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 che hanno in carico la gestione di questi sistemi, siano informate dell'accaduto in modo da operare scelte opportune o applicare le dovute correzioni (*bugfix*).

Gli sviluppatori sono soliti utilizzare messaggi di tracciamento (*log*) per stampare a video o salvare in *file* stati significativi delle applicazioni. I messaggi più frequenti riportati nei *log* sono quelli relativi a situazioni di errore (*Exception* e *Stack Trace*).

L'approccio comune alla creazione e gestione dei *log* presenta la criticità specifica della *località*, poiché tipicamente questi *file* 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 la tematica 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 e garantiscono l'alta accessibilità della soluzione.

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 di traffico.

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

Prima di entrare nello specifico delle tematiche trattate, è necessario riprendere e chiarire alcuni concetti chiave che utilizzeremo ripetutamente nel seguito della tesi.

Si definisce *processo* un programma in esecuzione e *log* l'insieme dei messaggi prodotti, a fini informativi, da tale processo.

I messaggi di *log* posso essere di vario tipo. È usanza comune caratterizzare ogni messaggio con un livello di gravità (*severity*) permettendo così una rapida identificazione degli errori critici allo scopo di rendere repentino l'intervento di riparazione dell'applicazione.

Benché esista un protocollo riconosciuto a livello internazionale e adottato negli ambienti *Unix-like* chiamato *SysLog*, nell'ambito dello sviluppo di applicativi, non esistono norme vincolanti per la strutturazione dei log stessi. Questo accade sia perché SysLog non rappresenta uno standard rigidamente definito, sia perché l'organizzazione delle informazioni contenute nei log, è spesso delegata agli sviluppatori, i quali implementano questa funzionalità

nel modo più conveniente rispetto alle specifiche esigenze dell'applicazione in costruzione.

All'interno dei log, di conseguenza, troveremo informazioni diversificate in base al caso d'uso, ma tra le più frequenti possiamo citare:

- dati specifici del sistema nel quale l'applicazione è in esecuzione;
- dati relativi all'utente che sta utilizzando l'applicazione;
- dati relativi allo stato del sistema in un preciso istante temporale;
- un marcatore temporale (*timestamp*)
- un messaggio in linguaggio naturale, 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, con l'obiettivo di 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. Questa operazione 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 *file* di testo 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*, questi *file* possono diventare molto grandi.

File di considerevoli dimensioni sono altamente complessi da gestire da parte degli addetti ai lavori. La problematica più evidente diviene infatti trovare informazioni significative all'interno di questa grande mole di dati. Questo processo richiede infatti tempo e attenzione in situazioni di emergenza, nelle quali il ripristino del sistema deve avvenire nel modo più rapido possibile.

Il problema dei file di grandi dimensioni non riguarda esclusivamente la scansione sequenziale delle informazioni, bensì l'individuazione del messaggio prodotto a fronte di una criticità e la correlazione di quest'ultima allo stato del sistema nell'istante in cui è stata generata.

L'individuazione degli errori non riguarda esclusivamente l'analisi dei log nell'istante in cui il malfunzionamento si è manifestato, bensì richiede di comprendere la catena di eventi pregressi, non sempre palese, che ha

portato il sistema nella condizione di errore. La capacità dell'analista sta nel riconoscere pattern ricorrenti che generano l'intera situazione.

Un'ulteriore complicazione dovuta alla dimensione eccessiva dei file di log non riguarda solo il riconoscimento delle cause di un problema ma anche l'individuazione di criticità simili occorse in istanti temporali differenti. Questo processo, noto come *clustering*, diviene ovviamente oneroso in termini di tempo, al crescere delle dimensione del *log*.

L'analista ha anche un'altra incombenza: verificare che le dimensioni dei file di *log* non eccedano al punto di compromettere il funzionamento dell'applicazione stessa a causa dell'assenza di spazio su disco fisso.

A livello aziendale il tempo che intercorre tra la scoperta di un *bug* e il relativo *bugfix* dovrebbe essere il più possibile contenuto. Spesso purtroppo le eccessive dimensioni dei file di *log* rendono questa procedura molto costosa.

1.2 La centralizzazione

L'azienda che ha ospitato lo *stage* di Tesi, Codice Plastico, è una realtà che opera nel mercato IT sviluppando applicazioni su misura di tipo mobile, web e desktop. Il contesto dal quale è nato lo sviluppo di questo progetto è stata la necessità di trovare la soluzione per centralizzare i *log* in un unico sistema, facilmente accessibile e con un'interfaccia utente *user-friendly*.

Ogni applicazione realizzata e posata dall'azienda, infatti, presenta criticità differenti rispetto alla gestione degli errori:

- Nelle applicazioni web, di norma, i log risiedono su server di proprietà dei clienti, spesso dislocati in aree geografiche differenti.
- Nel caso delle applicazioni mobile, i log risiedono sui device stessi, così come nel caso delle applicazioni desktop.

Tra i numerosi vantaggi della centralizzazione il principale è la riduzione del carico di lavoro dell'analista, il quale può avere accesso ai *log* senza l'onere di doverli attivamente cercare sui sistemi dei clienti.

Inoltre l'aggregazione dei dati permette all'analista di aver accesso ad informazioni già pre-elaborate, come ad esempio il *clustering* di errori simili avvenuti in istanti temporali differenti con l'evidente semplificazione del processo di riconoscimento delle correlazioni causa-effetto.

Il processo di centralizzazione richiede l'inversione del paradigma di segnalazione degli errori. Si passa da una modalità nella quale è l'analista a dover cercare attivamente i file sui sistemi, ad una in cui sono i sistemi stessi ad inviare i propri *log*. A questa logica è facilmente integrabile un sistema di notifiche in tempo reale, con lo scopo di rendere repentina la segnalazione dell'errore e il conseguente intervento di ripristino.

In commercio esistono svariati sistemi e tools che consentono la gestione delle problematiche legate ai log, come ad esempio l'archiviazione, l'analisi, il parsing, il monitoraggio e le segnalazioni. Anche in questo caso la centralizzazione degli strumenti di controllo, evita l'installazione e conseguente manutenzione di numerosi *tools* su ciascun sistema in produzione.

L'effetto evidente della centralizzazione, della semplificazione dell'accesso ai dati e della loro analisi è la riduzione di costi di manutenzione del software, oltre all'incremento della qualità del prodotto offerto.

1.3 Business intelligence

La Business Intelligence o BI indica l'operazione di inferenza di informazioni da una mole di dati, provenienti da fonti differenti nei processi aziendali.

In senso ampio, le fonti di informazioni utili possono essere tra le più disparate, da statistiche di utilizzo dei sistemi informativi, ai dati generati dal funzionamento di software di automazione, al campionamento di flussi di navigazione e modalità di utilizzo dei sistemi.

L'obiettivo della BI è di trarre informazioni e conclusioni utili ai fini aziendali, come l'individuazione delle cause di problemi, la misurazione delle performance, la progettazione di feature che potrebbero incrementare la qualità del prodotto o fare previsioni e stime di scenari futuri, sulla base della storia pregressa.

Poichè i *log* sono strettamente legati al funzionamento delle applicazioni stesse, possono essere utilizzati, oltre che come sistema di controllo dei malfunzionamenti anche come veicolo di raccolta di informazioni utili alla BI.

La BI, infatti, si avvale dei log con diversi fini:

- come strumento per la comprensione e il tracciamento del comportamento degli utenti che operano nel sistema
- ottenere statistiche di utilizzo del sistema, come ad esempio le funzionalità più utilizzate di un'applicazione o quali aree più visitate di un sito internet.

Poichè i dati utili da collezionare variano in maniera significativa da applicazione ad applicazione, da contesto a contesto, l'idea di costruire un sistema di centralizzazione dei log, che non ponga vincoli nella formulazione

degli stessi, va nell'ottica di poter offrire uno strumento utile di raccolta di informazioni diversificate a supporto alle decisioni.

In altre parole, il sistema può diventare un collettore di informazioni relative al funzionamento del sistema ma non necessariamente legate ai concetti di errore e criticità, divenendo di fatto uno strumento per la gestione strategica dei processi aziendali.

Capitolo 2

Log: software e applicazioni

Durante la prima fase del progetto si sono analizzate le soluzioni già proposte dal mercato IT, verificando quali tra i prodotti esistenti presentassero la peculiarità della centralizzazione dei Log.

In questo capitolo analizzeremo alcune tra le principali applicazioni dedicate alla raccolta e analisi dei dati, indicando per ciascuna quali features sono state di ispirazione al progetto, così come quali problemi e svantaggi sono stati identificati.

Benchè il panorama delle soluzioni esistenti fosse ampio, vedremo nel seguito del capitolo, quali sono state le motivazioni che hanno portato alla scelta dell'implementazione di un software proprietario e interno all'azienda.

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

sul mercato esistono svariati sistemi per la gestione centralizzata dei log

Airbrake + aggrega log e ne facilita l'analisi integrato con sistemi di tracking - formati dei messaggi fisso e solo errori

log.io + realtime - no aggregazione

rollbar + non logga solo errori, ma la gestione delle non eccezioni è un o' tirata per i capelli fa aggregazione

https://rollbar.com/vs/airbrake/ questo è mooooolto meglio di mole :(

- soluzione completa ma blindata e non estensibile (gniiiiiii)

papertrail + realtime aggregazione - logga solo tipi di errori noti, no formato flessibile

CACCHIO! fluentd + elasticsearch + kibana = mole

2.2 Una nuova applicazione: Mole.io

ci sono tante soluzioni sul mercato, prima di partire le abbiamo analizzate (stato dell'arte)

perchè lo abbiamo scritto comunque? - avere una soluzione interna e personalizzabile su misure per problematiche dell'azienda - sperimentazione di alcune tecnologie - estensibilità + plugin strutturazione a plugin - tenersi i dati

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

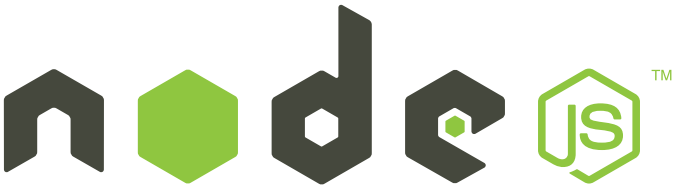

Figura 4.1: Il logo ufficiale di Node.js

La descrizione di Node.js riportata sul sito 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).

Figura 4.2: Il logo di NPM

NPM [2] 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 modu-

lo 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 [3] 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.

Figura 4.3: Il logo di RabbitMQ

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 architetture 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.

RabbitMQ permette di costruire *pattern* di comunicazione tra processi, i più utilizzati sono quelli riportati di seguito.

Work Queues

L'idea alla base questo tipo di configurazione, chiamata anche *Task Queues*, consiste nell'evitare picchi nel carico di lavoro e risorse del sistema, dovuti allo svolgimento di una operazione e all'attesa del suo completamento. Per ovviare a questo problema si configura un sistema nel quale esiste un produttore di dati e uno o più consumatori.

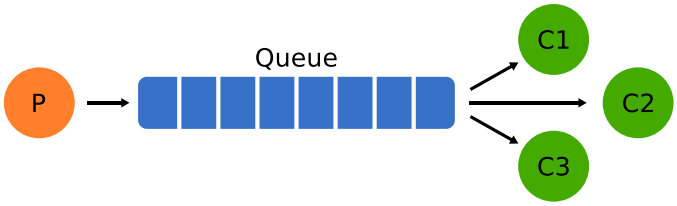

Figura 4.4: Work Queues in RabbitMQ

In figura 4.4 vediamo una rappresentazione schematica di questo tipo di configurazione. All'arrivo di una richiesta di elaborazione, il produttore (P) invia un messaggio sulla coda. I consumatori (C1, C2 e C3) in attesa estraggono un messaggio dalla coda e lo elaborano.

Nella sua configurazione base, la coda esegue un *load-balancing* dei messaggi, questo significa che fornisce esattamente lo stesso quantitativo di

messaggi, e quindi di carico di lavoro, ad ogni consumatore.

RabbitMQ permette di variare il numero di consumatori in ascolto sulla coda dinamicamente, a *runtime*. Non appena un nuovo consumatore si registra per la ricezione dei messaggi, il carico di lavoro viene automaticamente ricalcolato in modo da essere costante per tutti i nodi.

Il load-balancing automatico e la possibilità di sottoscrivere consumatori a runtime diventano funzionalità molto importanti in presenza di sistemi attivi su PaaS. All'aumentare delle richieste, infatti, il sistema potrebbe in automatico attivare nuovi consumatori e sottoscriverli, ottenendo in questo modo un sistema altamente scalabile.

Publish/Subscribe

Questa configurazione si ispira al concetto di *abbonamento*, l'idea di base infatti è poter inviare il medesimo messaggio a più destinatari.

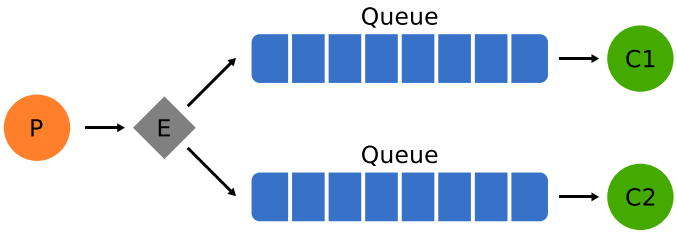

Figura 4.5: Publish/Subscribe in RabbitMQ

La figura 4.5 mostra una rappresentazione del sistema Publish/Subscribe. A differenza del modello *Work Queues*, qui è stato aggiunto un nuovo attore: un *exchange* (E) il cui compito è duplicare i messaggi e inviarli a tutte le code ad esso connesse.

In questo caso il produttore di messaggi (P) invia un messaggio ad un exchange (E), il quale lo duplica e lo invia a tutte le code ad esso connesse.

I consumatori in ascolto (C1 e C2), riceveranno lo stesso messaggio.

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

Il salvataggio dei dati è una operazione critica in moltissimi sistemi software. La scelta di quale database utilizzare per salvare le informazioni è altrettanto delicata e va ponderata alla luce di vari punti di vista. Per la nostra applicazione abbiamo scelto di utilizzare MongoDB. Di seguito cercheremo di illustrare le principali caratteristiche di questo database e le motivazioni che ci hanno spinto a sceglierlo per la nostra applicazione.

Figura 4.6: Il logo di MongoDB

I database relazionali (RDBMS), per loro natura, espongono il loro contenuto in un formato tabellare e utilizzano concetti come righe e colonne per fornire l'accesso ai dati o porzioni di essi. In questo tipo di database lo *schema* dei dati, cioè la loro struttura è ben definita e va studiata in fase di *design* della base di dati, essi si definiscono infatti database *schema-full*. Negli RDBMS la modifica del formato di un dato a sistema avviato è una operazione delicata. Essa infatti deve tenere conto dei dati già presenti all'interno del sistema e deve aggiornarli coerentemente con la nuova struttura assunta dalla tabella che li contiene.

MongoDB è un database *schema-less*, questo significa che la struttura di un dato non è definita a priori, ma soprattutto che essa può variare nel tempo senza richiedere aggiornamenti ai dati già presenti nella base dati. MongoDB utilizza infatti un modello documentale per il salvataggio dei

dati, essi salvati internamente come oggetti *BSON* e forniti all'esterno sotto forma di oggetti *JSON*.

Per comprendere i concetti alla base di MongoDB è possibile realizzare alcune similitudini tra concetti proposti da questa base dati e concetti presenti negli RDBMS. La tabella seguente mostra alcune di queste similitudini.

RDBMS	**MongoDB**
table	collection
tuple	document
column	field

Le *collection* sono insiemi di *document*, i quali, a loro volta, contengono vari *field* che rappresentano le chiavi per l'accesso ai dati veri e propri: i *value*.

Un documento BSON può contenere *field* di vario tipo: interi, stringhe, *array*, dati binari, oppure altri documenti (*embedded document*). Come abbiamo anticipato, in MongoDB, lo schema dei documenti non è fisso, questo significa che nella stessa collection potremo trovare document con struttura differente.

Altre funzionalità significative di MongoDB sono la possibilità di eseguire aggiornamenti atomici dei dati, la ricerca *full-text* all'interno dei documenti e degli *embedded document*, la possibilità di creare indici sui dati e indici di tipo geospaziale.

La possibilità di salvare dati aventi una struttura variabile è stato il motivo principale che ci ha spinto ad utilizzare questo tipo di database, nel capitolo 5.1.2 vedremo come questa funzionalità ci ha permesso di creare un sistema estremamente flessibile.

I creatori di MongoDB hanno fatto in modo che il database da loro realizzato realizzasse due caratteristiche fondamentali, che lo rendono il candidato

ideale per l'installazione in ambienti PaaS: l'alta accessibilità e la scalabilità.

Nella sezione seguente vedremo come MongoDB implementa questi due concetti e come essi si posso utilizzare sia per fronteggiare l'aumento di richieste da parte degli utenti, sia per rendere il sistema resistente a problemi di malfunzionamento dei server sui quali MongoDB viene installato.

Nella sezione 5.1 vedremo inoltre come queste caratteristiche hanno reso MongoDB il candidato ideale ad essere utilizzato all'interno del nostro sistema. Nella configurazione finale, infatti, esso verrà installato proprio su un servizio di tipo PaaS.

4.3.1 Fronteggiare le richieste

Come abbiamo anticipato, MongoDB è un database documentale che garantisce alte *performance*, alta accessibilità e permette facilmente di di scalare la sua struttura per fronteggiare le richieste. Vediamo brevemente come MongoDB ottiene ognuna di queste funzionalità:

Database documentale I documenti contenuti in MongoDB (oggetti) mappano molto bene gli oggetti e le strutture dati fornite dai principali linguaggi di programmazione, rendendo quasi inutile la necessità di un software di traduzione tra strutture dati utilizzate nel software e la loro rappresentazione nel database. Tipicamente i software di *Object-Relational Mapping* (ORM) sono necessari in presenza di database relazionali. Il vantaggio di avere gli *embedded document*, inoltre, permette di ridurre il numero di operazioni di *join* sui dati, rendendo superflua una delle caratteristiche fondamentali dei RDBMS. Com MongoDB diventa quindi molto semplice far *evolvere* le proprie strutture dati, rendendo lo sviluppo dell'applicazione molto più flessibile.

Alte *performance* La possibilità di inserire documenti in documenti, garantisce scritture veloci e la definizione degli indici può includere chiavi presenti negli *embedded documents*, in questo modo è possibile ottenere tempi di risposta del sistema molto contenuti.

Alta accessibilità Questa proprietà, chiamata tecnicamente, *High Availability* (HA), è realizzata con server replicati e organizzati in *cluster*, detti *replica-set*, nei quali è possibile identificare un *master* e altri *slave*. Il master riceve le richieste e le smista sugli slave nel cluster. In caso di problemi al server master, MongoDB esegue una elezione automatica del nuovo master tra gli slave rimanenti.

Facile scalabilità Lo *sharding* è la possibilità di suddividere una collection su più server in modo automatico. Questa caratteristica rende MongoDB altamente indicato per essere installato su piattaforme di tipo PaaS, all'aumentare dei dati presenti nel database, infatti, è sufficiente aumentare il numero di server a disposizione per accogliere più informazioni. Dal punto di vista dell'applicazione che utilizza questi dati, questa operazione è trasparente. In questo modo il numero di server necessari aumenta linearmente all'aumentare della quantità di dati salvata. È possibile aggiungere server in modo dinamico, senza arrestare il sistema, questa funzionalità è particolarmente importante quando MongoDB è utilizzato per contenere dati di applicazioni web che non ammettono momenti di *downtime*.

Oltre alla flessibilità offerta dal modello documentale, MongoDB include le funzionalità comuni degli RDBMS, quali indici, aggregazioni, *query*, ordinamenti, aggiornamenti di dati aggregati e *upsert*, cioè aggiornamento di un dato se già esistente o creazione di un nuovo dato.

Gli sviluppatori di MongoDB hanno cercato di realizzare un database che fosse semplice da installare e manutenere, infatti la filosofia alla base di MongoDB è fare la cosa giusta. Questo significa che il sistema cerca di adattarsi nel miglior modo possibile alla configurazione dell'*hardware* che lo ospita e fornisce un insieme limitato di parametri di configurazione, mantenendo così una interfaccia *user-friendly* verso gli amministratori e permettendo agli sviluppatori di concentrarsi sulle logiche applicative invece di occuparsi della configurazione del database.

Sebbene MongoDB supporti configurazioni *standalone* (o *single-instance*), la configurazione comune di questo database in produzione è quella distribuita. Combinando le funzionalità di *replica-set* e *sharding* è possibile ottenere

alti livelli di ridondanza per grandi basi di dati in modo completamente trasparente per l'applicazione.

4.4 AngularJS e altre tecnologie di frontend

4.4.1 Gestione delle dipendenze

4.5 Strumenti per il deploy

Capitolo 5

Mole.io

In questo capitolo descriveremo nel dettaglio Mole.io: un nuovo sistema per la gestione centralizzata dei log, realizzato come progetto di questa tesi.

Iniziamo immediatamente con la doverosa spiegazione dell'origine del nome di questo software.

Mole è una parola inglese che significa *talpa*. Nel gergo dello spionaggio, la talpa, è un infiltrato che viene inserito in un sistema avversario e cattura informazioni che riferisce all'*intelligence* della sua fazione.

Il nostro sistema, si comporta esattamente come un infiltrato: passa informazioni del sistema nel quale viene inserito, le applicazioni da monitorare, alla sua organizzazione, gli sviluppatori.

Ogni talpa che si rispetti ha alcuni contatti all'interno del sistema che gli riportano le informazioni rilevanti. Abbiamo chiamato questi contatti *mole-contacts* e le soffiate da loro riferite *whispers*.

Nel nostro sistema, i *mole-contacts* sono moduli software che risiedono all'interno dell'applicazione da monitorare, catturano le situazioni significative per il software nel quale operano ed inviano degli *whisper* ad un server chiamato *mole*.

Nell'immaginario collettivo l'infiltrato è una persona ben vestita, con un abito elegante, giacca, cravatta e cappello. Anche il nostro software, in un certo senso, è ben vestito, infatti possiede una interfaccia grafica realizzata per monitorare le applicazioni ed organizzare gli whisper in arrivo. Questo componente si chiama *mole-suit*.

Il progetto al momento è un prototipo, ma il è destinato ad essere ultimato per diventare un servizio vero e proprio fornito via web. in figura 5.1 è riportato il logo proposto per Mole.io.

Figura 5.1: Il logo di Mole.io

Nella sezione seguente vedremo quali sono le funzionalità specifiche di ogni componente in Mole.io, con particolare attenzione al modo nel quale tali componenti scambiano dati tra loro.

5.1 Architettura del sistema

Come abbiamo anticipato nell'introduzione di questo capitolo, Mole.io è organizzato in diverse componenti, ognuna delle quali possiede un compito specifico e interagisce con altre.

Nell'architettura di Mole.io si possono innanzitutto individuare due macro-componenti principali. Nelle sezioni seguenti ne illustreremo la struttura.

Insertion

Questo *layer* dell'applicazione si occupa dell'inserimento dei dati nel sistema e si avvale di diversi moduli per fare in modo che gli whisper vengano salvati, aggregati e organizzati nel database.

Nello schema **??** è riportata l'architettura del layer di insertion, essa è composta da diversi moduli:

mole-contacts

la porzione di sistema che si occupa dell'inserimento dei dati provenienti dall'esterno, è composta a sua volta dai *mole-contacts*, da *mole* e dai *denormalizers*.

insertion è la porzione di sistema che si occupa dell'inserimento dei dati provenienti dall'esterno, è composta a sua volta dai *mole-contacts*, da *mole* e dai *denormalizers*.

presentation si occupa dell'estrazione dei dati dal sistema e della loro presentazione all'utente.

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.

[2] Inc Joyent. Node package manager (npm) website, 2010.

[3] Rabbit Technologies Limited. Rabbitmq website, 2006.