Progetto di Middleware
Gruppo 5

Giovanni Donelli <donelli@cs.unibo.i >
Mara Mucci <mucci@cs.unibo.it >
Fabio G. Strozzi <fstrozzi@cs.unibo.it >

1 Ruoli e permessi

Le speciche del progetto richiedono che ad ogni utente sia assegnato un ruolo configurabile dall'amministratore del sistema; per ogni ruolo, inoltre, deve esser possibile assegnare dei privilegi che regolano la prenotazione delle aule.

Il punto cruciale di questo requisito è la realizzazione di un insieme di regole modulari, componibili da parte dell'amministratore e tali da esprimere con una certa libertà i privilegi da associare alle aule e ai ruoli.

Le regole che abbiamo implementato permettono di costruire dei vincoli temporali. Per dare un esempio del potere espressivo ottenuto, si considerino i seguenti divieti che l'amministratore può esprimere componendo più regole tra loro:

- divieto di prenotare più di 2 ore al giorno tutti i giorni di ogni settimana a marzo;
- divieto di prenotare il primo giorno di ogni mese tra marzo e settembre;
- divieto di prenotare il 18 di febbraio;
- divieto di prenotare più di 60 minuti al giorno;
- divieto di prenotare più di 200 minuti alla settimana tra dicembre e gennaio;

Per rendere operativi i divieti creati, l'amministratore deve associarli alle aule e ai ruoli. É possibile associare più divieti contemporaneamente ad ogni aula: in questo caso la prenotazione di quell'aula è accettata dal sistema solo se rispetta tutti i vincoli stabiliti.

Da un punto di vista implementativo ciò che viene creato quando l'amministratore compone un divieto, non è altro che una catena di regole, ovvero un insieme

di istanze di classi che implementano l'interfaccia `Rule`. Le catene sono create a partire dalle regole che operano sugli intervalli di tempo più piccoli (minuti e giorni).

Ogni regola è in grado di analizzare una prenotazione (un oggetto di tipo `Schedule`) e stabilire se è valida rispetto ai vincoli che implementa. Alcune classi (come `DayRule` e `WeelRule`), mantengono un riferimento ad un'altra regola (detta *sotto-regola*), alla quale delegano la verifica della correttezza di una prenotazione (o di una parte di essa).

Le regole, quindi, operano in uno dei seguenti modi:

- data una istanza di `Schedule` stabiliscono se essa è accettabile o meno: ad esempio, un oggetto di tipo `MinuteRule`, che viene istanziato con un limite di minuti stabilito dall'utente (nel costruttore all'atto della creazione), accetta una prenotazione solo se il numero di minuti è inferiore a quello stabilito e la scarta altrimenti.

- oppure, data un'istanza di `Schedule` e un intervallo temporale che la include, riducono la prenotazione in *sotto-prenotazioni* e stabiliscono se accettarla o meno in relazione all'applicazione delle sotto-regole alle sotto-prenotazioni. Ad esempio, il divieto di prenotare "più di due ore al giorno" è una catena composta dalla regola `DayRule` e dalla sotto-regola `MinuteRule` (quest'ultima impostata a 120 minuti di limite massimo). `DayRule` riduce la prenotazione passata in sotto-prenotazioni della durata di un giorno e applica la sotto-regola `MinuteRule` ad ognuna di esse: la prenotazione nel suo insieme sarà ritenuta valida solo se tutte le sotto-prenotazioni della durata di un giorno durano menodi 120 minuti.

Dal momento che alcune delle regole ideate si riferiscono a periodi di tempo precisi, è stato necessario costruire una classe di supporto (`CalendarInterval`) che fornisce dei metodi per estrarre sotto-intervalli e giorni specifici a partire da un qualunque intervallo di tempo. Grazie a questa classe, ad esempio, è possibile ottenere le date dei giorni di un periodo generico, o quelle di inizio e fine di una certa settimana del mese.

La schematizzazione in UML delle classi `Rule` e `CalendarInterval` è mostrata in figura 1 (per semplicità sono mostrate solo alcune delle implementazioni di `Rule`).

Come si può notare, `Rule` estende la classe `Serializable`. Questa condizione è necessaria per permettere di memorizzare le catene di regole come attributi di Entity EJB chiamati `Ban`. L'associazione dei divieti alle classi e ai ruoli viene fatta in un secondo momento tramite l'Entity EJB `RoomPolicy`.

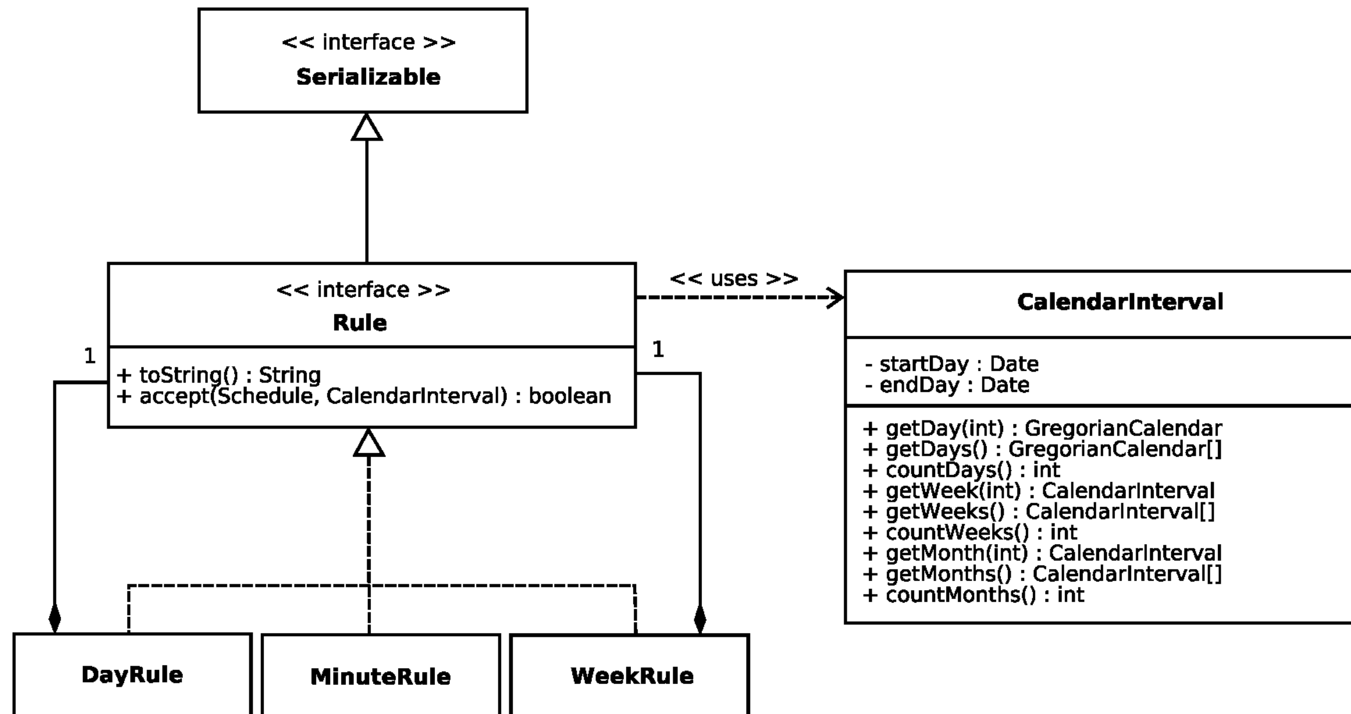

Figura 1: Diagramma di classe UML: l'interfaccia `Rule` fornisce l'astrazione della regola generica (sono mostrate solo alcune delle classi che la implementano); `CalendarInterval` permette di lavorare sugli intervalli di tempo.

2 Test di carico

Abbiamo messo a confronto il comportamento dell'applicazione in un ambiente client-server e in un ambiente distribuito. In entrambi i casi abbiamo misurato i valori del tempo di risposta per servire una pagina e del throughput, inteso come numero di pagine servite al minuto.

Data la complessità delle pagine web che abbiamo realizzato, si è deciso di testare i tempi di reazione alla richiesta di particolari pagine create ad-hoc ma tali da rispecchiare alcune delle operazioni che avvengono con maggior frequenza:

- `MonthSchedule`: visualizza il calendario delle prenotazioni di un mese (almeno 16 prenotazioni per mese), fa uso elevato del database e allo stesso tempo è fortemente cpu-bound;

- `PopulateDB`: effettua l'inserimento e la rimozione delle aule, in questo modo è possibile testare il sistema con le operazioni di creazione e rimozione di Entity EJB;

- `RoomManagement`: richiede l'elenco completo delle aule (circa 400), dipende principalmente dai tempi di risposta del database ed è pochissimo cpu-bound.

I test della soluzione distribuita sono stati effettuati su cluster omogenei composti da 8, 4 e 2 nodi. Raccogliere i dati nelle diverse configurazioni ci ha permesso di studiare la scalabilità dell'application server JBoss al variare del numero dei nodi.

Per effettuare la spartizione ed il load balancing delle richieste abbiamo utilizzato il web server Apache ed il modulo `mod_jk` che è in grado di smistare le richieste HTTP verso una batteria di web server Tomcat opportunamente configurati: nel nostro caso abbiamo fissato il load factor di ogni server a 1 dato che le macchine erano identiche. L'esecuzione del web server Apache è avvenuta su una macchina distinta dalle macchine del cluster.

Per ogni configurazione, abbiamo simulato con JMeter 8 richieste contemporanee, ripetute in serie 50 volte, per un totale di 400 richieste complessive. I risultati sono esposti nelle tabelle 1, 2 e 3.

Come si può notare, passando dalla soluzione non clusterizzata (un solo nodo) a quella clusterizzata, il tempo medio di risposta delle pagine diminuisce e il throughput aumenta. Putroppo, questa tendenza non è lineare rispetto al numero dei nodi in seguito al collo di bottiglia causato dal database Hypersonic centralizzato, del quale cioè esiste una sola istanza condivisa da tutti i nodi. Questo limite della configurazione distribuita è ben visibile nella tabella 3 che mostra i tempi di risposta di una pagina che legge 400 aule (un'operazione che dipende quasi esclusivamente dal tempo di risposta del database): un'aumento del numero di nodi

comporta addirittura una riduzione del tempo di risposta che sale da 31.35 secondi con 4 nodi a 33.21 con 8 nodi. Presumiamo che questo risultato sia causato dalla saturazione delle richieste che possono impegnare contemporaneamente il database. Sebbene i tempi di risposta della pagina `MonthSchedule` siano alti (una media di 11 secondi con 8 nodi), le maggiori dimensioni del cluster comportano un miglioramento dal momento che le operazioni coinvolte sono principalmente cpu-bound, cioè dipendono in prevalenza dall'elaborazione dei dati, e traggono vantaggio dalla maggiore parallelizzazione (con 8 nodi si può presumere che ognuna delle 8 richieste sia gestita da un singolo application server).

Rispetto alla configurazione client-server la soluzione distribuita che abbiamo adottatato presenta vantaggi e svantiggi. Un vantaggio è certamente la possibilità di parallelizzare le richieste e distribuire il carico tra le varie macchine in modo da aumentare la capacità di servizio. Inoltre, la replicazione dei nodi aumenta la tolleranza verso alcuni tipi di guasto. Durante i test abbiamo simulato dei possibili guasti ai nodi (uccidendo alcuni processi di JBoss durante l'elaborazione) e abbiamo verificato che il modulo `mod_jk` permette di mantenere attivo il servizio smistando le richieste ai nodi rimasti. Purtroppo, la centralità del database rappresenta anche in questo caso il punto debole del sistema: infatti, se un guasto colpisce il nodo che è responsabile anche dell'esecuzione di Hypersonic, allora l'intero sistema va in crash. Lo stesso discorso vale anche per il web server Apache verso il quali tutti i client inviano le richieste. Se un guasto avvenisse a questo tier, nel nostro caso sarebbe ancora possible richiedere le pagine direttamente alle macchine del cluster; ma in una situazione reale, dove il web server rappresenta l'unico nodo di accesso per l'utente, allora tutto il sistema risulterebbe inaccessibile.

Numero di nodi	**Media del tempo di risposta (sec)**	**Deviazione dalla media (sec)**	**Throughput (pagine/minuto)**
8	11.00	1.21	43.6
4	14.19	2.53	33.8
2	21.68	10.66	22.14
1	36.03	2.06	13.32

Tabella 1: Risultati dell'interrogazione della pagina `MonthSchedule` che restituisce il calendario con tutte le prenotazioni di un mese divise giorno per giorno.

Numero di nodi	Media del tempo di risposta (sec)	Deviazione dalla media (sec)	Throughput (pagine/minuto)
8	1.18	0.81	406.7
4	1.28	1.02	375
2	1.61	1.25	298.1
1	2.40	0.97	200

Tabella 2: Risultati dell'interrogazione della pagina `PopulateDB` che comporta la creazione e cancellazione di diversi Entity EJB.

Numero di nodi	Media del tempo di risposta (sec)	Deviazione dalla media (sec)	Throughput (pagine/minuto)
8	33.21	6.95	14.5
4	31.35	5.62	15.3
2	41.29	13.24	11.6
1	75.71	2.60	6.3

Tabella 3: Risultati dell'interrogazione della pagina `RoomManagement` che restituisce l'elenco completo delle aule.