60000</every>
  </repeat>
</trigger>
```

Definire un trigger che scatta immediatamente e si ripete ogni 10 secondi fino a 40 secondi da ora:

```
<trigger name="ttre">
  <repeat>
    <every>10000</every>
    <end>
      <relative to="now">40000</relative>
    </end>
  </repeat>
</trigger>
```

Definire un trigger che scatta il 17 Marzo del 2006 alle 10:30 e si ripete per 6 volte, con 30 secondi di intervallo tra ogni scatto:

```
<trigger name="tquattro">
  <repeat>
    <begin>
      <absolute>2006-03-17T10:30:00</absolute>
    </begin>
    <every>30000</every>
    <count>6</count>
  </repeat>
</trigger>
```

Definire un trigger che scatta il 31 Dicembre del 2001 alle 12:00, termina dopo un minuto, e si ripete per 10 volte:

```
<trigger name="tcinque">
  <repeat>
    <begin>
      <absolute>2001-12-31T12:00:00</absolute>
    </begin>
    <count>10</count>
    <end>
      <relative to="begin">60000</relative>
    </end>
  </repeat>
</trigger>
```

Definire un trigger che scatta tra 30 secondi da ora (con un intervallo di 30 secondi tra uno scatto e l'altro), si ripete per 10 volte o finisce dopo un minuto dall'inizio:

```
<trigger name="tsei">
  <repeat>
    <begin>
      <relative to="now">30000</relative>
    </begin>
    <every>30000</every>
    <count>10</count>
    <end>
      <relative to="begin">60000</relative>
    </end>
  </repeat>
</trigger>
```

Vale la condizione più restrittiva (in questo caso l'`<end>`), e il trigger scatterà due volte.

5.2.3 Task

Definire un task con più trigger:

```
<task id="idvalue">
  <job uri="http://www.somesite.org" http-method="POST">
    ...
  </job>
  <triggers>
    <trigger name="tuno">
      <repeat>
        <begin>
          <relative to="now">2000</relative>
        </begin>
        <every>5000</every>
      </repeat>
    </trigger>
    <trigger name="tdue">
      <repeat>
        <every>5000</every>
      </repeat>
    </trigger>
  </triggers>
</task>
```

5.3 XML Schema

È stato definito un semplice XML Schema per HRSML, di cui riportiamo il codice e il diagramma.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.iit.cnr.it/xmlgroup/hrs"
xmlns:hrs="http://www.iit.cnr.it/xmlgroup/hrs">

    <element name="tasks">
      <complexType>
        <sequence>
          <element ref="hrs:task" maxOccurs="unbounded" minOccurs="0"/>
        </sequence>
      </complexType>
    </element>

    <complexType name="DateTimeType">
      <choice>
        <element ref="hrs:absolute"/>
        <element ref="hrs:relative"/>
      </choice>
    </complexType>

    <simpleType name="HttpMethodType">
      <restriction base="string">
        <enumeration value="GET"/>
        <enumeration value="POST"/>
        <enumeration value="PUT"/>
        <enumeration value="DELETE"/>
      </restriction>
    </simpleType>

    <element name="task">
      <complexType>
        <sequence>
          <element ref="hrs:job"/>
          <element ref="hrs:triggers" maxOccurs="1" minOccurs="0"/>
        </sequence>
        <attribute name="id" type="ID" use="optional"/>
      </complexType>
    </element>

    <element name="job">
      <complexType>
        <sequence maxOccurs="unbounded" minOccurs="0">
          <any namespace="##any" processContents="lax"/>
        </sequence>
        <attribute name="uri" type="anyURI" use="required"/>
        <attribute name="http-method" type="hrs:HttpMethodType"
```

```
      use="required"/>
  </complexType>
</element>

<element name="trigger">
  <complexType>
    <choice>
      <element ref="hrs:at"/>
      <element ref="hrs:repeat"/>
    </choice>
    <attribute name="name" type="hrs:nameType" use="optional"/>
  </complexType>
</element>

<element name="at" type="hrs:DateTimeType"/>

<element name="repeat">
  <complexType>
    <sequence>
      <element ref="hrs:begin" maxOccurs="1" minOccurs="0"/>
      <choice>
        <sequence>
          <element ref="hrs:every"/>
          <element ref="hrs:count" maxOccurs="1" minOccurs="0"/>
          <element ref="hrs:end" maxOccurs="1" minOccurs="0"/>
        </sequence>
        <sequence>
          <element ref="hrs:count"/>
          <element ref="hrs:end"/>
        </sequence>
      </choice>
    </sequence>
  </complexType>
</element>

<element name="begin" type="hrs:DateTimeType"/>

<element name="every" type="long"/>

<element name="count" type="positiveInteger"/>

<element name="end" type="hrs:DateTimeType"/>

<element name="absolute" type="dateTime"/>

<element name="relative" type="hrs:relativeType"/>
```

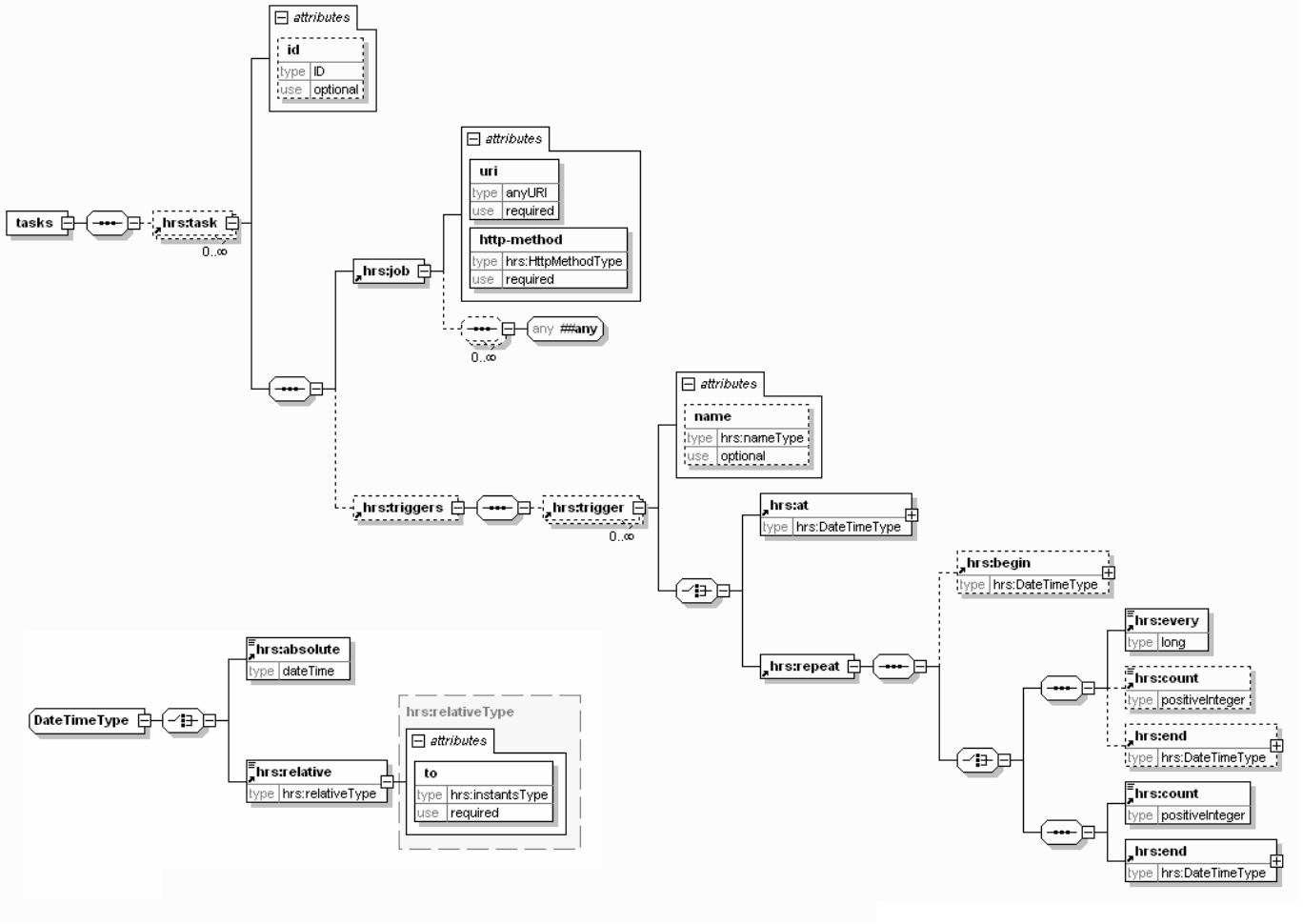
```
    <simpleType name="nameType">
      <restriction base="string">
        <pattern value="[a-zA-Z][\w_]*"/>
      </restriction>
    </simpleType>

    <simpleType name="instantsType">
      <restriction base="string">
        <enumeration value="now"/>
        <enumeration value="begin"/>
      </restriction>
    </simpleType>

    <complexType name="relativeType">
      <simpleContent>
        <extension base="long">
          <attribute name="to" type="hrs:instantsType" use="required"/>
        </extension>
      </simpleContent>
    </complexType>

    <element name="triggers">
      <complexType>
        <sequence>
          <element ref="hrs:trigger" maxOccurs="unbounded" minOccurs="0"/>
        </sequence>
      </complexType>
    </element>

</schema>
```


attributes
id
type ID
use optional
tasks
hrs:task
0..∞
hrs:job
attributes
uri
type anyURI
use required
http-method
type hrs:HttpMethodType
use required
any ##any
0..∞
attributes
name
type hrs:nameType
use optional
hrs:triggers
hrs:trigger
0..∞
hrs:at
type hrs:DateTimeType
hrs:repeat
hrs:begin
type hrs:DateTimeType
hrs:every
type long
hrs:count
type positiveInteger
hrs:end
type hrs:DateTimeType
hrs:count
type positiveInteger
hrs:end
type hrs:DateTimeType
DateTimeType
hrs:absolute
type dateTime
hrs:relative
type hrs:relativeType
hrs:relativeType
attributes
to
type hrs:instantsType
use required

Capitolo 6

Implementazione

6.1 Intenzioni

Il prototipo (ci si riferirà ad esso come HRS) si propone di realizzare un sottoinsieme delle funzionalità di HTTP Request Scheduler. In particolare HRS si potrà istruire per:

- schedulare uno o più nuovi task;
- sostituire il job di un task già in schedulazione con un nuovo job;
- rimuovere o aggiungere trigger ai task in schedulazione;
- rimuovere un task.

Queste direttive saranno espresse per mezzo dell'HRSML, e inviate ad HRS tramite HTTP.

La terza fase dello scambio di messaggi tra il client e HRS non sarà implementata. Quando i job configurati saranno eseguiti, la risposta ottenuta da HRS sarà stampata a video, e il client non potrà recuperarla.

6.2 Servlet e J2EE

Per poter essere configurato attraverso HTTP, HRS deve necessariamente contenere una parte server. In questa implementazione si è scelto di sviluppare delle servlet Java[1], una con un'interfaccia SOAP (`SOAPServlet`) e l'altra sullo stile REST (`RESTServlet`). Il servlet container scelto per il testing è Jetty[2].

[1] Vedi [14]
[2] Vedi [17]

6.3 Quartz

La schedulazione vera e propria sarà affidata a Quartz. Attraverso la servlet inclusa nel framework verrà creata un'istanza dell'oggetto `org.quartz.Scheduler` all'avvio del servlet container, i cui metodi saranno invocabili dalle nuove servlet attraverso una piccola libreria definita in 6.6.

6.3.1 Creare un task per Quartz

Quartz supporta il concetto di task definendo direttamente i job e associando ad esso dei trigger.

6.3.2 Creare un job per Quartz

Quartz definisce i job come classi che realizzano l'interfaccia `org.quartz.Job`. Ogni volta che lo schedulatore decide di lanciare un job, chiamerà il suo metodo `execute`, il quale deve contenere la logica del job. I parametri necessari per la sua esecuzione devono essere stati inseriti precedentemente in un oggetto associato al job di tipo `JobDataMap`, che implementa `java.util.Map`. Come in tutte le `Map`, l'oggetto (in questo caso stiamo usando delle stringhe) sarà accessibile attraverso una *chiave*, in questo caso anch'essa di tipo `String`. Una volta definito il job, sarà sufficiente inserirlo nello scheduler attraverso il metodo `addJob`.

6.3.3 Creare un trigger per Quartz

Quartz supporta diversi tipi di trigger, di cui noi useremo solo il più semplice: `SimpleTrigger`. Un `SimpleTrigger` si comporta esattamente come `<repeat>` dell'HRSML. Infatti, i parametri necessari per definirne uno sono:

- la data e l'ora di inizio, con un oggetto `Date`;
- il tempo tra uno scatto e l'altro in millisecondi;
- il numero di ripetizioni, che può essere indefinito;
- la data e l'ora di fine (opzionale), ancora con un oggetto `Date`.

Questo trigger si presta però anche ad implementare il nostro `<at>`: sarà sufficiente impostare il momento di inizio, un periodo qualsiasi, un numero di ripetizioni pari ad uno[3] ed omettere la fine. Una volta creato un Trigger, sarà possibile associarlo ad un job attraverso lo scheduler, con il metodo `scheduleJob`.

[3] In realtà Quartz considera questo numero pari a zero in questo caso, inteso come nessuna ripetizione

6.4 HTTPJob

Allo scattare di un trigger, HRS dovrà caricare i parametri di una richiesta HTTP dal job associato e comportarsi da client. Per questo scopo è stata sfruttata la libreria `org.apache.commons.httpclient`. HTTPJob è una classe che realizza l'interfaccia `org.quartz.Job`, in modo che il suo metodo `execute` sia invocabile dallo scheduler. All'interno del metodo si procede come segue:

1. si recuperano i parametri dalla `JobDataMap` del job il cui trigger è scattato (uri, metodo HTTP ed eventuale contenuto);
2. si esegue una nuova richiesta HTTP utilizzando i parametri;
3. si stampa il risultato a video.

6.5 Altre librerie

Per poter manipolare documenti XML come HRSML o SOAP è stata utilizzata la libreria JDOM [18], che offre un'interfaccia DOM all'XML. La corrispondenza tra il tipo Java `Date` e `xsd:DateTime` è stata realizzata con la libreria xBean [19]. Sono state ovviamente incluse le dipendenze di tutte le librerie utilizzate.

6.6 HRSlib

È stata definita una piccola libreria per lavorare con lo scheduler Quartz. In pratica si tratta di mappare le strutture dell'HRSML in classi Java, e di fare in modo che esse si interfaccino con lo scheduler. Troviamo infatti le interfacce Task e Trigger e la classe Job. Inoltre, due classi che implementano l'interfaccia `java.util.Map` sono state previste per rappresentare gli elementi `<tasks>` e `<triggers>`.

Per poter lavorare con queste classi, il programma ospitante (nel nostro caso una delle servlet) istanzia un oggetto di tipo `HttpRequestSchedulerQuartzImpl` passando come parametro l'istanza dello scheduler Quartz. La classe funge da interprete, realizzando attraverso l'uso di Quartz i metodi dell'interfaccia `HttpRequestScheduler` che implementa:

- `void addTask(Task t, String taskId);`
- `Task getTask(String taskId);`
- `void removeTask(String taskId).`

Il secondo metodo in realtà ritorna un riferimento a `TaskQuartzImpl`, tramite il quale è possibile modificare lo stato dell'effettivo task in schedulazione utilizzando i metodi dell'interfaccia `Task` che implementa:

- `void replaceJob(Job j);`

- `void addTrigger(Trigger t, String tName);`
- `void addTriggers(Triggers ts);`
- `void removeTrigger(String tName);`
- `Job getJob();`
- `Trigger getTrigger(String tName);`
- `Triggers getAllTriggers()`.

`TaskInstance`, `Job`, `AtTrigger` e `RepeatTrigger` sono invece rappresentazioni non legate a Quartz, e possono essere utilizzate per creare istanze di questi oggetti al di fuori dello schedulatore.

Come esempio, vediamo come un programma ospitante potrebbe schedulare un nuovo task:

```
HttpRequestScheduler hrs = new HttpRequestSchedulerQuartzImpl(quartzScheduler);
Job j = new Job("http://www.example.org", "GET", null);
Trigger tr = new AtTrigger(new Date());
Task t = new TaskInstance(j, tr, "tuno");

hrs.addTask(t, "task1");
```

e come potrebbe accedere successivamente al task già in schedulazione per, ad esempio, rimuovere il trigger:

```
Task task1 = hrs.getTask("task1");
task1.removeTrigger("tuno");
```

6.7 La servlet REST

Per realizzare un servizio in stile REST, bisogna affrontare diversi passi [9]:

1. determinare i tipi di risorsa richiesti;
2. determinare le operazioni richieste (GET, POST, PUT o DELETE) per ogni tipo di risorsa;
3. assegnare un identificatore (URI) ad ogni risorsa;
4. creare delle classi per modellare ogni tipo di risorsa;
5. scrivere degli schemi per descrivere le richieste e le risposte;
6. creare la servlet vera e propria;
7. creare un *deployment descriptor* che leghi la servlet ai tipi di risorsa;
8. creare un file WAR che contenga il deployment descriptor, la servlet e le librerie utilizzate;

9. inserire il file WAR in un servlet container.

Per i primi tre punti, le scelte fatte sono riassunte nella seguente tabella:

Risorsa	Operazioni	URI
task	PUT, DELETE	`/rest/tasks/`*taskId*
job	PUT	`/rest/tasks/`*taskId*`/job`
trigger	PUT, DELETE	`/rest/tasks/`*taskId*`/triggers/`*triggerName*
tasks	POST	`/rest/tasks`
triggers	POST	`/rest/tasks/`*taskId*`/triggers`

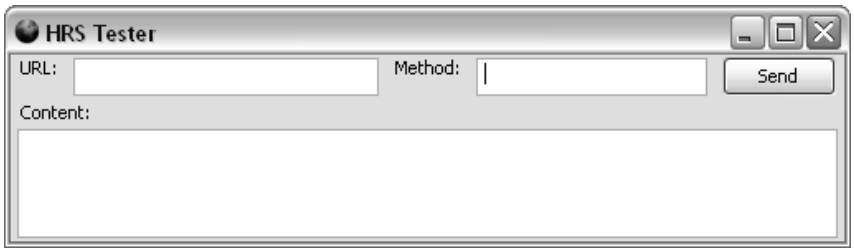
Il messaggio HRSML associato ad ogni richiesta PUT o POST deve avere la radice corrispondente al tipo di risorsa trattato. Con il metodo PUT è possibile aggiungere una nuova risorsa, a cui sarà assegnato l'URI indicato. Tramite DELETE può esssere invece cancellata. Il metodo POST è stato scelto negli ultimi due casi, dove l'identificatore della risorsa effettivamente creata non è quello indicato dall'URI, ma piuttosto suo discendente.

La servlet riconosce la risorsa e i metodi richiesti, estrae le informazioni di configurazione dall'eventuale corpo del messaggio ed utilizza le classi di HRSlib per configurare lo scheduler Quartz.

Per i punti successivi si rimanda a 6.6 per le classi e a 5.2.3 per lo schema delle richieste. Gli ultimi tre punti sono comuni ad entrambe le servlet. In 6.9 si riporterà il contenuto del deployment descriptor. I file WAR sono stati generati attraverso Eclipse[4], l'IDE utilizzato per sviluppare il progetto.

6.7.1 Testing

Sul lato server, la servlet REST di HRS è stata provata all'interno del servlet container Jetty, su piattaforma Windows XP. L'interfaccia usata sul client è una semplice pagina scritta in XUL per l'ambiente Mozilla XULRunner [5], che si basa sullo stesso motore dei browser Firefox e Mozilla suite. La pagina permette di specificare l'indirizzo di HRS, il metodo HTTP da utilizzare per la richiesta ed, opzionalmente, il contenuto espresso in HRSML. Con l'utilizzo dell'oggetto javascript XMLHttpRequest, verrà originata una richiesta per la servlet REST di HRS.



[4]Eclipse è un famoso ambiente IDE open source per la programmazione. Per maggiori informazioni, vedere [20]

[5]Per maggiori informazioni, si veda [21]

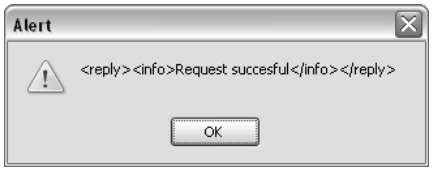
Il test consiste nello schedulare una richiesta HTTP GET all'indirizzo `http://localhost:8080/exist/servlet/db/universe/universe.xml` con un trigger che scatta ogni 10 secondi a partire da ora continuando indefinitamente. Il task così definito verrà nominato `universe` e il trigger `repeat10sec`. Successivamente aggiungeremo un trigger simile al precedente, con la differenza che l'intervallo sarà di 20 secondi. Infine cancelleremo il primo trigger.

Innanzitutto descriviamo il task iniziale in HRSML.

```
<task xmlns="http://www.iit.cnr.it/xmlgroup/hrs">
  <job uri="http://localhost:8080/exist/servlet/db/universe/universe.xml"
    http-method="GET" />
  <triggers>
    <trigger name="repeat10sec">
      <repeat>
        <every>10000</every>
      </repeat>
    </trigger>
  </triggers>
</task>
```

Attraverso l'interfaccia, selezioniamo il metodo PUT e inseriamo l'indirizzo di HRS, completandolo con il percorso che vogliamo definire per il nostro task, completo di nome (`/tasks/universe`):

Premendo il tasto *Send*, comparirà un popup con la risposta del server:

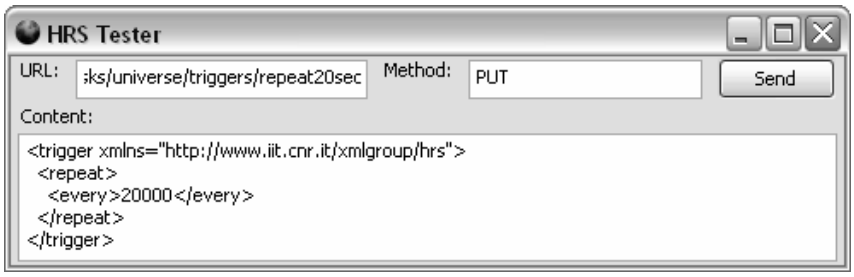


Siamo ora in grado di controllare che tutto vada come sperato visualizzando l'output di Jetty, che in effetti ora sta mostrando a intervalli regolari di 10 secondi il contenuto del documento `http://localhost:8080/exist/servlet/db/universe/universe.xml`:

Definiamo ora il nuovo trigger:

```
<trigger xmlns="http://www.iit.cnr.it/xmlgroup/hrs">
  <repeat>
    <every>20000</every>
  </repeat>
</trigger>
```

Lasciamo il metodo PUT e modifichiamo l'URI in modo da identificare il task precedentemente creato e il nome del nuovo trigger (`/tasks/universe/triggers/repeat20sec`):



Adesso possiamo selezionare il metodo DELETE e inserire l'URI che identifica il vecchio trigger (`/tasks/universe/triggers/repeat10sec`). Una volta eseguita la richiesta, sarà possibile osservare dall'output di Jetty la nuova frequenza con cui viene mostrato il contenuto del documento.

6.8 La servlet SOAP

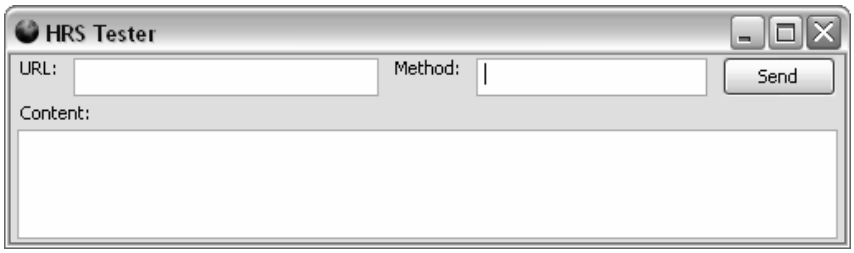
È stata realizzata una servlet SOAP molto semplice, che sfrutta in una minima parte i vantaggi di questo protocollo. La servlet riceve il messaggio SOAP e ne scorre il campo `Body`, cercando le invocazioni di metodi presenti. Sono stati previsti i seguenti metodi:

- `addTasks`, per creare di una lista di task a partire da `<tasks>`.
- `addTask`, per creare un task a partire da `<task>`. Necessita di un parametro aggiuntivo, *taskId*, che sarà contenuto all'interno di un elemento `<taskId>`.
- `replaceJob`, per sostituire un job definito da `<job>` di un task indicato da `<taskId>`.
- `addTriggers`, per aggiungere una lista di trigger a partire da `<triggers>` per un task indicato da `<taskId>`.
- `addTrigger`, per aggiungere un trigger definito da `<trigger>` con nome indicato da un elemento `<triggerId>` in un task indicato da `<taskId>`.
- `removeTask`, per rimuovere il task `<taskId>`.
- `removeTrigger`, per rimuovere il trigger `<triggerId>` da un task `<taskId>`.

Una volta stabilito cosa fare, la servlet procede estraendo dal frammento di HRSML contenuto nel corpo del metodo le informazioni necessarie per costruire le classi HRSlib, e le usa per istruire lo scheduler Quartz.

6.8.1 Testing

Sul lato server, la servlet SOAP di HRS è stata provata all'interno del servlet container Jetty, su piattaforma Linux Slackware. Il sistema operativo scelto per il client è invece Windows XP. L'interfaccia usata sul client è una semplice pagina scritta in XUL per l'ambiente Mozilla XULRunner [6], che si basa sullo stesso motore dei browser Firefox e Mozilla suite. La pagina permette di specificare l'indirizzo di HRS, il metodo HTTP da utilizzare per la richiesta (in questo caso POST) ed il contenuto SOAP. Con l'utilizzo dell'oggetto javascript XMLHttpRequest, verrà originata una richiesta per la servlet SOAP di HRS. Come ulteriore prova, è stato usato un semplice script visual basic che svolge la stessa funzione utilizzando oggetti Microsoft.



Il test consiste nello schedulare una richiesta HTTP POST all'indirizzo `http://192.168.0.3:8080/exist/servlet/db/universe/universe.xml` in modo che venga eseguita fra due minuti. Inizialmente non prevederemo nessun contenuto per la richiesta, simulando un errore dell'utilizzatore. Subito dopo simuleremo una correzione: invieremo un nuovo comando allo scheduler, in cui specificheremo il job corretto che andrà a sostituirsi al precedente.

Creiamo il messaggio SOAP per creare i due task:

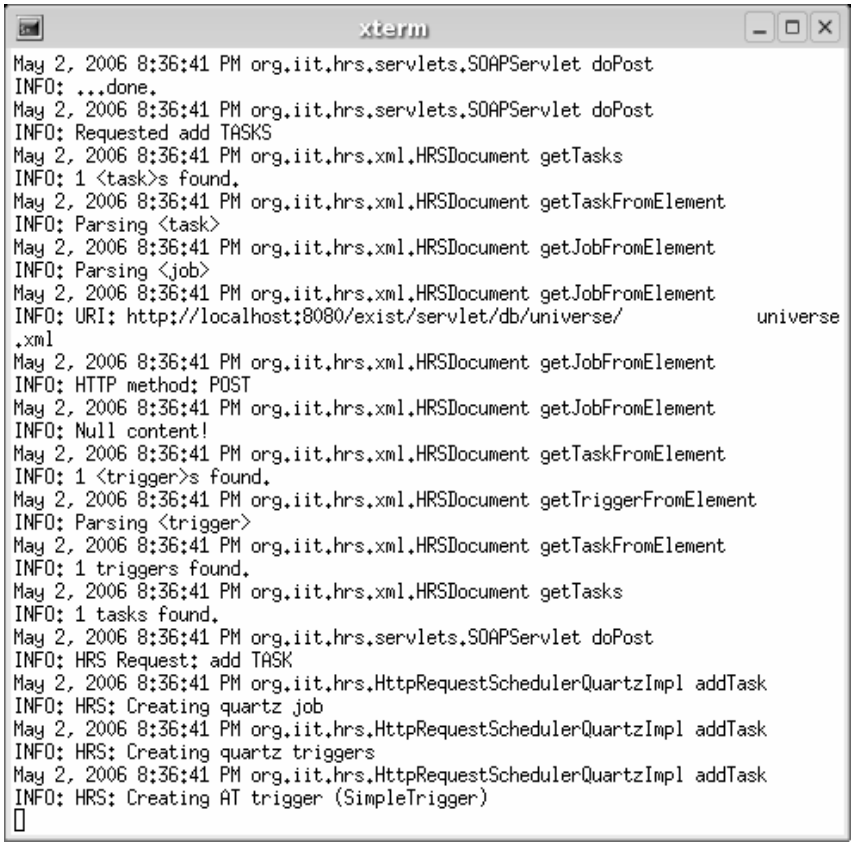
```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <addTasks xmlns="http://www.iit.cnr.it/xmlgroup/hrs">
      <tasks>
        <task id="update">
          <job uri="http://localhost:8080/exist/servlet/db/universe/
            universe.xml" http-method="POST">
          </job>
          <triggers>
            <trigger name="tuno">
              <at><relative to="now">120000</relative></at>
```

[6] Per maggiori informazioni, si veda [21]

```
            </trigger>
          </triggers>
        </task>
      </tasks>
    </addTasks>
  </soap:Body>
</soap:Envelope>
```

Utilizzando il metodo POST, spediamo le direttive alla servlet SOAP di HRS:

Il server riceve la richiesta correttamente, e stampa il seguente messaggio:



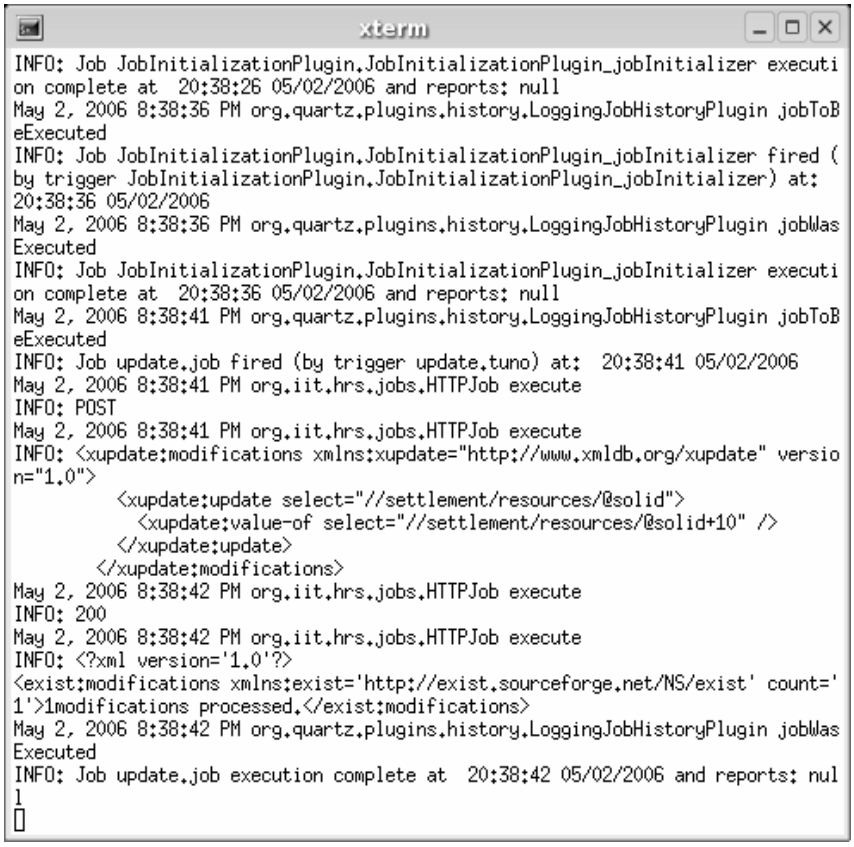
Adesso definiamo il job corretto e l'envelope SOAP per sostituirlo a quello errato:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <replaceJob xmlns="http://www.iit.cnr.it/xmlgroup/hrs">
      <taskId>update</taskId>
      <job uri="http://localhost:8080/exist/servlet/db/universe/
        universe.xml" http-method="POST">
        <xupdate:modifications version="1.0"
          xmlns:xupdate="http://www.xmldb.org/xupdate">
```

```
          <xupdate:update select="//settlement/resources/@solid">
            <xupdate:value-of select="//settlement/resources/@solid+10" />
          </xupdate:update>
        </xupdate:modifications>
      </job>
    </replaceJob>
  </soap:Body>
</soap:Envelope>
```

Inviamo la correzione ed attendiamo lo scadere dei due minuti.

Il server mostrerà il seguente output, ad indicare la corretta esecuzione del job:



L'uso del client visual basic script produce gli stessi risultati.

6.9 Il deployment descriptor

Si riporta il contenuto del file web.xml utilizzato per la configurazione dell'applicazione:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
```

```
    <servlet-name>RESTServlet</servlet-name>
    <servlet-class>org.iit.hrs.servlets.RESTServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>QuartzInitializer</servlet-name>
    <display-name>Quartz Initializer Servlet</display-name>
    <servlet-class>
      org.quartz.ee.servlet.QuartzInitializerServlet
    </servlet-class>
    <init-param>
      <param-name>config-file</param-name>
      <param-value>org/iit/hrs/quartz/quartz.properties</param-value>
    </init-param>
    <init-param>
      <param-name>shutdown-on-unload</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>start-scheduler-on-load</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
    <servlet-name>SOAPServlet</servlet-name>
    <display-name>SOAPServlet</display-name>
    <description></description>
    <servlet-class>org.iit.hrs.servlets.SOAPServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RESTServlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>QuartzInitializer</servlet-name>
    <url-pattern>/QuartzInitializer</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>SOAPServlet</servlet-name>
    <url-pattern>/soap</url-pattern>
  </servlet-mapping>
</web-app>
```

Riferimenti

[1] David A. Chapell Tyler Jewell. *Java Web Services*. HopsLibri, 2002.

[2] The Free Encyclopedia. Wikipedia. Web service. http://en.wikipedia.org/w/index.php?title=Web_service&oldid=50701690.

[3] W3C. Xml in 10 points. http://www.w3.org/XML/1999/XML-in-10-points.html.

[4] The Free Encyclopedia. Wikipedia. Hypertext transfer protocol. http://en.wikipedia.org/w/index.php?title=HyperText_Transfer_Protocol&oldid=48476946.

[5] The Free Encyclopedia. Wikipedia. Osi model. http://en.wikipedia.org/w/index.php?title=OSI_model&oldid=50876080.

[6] W3C. Uniform resource locator. http://www.w3.org/Addressing/#background.

[7] W3C. Soap. http://www.w3.org/TR/soap12-part1/#intro.

[8] Roy Thomas Fielding. Representational state transfer. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[9] Mark Volkmann. Rest architectural style. http://www.ociweb.com/jnb/jnbNov2004.html.

[10] OpenSymphony. Quartz. http://www.opensymphony.com/quartz.

[11] Paul Vixie. cron(8) - linux man page. http://www.die.net/doc/linux/man/man8/cron.8.html.

[12] The Free Encyclopedia Wikipedia. Ajax. http://en.wikipedia.org/w/index.php?title=Ajax_%28programming%29&oldid=50868786.

[13] Apache Software Foundation. Apache license, version 2.0. http://www.apache.org/licenses/LICENSE-2.0.

[14] Sun Developement Network. Java servlet technology. http://java.sun.com/products/servlet/.

[15] XML:DB Initiative. Xupdate - xml update language. `http://xmldb-org.sourceforge.net/xupdate`.

[16] W3C. Xml schema part 2: Datatypes second edition. `http://www.w3.org/TR/xmlschema-2`.

[17] Jetty. Jetty://. `http://jetty.mortbay.org/jetty/index.html`.

[18] jdom.org. Jdom. `http://www.jdom.org/`.

[19] The Apache XML Project. Apache xmlbeans. `http://xmlbeans.apache.org/`.

[20] Eclipse.org. Eclipse. `http://www.eclipse.org/`.

[21] The Free Encyclopedia Wikipedia. Xulrunner. `http://en.wikipedia.org/wiki/XULRunner`.