UNIVERSITÀ DEGLI STUDI DI MILANO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in

Scienze e Tecnologie dell'Informazione

Mole.io

un sistema per la gestione centralizzata dei log applicativi

RELATORE

Prof. Ernesto DAMIANI

TESI DI LAUREA DI

Federico GANDELLINI

CORRELATORE

Matr. 123456

Prof. Nome COGNOME

Anno Accademico 2013/2014

Ringraziamenti

Un grazie a ...

Indice

Introduzione

In questa tesi descriveremo Mole.io: un sistema centralizzato per la raccolta e l'aggregazione di messaggi provenienti da applicazioni remote.

Durante il loro ciclo di lavoro o *processing*, le applicazioni software eseguono operazioni significative o entrano in situazioni di errore, in questi casi è importante che le persone cha hanno in carico la gestioni di questi sistemi, siano informate dell'accaduto in modo da operare scelte opportune o applicare le dovute correzioni (*bugfix*).

Gli sviluppatori spesso utilizzano messaggi di tracciamento (*log*) per stampare a video o salvare in *files* stati significativi delle applicazioni. Gli stessi *log* sono utilizzati più spesso per riportare situazioni di errore (*Exception* e *Stack Trace*).

Il problema principale di questo approccio è la *località* dei *log*, solitamente questi *files* vengono salvati, nella stessa macchina sulla quale sta operando l'applicazione.

All'aumentare del numero di applicazioni da gestire e del numero di macchine in produzione, capita spesso che i server siano in luoghi geograficamente distanti tra loro. Questa situazione rende evidente la difficoltà di ottenere un feedback veloce dello stato di ogni software e delle eventuali situazioni di errore in cui le applicazioni si trovano.

Mole.io cerca di risolvere il problema facendo in modo che i software

che lo utilizzano, siano in grado di inviare le informazioni che ritengono significative ad un server centrale, che le raccoglie, le cataloga e le aggrega per essere facilmente supervisionate da parte degli sviluppatori.

Nel primo capitolo tratteremo approfonditamente il problema dei *log*, i contesti nei quali essi vengono utilizzati e le problematiche legate alla gestione di questo tipo di soluzione di tracciamento. Vedremo anche come utilizzare i *log* per ottenere informazioni di supporto alla *business intelligence*.

Il secondo capitolo riporterà un elenco dei principali *software* per la gestione centralizzata dei *log* presenti sul mercato e delle soluzioni *Open Source* che sono state prese a modello per la realizzazione di Mole.io. Descriveremo ogni applicazione e mostreremo come Mole.io possa essere una soluzione innovativa sotto svariati punti di vista.

I due capitoli seguenti permetteranno di approfondire i dettagli tecnici delle metodologie di sviluppo applicate durante il *design* del software e alcune tra le principali tecnologie utilizzate per la realizzazione del sistema.

Il quinto capitolo descriverà la struttura di Mole.io e le varie componenti software che rendono l'applicazione scalabile, sicura e garantiscono l'alta affidabilità della soluzione. Uno spazio particolare sarà inoltre riservato alle problematiche incontrate durante lo sviluppo.

Nel sesto capitolo vedremo in modo oggettivo, con *benchmark* e *stress test* il comportamento di Mole.io all'aumentare del carico di lavoro e dimostreremo come le soluzioni di design applicate garantiscano buone *performance* anche in condizioni critiche.

Infine discuteremo i risultati ottenuti e proporremo alcune interessanti funzionalità che trasformeranno Mole.io dall'attuale *proof of concept* ad un vero e proprio servizio.

Capitolo 1

Log:
contesti e problematiche

Iniziamo riportando alcune definizioni che utilizzeremo spesso nel seguito della tesi. Diciamo *processo* un programma in esecuzione e *log* l'insieme dei messaggi prodotti, a fini informativi, da tale processo.

La decisione di quali e quante informazioni salvare, spetta tipicamente allo sviluppatore o al personale addetto alla gestione dell'applicazione.

I messaggi di log posso essere di vario tipo, è usanza comune caratterizzare ogni messaggio con un livello di gravità (*severity*) permettendo così una identificazione più rapida degli errori più gravi, rendendo repentino l'intervento di riparazione dell'applicazione.

Informazioni spesso salvate all'interno dei log sono dati specifici del sistema nel quale l'applicazione è in esecuzione, dati relativi all'utente che la sta utilizzando, oppure relativi allo stato del sistema in un preciso istante temporale. Ogni log è infine corredato da un messaggio significativo che lo rende immediatamente identificabile tra altri.

1.1 Trattare gli errori applicativi

Il salvataggio dei log, come abbiamo anticipato, è una operazione molto comune nei software, ma diventa fondamentale quando si vuole monitorare lo stato interno di una applicazione in esecuzione e si vuole essere informati riguardo alle situazioni di errore nelle quali quest'ultima incorre.

Salvare le informazioni relative alle situazioni di errore è importante per gli sviluppatori, questo permette, infatti di velocizzare l'individuazione di errori (*bug*) nel flusso di lavoro del programma e, di conseguenza la loro risoluzione (*bugfix*).

Il salvataggio dei log avviene tipicamente su uno o più *files* presenti nella stessa macchina nella quale sta funzionando l'applicazione. A seconda del tempo di esecuzione di una applicazione e della frequenza con la quale essa produce messaggi di log, i files sui quali vengono salvate le informazioni possono diventare molto grandi. Un file di questo tipo è molto complesso da gestire da parte degli addetti ai lavori, infatti diventa lungo e complesso trovare informazioni significative all'interno di esso e soprattutto diventa complesso correlare le situazioni di errore e capire con quale frequenza o in quali condizioni si presenta un particolare malfunzionamento.

problema località: se ho molte applicazioni e molti clienti devo tener monitorata (da remoto) la situazione dei log per ogni cliente/app/macchina, verificare che non esplodano i files e quando diventano troppo grandi, archiviare parte di questi.

problema spazio, località difficili da leggere e difficile tirarci fuori delle info significative

1.2 La centralizzazione

tante applicazioni (anche tipi diversi) che loggano tanti clienti da gestire dislocati sul territorio

veremo che mole usa un db documentale perhcé si presta meglio a salvare dati non fortemente strutturati

1.3 Business intelligence

capire come gli utenti usano il sistema statistiche sul sistema decidere come indirizzare lo sviluppo

Capitolo 2

Log: software e applicazioni

ci sono tante soluzioni sul mercato, di seguito alcune ma non ci piacciono

2.1 Prodotti e soluzioni sul mercato

overview di alcuni sistemi di logging con le relative funzioni specifiche i competitor airbreak - logga solo rollbar - aggrega papertrail - live log

2.2 Una nuova applicazione: Mole.io

perché le soluzioni sul mercato non ci piacciono le peculiarità di mole.io

Capitolo 3

Metodologie di sviluppo

3.1 User stories

3.2 Test e behavior driven development

Capitolo 4

Tecnologie utilizzate

In questo capitolo approfondiremo i dettagli delle tecnologie utilizzate per realizzare Mole.io. Illustreremo, per ognuna di esse, le motivazioni che ci hanno spinto alla scelta di particolari soluzioni software e le problematiche incontrate durante il loro utilizzo.

Mole.io è stato interamente sviluppato utilizzando il linguaggio JavaScript. L'utilizzo di questa tecnologia è abbastanza comune all'interno delle pagine web e si presta bene all'utilizzo *client side*, meno comune è invece la sua applicazione nella parte *server*. La piattaforma Node.js permette di utilizzare JavaScript per sviluppare la parte *backend* delle applicazioni. Rendere uniforme il linguaggio utilizzato permette facilitare lo sviluppo e le fruibilità del progetto da parte degli sviluppatori. Per poter lavorare al progetto è infatti richesta solo la conoscenza di JavaScript e non di altri linguaggi, questo è un ottimo requisito quando si pensa ad un team di sviluppo in espansione.

Quella del paragrafo precedente è solo una delle motivazioni che ci hanno spinto a scegliere Node.js come tecnologia di sviluppo, iniziamo quindi a vedere in dettaglio questa tecnologia.

4.1 Node.js

La *homepage* del sito ufficiale [1] di Node.js fornisce una sintetica ma precisa descrizione di questa tecnologia, partiremo proprio da essa per illustrarne le peculiarità.

La descrizione ufficiale recita:

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Scopriamo immediatamente che Node.js è una *piattaforma*. L'utilizzo di questo termine mette l'accento su un aspetto fondamentale di questa tecnologia: fornire un ambiente nel quale le applicazioni sviluppate possano funzionare con il supporto di librerie di sistema fornite da Node.js stesso.

La piattaforma Node.js utilizza JavaScript come linguaggio di sviluppo. Per farlo si avvale del potente interprete *V8* presente all'interno del browser *Chrome* di *Google*. L'utilizzo di un linguaggio altamente popolare e di una base solida come quella fornita dal popolare *browser* permettono di costruire applicazioni in modo semplice e veloce.

L'ultimo importante concetto, che leggiamo dalla prima frase della descrizione, è che Node.js è principalmente orientato allo sviluppo di applicazioni che lavorano con la rete. Per sua natura, la piattaforma ci aiuta a fare in modo che esse siano scalabili.

La seconda parte della descrizione spiega sinteticamente alcune caratteristiche peculiari di Node.js e ne definisce meglio il contesto applicativo.

L'intera piattaforma è centrata sul concetto di *evento*. Si dice evento un messaggio che viene scatenato in un determinato istante dell'elaborazione e che successivamente è catturato e gestito dalle componenti del sistema che sono preposte alla gestione di quell'evento specifico.

Il sistema ad eventi viene utilizzato da Node.js congiuntamente ad una gestione non bloccante delle operazioni di *Input/Output*. Questo significa che una operazione potenzialmente lunga, come ad esempio la comunicazione con il *filesystem* o con i dispositivi di *rete* non bloccano il flusso di esecuzione del programma principale. Dopo aver richiesto ad altri attori del sistema il dato di cui necessita, il programma continua il suo normale flusso di esecuzione e verrà informato, utilizzando un evento, quando il dato richiesto sarà disponibile. Questa gestione non bloccante delle operazioni di *I/O* è chiamata *I/O* asincrono.

Questa modalità di gestione delle operazioni non strettamente legate alla logica applicativa, permette a Node.js di essere molto efficiente se utilizzato per la realizzazione di applicazioni che elaborano grandi quantità di dati ma devono rimanere *reattive* nei confronti di nuove richieste di elaborazione.

4.1.1 Le operazioni asincrone

Abbiamo introdotto il concetto di I/O asincrono, di seguito illustreremo come Node.js riesca a gestirlo utilizzando una quantità limitata di risorse di sistema.

I componenti *hardware* di un sistema elaborano dati con velocità differenti. La rapidità di ogni componente è fortemente legata al modo nel quale questo è stato realizzato. La tabella seguente riporta un elenco di componenti e le relative velocità indicative riferite ad un singolo ciclo di *CPU*.

Componente	**Numero di cicli**
CPU	1
Cache di livello 1	3
Cache di livello 2	14
RAM	250
Hard Disk	41.000.000
Dispositivo di rete	240.000.000

Guardando la tabella si nota immediatamente come i dispositivi di *I/O* siano ordini di grandezza più lenti rispetto ai dispositivi di elaborazione delle informazioni. Se il flusso del programma dovesse aspettare in modo sincrono ogni singola operazione di lettura o scrittura su disco, ad esempio, esso perderebbe la possibilità di eseguire circa quaranta milioni di operazioni di calcolo, con un conseguente degrado delle performances del software.

Una tecnica comune per affrontare il problema dell'*I/O* è l'utilizzo di *threads*. Un thread è spesso definito con il termine sotto-processo leggero, in effetti esso condivide codice e risorse di sistema con altri threads appartenenti allo stesso processo padre.

Un thread permette al processo di parallelizzare l'esecuzione di una parte del suo flusso di lavoro, ma al tempo stesso sono necessarie risorse di sistema per creare il thread stesso e per distruggerlo. Node.js ovvia a questo problema utilizzando un *thread pool*, cioè un insieme di threads che fungono da *worker* per il processo. Quando è necessario eseguire un particolare task, esso viene sottoposto al thread pool, di conseguenza viene assegnato ad un worker, esso lo esegue e al termine del lavoro comunica l'esito dell'elaborazione al chiamante tramite una funzione detta *callback*.

L'utilizzo di un thread pool comporta un incremento di prestazioni da parte delle applicazioni che lo sfruttano: i thread pool ottimizzano l'utilizzo

della memoria e del processore, diminuendo l'overhead di gestione dei thread.

4.1.2 Le callback

Una delle principali difficoltà quando si utilizza Node.js per la prima volta è la gestione delle callback. Di seguito è riportato un frammento di codice Node.js che esegue il salvataggio di un utente su Database.

```
function save(user) {
  console.log('saving user into db');
  db.insert(user, function(err, user) {
    console.log('user successfully saved!');
  });
  console.log('all we need is just a little patience...');
}
```

la funzione `db.insert()` è asincrona, ed accetta due parametri: il dato da salvare e la callback da eseguire una volta eseguito l'inserimento nel Database. L'esecuzione di questo codice produce un output in console simile a quello mostrato qui sotto.

```
saving user into db
all we need is just a little patience...
user successfully saved!
```

Se la funzione `db.insert()` non fosse asincrona otterremmo un output come quello che segue.

```
saving user into db
user successfully saved!
all we need is just a little patience...
```

Questo piccolo esempio, lascia intuire un potenziale problema nella gestione delle callback: se, ad esempio, dovessimo assegnare alcuni privilegi di *default* a tutti i nuovi utenti inseriti nel sistema, potremmo scrivere il seguente codice.

```
function save(user) {
  console.log('saving user into db');
  db.insert(user, function(err, user) {
   db.insert(grants, user, function(err, grants) {
      console.log('user successfully saved!');
    });
  });
  console.log('all we need is just a little patience...');
}
```

Ora il problema è evidente: se i dati ottenuti in modo asincrono sono necessari per eseguire altre procedure, essi vanno gestiti internamente alla callback stessa. Se utilizzate in modo improprio, le callback, portano al cosiddetto *callback-hell*, ovvero un codice nel quale il lettore non riesce più a seguire il flusso logico dell'applicazione.

Fortunatamente JavaScript ci fornisce una soluzione semplice al problema, ma che richiede allo sviluppatore disciplina nello scrivere il codice sorgente. L'esempio precedente, infatti potrebbe essere riscritto come segue.

```
function onGrantsSaved(err, grants) {
  console.log('user successfully saved!');
}

function onUserSaved(err, user) {
```

```
  db.insert(grants, user, onGrantsSaved);
}

function save(user) {
  console.log('saving user into db');
  db.insert(user, onUserSaved);
  console.log('all we need is just a little patience...');
}
```

Il codice è ora molto più leggibile, semplicemente evitando la dichiarazione di funzioni anonime in linea.

4.1.3 La storia

4.1.4 Npm e moduli

L'organizzazione del codice è un aspetto fondamentale dello sviluppo. Il *Single Responsibility Principle* (SRP) è uno dei principi di design del software più importanti. Applicato alla programmazione ad oggetti, esso sostiene che ogni oggetto deve essere responsabile di un singolo aspetto del comportamento del sistema.

Node.js incarna questo principio nelle fondamenta della sua struttura, infatti permette di organizzare il codice in unità indipendenti tra loro chiamate *moduli*. Ogni modulo nella piattaforma può decidere quali dati gestire al suo interno e quali esporre all'esterno. Si creano in questo modo delle *black box* che funzionano tanto meglio quanto il loro compito è specifico.

Poco dopo la nascita di Node.js la comunità di sviluppatori che lo utilizzava ha deciso di arricchire questa piattaforma con un *tool* ormai insostituibile: *Node Package Manager* (NPM).

NPM è un tool, utilizzabile da linea di comando, che si occupa della gestione dei moduli e delle loro dipendenze, per farlo utilizza un *repository* in rete nel quale sono registrati tutti i moduli pubblici sviluppati dai vari *contributor* della *community* Node.js.

Abbiamo introdotto il concetto di *dipendenza* tra moduli. Un modulo si dice dipendente da un altro quando necessita di quest'ultimo per eseguire il suo compito. NPM impone che ogni modulo sia descritto dal file `Package.json` che ne riporta le informazioni principali ed elenca gli altri moduli dai quali dipende. Di seguito riportiamo un esempio di `Package.json` per un modulo chiamato `mole` che fa parte del sistema realizzato per questa tesi.

```
{
  "name": "mole",
```

```
"version": "0.0.1",
"author": "Federico Gandellini",
"description": "whisper collector and denormalizer",
"private": true,
"scripts": {
  "start": "node mole.js"
},
"engines": {
  "node": ">=0.8.0",
  "npm": ">=1.2.0"
},
"dependencies": {
  "express": "3.3.4",
  "underscore": "~1.5.2",
  "mongo-make-url": "0.0.1",
  "mongoskin": "~0.6.0",
  "mongodb": "~1.3.19",
  "require-all": "0.0.8",
  "rabbit.js": "~0.3.1"
},
"devDependencies": {
  "grunt-contrib-jshint": "~0.6.4",
  "grunt": "~0.4.1",
  "grunt-mocha-cli": "~1.2.1",
  "grunt-contrib-watch": "~0.5.3",
  "mocha": "~1.13.0",
  "should": "~1.3.0",
```

```
    "supertest": "~0.8.0",
    "Faker": "~0.5.11"
  }
}
```

Nella prima parte del file possiamo trovare i dati principali del modulo, come il suo nome, l'autore, la versione, e una descrizione. Seguono un paio di parametri che definiscono le regole di pubblicazione, il comando per lanciare questo modulo e le versioni richieste della piattaforma Node.js e di NPM.

Le due sezioni seguenti nel file elencano le dipendenze, la prima indica i moduli che devono essere presenti perché esso possa essere messo *in produzione*, le altre sono dipendenze che sono richieste esclusivamente durante lo sviluppo o il *testing* del modulo stesso. Come si può notare, nel file, non sono presenti solo i nomi, ma anche le versioni richieste degli altri moduli.

Uno dei comandi più utlizzati di NPM è `npm install`, esso legge il file `Package.json` presente nella *directory* corrente e scarica dalla rete tutti i pacchetti richiesti alle rispettive versioni e permette all'utilizzatore del modulo di essere immediatamente operativo.

Alcuni moduli utilizzati

Per poter utilizzare le funzionalità fornite da un modulo aggiuntivo, Node.js fornisce un comando che permette di importarlo dall'esterno. Questo comando è `require()` e si utilizza passando come parametro il nome del modulo da caricare.

```
var express = require('express');
```

Una volta eseguito il comando, la variabile `express` conterrà il modulo appena caricato e sarà possibile utilizzarne le funzionalità.

Illustriamo ora alcuni dei moduli utilizzati per realizzare la nostra applicazione.

express È uno dei moduli più importanti: permette di creare facilmente un *server web* in grado di rispondere a richieste provenienti dai *client*. Utilizza un sistema di *rotte* per legare l'azione da eseguire alla richiesta HTTP in arrivo. Express eredita da *Connect* una funzionalità chiave per il suo funzionamento, i *middleware*. In Connect, un middleware è una funzione che filtra tutte le richieste in ingresso e le risposte in uscita e le restituisce rielaborate. I middleware sono costruiti in modo da essere accodati l'uno con l'altro, dando la possibilità di costruire filtri molto complessi. Illustreremo la configurazione delle rotte e dei middleware di express nella sezione 5.1.

passport Il compito di questo modulo è gestire l'autenticazione degli utenti. Passport fornisce un comodo middleware per express, con il quale è possibile verificare i dati di autenticazione dell'utente quando questo voglia accedere a specifiche rotte. Più avanti, nella sezione 5.2, approfondiremo questo tema.

mongoose e mongoskin Sono i due principali *driver* Node.js per MongoDB, il sistema per l'archiviazione dei dati che abbiamo utilizzato nell'applcazione. Nella sezione 4.3 vedremo nel dettaglio questo database documentale e illustreremo come è possibile accedere alla sua interfaccia utilizzando Node.js.

require-all Permette di caricare tutti i moduli presenti in una directory. Questo modulo ci è stato molto utile per garantire e semplificare l'estensibilità del sistema. Nella sezione 5.1.1 vedremo nel dettaglio come abbiamo sfruttato questa semplice ma potente funzionalità.

mocha Questo modulo non fornisce funzionalità vere e proprie da spendere in produzione, bensì un insieme preziosissimo di *tool* per poter testare la propria applicazione Node.js. Abbiamo largamente utilizzato questo strumento durante la realizzazione del sistema per renderlo, quanto più possibile, *bug-free*.

4.2 RabbitMQ

Quando si progetta una applicazione web che offre un servizio, bisogna sempre porre molta attenzione a non introdurre nel sistema i cosiddetti *colli di bottiglia*, cioè componenti dell'architettura che non riescono a sopportare il carico delle richieste in arrivo dagli utenti.

Una tecnica piuttosto efficace per evitare i colli di bottiglia, consiste nel realizzare un *disaccoppiamento* delle varie componenti presenti nel sistema. Una applicazione ben disaccoppiata è una applicazione nella quale ogni singolo componente svolge una funzione specifica e comunica con le altre parti del sistema secondo protocolli predefiniti. Disaccoppiare quindi non è sempre facile, perché è necessario costruire infrastrutture che permettano alle varie parti, ormai slegate, di comunicare tra loro.

RabbitMQ è un software che permette di realizzare, configurare e monitorare complessi sistemi di code di messaggi. Il suo utilizzo permette di realizzare una infrastruttura nella quale le diverse parti di un sistema possano comunicare tra loro scambiandosi messaggi attraverso le code utilizzando un protocollo determinato dallo sviluppatore.

L'utilizzo di RabbitMQ fornisce un vantaggio evidente in termini di flessibilità: è possibile agganciare o rimuovere parti da un sistema attivo senza comprometterne l'intero funzionamento.

La flessibilità non è l'unico vantaggio, infatti RabbitMQ permette di ottenere strutture perfettamente gestibili su sistemi di tipo *Platform as a Service* (PaaS), sui quali si può decidere di aumentare o diminuire dinamicamente le risorse fornite ad un sistema in produzione in accordo con il numero di richieste utente da soddisfare.

Nella sezione 5.1 vedremo nel dettaglio la configurazione di RabbitMQ utilizzata per realizzare Mole.io e quali tecniche abbiamo messo in atto per

permettere al sistema di essere installato su una piattaforma di tipo PaaS.

4.3 MongoDB

4.3.1 Fronteggiare le richieste

4.4 AngularJS e altre tecnologie di frontend

4.4.1 Gestione delle dipendenze

4.5 Strumenti per il deploy

Capitolo 5

Mole.io

5.1 Architettura del sistema

5.1.1 CQRS ed estensibilità

5.1.2 mole

I denormalizzatori

5.1.3 mole-suit

I plugin e gli widget

5.2 Autenticazione degli utenti

5.3 Scalabilità e affidabilità

5.4 Problematiche di sviluppo

Capitolo 6

Configurazioni e benchmark

problemi con i benchmark - dipendi dalla rete su cui sei - nostro client fatto con node - perché non l'abbiamo usato - come sono stati fatti i benchmark - specifiche del sistema VM, ram, hdd, ... - risultati ottenuti

non ha senso testare mole (server) per la parte rabbit (spiegare) il vero collo di bottiglia è il db abbiamo testato con configurazioni di db differenti

Conclusioni e sviluppi futuri

Bibliografia

[1] Inc Joyent. Node.js website, 2009.