



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. 703156

Dott. Emanuele DelBono

Anno Accademico 2014/2015

Ringraziamenti

Un grazie a ...

Indice

| | |
|--|-----------|
| Introduzione | 3 |
| 1 Log: | |
| Contesti e Problematiche | 6 |
| 1.1 Trattare gli Errori Applicativi | 8 |
| 1.2 La Centralizzazione | 10 |
| 1.3 Business Intelligence | 12 |
| 2 Log: Software e Applicazioni | 15 |
| 2.1 Prodotti e Soluzioni sul Mercato | 16 |
| 2.2 Una Nuova Applicazione: Mole.io | 27 |
| 3 Metodologie di Sviluppo | 29 |
| 3.1 User Story | 33 |
| 3.2 Il Test Driven Development | 36 |
| 4 Tecnologie Utilizzate | 39 |
| 4.1 Node.js | 43 |
| 4.1.1 Le operazioni asincrone | 44 |
| 4.1.2 Le callback | 48 |
| 4.1.3 Npm e Moduli | 52 |
| 4.2 RabbitMQ | 57 |

| | | |
|----------|--|------------|
| 4.3 | MongoDB | 61 |
| 4.3.1 | Fronteggiare le Richieste | 64 |
| 4.4 | AngularJS e Altre Tecnologie di Frontend | 67 |
| 4.4.1 | Gestione delle Dipendenze | 76 |
| 4.5 | Strumenti per il Deploy | 80 |
| 5 | Mole.io | 83 |
| 5.1 | Architettura del Sistema | 85 |
| 5.1.1 | Command Query Responsibility Segregation | 93 |
| 5.1.2 | Mole | 97 |
| 5.1.3 | Mole Suit | 103 |
| 5.2 | Autenticazione degli Utenti | 109 |
| 5.3 | Scalabilità e Affidabilità | 114 |
| 6 | Configurazioni e Benchmark | 119 |
| | Conclusioni e sviluppi futuri | 122 |
| | Bibliografia | 125 |

Introduzione

In questa tesi si descriverà 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, il quale le raccoglie, le cataloga e le aggrega per essere facilmente supervisionate da parte degli sviluppatori.

L'esigenza di una applicazione per la centralizzazione dei log nasce da CodicePlastico [1], una azienda con sede a Brescia, che si occupa di realizzare applicazioni su misura per i propri clienti.

La gestione di un gran numero installazioni dislocate sul territorio e di molte realtà aziendali con esigenze differenti ha reso, per CodicePlastico, particolarmente complesso il tracciamento dello stato di ogni software in produzione. Questa situazione ha spinto l'azienda a decidere di dotarsi di un sistema centralizzato in grado di collezionare i log prodotti dai diversi applicativi, organizzarli e catalogarli in modo automatico.

Il nuovo approccio permette agli sviluppatori di identificare, in breve tempo, il manifestarsi di un malfunzionamento in qualunque applicazione installata presso uno dei propri clienti e reagire rapidamente proponendo una azione risolutiva.

CodicePlastico opera nel settore IT avvalendosi di strumenti software di vario genere, sia proprietari, sia *open source*. Per il *deploy* in produzione di Mole.io, si è scelto di utilizzare *Microsoft Azure*, un sistema *PaaS* distribuito con il quale è possibile creare architetture facilmente scalabili per supportare la variabilità del carico di lavoro richiesto al sistema.

Durante una prima fase di *test*, Mole.io sarà utilizzato per gestire tre o quattro installazioni, che fungeranno da *pilota* per il progetto. La pianificazione dell'azienda, per l'immediato futuro, prevede di aumentare velocemente il numero di clienti attivi nel sistema. Una architettura scalabile, di conseguenza, permetterà di affrontare le esigenze di reattività e stabilità del sistema al variare del carico di lavoro.

Nel primo capitolo saranno trattati 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. Sarà analizzato 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. Verrà descritta ogni applicazione e sarà mostrato 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 verrà mostrato in modo oggettivo, con *benchmark* e *stress test* il comportamento di Mole.io all'aumentare del carico di lavoro e sarà dimostrato come le soluzioni di design applicate garantiscono buone *performance*, anche in condizioni critiche di traffico.

Infine verranno discussi i risultati ottenuti e saranno proposte 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 verranno utilizzati 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 marcitore 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, risulta infatti difficoltosa anche 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 una soluzione per centralizzare i log in un unico sistema, facilmente accessibile e con un’interfaccia utente *user-friendly*.

Ogni applicazione realizzata e installata 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 *tool* 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 tool 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 (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.

Poiché 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
- per 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.

Poiché 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. Un esempio di tool adatto a questo tipo di elaborazioni è SpagoBI [2].

SpagoBI

SpagoBI è una *suite* open-source ideata appositamente per la Business Intelligence. Questo strumento offre molte funzionalità per l'elaborazione dei dati e possiede una serie di tool per l'analisi semantica. La fruizione dei dati da parte degli utenti è molto efficace, SpagoBI offre infatti *widget* grafici per la visualizzazione di dati aggregati e complessi, come grafici e riferimenti geospatiali.

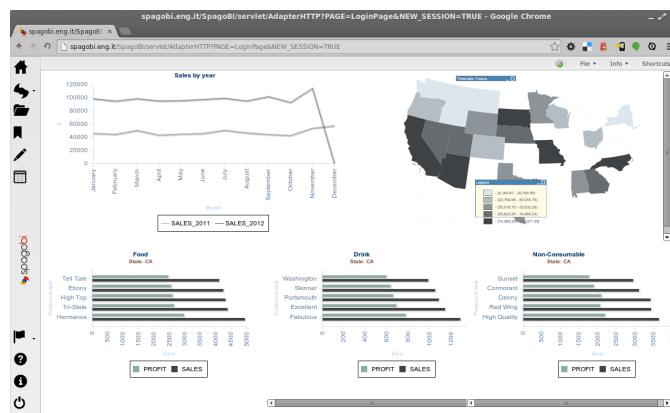


Figura 1.1: Esempio di applicazione realizzata con SpagoBI

SpagoBI possiede una architettura modulare, nella quale ogni componente software possiede un compito specifico e coopera con le altre per realizzare funzionalità complesse.

SpagoBI Server il modulo principale. Il suo compito è offrire le funzionalità di base per la raccolta, l'analisi e il filtraggio dei dati.

SpagoBI Studio fornisce agli sviluppatori la possibilità di modificare i *report* generati da SpagoBI Server secondo le necessità applicative specifiche.

SpagoBI Meta offre funzionalità di elaborazione dei *meta-dati* gestiti dall'applicazione.

SpagoBI SDK è un tool per l'integrazione di funzionalità specifiche realizzate da terze parti, ad esempio servizi web o portali esterni all'applicazione.

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 saranno analizzate alcune tra le principali applicazioni dedicate alla raccolta e analisi dei dati. Per ciascuna soluzione si andranno ad indicare le *feature* di ispirazione al progetto, così come i problemi e gli svantaggi identificati.

Lo stato dell'arte della tecnologia in questo settore è attualmente molto sviluppato: esistono soluzioni complesse e complete che in parte copriranno, con le proprie funzionalità, le necessità individuate in fase di analisi preliminare. Tuttavia si andranno ad analizzare le motivazioni che hanno portato l'azienda ad optare per una soluzione *custom* sviluppata internamente.

2.1 Prodotti e Soluzioni sul Mercato

I prodotti per la gestione centralizzata dei *log* offerti dal mercato sono svariati. Tuttavia, molte di queste soluzioni nascono con l'obiettivo di essere verticalizzate su uno specifico ambito di utilizzo applicativo. Ad esempio alcune soluzioni vengono costruite appositamente per affrontare le problematiche dell'ambito mobile, come altre sono realizzate specificamente per un determinato framework o ambiente di sviluppo.

Per questo motivo sono state escluse dall'analisi soluzioni altamente legate a precisi casi d'uso. Si sono andate, invece, ad individuare le cinque software che affrontano la problematica dei log centralizzati in maniera generica, integrabili con diversi ambienti di sviluppo, e utilizzabili, contemporaneamente, da differenti tipologie di applicazioni.

Airbrake

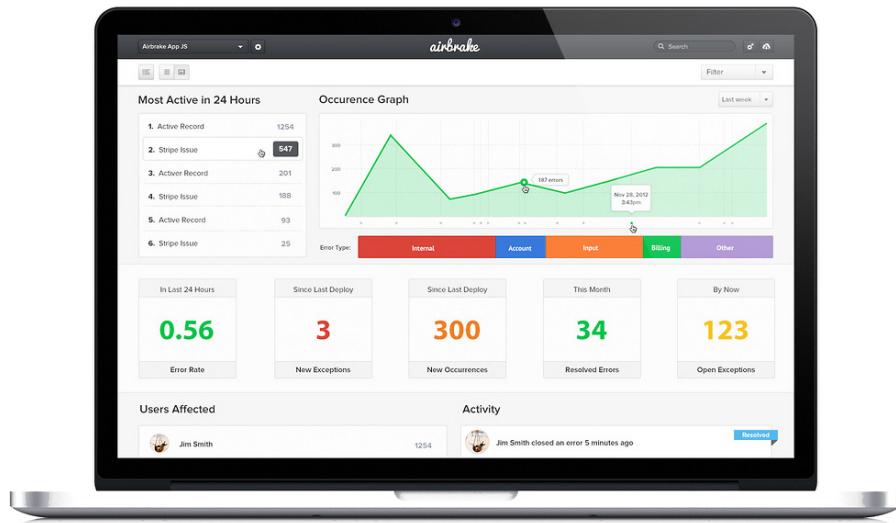


Figura 2.1: Il sito web di Airbrake

Airbrake [3] è una delle più conosciute applicazioni per il monitoraggio dei log prodotti da applicazioni mobile. Benché sia diffusa tra le applicazioni per smartphone e dispositivi mobili, offre moduli di integrazione per i principali linguaggi di programmazione e può essere utilizzata anche in ambito web o desktop.

A partire dalla versione 2.0 propone un pannello di gestione, ricerca e aggregazione dei messaggi di errore completamente rinnovato. L'interfaccia grafica è gradevole e ben organizzata.

La *dashboard* principale dell'applicazione riporta tutti i principali errori ottenuti dai software monitorati, questo aiuta il *team* di sviluppo a concentrarsi sulle problematiche più urgenti, massimizzando il valore applicativo percepito dagli utenti e contribuendo ad aumentare la produttività del team stesso.

Oltre all'aggregazione automatica dei messaggi d'errore, Airbrake possiede moduli di integrazione con i maggiori sistemi di *bug-tracking*. Al momento della ricezione di un messaggio di errore, viene creato *task* a carico di uno sviluppatore che si occuperà poi della gestione della problematica e bugfix.

Airbrake si avvale dell'utilizzo di connessioni sicure SSL al fine di garantire la riservatezza dei dati raccolti.

Il formato dei messaggi di errore interpretato da questo *tool* ha una struttura fissa e non consente l'aggiunta di dati addizionali custom. Questo aspetto rende Airbrake uno strumento poco adatto al tracciamento di messaggi con formati diversi dagli eventi di errore.

Log.io

La peculiarità di Log.io [4] è l'aspetto *realtime*, infatti esso permette di ottenere in tempo reale i log in arrivo dalle applicazioni monitorate.



Figura 2.2: Il sito web di Log.io

Log.io non possiede un sistema di persistenza dei dati, i quali vengono salvati dai diversi sistemi monitorati in modo indipendente. Esso si occupa esclusivamente di fornire un punto di raccolta dei messaggi in arrivo dalle diverse sorgenti e di applicare chiavi di filtraggio in tempo reale.

I log vengono gestiti come flussi di dati. L'analista ha la possibilità di utilizzare dei filtri sui dati con lo scopo di limitare la quantità di informazioni riportate, mostrando i soli dati utili all'analisi. Purtroppo questo sistema non esegue alcun tipo di aggregazione dei dati in arrivo e non permette quindi di avere una visione d'insieme della situazione dell'applicazione monitorata.

L'invio dei messaggi al server, da parte delle applicazioni, avviene tramite messaggi TCP con una formattazione fissa. Questo aspetto contribuisce a

rendere Log.io uno strumento strumento difficilmente utilizzabile quando si vogliono notificare informazioni più strutturate di una semplice stringa di testo.

Rollbar

The screenshot shows the Rollbar website interface. At the top, there's a navigation bar with links for FEATURES, PRICING, DOCS, LOG OUT, and PROJECTS. Below the navigation is a main header with the Rollbar logo and a sub-header 'Take control of your errors'. A sub-copy below it reads 'Rollbar collects and analyzes errors so you can find and fix them faster.' There are two buttons: 'See Live Demo' and 'Explore Features'. To the right of this is a screenshot of the Rollbar dashboard, which displays a list of error items and two charts showing error trends over time. Below the dashboard, there's a section titled 'Built for Faster Cycles' with three columns: 'Collect all your errors', 'Surface the top issues', and 'Find, fix, and resolve'. Each column has a brief description and a small icon.

Figura 2.3: Il sito web di Rollbar

Tra le applicazioni a pagamento censite, questa risulta essere la più completa [5], permettendo infatti di registrare sia messaggi di errore sia messaggi generici.

E' possibile arricchire i messaggi inviati a Rollbar con alcuni dati semplici o strutturati, definibili dall'utente. Le funzionalità di aggregazione dei dati sui tali messaggi, sono disponibili solo ad alcuni campi obbligatori che il sistema è in grado di gestire. Ad esempio è possibile aggregare i dati per

livello di gravità, o per similitudine tra i messaggi, ma non è in grado di fare assunzioni automatiche sui campi custom aggiunti.

Ogni volta un sistema viene pubblicato in produzione, è possibile inviare una notifica a Rollbar. Questo permette agli analisti di correlare gli errori in arrivo con la specifica versione dell'applicazione dalla quale essi sono stati generati.

Rollbar offre, inoltre, la possibilità di aggiungere collaboratori ad un progetto, dando la possibilità ad un intero team di sviluppo di condividere le informazioni relative allo stato delle applicazioni monitorate.

In questa applicazione è possibile configurare la notifica via email per alcuni tipi di messaggi e, infine, possiede una modalità di visualizzazione dei dati *realtime*.

Permette infine di notificare gli eventi aggiungendo automaticamente alcuni *task* nei principali sistemi di *time tracking* e *project management* esistenti sul mercato.

Papertrail

Papertrail è un tool orientato agli amministratori di rete [6]. Nei sistemi *unix-like*, il comando `tail`, permette di stampare in *console* in tempo reale le ultime righe aggiunte ad un file utilizzato da un processo. Analogamente, lo stesso comando in Papertrail, permette di ottenere la coda degli ultimi messaggi ricevuti in tempo reale con una operazione chiamata *Tail and Search*. Papertrail salva periodicamente i log ricevuti e permette di scaricarli, rendendoli così disponibili per eventuali elaborazioni successive.

A livello di analisi automatizzata, il tool, esegue una classificazione dei messaggi ricevuti estrapolando autonomamente la chiave di aggregazione dal formato stesso del messaggio.

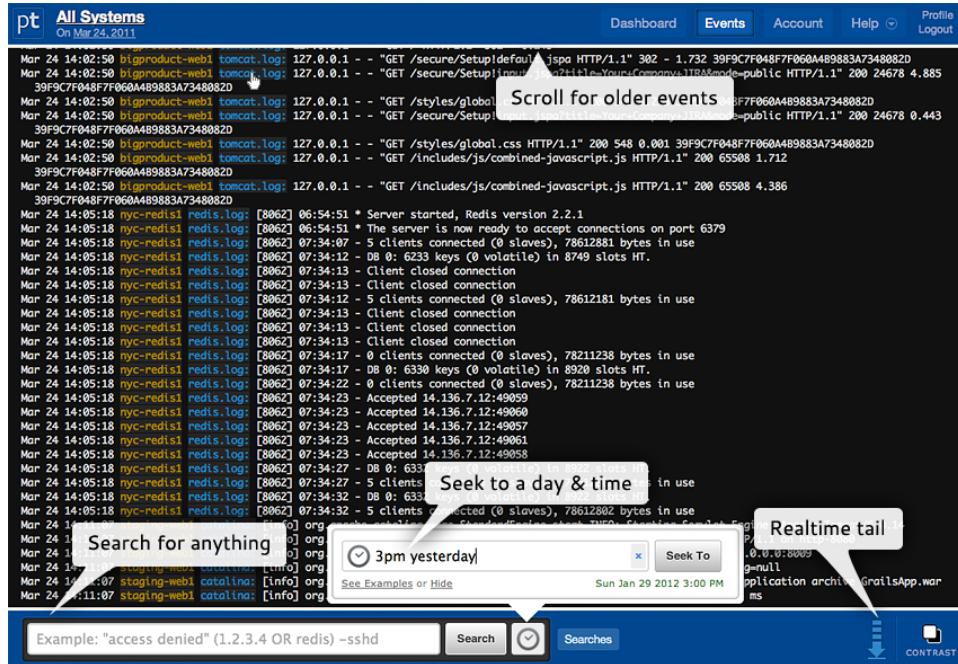


Figura 2.4: Le principali funzionalità di Papertrail

Aggrega, inoltre, messaggi di tipo uniforme, come ad esempio messaggi di riavvio del sistema, eventi di *login* degli utenti, o errori di Apache.

E' possibile utilizzare diversi criteri per il filtraggio dei messaggi di errore.

Tra questi è di particolare interesse il filtro temporale, in cui è possibile selezionare un intervallo di tempo e ottenere l'elenco dei messaggi ricevuti in quel momento.

L'interfaccia di Papertrail facilita il lavoro dell'analista formattando con colori differenti l'elenco dei messaggi ricevuti. I colori sono correlati al testo in maniera semantica (date, indirizzi ip, ecc...).

Dal punto di vista della personalizzazione dei dati, questa applicazione, non consente di aggiungere informazioni strutturate ai messaggi inviati. Questo perché il sistema di aggregazione si basa sull'analisi del formato delle stringhe ricevute. Conseguentemente, non solo Papertrail accetta come

messaggi esclusivamente i formati stringa e non dati strutturati come accade nelle altre applicazioni prese in esame, ma non è possibile aggiungere ulteriori informazioni ai messaggi stessi.

Fluentd, ElasticSearch e Kibana

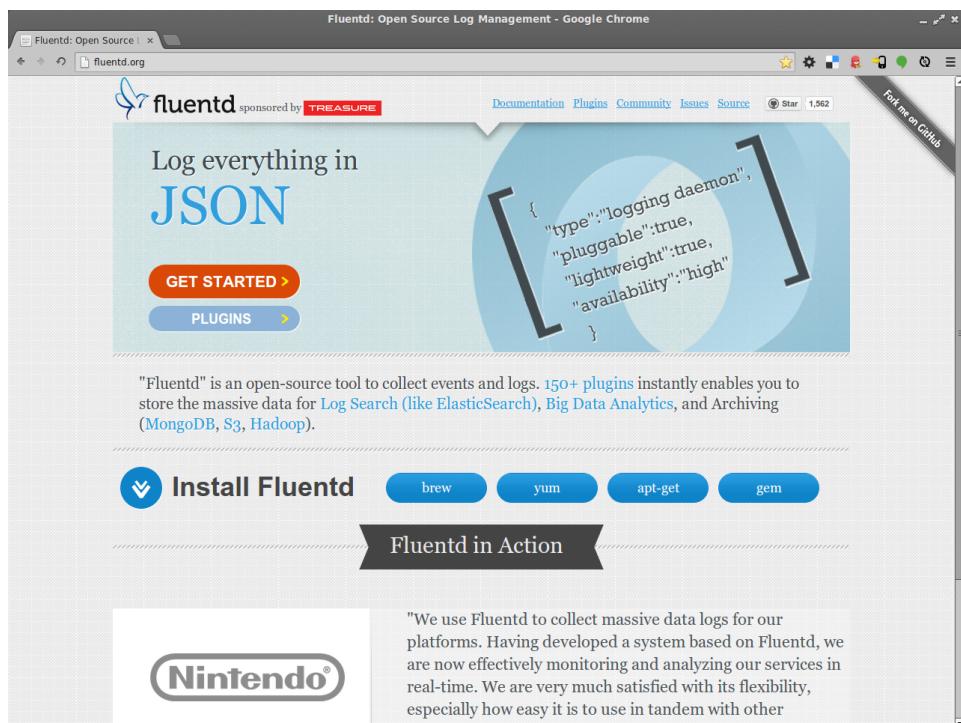


Figura 2.5: Il sito web di Fluentd

Nella analisi comparativa in corso è necessario inserire anche la seguente architettura composta da tre applicazioni che collaborano sinergicamente al fine di creare uno strumento flessibile e completo per la raccolta, l'archiviazione, la ricerca e l'analisi dei log.

Nonostante questa soluzione software sia la più completa architettura *open source* centralizzata per la raccolta di log, non è stata rilevata da una prima fase di ricerca. Questo si suppone sia dovuto al fatto che ciascun applicativo che compone l'architettura è nato con lo scopo di trattare un

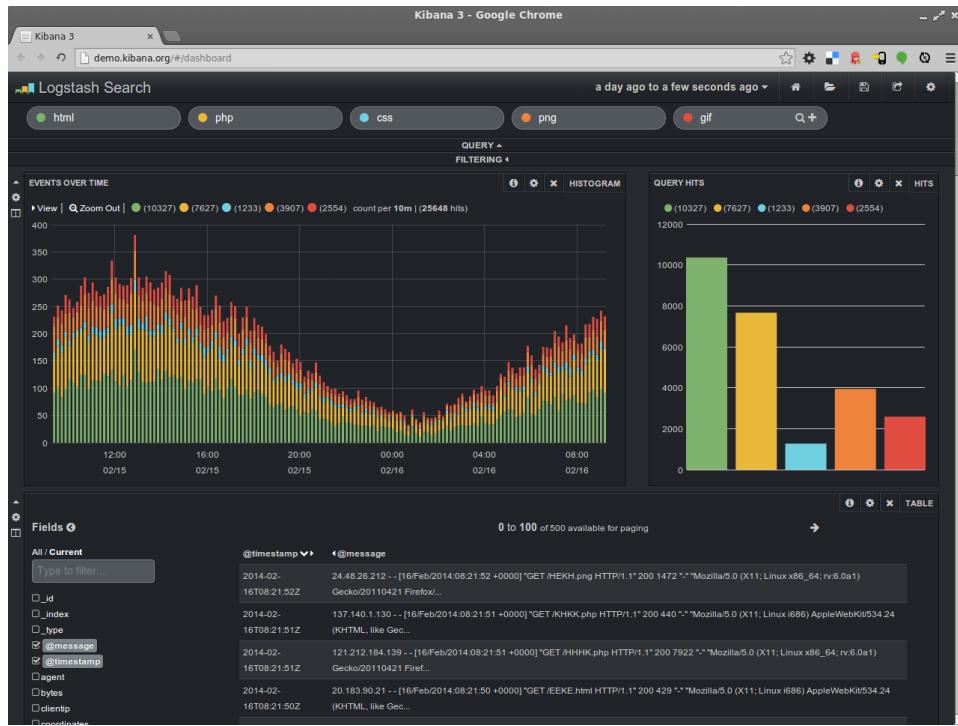


Figura 2.6: Un esempio di interfaccia generata con Kibana

sotto-problema specifico, rispetto al tema della gestione centralizzata dei log.

Fluentd E’ un collettore di messaggi in formato JSON [7]. Sistemi differenti possono inviare dati all’applicazione che si occupa di redirigerli su *output* differenti, come ad esempio server per l’invio di *email*, file di testo, basi di dati o altri servizi. Grazie ad una vasta gamma di plugin, sviluppati dalla *community*, Fluentd può ricevere i più svariati tipi di messaggi che vengono tradotti in JSON dai plugin stessi e in seguito gestiti dall’applicazione. L’intero software è stato ideato per l’installazione su una architettura distribuita, fornendo la possibilità di eseguire un *load balancing* automatico del carico di lavoro dovuto

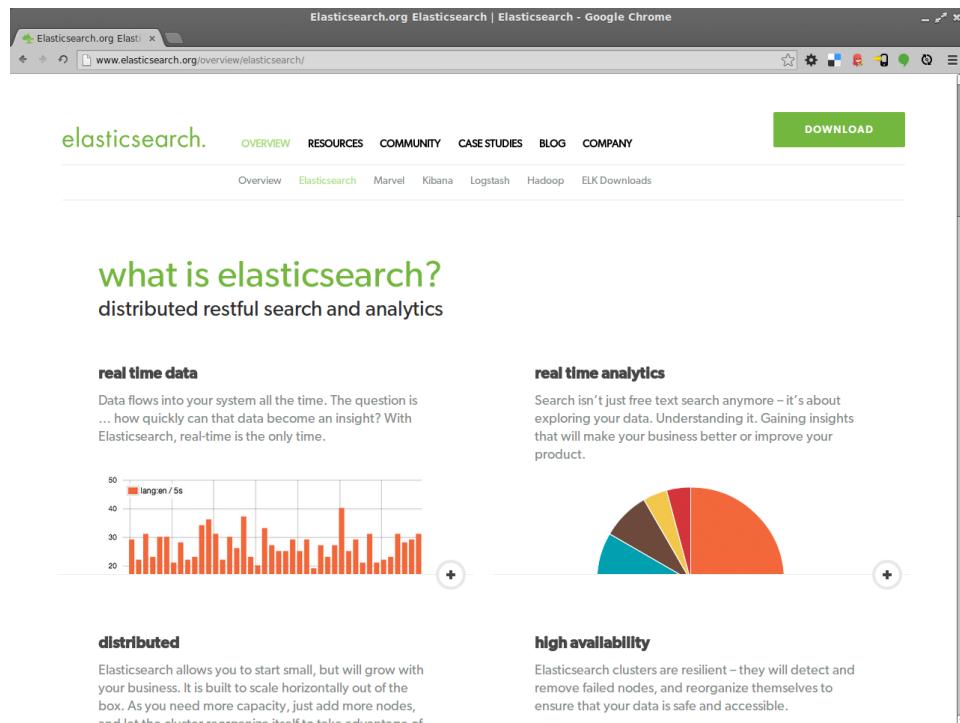


Figura 2.7: Il sito web di Elastic Search

alla ricezione dei messaggi.

Elastic Search E' un sistema distribuito per effettuare ricerche e analisi su grandi quantità di dati [8]. Tra le caratteristiche fondamentali di questo software possiamo annoverare l'alta affidabilità, garantita da un sistema di nodi ridondanti, e la possibilità di lavorare in *realtime*. Espone in maniera automatica una interfaccia software che rende molto agevoli le operazioni di filtraggio e aggregazione dei risultati ottenuti. Ad esempio ipotizzando di voler gestire informazioni di log con la seguente struttura:

```
{
  timestamp: '2014-01-16T20:19:28.871Z',
  user: 'mario rossi',
```

```
    message: 'login error'  
}
```

Elastic Search genererà in automatico alcune chiavi di ricerca che permetteranno di filtrare i messaggi ricevuti per `timestamp`, `user` e `message`. Le interfacce per la ricerca e aggregazione dei dati sono rese facilmente fruibili con l'aiuto di *API REST* anch'esse generate automaticamente a partire dall'analisi della struttura dei dati raccolti. La flessibilità fornita da Elastic Search è ottenibile anche grazie all'aiuto di un database documentale *schema less*, che il sistema utilizza per il salvataggio dei dati applicativi ricevuti.

Kibana Si occupa di mostrare a video i dati ottenuti dall'interrogazione dei Elastic Search. Con questo tool [8] è possibile realizzare, in pochi passaggi, svariate tipologie di grafici e tabelle filtrabili e ordinabili.

L'utilizzo di questi tre tool in maniera sinergica permette di costruire un sistema di gestione e analisi dei log efficiente basato sulla architettura mostrata in figura 2.8. I messaggi, provenienti da sorgenti differenti, vengono raccolti dai plugin di Fluentd e tradotti in formato JSON. A questo punto un ulteriore plugin si occuperà di inviare i dati ad Elastic Search il quale li salverà all'interno del proprio database documentale. Sempre Elasti Search avrà il compito di interrogare la base di dati ed esporre tali informazioni aggregate a Kibana attraverso una *REST API*. Questo quindi permetterà all'analista di creare *report* personalizzabili secondo le esigenze specifiche.

Benché questa architettura presenti caratteristiche e funzionalità aderenti alle esigenze di Codice Plastico, essa richiede una ampia e approfondita conoscenza delle singole applicazioni che la compongono. La frammenta-

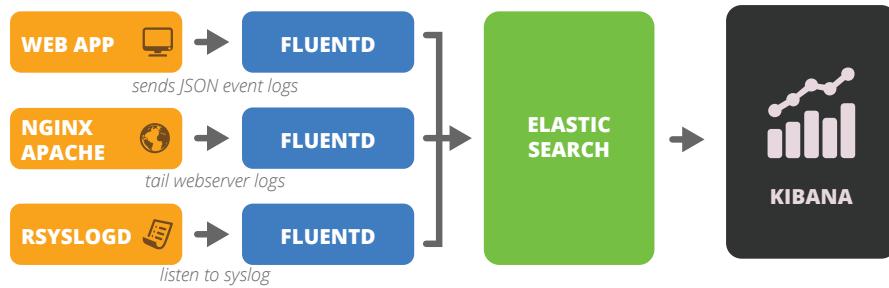


Figura 2.8: L’architettura realizzata con Fluentd, Elastic Search e Kibana

zione delle funzionalità su tre diverse applicazioni, inoltre, rende necessaria un’opera di manutenzione attenta e puntuale su tutte le componenti del sistema (aggiornamenti, compatibilità dei software, ecc...). Questo genera un *overhead* di gestione rispetto alla manutenzione di una singola applicazione.

Riepilogo Comparativo

Nella tabella 2.1 sono riportate schematicamente le principali caratteristiche analizzate nelle soluzioni, per la centralizzazione dei log, presenti in esame.

| | Airbrake | Log.io | Rollbar | Papertrail | ElasticSearch | Kibana |
|--------------------------------|----------|--------|---------|------------|---------------|--------|
| Open source | | ✓ | | | ✓ | |
| Persistenza dati | ✓ | | ✓ | ✓ | ✓ | |
| Dati utente strutturati | | | ✓ | | ✓ | |
| Aggregazione automatica | ✓ | | ✓ | ✓ | ✓ | |
| SaaS | ✓ | ✓ | ✓ | ✓ | | |

Tabella 2.1: Tabella comparativa delle funzionalità

2.2 Una Nuova Applicazione: Mole.io

Il lavoro preliminare di analisi dello stato dell'arte delle tecnologie esistenti sul mercato, svolto in collaborazione con CodicePlastico, ha portato a perseguire la scelta dello sviluppo di una nuova applicazione realizzata internamente all'azienda stessa.

Le motivazioni che hanno determinato questa scelta, sono molteplici:

Business la maggior parte delle soluzioni analizzate è fornita come *Software as a Service* (SaaS). In questo caso sarebbe d'obbligo far adottare tale soluzione ad ogni cliente, perdendo di fatto, il vantaggio di avere un sistema centralizzato di raccolta dei dati.

Riservatezza le soluzioni analizzate che sono fornite come SaaS, prevedono che i messaggi vengano salvati su basi di dati remote e non di proprietà dell'azienda. Questo espone potenzialmente a problematiche di gestione della riservatezza dei dati sensibili che sarebbero più facilmente controllabili utilizzando un sistema *self-hosted*.

Personalizzazione a parte l'ultima soluzione analizzata (Fluentd, Elastic Search e Kibana) la maggior parte dei software presenta difficoltà nella personalizzazione del contenuto dei messaggi, fondamentale per l'azienda, sia per tracciare con precisione problematiche specifiche delle applicazioni in produzione, sia nell'ottica della raccolta dei dati per la Business Intelligence.

Manutenzione limitare gli interventi di manutenzione ad un'unica applicazione proprietaria, della quale, conseguentemente, si ha pieno controllo dello sviluppo.

Estensibilità l'intero sviluppo delle varie componenti dell'applicazione è stato guidato dal concetto di estensibilità. Il sistema risultante dovrà essere facilmente estendibile con nuove funzionalità e *feature* per adattarsi nel modo più aderente possibile alle esigenze di ogni cliente. Un ulteriore aspetto che ha motivato alla costruzione di una nuova applicazione, è la possibilità di rilasciare gradualmente agli utenti le nuove funzionalità introdotte, man mano, nel sistema (*deploy* graduale).

Sperimentazione tra i valori aziendali fondamentali di CodicePlastico vengono annoverati l'aggiornamento costante del personale e il miglioramento delle *skill* di ogni sviluppatore, anche attraverso pratiche agili come *pair programming*, *randori* e progetti personali. Lo sviluppo di una applicazione con l'ausilio di tecnologie relativamente recenti e innovative nel panorama IT, è in completo accordo con la *vision* aziendale.

Capitolo 3

Metodologie di Sviluppo

Per lo sviluppo del progetto di tesi, sono state adottate tecniche di progettazione e sviluppo del software che provengono dall'ambito delle metodologie di sviluppo agili.

Il modello di sviluppo agile è un insieme di pratiche basate sulla costruzione iterativa e incrementale del software. Il flusso di lavoro è organizzato in cicli di breve durata, che hanno come oggetto l'implementazione di una piccola quantità di *feature*. Questa pratica di sviluppo, offre un immediato vantaggio: la possibilità di riesaminare il lavoro effettuato al ciclo precedente e decidere, a seconda delle esigenze correnti, nella successiva iterazione, se continuare lo sviluppo nella stessa direzione o cambiare radicalmente approccio.

Questa possibilità è fondamentale nell'ottica della buona riuscita di un progetto, poiché è molto frequente che, durante lo sviluppo di applicativi software, i committenti decidano, per vari motivi, di sostituire logiche di funzionamento del software. Lo sviluppo iterativo, quindi, mette in condizione il team di lavoro di proporre soluzioni che si adattano nel tempo alle specifiche, generando così un prodotto strettamente aderente alle aspettative

del cliente.

Tra le tecniche operative più frequenti adottate durante il processo di sviluppo del progetto, citiamo:

- pianificazione adattiva
- sviluppo evolutivo
- rilasci frequenti
- *time-boxing*

Lo scopo di queste tecniche è ottenere un modello di sviluppo che possa essere flessibile e adattarsi bene al cambiamento.

Lo Sviluppo Iterativo e Incrementale

Il processo di sviluppo è suddiviso in unità base, con una durata temporale che va da una a quattro settimane. Conseguentemente non è possibile organizzare le attività di sviluppo tenendo conto dei soli obiettivi globali del progetto, ma è necessario ri-contestualizzarli rispetto alla finestra temporale adottata. Gli obiettivi globali vengono così suddivisi in task più piccoli, ciascuno focalizzato su una problematica specifica.

Al termine di ogni unità base, si procede con la successiva, in un processo iterativo. Il risultato di una iterazione dovrebbe essere un prodotto parzialmente funzionante da mostrare al cliente.

Questo approccio minimizza il rischio di portare lo sviluppo fuori dal contesto e permette di avvicinare il cliente al processo di realizzazione del software, rendendolo partecipe dei problemi incontrati e permettendogli di ottenere un prodotto molto aderente alle proprie esigenze.

La Comunicazione Rapida

Le metodologie agili suggeriscono una organizzazione gerarchica, con un ristretto numero di livelli, dei team di sviluppo. Ogni team elegge il proprio *owner* che è responsabile del gruppo di lavoro ed è l'unica interfaccia con i superiori. Anche in questo caso l'esigenza è creare una catena di comunicazione il più corta possibile, in modo da permettere agli sviluppatori di ottenere, in brevissimo tempo, un *feedback* riguardo ai problemi incontrati.

Un altro strumento utile a questo scopo sono gli *stand-up meeting*: riunioni giornaliere brevissime, nelle quali, per salvaguardare la brevità dell'incontro, i partecipanti formano un cerchio stando in piedi. In questi incontri ogni sviluppatore riporta all'*owner* l'elenco dei lavori sul quale è impegnato, stime di tempi per la chiusura delle *feature* in sviluppo, eventuali problematiche incontrate e programmazione delle attività di sviluppo nell'immediato futuro.

Mantenere Alta la Qualità

Le metodologie agili prevedono l'impiego di strumenti software avanzati per supportare le tecniche descritte e facilitare il raggiungimento degli obiettivi prefissati. In [9] e [10] Robert C. Martin descrive alcuni strumenti e varie tecniche utilizzate per perseguire l'obiettivo della qualità del software. Nella trattazione, tra gli altri, sono annoverati:

- *continuous-integration*
- test automatici
- *pair programming*
- *test-driven development* (TDD)

- *design patterns* [11]

- *user stories*

Nelle sezioni seguenti approfondiremo alcune delle tecniche utilizzate durante la progettazione e lo sviluppo del progetto.

3.1 User Story

Le pratiche agili suggeriscono l'utilizzo di *user story* per la descrizione del comportamento del sistema. Una *user story* è una frase, composta utilizzando il linguaggio naturale, che descrive una particolare caratteristica dell'applicazione da implementare.

La struttura della frase da redigere è prefissata e si compone di tre entità fondamentali: *chi*, *cosa* e *perché*. Attraverso questi tre concetti è possibile descrivere il comportamento atteso da un attore nel sistema, umano o software, che esegue una determinata azione, al fine di ottenere un risultato.

Ad esempio, per descrivere la *feature* di autenticazione di un utente nel sistema, si potrebbe procedere nel modo seguente:

Come utente

Voglio poter inserire username e password

Al fine di accedere al sistema

Come mostra l'esempio, le frasi devono essere concise, in modo da rappresentare una funzionalità ben definita del sistema.

Le user story sono la prima fase della progettazione e hanno lo scopo, oltre a quello di stilare una lista condivisa di funzionalità da implementare, di definire un ordine di priorità e la stima dei tempi di realizzazione delle stesse. Un metodo semplice ed efficace per stimare le user story è riportare il tempo di realizzazione di ognuna a fianco della descrizione.

La *user story* è tipicamente scritta su un biglietto adesivo simile a quello in figura 3.1, e incollata ad una lavagna che riporta l'insieme dei requisiti del sistema. Un esempio è la *board* in figura 3.2.

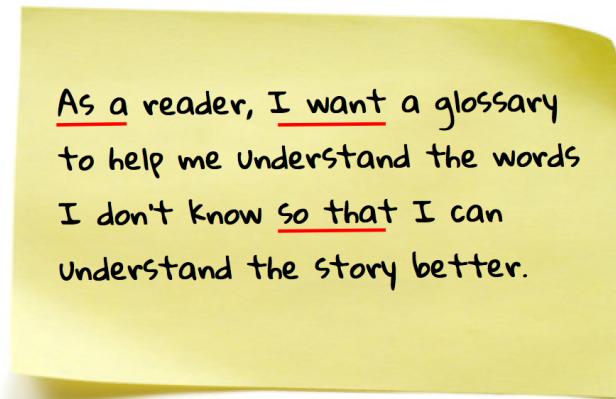


Figura 3.1: Un esempio di *user story*

La creazione delle storie, tipicamente, avviene in collaborazione con il cliente. Durante questo processo l'*owner* lo guida, attraverso domande specifiche, alla stesura e formalizzazione delle funzionalità necessarie.

Ogni storia rappresenta una feature ben precisa sviluppabile in un periodo che va, tipicamente, da alcuni giorni a un paio di settimane.

L'operazione fisica di scrittura della user story, aiuta il cliente a concretizzare la sua idea di progetto, visualizzandone i dettagli del funzionamento. Questo è molto utile sia per gli sviluppatori, sia per il cliente stesso. Infatti, spesso, i requisiti richiesti non sono dettagliati e specifici a sufficienza. Come ulteriore vantaggio si ha la costruzione di un linguaggio comune, condiviso, tra cliente e sviluppatori che rende agevole la realizzazione dell'intero progetto e dei futuri feedback.

Nel caso in cui le esigenze di progetto cambino, è molto facile, in questo processo, sia sostituire le user story, sia rendersi conto di quale impatto comporti il cambiamento, sull'intero sistema.

Come sarà descritto nella sezione successiva, l'utilizzo delle user story si lega a quello del TDD, poiché i *test* diverranno la dimostrazione oggettiva

dell'effettiva implementazione e garanzia di correttezza di ogni storia.

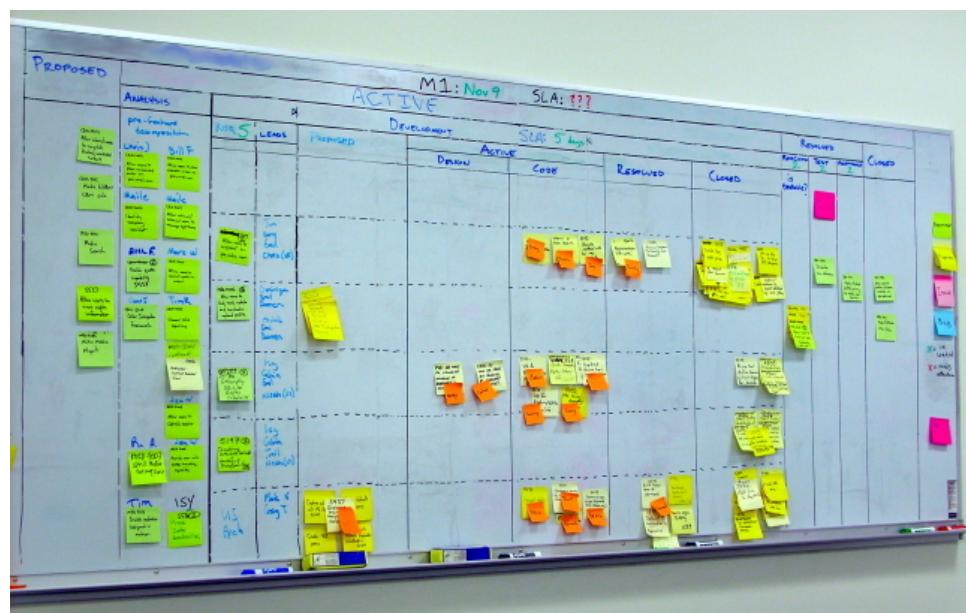


Figura 3.2: Una lavagna con user story

3.2 Il Test Driven Development

Il *test driven development* (TDD) è una tecnica di programmazione che prevede la scrittura di un test, ovvero di una porzione di codice che si occupa di verificare una funzionalità, prima della implementazione della funzionalità stessa.

L'ideazione e la diffusione di questa tecnica di programmazione è stata curata da Kent Beck, che nel suo libro *Test Driven Development: By Example* [12] spiega come applicare il TDD a svariati contesti e dettaglia questa tecnica nelle sue tre fasi fondamentali, definite abitualmente con il termine *red-green-refactor*.

I nomi *red* e *green* delle prime due fasi derivano dalla convenzione, utilizzata dalla maggior parte dei framework di test, di indicare in colore rosso un test fallito e in verde un test andato a buon fine.

Di seguito sono riportati i dettagli di ciascuna fase.

Prima fase: Red

Ogni nuova feature da implementare inizia con la stesura di un nuovo test, questo aiuta il programmatore a definire, a priori, le specifiche funzionali della porzione di sistema che dovrà implementare. La funzionalità descritta dal test può mappare direttamente una *user story* o può essere una porzione di essa.

Poiché la feature oggetto del test non è stata implementata, l'esecuzione del test riporterà esito negativo e, conseguentemente, esso risulterà fallito (rosso).

Questa pratica, a prima vista inutile, ha lo scopo di validare il test stesso. Infatti se il risultato fosse verde, questo significherebbe che esso è superfluo oppure errato.

Seconda fase: Green

Questa fase del processo è quella nella quale si esegue lo sviluppo vero e proprio della nuova funzionalità, in modo da far diventare il test verde (soddisfatto).

L'obbiettivo è esclusivamente soddisfare i requisiti quindi lo sviluppatore, tipicamente, cerca di arrivare alla soluzione nel modo più semplice possibile, non curandosi dei dettagli stilistici del codice.

Terza fase: Refactor

L'ultimo passo da eseguire è la rifattorizzazione del codice, cioè la riorganizzazione delle funzionalità realizzate al passo precedente al fine di ottenere una architettura ben congegnata e una struttura chiara.

Una delle tecniche principali per realizzare questo compito è chiamata *Don't Repeat Yourself* (DRY) e consiste nel cercare di rimuovere quanto più possibile il codice duplicato mantenendo le medesime funzionalità.

L'utilizzo di questa tecnica favorisce il riuso e, tipicamente, veicola verso un buon design del software. In questa fase lo sviluppatore ha la massima libertà di sperimentazione, in quanto la correttezza del software rifattorizzato è garantita dai test scritti in precedenza.

Il processo del TDD è iterativo, al termine della fase di refactoring, esso ricomincia da capo, con la stesura di un nuovo test o la riformulazione di test esistenti. Il grafico 3.3 riporta schematicamente l'andamento circolare di questo processo.

La scelta di utilizzare il TDD all'interno di un progetto software garantisce la corretta implementazione delle user story. L'adozione di questa tecnica è, inoltre, facilitata dalla esistenza di innumerevoli framework di test per la maggior parte dei linguaggi di programmazione esistenti.

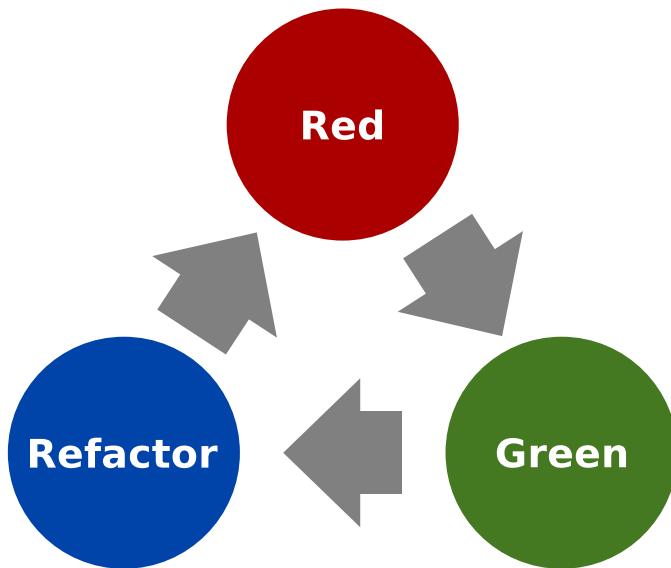


Figura 3.3: Il flusso di lavoro circolare del TDD

L'utilizzo del TDD fornisce agli sviluppatori svariati vantaggi, alcuni dei quali sono:

- riduzione drastica della necessità di un *debugger* per la ricerca degli errori;
- acquisizione di fiducia nella sperimentazione, poiché l'impatto dell'adozione ogni soluzioni è sempre validato dai test e quantificabile;
- documentazione del codice per gli altri sviluppatori del team, i test forniscono infatti esempi di utilizzo delle diverse funzionalità implementate;
- tracciamento dello stato di avanzamento dei lavori;
- modularizzazione della struttura del progetto, tramite la suddivisione delle feature in unità indipendenti e cooperanti tra loro;
- *focus* su una singola funzionalità;

Capitolo 4

Tecnologie Utilizzate

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

Il primo aspetto interessante dello sviluppo di Mole.io è il linguaggio di programmazione utilizzato per realizzarlo. L'intera applicazione infatti è stata sviluppata utilizzando il linguaggio *JavaScript* (JS).

JavaScript è un linguaggio di *scripting* dinamico, comunemente utilizzato, all'interno del browser, come supporto alla realizzazione di pagine web. Le sue applicazioni sono svariate: dall'animazione di porzioni di interfaccia grafica, alla comunicazione asincrona con il server, alla realizzazione di interi giochi all'interno del browser.

JS è un linguaggio *prototype-based*, utilizza tipizzazione dinamica delle variabili, e presenta funzioni *first-class*, è possibile infatti passarle come parametri, assegnarle ad una variabile e restituire valori da una funzione.

Questo è un linguaggio molto contaminato, la sua sintassi, infatti, si ispira a quella del C e di Java, ma utilizza alcuni principi di design provenienti

dai linguaggi Lisp e Scheme [13]. Questo fa sì che JavaScript possa supportare differenti stili di programmazione, è infatti adatto ad essere utilizzato sia come linguaggio imperativo, sia Object Oriented, sia funzionale [14].

Negli ultimi anni il mondo IT ha assistito ad una serie di utilizzi alternativi di JavaScript, ci sono state infatti applicazioni di questo linguaggio negli ambiti *desktop*, *mobile* ed *embedded*.

L'aspetto più interessante, ai fini della realizzazione del progetto di tesi, è l'utilizzo di JS per la realizzazione di applicazioni *server-side*. Questo approccio permette di ottenere un intero *stack* applicativo uniforme che facilita la manutenzione dell'applicazione stessa.

Uniformare il linguaggio utilizzato permette di facilitare lo sviluppo e la fruibilità del progetto da parte degli sviluppatori. Per poter lavorare al software è infatti richiesta solo la conoscenza di JavaScript e non di altri linguaggi. Questo è un ottimo requisito quando si lavora con un team di sviluppo in espansione.

La piattaforma Node.js [15] permette di utilizzare JavaScript per sviluppare la parte *backend* delle applicazioni.

Per il salvataggio dei dati, la scelta è ricaduta su MongoDB [16], un database documentale *schema-less*, che permette di salvare dati direttamente in formato *JavaScript Object Notation* (JSON). La possibilità di salvare nel database strutture dati gestite in modo nativo da JavaScript facilita notevolmente il lavoro di gestione delle informazioni.

Il *deploy* in produzione di Mole.io verrà eseguito in un ambiente di tipo PaaS. Sistemi di questo tipo sono caratterizzati dalla possibilità di fornire all'utente *container* virtuali o macchine virtuali nelle quali eseguire le applicazioni.

Quando si realizza il design di applicazioni per sistemi PaaS, quindi, è

importante riuscire ad identificare i sotto-componenti software e gli specifici ruoli e compiti di ciascuno di essi. I sotto-sistemi devono quindi essere realizzati in modo da cooperare tra loro. Il *disaccoppiamento* dei diversi servizi permette di scalare il sistema in modo da adattarne la configurazione alle specifiche esigenze in ogni istante.

Una volta determinate le diverse componenti del sistema è necessario metterle in comunicazione tra di loro in modo da permettere la cooperazione e lo scambio di informazioni. A questo scopo si è deciso di introdurre nell'architettura applicativa RabbitMQ [17], una piattaforma per la gestione di code di messaggi, che permette lo scambio di dati tra le diverse componenti del sistema.

Il panorama attuale delle tecnologie per la realizzazione di applicazioni web *client-side* è vastissimo. Per la costruzione di Mole.io la scelta è ricaduta su AngularJS, un popolare *framework* sviluppato da Google appositamente per la creazione di *single-page application*. Questo strumento è stato scelto principalmente per la sua naturale attitudine a lavorare con interfacce di *backend* di tipo REST.

Per lo sviluppo dell'interfaccia grafica si è scelto di utilizzare Bootstrap [18], un framework CSS che facilita la realizzazione e la stilizzazione di pagine web fornendo un *set* di classi CSS preconfigurate. Questo strumento ha permesso di realizzare velocemente un prototipo dell'applicazione con una interfaccia grafica decorosa. Le varie librerie JavaScript necessarie per il funzionamento dell'interfaccia sono state organizzate utilizzando Bower [19], un tool per la gestione delle dipendenze.

Come strumento per il deploy dell'applicazione, si è scelto di utilizzare Dokku [20]. Dokku è un software che permette di eseguire la posa in produzione dell'intera applicazione con un singolo comando lanciato direttamente

dalla directory del repository di lavoro.

Nelle sezioni seguenti saranno mostrate nel dettaglio alcune delle tecnologie utilizzate per realizzare e mettere in produzione Mole.io.

4.1 Node.js

La *homepage* del sito ufficiale di Node.js [15] fornisce una sintetica ma precisa descrizione di questa tecnologia, un buon punto di partenza per illustrarne le peculiarità.

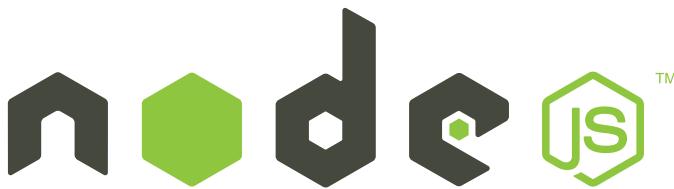


Figura 4.1: Il logo ufficiale di Node.js

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.

E' immediato comprendere 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 di una versione appositamente riadattata del potente interprete V8 presente all'interno del browser *Chrome* di *Google*. L'utilizzo di un linguaggio altamente diffuso e di una base solida come quella fornita dal popolare *browser* permettono di costruire applicazioni affidabili in modo semplice e veloce.

L'ultimo importante concetto, che si apprende 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 facilmente 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 blocca 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*.

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

4.1.1 Le operazioni asincrone

Interessante, a fronte dell'introduzione del concetto di I/O asincrono, è vedere 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.

L'utilizzo dei thread permette ad un processo di parallelizzare l'esecuzione di una parte del suo flusso di lavoro. Al tempo stesso sono necessarie risorse di sistema sia per creare il thread sia per distruggerlo. Node.js ovvia a questo problema utilizzando un *thread pool*, cioè un insieme di thread

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.

Per comprendere come Node.js sia in grado di realizzare tale funzionalità, è necessario spiegare nel dettaglio il modello di gestione delle richieste implementato da questo framework.

Innanzitutto è necessario chiarire che Node.js, a differenza di altri sistemi per lo sviluppo *server-side* non necessita di una applicazione che funga da *web server*, come accade ad esempio nel caso di Apache per PHP. Quando si sviluppa in Node.js infatti l'applicazione realizzata è il server. Il framework mette a disposizione funzionalità apposite per la creazione di un server interno ad ogni applicazione. Questo approccio favorisce la strutturazione a *servizi* dell'applicazione, che può essere suddivisa in processi Node.js separati e quindi in veri e propri server in comunicazione fra loro.

Di seguito è riportato un server web minimale realizzato in Node.js.

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

http.createServer(function (req, res) {
  res.end('Hello World');
}).listen(3000, '127.0.0.1');

console.log('Server running at http://127.0.0.1:3000/');
```

La prima operazione eseguita dal programma è l'importazione del modulo `http`, che fornisce le funzionalità di rete. Successivamente viene chiamata la funzione `createServer()` che si occupa di generare il server web. Utilizzando la funzione `listen()`, successivamente, il server viene attivato e messo in ascolto all'indirizzo `http://127.0.0.1:3000/`. Il comando `console.log()` stampa in console l'informazione che il server è avviato. Ad ogni richiesta ricevuta dal server, viene eseguita la funzione passata come parametro alla `createServer()`, la quale risponde alla richiesta inviando al client la stringa `Hello World`.

Come si vede da questo esempio, con Node.js, la logica applicativa e il *web-server* risiedono all'interno dello stesso software.

Node.js utilizza internamente un modello chiamato *Event Loop* per la gestione delle richieste in arrivo. All'avvio dell'applicazione, vengono attivati un thread principale e un insieme finito (il default è quattro) di thread secondari definiti *thread-pool*. Nell'istante in cui arriva una nuova richiesta da parte di un client, il thread principale la prende in carico ed inserisce i dati della richiesta e la funzione *callback* per gestirla in una coda. A fronte di questa operazione, viene attivato il primo thread libero nel thread-pool, detto *worker*, il quale esegue la callback e raccoglie il risultato. Una volta generato il messaggio di risposta da inviare al client, il worker lo restituisce al thread principale, il quale provvede ad inviarlo al client.

Questo approccio presenta un grande vantaggio rispetto ad altri sistemi non *event-based*: il thread principale è quasi sempre in stato *idle*. Esso si occupa infatti esclusivamente di smistare richieste e raccogliere risposte, quindi rimane sempre reattivo. Il risultato di questo approccio è un sistema responsivo, che riesce a gestire un grande numero di richieste sfruttando una bassissima quantità di risorse di sistema.

In figura 4.2 è illustrato schematicamente l'event-loop per la gestione delle richieste di Node.js.

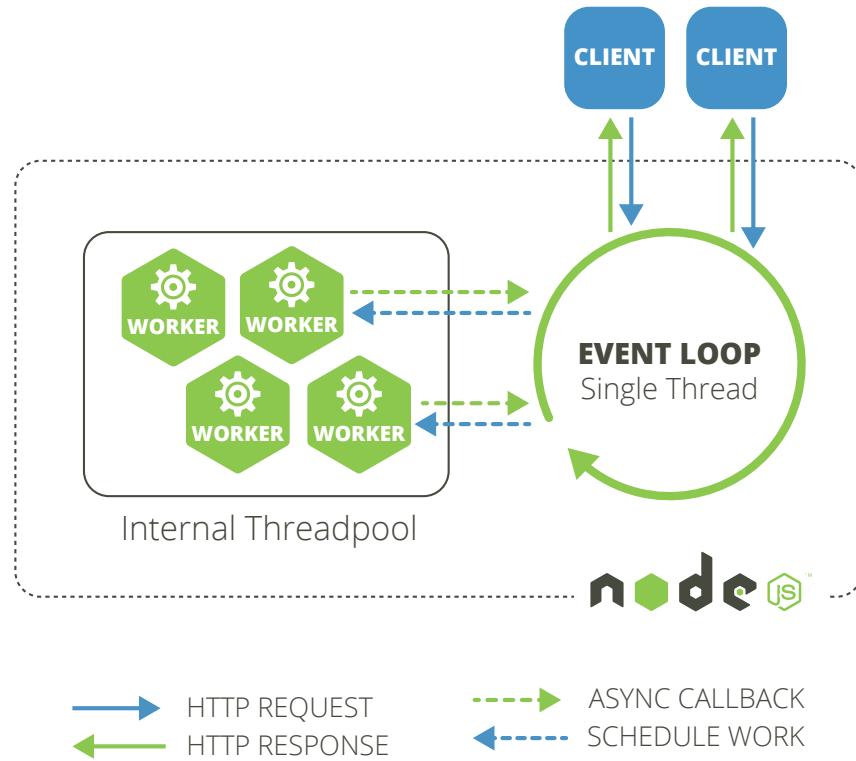


Figura 4.2: L'*event loop* di Node.js

4.1.2 Le callback

Una delle principali difficoltà che si presenta ad uno sviluppatore che si appresta ad utilizzare 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 effettuato 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 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 4.3: Il logo di NPM

NPM [21] 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 di NPM più utilizzati è `npm install`. Il comando indica a NPM di leggere il file `Package.json` presente nella *directory* corrente e scaricare dalla rete tutti i pacchetti richiesti alle rispettive versioni. In questo modo, l'utilizzatore del modulo è 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.

Di seguito verranno presi in considerazione alcuni dei principali moduli utilizzati per realizzare l'applicazione oggetto della tesi.

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 realizzare filtri molto complessi. La configurazione delle rotte e dei middleware di express verrà illustrata 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, sarà approfondito 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'applicazione. Nella sezione 4.3 verrà illustrato nel dettaglio questo database documentale e si mostrerà 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 ?? si analizzerà nel dettaglio come è stata sfruttata 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. Tale strumento è stato largamente utilizzato durante la realizzazione del sistema, applicando le tecniche illustrate nel capitolo 3.2, per rendere Mole.io, 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 [17] 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 possono comunicare tra loro scambiandosi messaggi attraverso le code utilizzando un protocollo determinato dallo sviluppatore.



Figura 4.4: 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 di 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.

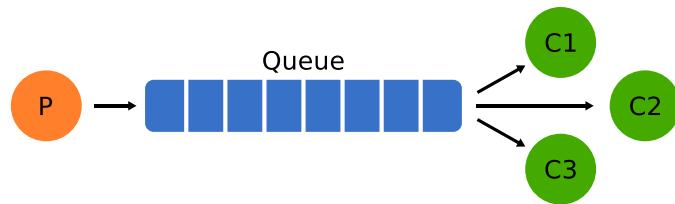


Figura 4.5: Work Queues in RabbitMQ

In figura 4.5 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.

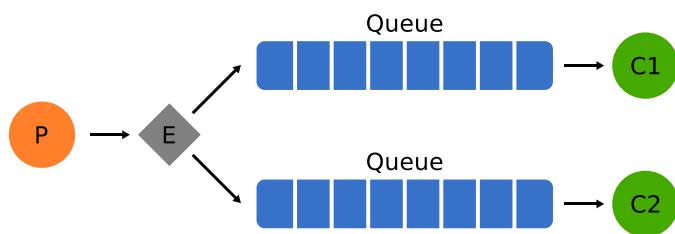


Figura 4.6: Publish/Subscribe in RabbitMQ

La figura 4.6 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 5.1.2 sarà illustrata nel dettaglio la configurazione di RabbitMQ utilizzata per realizzare Mole.io e quali tecniche si sono messe 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 del tipo di database da utilizzare per salvare le informazioni è altrettanto delicata e va ponderata alla luce di svariati punti di vista. Per la realizzazione della applicazione oggetto di tesi, si è scelto di utilizzare MongoDB [16]. Di seguito verranno illustrate le principali caratteristiche di questo database e le motivazioni che sottendono tale scelta.



Figura 4.7: 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 struttura delle informazioni 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 la rappresentazione dei dati,

essi sono 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 documenti con struttura differente.

Altre funzionalità significative di MongoDB sono:

- la possibilità di eseguire aggiornamenti atomici dei dati;
- la ricerca *full-text* all'interno di documenti e di *embedded document* ;
- la possibilità di creare indici su dati;
- la creazione di indici di tipo geospaziale;

Si è scelto questo tipo di database perché fornisce la possibilità di salvare dati aventi una struttura variabile. Nel capitolo 5.1.2 si vedrà come questa funzionalità permette di creare un sistema estremamente flessibile.

I creatori di MongoDB hanno fatto in modo che il database da loro realizzato possedesse due caratteristiche fondamentali, che lo rendono il candidato ideale per l'installazione in ambienti PaaS: l'alta accessibilità e la scalabilità.

Nella sezione seguente si illustrerà come MongoDB implementa questi due concetti e come essi si possano 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.

La sezione 5.1.2 fornirà inoltre elementi per comprendere come queste caratteristiche hanno reso MongoDB lo strumento più adatto ad essere integrato nell'applicazione oggetto di tesi. Nella configurazione finale, infatti, esso verrà installato proprio su un servizio di tipo PaaS.

4.3.1 Fronteggiare le Richieste

Come è stato anticipato, MongoDB è un database documentale che garantisce alte *performance*, alta accessibilità e permette facilmente di scalare la sua struttura per fronteggiare le richieste. Di seguito sarà descritto brevemente come MongoDB realizza 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 all'interno di altri 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

AngularJS [22] è un framework open-source per la creazione di *single-page application*. Una applicazione single-page è una pagina web che carica dinamicamente dati dal server attraverso l'utilizzo di comunicazioni asincrone. Dal punto di vista dell'utente utilizzatore di questo tipo di applicazioni, la principale differenza con le comuni applicazioni web è la modalità di navigazione. Muovendosi da una pagina all'altra, infatti, il client non richiede nuove pagine al server, bensí esegue richieste in *background* e carica dinamicamente i nuovi contenuti all'interno della pagina corrente con l'utilizzo di JavaScript.

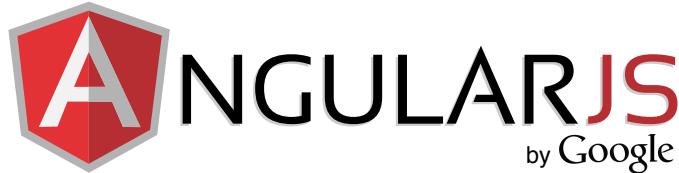


Figura 4.8: Il logo di AngularJS

Il principale vantaggio offerto dalle applicazioni single-page si manifesta durante operazioni di gestione dei dati utente. Nelle comuni applicazioni web, infatti, i dati di sessione sono passati da una pagina alla successiva utilizzando svariate tecniche. Questa operazione è resa necessaria dalla natura *state-less* del protocollo HTTP. Le single-page application risolvono alla radice il problema evitando il cambio di pagina e inviando i dati di autenticazione al server solo quando l'utente esegue operazioni per le quali è richiesta la sua identificazione.

L'utilizzo di AngularJS permette inoltre di realizzare applicazioni web utilizzando un pattern *Model-View-ViewModel* (MVVM), semplificando mol-

to la testabilità del sistema. Una delle grandi problematiche di cui risentono le applicazioni web classiche infatti è la possibilità di testare il codice che le compone. Solitamente questa difficoltà nasce dal fatto che la logica applicativa e quella di visualizzazione sono codificate simultaneamente e quindi strettamente legate tra loro.

Il pattern MVVM facilita il processo di *testing* delle applicazioni separando nettamente le competenze di ogni componente software che partecipa nel sistema. Questo pattern si basa sulla identificazione di tre componenti fondamentali:

Model è la parte del sistema che si occupa di trattare i dati, spesso è implementata impiegando librerie per l'accesso a database oppure servizi esterni utilizzati come sorgenti di dati per l'applicazione.

View è l'interfaccia grafica vera e propria. Nel caso di applicazioni web si identifica spesso con le porzioni di codice HTML e quello CSS utilizzato per definire lo stile.

ViewModel è simile ad un *controller*, implementa la logica applicativa, anche definita *logica di business*. In questo strato software è possibile trovare algoritmi per la manipolazioni dei dati specifici dell'applicazione e funzionalità per la gestione delle interazioni dei diversi servizi applicativi.

Le tre componenti del pattern MVVM cooperano tra loro scambiandosi messaggi secondo interfacce definite in fase di design dell'applicazione. La figura 4.9 mostra le principali interazioni richieste per permettere la completa collaborazione di tutte le componenti software.

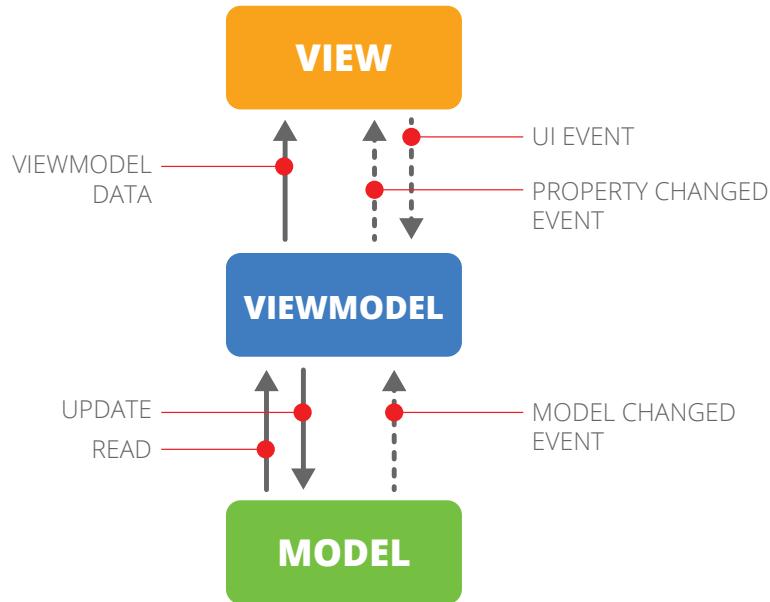


Figura 4.9: Schema delle comunicazioni nel modello MVVM

Questo approccio porta ad avere Model e ViewModel popolati con la maggior parte del codice applicativo, che, di conseguenza, viene rimosso dalla View. Il vantaggio dell'utilizzo dell'MVVM risiede proprio nel fatto che la sezione View, complessa da testare, diventa molto semplice e di fatto è quasi inutile eseguire test sul codice che la genera.

La principale funzionalità fornita da AngularJS è il *binding* dinamico delle variabili JavaScript con i controlli HTML presenti nella pagina. Il binding è una operazione che permette di collegare una parte di Model con una porzione definita di View. Molto spesso questa funzionalità è utilizzata per collegare una variabile JavaScript con un controllo HTML, il binding garantisce che quando la variabile assume un nuovo valore, questo sia mostrato sulla interfaccia e, viceversa, quando un utente modifica il valore nella vista, questo venga immediatamente riflesso sulla variabile nel modello. Questo

tipo di binding è definito *bidirezionale* o *two-way data binding*.

Il binding è una funzionalità molto interessante, perché solleva il programmatore dall'incombenza di mantenere la coerenza tra dati e interfaccia. Automatizzando questa operazione si ottiene anche una semplificazione del codice e una maggiore garanzia di funzionamento dell'applicazione.

Per implementare il binding bidirezionale, AngularJS si serve di nuovi costrutti che arricchiscono il vocabolario a disposizione del programmatore per la stesura del codice HTML. Segue un esempio di codice contenente i costrutti AngularJS per realizzare un binding.

Contenuto del file `view.html`:

```
<!doctype html>

<html ng-app='greetingsApp'>
  <head>
    <script src="angular.min.js"></script>
    <script src="app.js"></script>
  </head>
  <body ng-controller="GreetingsCtrl">
    Hello {{subject}}!
  </body>
</html>
```

Contenuto del file `app.js`:

```
angular.module('greetingsApp', [])
  .controller('GreetingsCtrl', function ($scope) {
    $scope.subject = 'World';
});
```

Esaminando il codice HTML del file `view.html` nell'esempio, si nota innanzitutto la necessità di importare libreria JavaScript `angular.min.js` nel-

la pagina web. Questa operazione permette di utilizzare AngularJS all'interno della pagina stessa. Per attivare la libreria all'interno della pagina web è, inoltre, necessario specificare l'attributo `ng-app`. All'applicazione nell'esempio è stato dato nome `greetingsApp`.

AngularJS necessita di uno *scope* nel quale monitorare le variabili che fanno uso del binding. Esso è definito con il l'attributo `ng-controller`. Nell'esempio riportato lo scope sarà esteso a tutto il contenuto del tag `body`. All'interno dello scope, si identificano le variabili della view agganciate al modello con l'utilizzo di una doppia parentesi graffa `{::}`.

Nell'esempio, la logica dell'applicazione risiede nel file `app.js`, infatti al suo interno è possibile identificare una dichiarazione di controller e una funzione che ne implementa il comportamento. Nel caso riportato il comportamento è molto semplice, si limita all'assegnamento di una variabile utilizzata come modello.

AngularJS fornisce un insieme piuttosto nutrito di librerie di utilità che risolvono i più comuni problemi di comunicazione con sorgenti esterne di dati. Mole.io utilizza una di queste librerie per mettere in comunicazione i modelli AngularJS con una API REST presente sul server che si occupa di fornire i dati necessari a popolare le variabili. Non appena viene eseguito l'aggiornamento di tali variabili, AngularJS provvede ad aggiornare l'interfaccia utente con i dati ottenuti utilizzando del codice molto simile a quello riportato nell'esempio.

La struttura di questo framework, per sua natura, induce ad un design delle applicazioni per *componenti* indipendenti ma comunicanti. E' possibile infatti immaginare una pagina web contenente diversi costrutti `ng-controller`, ognuno dei quali si occupa di aggiornare una porzione della View con logiche specifiche. Non è difficile comprendere come questo ap-

proccio possa favorire la scrittura di software di buona qualità, attraverso il riuso di componenti e lo studio attento della separazione delle responsabilità di ogni componente.

AngularJS utilizza il formato *JavaScript Object Notation* (JSON) per lo scambio di dati tra il backend e i modelli. In [23] Douglas Crockford illustra questo formato da lui ideato. La definizione di JSON è aderente alle specifiche ECMA-262 del Dicembre 1999, e permette a JavaScript di interpretare questo formato nativamente. JSON quindi è un formato facilmente maneggiabile sia lato frontend, sia lato backend, questo ha contribuito a farlo diventare molto famoso e utilizzato per la realizzazione di applicazioni Node.js.

Per la realizzazione della parte frontend di Mole.io sono state utilizzate diverse librerie JavaScript e framework di supporto alla stesura del codice HTML. Dell'insieme delle librerie utilizzate, si sono estratte le principali, delle quali seguirà una descrizione:

- Leaflet
- D3.js
- Bootstrap

Leaflet e D3.js

Uno degli scopi per i quali è stata creata l'applicazione Mole.io è fornire una visione immediata della situazione nella quale le si trovano i sistemi monitorati. Per rendere a *colpo d'occhio* fruibili le informazioni si è pensato di utilizzare mappe geografiche e grafici. Le librerie *leaflet* e *d3.js* hanno rispettivamente questi compiti.



Figura 4.10: Il logo di Leaflet



Figura 4.11: Il logo di D3.js

leaflet [24] è una libreria JavaScript che si occupa di gestire e mostrare a video mappe geografiche. La principale particolarità di questo software è il fungere da *layer di astrazione* rispetto ai vari sistemi per la gestione di mappe geografiche esistenti. L'utilizzo di plugin, infatti, permette a leaflet di interfacciarsi con svariati *provider* di mappe online tra i quali Google Maps, CloudMade, OpenStreetMap, Esri e Nokia. Questa libreria offre molte funzionalità per interagire con le mappe caricate; tra le principali, la possibilità di utilizzare *overlay*, per evidenziare specifiche aree o porzioni di territorio e *marker* per indicare punti specifici nella mappa. È possibile, inoltre, arricchire i marker con testo e immagini caricabili quando l'utente clicca sull'icona del marker stesso. Uno dei principali motivi che hanno contribuito alla scelta di leaflet come libreria per rappresentare mappe, è l'aspetto della compatibilità con i diversi browser. Questa libreria infatti garantisce un alto grado di compatibilità con i principali browser per desktop e *smartphone*.

d3.js [25] il nome di questa libreria significa *Data-Driven Documents* e ne riassume perfettamente le funzionalità. Essa permette, infatti, di ge-



Figura 4.12: Un esempio di mappa realizzata con Leaflet

nerare grafici e documenti a partire da *set* di dati. Come Leaflet, anche D3 è altamente compatibile con i diversi browser, in quanto i grafici sono prodotti utilizzando i linguaggi HTML o SVG. L'SVG è un linguaggio derivato da XML che offre costrutti in grado di rappresentare grafiche *raster* e *vettoriali*. L'uso di queste tecnologie largamente diffuse all'interno dei diversi browser permette a D3 di garantire una discreta compatibilità *cross-browser*.

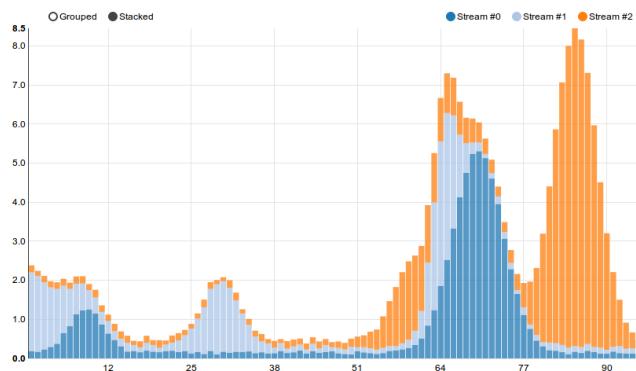


Figura 4.13: Un esempio di grafico realizzato con D3.js

Bootstrap

Questa libreria merita una trattazione separata, in quanto non si occupa, come le precedenti, di risolvere in modo specifico un singolo problema, co-

me nel caso della gestione delle mappe o dei grafici, bensì il suo campo di applicazione è molto più vasto.

Bootstrap [18] è infatti una collezione di strumenti che cooperano per fornire allo sviluppatore una piattaforma di templating CSS e JavaScript per la costruzione di moderne pagine web. Essa contiene set, con design pre-configurati, di componenti come buttoni, form, elementi per la navigazione e tipografici.

Seguendo semplici regole di stesura del codice HTML, anche uno sviluppatore con scarsa conoscenza del linguaggio CSS e scarse nozioni di design, è in grado di creare pagine web completamente stilizzate e funzionanti in pochissimo tempo.

L'idea alla base di Bootstrap è la standardizzazione di pattern di sviluppo e di design utilizzati nella realizzazione di applicazioni web. L'utilizzo di questo framework, infatti, ne velocizza e semplifica molto lo sviluppo.

Tra le principali funzionalità e le caratteristiche che hanno reso Bootstrap famoso e utilizzato da moltissimi sviluppatori, troviamo:

- compatibilità con i maggiori browser
- *gracefully-degrade* se usato su browser più datati
- supporto del design responsivo
- adattamento a diversi dispositivi (desktop, smartphone, tablet)
- licenza open source
- struttura modulare ed estensibile

4.4.1 Gestione delle Dipendenze

Per una azienda o uno sviluppatore è molto dispendioso, in termini di tempo e denaro, realizzare le diverse funzionalità implementandole in modo completamente autonomo. Per questo motivo è comune l'utilizzo di librerie che realizzano sotto-funzionalità e possono essere incluse nelle applicazioni come *black-box*.

Gli sviluppatori di frontend, oggi, hanno a loro disposizione un panorama vastissimo di librerie per la gestione delle più disparate problematiche legate alla realizzazione delle funzionalità client-side: gallerie di immagini, strumenti per la navigazione, maschere per l'inserimento di dati da parte dell'utente e molto altro.

Il processo che porta all'integrazione di una libreria esterna nel proprio progetto, si articola in diverse fasi:

- ricerca della libreria appropriata;
- verifica della compatibilità e degli eventuali problemi di integrazione con altre librerie già in uso;
- inclusione nei diversi file dell'applicazione;
- utilizzo delle funzionalità fornite;

Questa procedura, si ripete per tutte le librerie incluse nell'applicazione.

Si può immaginare come esso sia facilmente automatizzabile, nell'ottica di velocizzare il processo di sviluppo e assicurare una solida integrazione tra i sistemi.

La costruzione del frontend risente di problematiche simili a quella del backend e di soluzioni altrettanto simili. Come illustrato nel capitolo 4.1.3, l'utilizzo di tool di gestione delle dipendenze aiuta a evitare eventuali errori

ascrivibili al fattore umano, durante integrazione di librerie di terze parti, inoltre permette di velocizzare tali operazioni.

Yeoman è un insieme di strumenti per lo sviluppo di applicazioni client-side. Nasce con l'obiettivo di fornire allo sviluppatore un framework per svolgere il proprio lavoro in modo semplice e veloce, ma, al tempo stesso, produrre applicazioni web di alta qualità.



Figura 4.14: I loghi di Yo, Grunt e Bower

Yeoman [26] è un tool che si utilizza da linea di comando, offre molte funzionalità per l'automazione dei diversi task e per il supporto allo sviluppo, di seguito le principali:

- generazione automatica di *template*;
- gestione delle dipendenze delle diverse librerie incluse nel progetto;
- esecuzione automatica di test unitari;
- gestione di un server da utilizzare durante la fase di sviluppo;
- ottimizzazione del codice realizzato al fine di eseguire il deploy del progetto.

Questo strumento combina diversi software open-source che si occupano di fornire le varie funzionalità offerte. Utilizza un concetto di *generator*: un

processo automatico per la produzione di una applicazione, preconfigurata con varie librerie di supporto. Yeoman, inoltre, dispone di una modalità interattiva, nella quale guida lo sviluppatore nella scelta delle diverse librerie da includere nel progetto generato.

I software che concorrono a rendere Yeoman un tool potente e versatile sono:

Yo il vero e proprio sistema di generazione delle applicazioni. Si occupa di scrivere i diversi file di configurazione degli altri tool e includere le dipendenze necessarie per lo sviluppo del progetto.

Grunt [27] è un software di *task automation*. Il suo compito è l'automatizzazione di compiti ripetitivi. È sviluppato in Node.js e permette di definire, usando un file di configurazione in formato JSON, la sequenza delle operazioni da eseguire e l'elenco dei file sulle quali esse operano. Grunt è una applicazione modulare, supporta infatti l'utilizzo di variati plugin per l'esecuzione di specifici task, come il *linting* (controllo della sintassi) dei file di progetto, la *minificazione* degli script CSS e l'esecuzione dei test unitari tramite l'utilizzo di apposite librerie.

Bower [19] si occupa della gestione delle dipendenze. Esegue il *download* della libreria desiderata, se necessario, decomprime il file ottenuto e ne posiziona il contenuto in una specifica directory del progetto, in modo da renderlo fruibile all'interno dell'applicazione. Può accadere che il funzionamento della libreria scaricata, a sua volta, dipenda dalla presenza di altre librerie, dette *dipendenze*. In questo caso, Bower eseguirà un controllo delle versioni di ogni libreria richiesta e avvertirà l'utente in caso di possibili incompatibilità.



Figura 4.15: Il logo di Yeoman

Per lo sviluppo di Mole.io è stato utilizzato un generatore di applicazioni AngularJS. Dopo aver lanciato il comando di generazione, Yeoman ha prodotto un progetto vuoto, ma completamente funzionante e predisposto per l'esecuzione di test unitari con l'uso di *Karma* (un framework messo a disposizione dagli stessi creatori di AngularJS) e un server Node.js per servire staticamente i file contenenti l'applicazione. Grunt è stato anch'esso automaticamente configurato con task per l'esecuzione dei test, la minificazione, e l'avvio del server di supporto. Bower è stato utilizzato per ottenere le dipendenze di base quali jQuery.js, Modernizr, Less.

4.5 Strumenti per il Deploy

Il *deploy* è tecnicamente la fase di installazione dell'applicazione web su un server, sia esso di test o di produzione.

La fase del deploy è molto delicata, in quanto, il manifestarsi di un errore durante l'installazione del sistema su un server di produzione, potrebbe impedire agli utenti di usufruire di tale servizio. La conseguenza peggiore, in questo caso, potrebbe essere la perdita di profitto (per servizi a pagamento) o di credibilità e di fiducia nell'applicazione da parte degli utenti.

Le operazioni tipiche di un deploy sono:

- minificazione degli script;
- backup della precedente versione del software in produzione;
- copia dei file di progetto sul server;
- avvio del nuovo sistema in produzione.

È, ovviamente, possibile eseguire manualmente queste operazioni, ma si incorre nel rischio di commettere un errore, spesso causato dalla distrazione, durante una delle fasi.

Per limitare quanto più possibile la probabilità di errori durante il deploy, sono stati costruiti tool appositi per l'esecuzione di questi task ripetitivi.

Docker è un sistema open-source per l'automatizzazione del processo di deploy di applicazioni. Per eseguire questo compito si avvale di *container*, veri e propri contenitori virtuali di applicazioni con le relative dipendenze, in grado di essere eseguiti su server Linux.

L'utilizzo di container offre svariati vantaggi, il principale è la portabilità delle applicazioni su diversi server fisici o servizi *cloud* pubblici e privati.

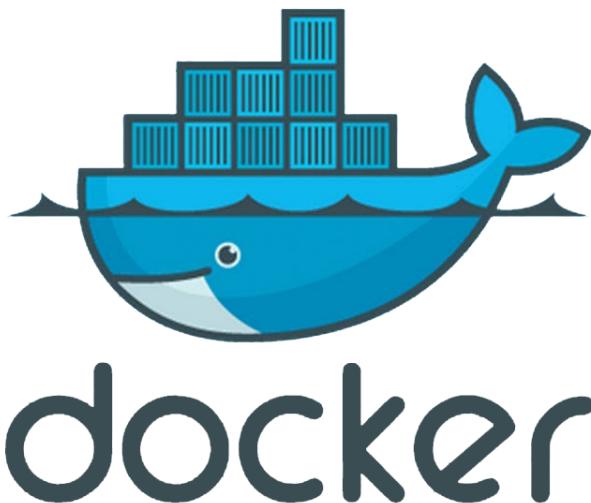


Figura 4.16: Il logo di Docker

Durante lo sviluppo di Mole.io, è stato utilizzato un sistema di container per eseguire il deploy dell'applicazione su un server di test.

È stata realizzata una macchina virtuale con sistema operativo Ubuntu Server, all'interno della quale è stato installato Dokku, un software in grado di trasformare il sistema in un server di tipo PaaS.

Dokku realizza in un server Ubuntu un clone privato del servizio Heroku e permette di interfacciarsi al proprio server esattamente nello stesso modo con cui si utilizzerebbe questo PaaS.

Per eseguire un deploy, infatti, è sufficiente scrivere un semplice comando in console:

```
git push test master
```

Con questo comando, sfruttando `git`, il sistema di *versioning* del codice, è possibile eseguire una operazione di `push` del proprio *branch* `master` sul server `test`.

A fronte del precedente comando, Dokku si preoccupa di:

- eseguire l'invio del branch master al server;
- generare un container Docker;
- copiare i file dell'applicazione nel nuovo container;
- installare, con `npm install`, come descritto nel capitolo 4.1.3, le dipendenze dell'applicazione;
- salvare il container nel quale risiedeva la precedente versione dell'applicazione;
- eseguire lo *switch* dei container, al fine di mettere in produzione la nuova versione;

Non è difficile quindi immaginare come questo tool semplifichi la delicata fase di deploy e minimizzi i rischi che essa comporta.

Capitolo 5

Mole.io

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

Ogni componente del progetto è stato battezzato con un nome di fantasia. Di seguito, l'idea che sottende la nomenclatura.

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.

L'applicazione realizzata, si comporta esattamente come un infiltrato: passa informazioni del sistema nel quale viene inserito, cioè 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. Mole.io possiede agenti con uno scopo simile, chiamati *mole-contact*. Le *soffiate* riferite da ogni mole-contact sono dette *whispers*.

I mole-contact 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 centrale chiamato

mole.

Nell’immaginario collettivo, l’infiltrato, è una persona ben vestita, porta un abito elegante, giacca, cravatta e cappello. Anche l’applicazione realizzata per questa tesi, in un certo senso, è ben vestita, infatti, possiede una interfaccia grafica realizzata per monitorare le applicazioni ed organizzare gli whisper in arrivo. Questo componente è stato chiamato *mole-suit*.

Lo sviluppo dell’applicazione è stato arricchito con uno studio grafico di alcune parti dell’interfaccia, e uno studio del logo. In figura 5.1 è riportato il logo proposto per Mole.io.



Figura 5.1: Il logo di Mole.io

Nelle sezioni seguenti verranno illustrate le funzionalità specifiche di ogni componente in Mole.io, con particolare attenzione alle modalità con le quali tali sotto-sistemi interagiscono e scambiano dati tra loro.

5.1 Architettura del Sistema

Ogni componente in Mole.io è stato realizzato cercando di rispettare il più possibile due concetti fondamentali, che sono stati il denominatore comune del progetto: l'estensibilità e la flessibilità.

Le comunicazioni tra i diversi componenti del sistema sono realizzate utilizzando il protocollo HTTP. I diversi attori si scambiano informazioni attraverso dei *contratti*, le interfacce *REpresentational State Transfer* (REST).

Una interfaccia REST [28] è uno stile architettonico composto da regole, ed elementi che permettono l'accesso a insiemi dati organizzati in maniera strutturata.

Nei sistemi REST, le risorse sono indirizzate secondo regole ben definite e note a priori. Immaginando di modellare una interfaccia REST che permetta l'accesso ad un sistema di utenti e di relazioni tra essi, potremmo definire i seguenti *Uniform Resource Identifier* (URI) per estrarre informazioni:

- `/users`: restituisce l'elenco di tutti gli utenti presenti nel sistema;
- `/users/marco`: restituisce i dati relativi all'utente di nome `marco`;
- `/users/marco/friends`: restituisce l'elenco degli amici di `marco`;
- `/users/marco/friends/fabio`: restituisce i dati dell'utente `fabio`, amico di `marco`;

Come si può notare, è stata definita una convenzione, una regola, per ottenere i dati dal sistema e modellare le relazioni tra essi. È stata inoltre, implicitamente, generata una modalità logica di navigazione attraverso i dati che permette di raggiungere le informazioni desiderate.

Le interfacce REST implementate in Mole.io restituiscono dati in formato JSON [23]. Riprendendo l'esempio precedente, si potrebbe immaginare che il sistema risponda alle richieste effettuate nell'ordine, restituendo i seguenti dati:

```
GET /users
(RISPOSTA) [ 'marco', 'fabio', 'paola', 'gianni' ]
GET /users/marco
(RISPOSTA) { name: 'marco', surname: 'bianchi', age: '25' }
GET /users/marco/friends
(RISPOSTA) [ 'fabio', 'paola' ]
GET /users/marco/friends/fabio
(RISPOSTA) { name: 'fabio', surname: 'verdi', age: '31' }
```

Come è stato anticipato, le interfacce REST, utilizzano il protocollo HTTP per scambiare dati. Questo protocollo mette a disposizione diverse tipologie di messaggi, ognuna di queste si può immaginare come una *azione* compiuta su una risorsa: quella indicata dall'*endpoint* dell'interfaccia. I comandi utilizzati per dialogare con un servizio REST sono:

POST permette di creare una nuova risorsa. Di solito, in caso di successo, il servizio restituisce il dato appena creato;

GET richiede lo stato corrente di una risorsa;

PUT applica una modifica ad una risorsa esistente;

DELETE elimina una risorsa;

In letteratura, questa modalità di interazione con i dati, è definita *Create-Read-Update-Delete* (CRUD).

Ad esempio, volendo inserire un nuovo utente si potrebbe utilizzare la seguente richiesta:

```
POST /users  
{ name: 'guido', surname: 'rossi', age: '22' }
```

A questo punto, è possibile verificare l'esito dell'inserimento con:

```
GET /users  
(RISPOSTA) [ 'marco', 'fabio', 'paola', 'gianni', 'guido' ]  
GET /users/guido  
(RISPOSTA) { name: 'guido', surname: 'rossi', age: '22' }
```

Una delle caratteristiche di REST è l'assenza del supporto alle *session*. I sistemi modellati secondo queste interfacce sono, infatti, *state-less*, cioè non presentano correlazioni tra richieste successive. L'approccio adottato da REST, in questo caso, è esattamente allineato con quello adottato da HTTP, anch'esso state-less. Come anticipato nella sezione 4.4, questa peculiarità rende i framework per realizzare single-page application, i tool ideali per lavorare in tale ambito.

Nell'architettura di Mole.io si possono individuare due *layer* principali:

Insertion è la porzione di sistema che si occupa dell'inserimento dei dati provenienti dall'esterno. Questa sezione è composta, a sua volta, dai *mole-contacts* e dal server *mole*. La sezione 5.1.2 si occupa di descrivere i dettagli dell'architettura del layer di insertion.

Presentation si occupa dell'estrazione dei dati dal sistema e della loro presentazione all'utente. Le componenti di questo modulo sono la *user-interface* (UI) AngularJS e il server *mole-suit*. Il layer di presentation verrà approfondito nella sezione 5.1.3.

Ogni componente ricopre un ruolo ben preciso nel sistema e comunica con gli altri attori attraverso interfacce di tipo REST.

mole-contact risiede all'interno di una applicazione detta *source* e si occupa dell'invio delle informazioni da salvare (log e messaggi) al server mole;

mole il suo compito è validare le informazioni in arrivo dai diversi mole-contact e salvarle all'interno del database;

user interface è la vera e propria interfaccia grafica di Mole.io. Permette agli utenti di interagire con il sistema e dialoga direttamente con mole-suit per ottenere i dati da mostrare;

mole-suit è il server che si occupa di estrarre dal database i dati richiesti dagli utenti;

La figura 5.2 rappresenta schematicamente le comunicazioni instaurate dalle diverse componenti del sistema. Come è possibile notare dal grafico, Mole.io è in grado di interagire con svariate tipologie di applicazioni (*source*): web, desktop oppure mobile. Nella sezione 5.1.2 verranno illustrate le interazioni tra source e mole-contact.

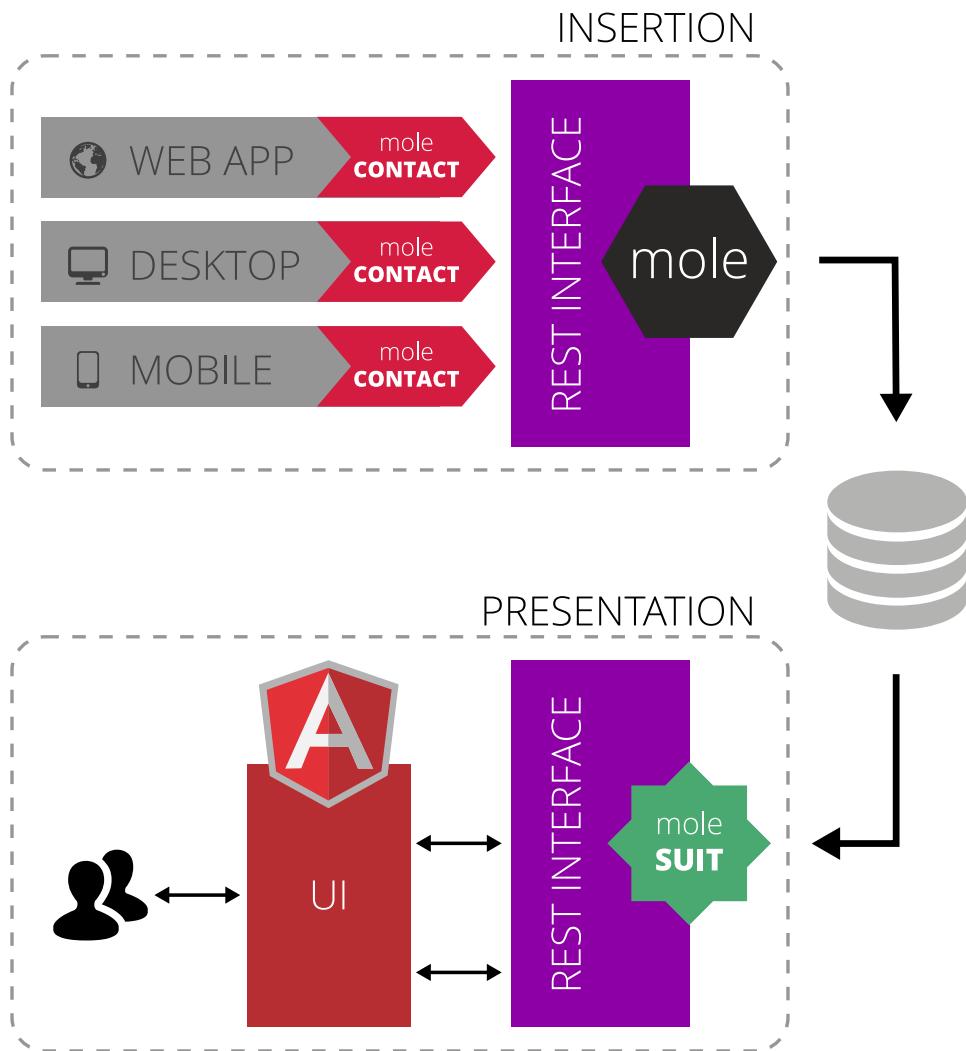


Figura 5.2: L'architettura di Mole.io

In tabella 5.1 sono riportati i principali endpoint delle interfacce REST esposte da ciascun componente in Mole.io. Per ciascun endpoint sono state riportate la tipologia di comando utilizzata per interagire con l'URI (GET o POST) e una descrizione delle funzionalità offerte da esso.

| | Tipo | Endpoint | Descrizione |
|------------------|-------------|---------------------------|---|
| mole | POST | /whispers | permette ai mole-contact di inviare whisper a mole |
| mole-suit | GET | /sources | restituisce l'elenco delle source |
| | POST | /sources | permette di inserire una nuova source |
| | GET | /sources/sID | restituisce i dati della source con id sID |
| | GET | /sources/sID/whispers | restituisce l'elenco degli whisper ricevuti dalla source con id sID |
| | GET | /sources/sID/whispers/wID | restituisce l'elenco al whisper con id wID ricevuto dalla source con id sID |

Tabella 5.1: I principali endpoint REST forniti da Mole.io

Implementare una interfaccia REST con Node.js è piuttosto semplice, specialmente se si utilizza il modulo Express, descritto nella sezione 4.1.3. Express, infatti, mette a disposizione dello sviluppatore un sistema molto potente di *route*, che permettono di identificare la risorsa desiderata a fronte di una richiesta in arrivo. Il codice seguente implementa un esempio di rotta con Express.

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

var app = express();

app.use(app.router);

app.get('/now', function(req, res) {
  res.json({ now: new Date().toString() });
});
```

L'applicazione Node.js carica il modulo Express e genera una applicazione utilizzando tale modulo. Successivamente viene registrata una rotta che corrisponde all'URI `/now`. All'arrivo di una richiesta di tipo `GET` verso la risorsa `/now`, Express esegue la callback associata e restituisce un documento JSON simile a quello seguente.

```
{ "now": "Sun Mar 09 2014 20:48:35 GMT+0100 (CET)" }
```

Il `router` Express, come molti altri plugin di questo sistema, rappresenta un *middleware*. Il comando `app.use(app.router);`, infatti, indica ad Express di utilizzare il middleware `router` per filtrare le richieste in arrivo.

Un middleware è un modulo software che filtra ogni richiesta in arrivo dai client. I middleware cooperano tra loro e vengono attivati in successione. Il flusso di lavoro di un middleware è riassumibile in tre fasi principali:

1. ottenere la richiesta;
2. validare la richiesta ed eseguire funzionalità specifiche associate al middleware stesso;
3. passare la richiesta al prossimo middleware;

Seguendo il medesimo schema di funzionamento, i middleware, elaborano le risposte provenienti dall'applicazione e restituite al client. La figura 5.3 illustra il funzionamento di questo sistema.

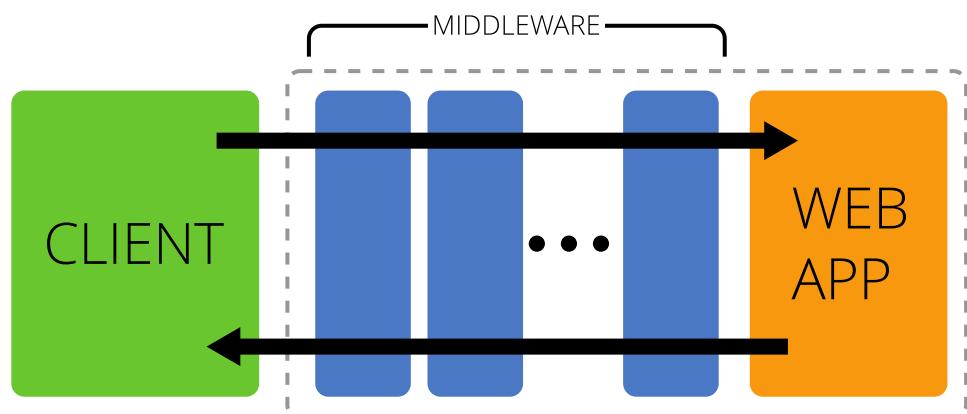


Figura 5.3: Schema di funzionamento dei middleware

5.1.1 Command Query Responsibility Segregation

Durante fase di progettazione di applicazioni come Mole.io, non si può prescindere dal prendere in considerazione il carico di lavoro che subirà il sistema in produzione ed i tempi di risposta da esso attesi.

Le tecniche di ottimizzazione del Database e di sharding dei dati, da sole, non bastano a garantire buone performance del sistema. È necessaria, infatti, una organizzazione dell'architettura del sistema che renda performanti le diverse operazioni verso il Database e, nello stesso tempo, non limiti le possibilità di estensione futura del sistema stesso.

Il *Command Query Separation* (CQS) è un principio di programmazione, discusso da Bertrand Meyer in [29]. Dichiara che, in un sistema software, ogni metodo deve appartenere ad una sola delle seguenti categorie:

Command un metodo che esegue azioni, modifica i dati, non restituisce alcun risultato e crea potenzialmente *inconsistenza* nel sistema;

Query un metodo che ottiene dati esistenti, esegue richieste e restituisce risultati senza alterare lo stato del sistema;

La descrizione del principio in una sola frase potrebbe essere: *porre una domanda non dovrebbe cambiare la risposta*.

Nonostante il CQS sembri un concetto molto semplice, la sua implementazione all'interno dei sistemi è tutt'altro che immediata. Il principio, infatti, è inteso come linea guida e come invito alla separazione netta delle competenze di ogni funzione o modulo.

Il *Command Query Responsibility Segregation* (CQRS) è un pattern architettonico che segue le regole indicate dal CQS e le applica al design di sistemi software. L'idea alla base di questo pattern, infatti, è la separazione netta dei moduli di scrittura (command) e lettura (query) presenti in una

applicazione. Spesso la separazione si estende al database stesso, ottenendo così sistemi con due basi di dati: una utilizzata esclusivamente per eseguire scritture e l'altra per eseguire letture.

I sistemi CQRS sono disegnati in modo da separare le responsabilità di scrittura e lettura. Questo approccio solleva immediatamente un problema: come è possibile scrivere dati all'interno di un database e pretendere di leggerli da un archivio differente? La risposta risiede nei *denormalizzatori*.

Un denormalizzatore è un modulo software che ha il compito di creare ridondanza all'interno di informazioni. Esso, infatti, legge dati salvati in formato grezzo e li elabora, ristrutturandoli, per fare in modo che essi diventino facilmente fruibili per la lettura. A questo punto li salva nuovamente all'interno del database. Nel caso di sistemi con due database, i denormalizzatori sono gli unici attori nel sistema che leggono dati dal database di scrittura e li scrivono nel database di lettura.

Il principale vantaggio offerto da CQRS è la possibilità di ottimizzare separatamente i database di lettura e scrittura. Ad esempio, la presenza di *indici* sulle tabelle di lettura, velocizza molto le operazioni di recupero dei dati, mentre penalizza quelle di scrittura, a causa della necessità di aggiornamento degli indici a fronte di un inserimento. La separazione delle tabelle di lettura e scrittura permette di attivare gli indici esclusivamente nel sistema di output, rendendo, di conseguenza, anche il sistema di input più reattivo.

L'approccio CQRS garantisce un alto *throughput* del sistema, cioè un'elevata frequenza nel fornire dati in lettura. Le informazioni pre-lavorate dai denormalizzatori, infatti, non necessitano di ulteriore elaborazione da parte dei moduli che si occupano dell'estrazione, i quali possono prelevarle direttamente e fornirle ai richiedenti.

Nei sistemi che fanno uso di MongoDB per il salvataggio dei dati, l'applicazione del pattern CQRS possiede un ruolo ancora più importante a causa del modo in cui MongoDB effettua le operazioni di *lock*.

MongoDB applica un *read-lock* a fronte di ogni operazione di lettura, e un *write-lock* a fronte di ogni scrittura. Il primo tipo di lock accoda ogni successiva lettura sull'intero database, mentre il secondo accoda tutte le letture e le scritture *pending* sull'intera base dati. È chiaro, quindi, come la possibilità di eseguire lock su database separati possa migliorare le performances globali del sistema evitando colli di bottiglia.

L'architettura di Mole.io trae ispirazione dal modello CQRS. Nel sistema realizzato non esiste una distinzione tra database di lettura e scrittura, bensì tra collection 4.3 per il salvataggio di dati grezzi e altre contenenti dati denormalizzati. L'applicazione è stata comunque disegnata in modo da rendere agevole la separazione di tali collection su database differenti, in caso di necessità future.

Mole.io applica il modello CQRS anche per quanto riguarda la separazione dei ruoli di scrittura e lettura. Il modulo *mole*, infatti, si occupa del salvataggio dei dati all'interno del database, mentre il compito di *mole-suit* è, principalmente, l'estrazione dei dati per la pubblicazione su interfaccia utente. In figura 5.4 è riportata schematicamente l'architettura CQRS realizzata in Mole.io.

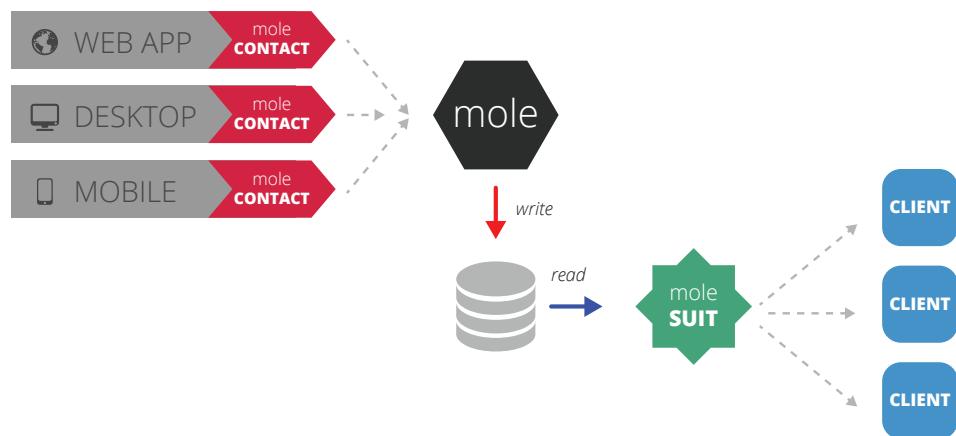


Figura 5.4: Architettura CQRS di Mole.io

5.1.2 Mole

Il layer di *insertion* in Mole.io è rappresentato da *mole*. Questo server si occupa di salvare all'interno del database MongoDB gli *whisper* ricevuti dai *mole-contact* presenti all'interno delle *source*.

I Mole Contact

Nella sezione 5.1 è stato introdotto il concetto di *source*: in Mole.io è definita source una applicazione da monitorare. È possibile monitorare differenti tipologie di applicazioni, l'unico requisito è che esse abbiano la possibilità di collegarsi ad internet per inviare i rispettivi messaggi (*whisper*) verso il server.

Le principali tipologie di source monitorate con Mole.io saranno applicazioni di tipo desktop, mobile e web app. Ogni applicazione verrà sviluppata utilizzando linguaggi di programmazione differenti, motivo per il quale, nella sua configurazione finale, Mole.io prevederà un *mole-contact* specifico per ogni linguaggio di programmazione gestito dal sistema. Alcuni esempi potrebbero essere: *mole-contact-js*, *mole-contact-net*, *mole-contact-java*, e così via.

I mole-contact inviano gli whisper in formato JSON al server mole utilizzando il protocollo HTTP. Il sistema prevede che all'header HTTP vadano aggiunti due campi personalizzati: **X-Mole-SourceId** e **X-Mole-SourceKey** popolati rispettivamente con l'id e la chiave assegnati da Mole.io alla sorgente in fase di registrazione.

Gli whisper

I messaggi inviati dalle sources verso mole, utilizzando i mole-contact, sono definiti whisper. In Mole.io, un whisper è un documento codificato in

formato JSON con alcuni campi obbligatori:

time l'istante nel quale il messaggio è stato generato dalla source;

sender il mittente del messaggio, è un campo di testo libero;

message il testo che caratterizza il messaggio stesso;

severity un livello di gravità del messaggio;

Ogni source, costruisce un whisper contenente i campi obbligatori e ne aggiunge altri specifici per l'applicazione stessa, ma a priori non noti a Mole.io.

Mole.io utilizza un approccio *agnostico* rispetto ai campi forniti da ogni applicazione. Questo è reso possibile grazie alla scelta di un database schema-less come MongoDB. Non dovendo definire una struttura dei dati a priori, infatti, mole può salvare nel database whisper con formati completamente differenti.

Di seguito è riportato un esempio di whisper utilizzato durante i test. Come si può notare, esso contiene, oltre ai campi di *default*, altri valori provenienti da una applicazione per *device* mobile.

```
{  
  "time" : "2014-03-13T11:53:20.197Z",  
  "sender" : "iPhone-02",  
  "severity" : 2,  
  "message" : "A first chance exception of type  
  'System.IO.FileNotFoundException' occurred",  
  "ip" : "34.46.123.45",  
  "email" : "mymail@mysite.com",  
  "os" : "ios",
```

```
"geo" : {  
    "lat" : 45.43356707013582,  
    "lon" : 10.32409153589988  
}  
}
```

I Denormalizzatori

Come è stato anticipato nella sezione 5.1.1, il compito dei denormalizzatori è pre-elaborare i dati grezzi, in modo da renderli facilmente fruibili per la lettura. La chiave del funzionamento di Mole.io risiede proprio nei denormalizzatori. Essi, infatti, si occupano di estrarre le informazioni aggiuntive presenti negli whisper in arrivo e le aggregano in collection specifiche all'interno di MongoDB.

Il sistema di gestione dei denormalizzatori è stato realizzato in modo che essi possano essere attivati a *runtime*. Questo approccio garantisce la possibilità di estendere il sistema per adattarlo a nuove esigenze applicative.

Con il passare del tempo, infatti una source potrebbe avere la necessità di cominciare a salvare informazioni differenti rispetto a quelle salvate fino a quel momento. L'unica operazione necessaria per adattare Mole.io alla nuova esigenza, sarà la realizzazione di un denormalizzatore specifico, che si occupi di trattare la nuova porzione di dato e di pre-elaborarla per garantirne una immediata fruizione da parte di mole-suit.

Un nuovo denormalizzatore attivato, dovrebbe occuparsi principalmente di elaborare i nuovi dati in arrivo e pre-processarli in modo da renderli fruibili per la sezione di presentation. Potrebbe però essere interessante implementare tale denormalizzatore in modo che prima di elaborare i nuovi dati in arrivo, esegua il *processing* dei dati grezzi precedentemente salvati

nel sistema. Seguendo questo approccio è possibile ottenere nuove modalità di aggregazione di dati preesistenti, generando un valore aggiunto per gli utilizzatori del sistema, i quali vedrebbero i vecchi dati aggregati secondo la nuova esigenza.

La possibilità di attivare un nuovo denormalizzatore a *runtime* è garantita dall'utilizzo di RabbitMQ 4.2 per la distribuzione dei dati in arrivo. All'arrivo di un nuovo whisper, infatti, vengono eseguite le seguenti operazioni:

1. Il server mole riceve il messaggio attraverso l'interfaccia REST e lo valida. La validazione avviene controllando l'esistenza della source riportata nell'header del messaggio e la conformità del corpo del messaggio con il formato JSON, nonché la presenza dei campi obbligatori.
2. Il dato validato viene salvato all'interno di MongoDB in una collection denominata `whispers_<id_source>`, dove `<id_source>` rappresenta l'identificativo MongoDB della source che ha inviato il messaggio.
3. Il dato grezzo, viene inserito in una coda RabbitMQ configurata per il funzionamento in modalità *Publish/Subscribe*
4. I denormalizzatori in ascolto sulla coda, ottengono il messaggio e lo elaborano estraendone la porzione di competenza.
5. Ogni denormalizzatore salva i dati da esso elaborati all'interno di una collecion MongoDB denominata `<denormalizer_name>_<id_source>`, dove `<denormalizer_name>` indica il nome del denormalizzatore che ha eseguito l'operazione e `<id_source>` rappresenta l'identificativo MongoDB della source che ha inviato il messaggio. Al momento della scrittura di questa tesi, i denormalizzatori realizzati sono:

- **geo**, estrae informazioni di geolocalizzazione;
- **summary**, aggrega gli whisper secondo il grado di severity e genera contatori specifici;
- **stats**, realizza statistiche orarie di ricezione degli whisper;
- **cluster**, aggrega gli whisper secondo severity, sender e message per creare *cluster* di messaggi simili;

La crescita del carico di lavoro richiede che i denormalizzatori elaborino una quantità crescente di whisper. Il problema del collo di bottiglia, a prima vista, è stato semplicemente spostato dal server centrale mole ai denormalizzatori. Per ovviare questo inconveniente, è stata sfruttata una peculiarità delle code RabbitMQ: i subscriber di una coda gestita con il sistema Publish/Subscribe, ricevono i messaggi in modalità *round robin*. Questa proprietà permette di distribuire il carico di lavoro su più denormalizzatori che lavorano in parallelo. A questo punto, è possibile attivare o disattivare denormalizzatori a *runtime* in accordo con il numero di richieste da gestire.

In figura 5.5 sono mostrate schematicamente la struttura interna del server mole e le comunicazioni instaurate tra i diversi componenti del sistema.

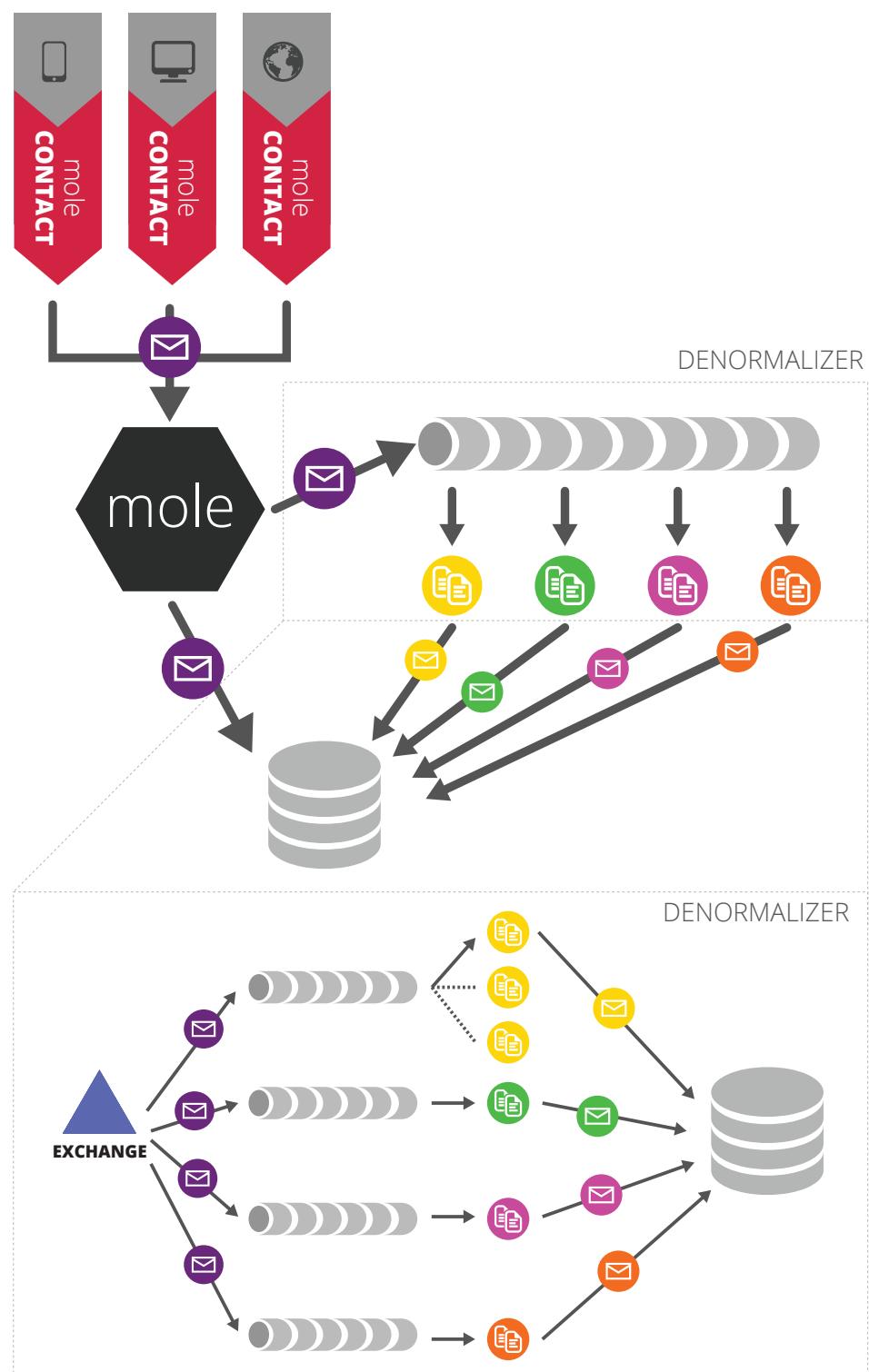


Figura 5.5: Architettura di mole

5.1.3 Mole Suit

Il layer di *presentation* di Mole.io è rappresentato da mole-suit.

Il server mole-suit è realizzato in Node.js e

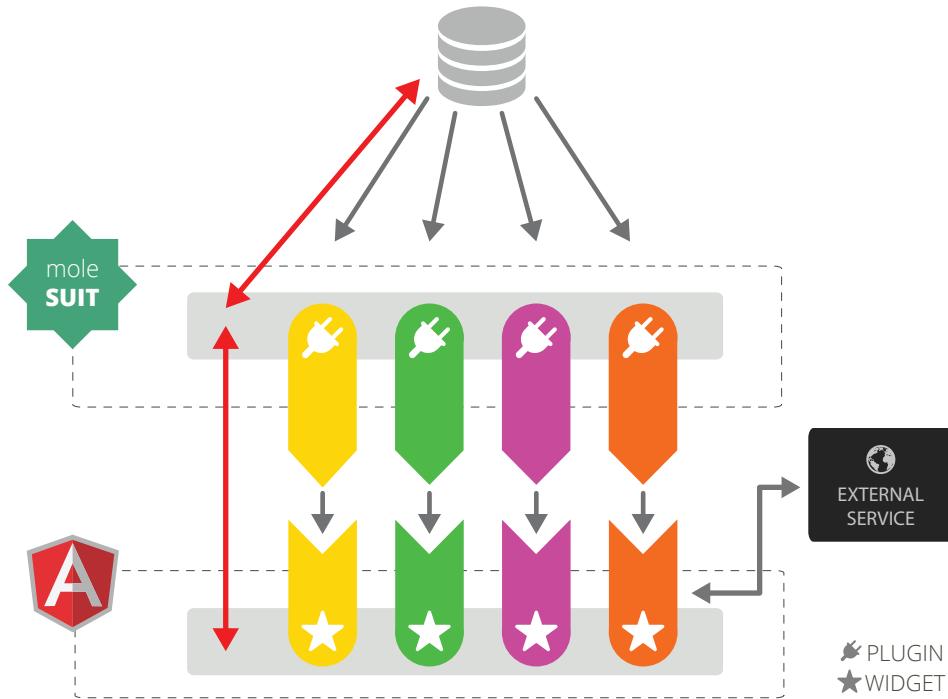


Figura 5.6: Architettura di mole-suit

La figura 5.6 fornisce una rappresentazione schematica delle componenti di Mole Suit.

Gli utilizzatori accedono al sistema attraverso l'interfaccia utente AngularJS. La UI è composta da widget che hanno il compito di interrogare i rispettivi plugin, residenti nel server mole-suit, al fine di estrarre i dati necessari al popolamento delle pagine web richieste.

al fine di ottenere i dati necessari al popolamento delle pagine web richieste.

Il server mole-suit si interfaccia con il database ed estrae i dati denor-

malizzati. Le informazioni ottenute vengono quindi inviate alla User Interface (UI) AngularJS che si occupa della vera e propria costruzione della pagina web all'interno del browser dell'utente.

I Plugin

La User Interface

L'interfaccia utente di Mole.io è interamente realizzata utilizzando AngularJS, descritto in 4.4. Questa tecnologia permette di implementare applicazioni web in grado di essere eseguite completamente client-side.

La costruzione delle pagine HTML da mostrare all'utente è, quindi, a carico del computer dell'utente stesso, non del server che fornisce l'applicazione. Ogni client ha, ovviamente, la necessità di comunicare con il server, al fine di ottenere i dati per popolare le pagine da mostrare all'utente.

Questo approccio permette di minimizzare la quantità di informazioni inviate dal server al client con il conseguente aumento delle performance dell'applicazione stessa.

In gergo tecnico, le applicazioni realizzate con AngularJS, si definiscono *servibili staticamente*. Questo termine sottolinea il fatto che non esiste la necessità di costruire pagine lato server e, di conseguenza, è possibile utilizzare server minimali, molto economici in termini di risorse di sistema, per fornire questo tipo di applicazioni.

Gli Widget

Il Workflow con Mole.io

Al primo accesso Mole.io, mostra la *homepage* pubblica dell'applicazione. Questa pagina riporta un messaggio di benvenuto ed un bottone, con il quale

è possibile procedere all'autenticazione con Twitter, utilizzando il protocollo OAuth 2.0, come descritto in 5.2.

Una volta eseguita la procedura di accesso, si raggiunge una pagina *Getting Started*, nella quale è descritta la sequenza delle operazioni necessarie per utilizzare Mole.io. L'immagine 5.7 riporta le schermate proposte dal sistema durante le operazioni descritte.

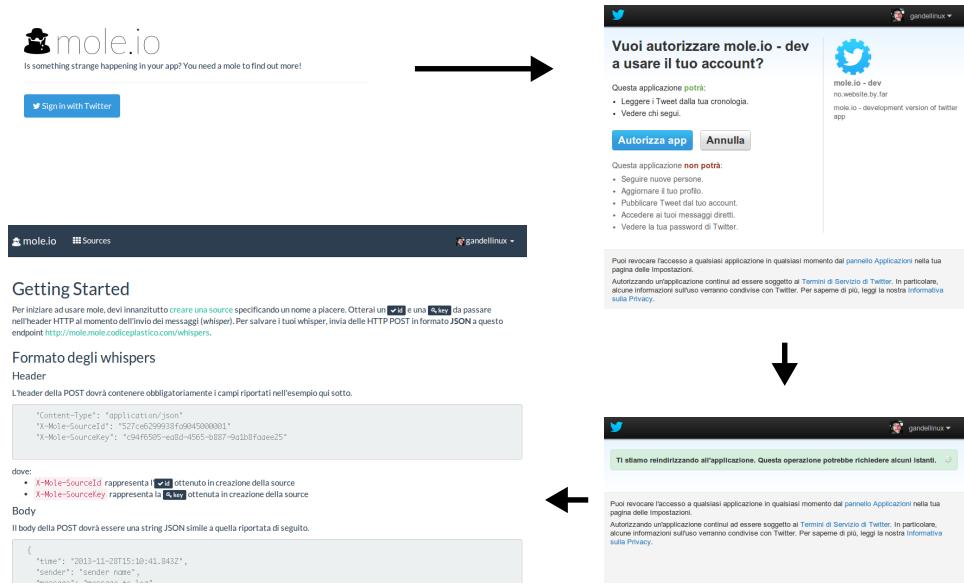


Figura 5.7: Accesso a Mole.io

Cliccando sulla voce *Sources*, nella barra superiore dell'applicazione, è possibile accedere alla pagina per la creazione e visualizzazione delle sources: le applicazioni da monitorare. Per aggiungere una nuova source è sufficiente inserire il nome desiderato nell'apposito campo di testo e premere il pulsante con il simbolo “+”. L'immagine 5.8 mostra le fasi di questa procedura.

Una volta creata la source, Mole.io restituisce due codici associati ad essa: un **id** e una **key**. Questi dati dovranno essere utilizzati per configurare il mole-contact presente all'interno dell'applicazione da monitorare, e per-

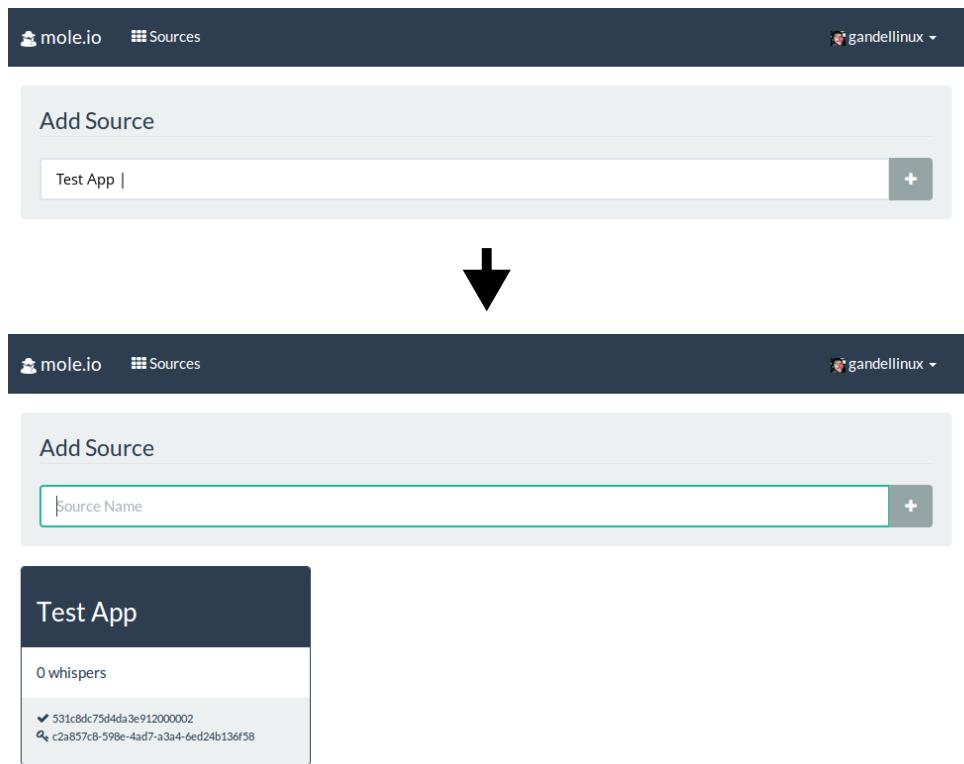


Figura 5.8: Creazione di una Source

metteranno al server mole di identificare la provenienza dei messaggi inviati da tale applicazione.

Una volta configurato il mole-contact, man mano l'applicazione invierà i proprio whisper, questi saranno raccolti dal server mole, il quale li salverà all'interno del database MongoDB ed eseguirà i denormalizzatori per preparare i dati alla fruizione da parte di mole-suit.

Accedendo all'elenco delle source e cliccando sul nome della source desiderata, è possibile accedere alla *dashboard* dell'applicazione. Questa pagina, visibile in figura 5.9, fornisce una visione generale dello stato della source, riportando il numero di whisper raccolti per ogni grado di *severity*, e grafici con statistiche orarie e geografiche.

Nel caso riportato come esempio, ogni whisper possiede un dato *geo*

associato. Tale informazione è denormalizzata da mole, caricata da mole-suit utilizzando uno specifico plugin e mostrata a video tramite un apposito widget AngularJS.

dove lo mettiamo? A differenza di questo caso, le operazioni di creazione e visualizzazione delle source, non necessitano di denormalizzatori: la UI interroga direttamente il server mole-suit per ottenere l'elenco delle source associate all'utente e le mostra a video.

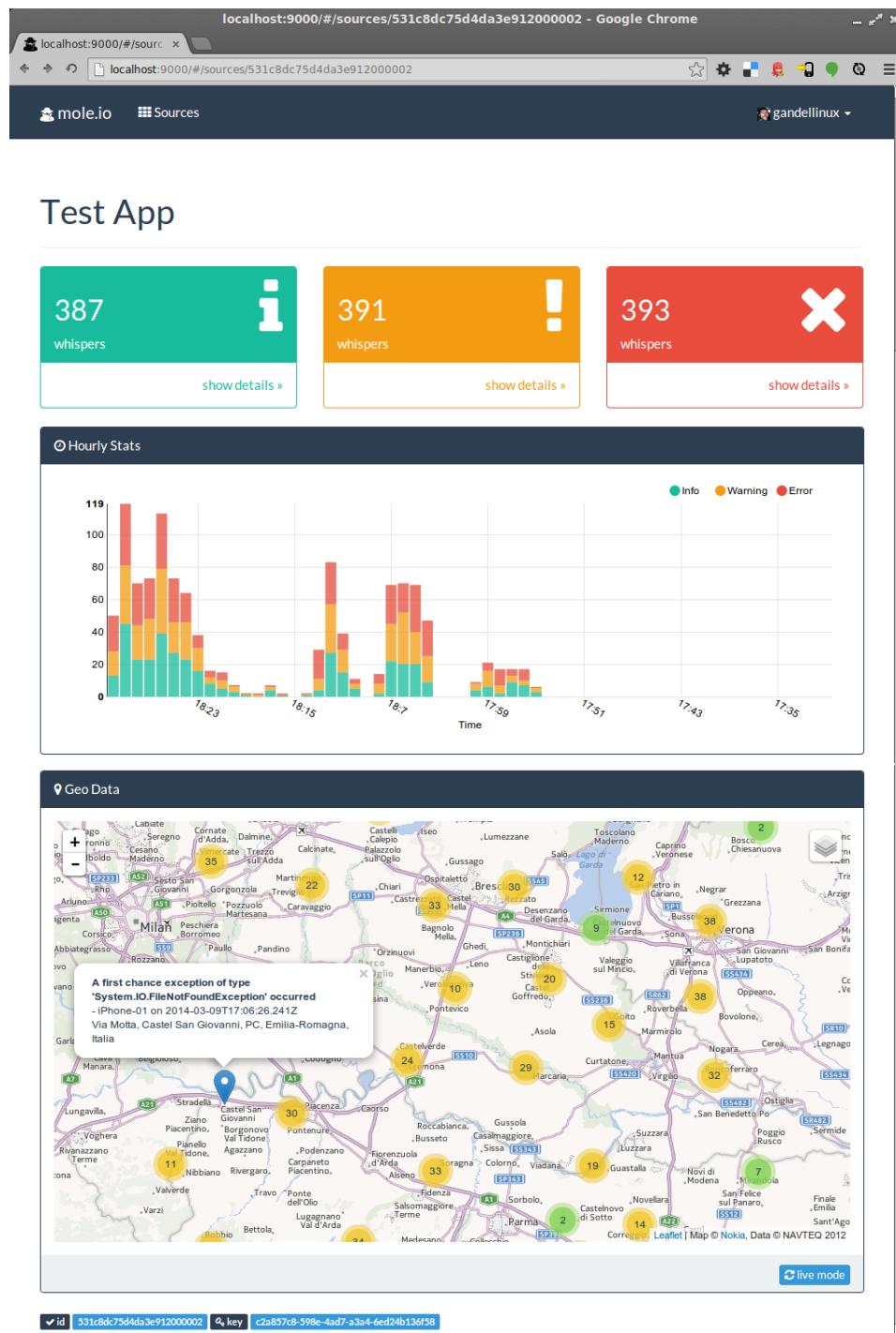


Figura 5.9: Dashboard di una Source

5.2 Autenticazione degli Utenti

L'avvento del web 2.0 ha visto la nascita di un numero sempre crescente di servizi che permettono agli utenti di pubblicare e condividere i propri contenuti in rete. Ognuna di queste applicazioni, necessita di dati identificativi dell'utente al fine di garantirgli l'accesso alle proprie risorse. Con il sistema di autenticazione standard, viene chiesto ad ogni utilizzatore di fornire uno *username* e una *password* per associare la propria identità ad uno specifico *account*. Il grande numero di servizi disponibili, però, obbliga gli utenti a dover gestire molte informazioni di autenticazione, inoltre accade spesso che tali utenti utilizzino la stessa password per servizi differenti, con un evidente rischio per la sicurezza.

Molte automobili di lusso possiedono una chiave per il posteggiatore. Una chiave speciale, che permette di attivare un numero limitato di funzionalità dell'autovettura e di guidarla entro uno spazio limitato.

OAuth è un protocollo standard per autorizzazioni che, in modo simile a quanto succede per le chiavi posteggiatore per le automobili, permette ad un utente di garantire, ad applicazioni di terze parti, l'accesso ad un numero limitato di contenuti personali.

Con OAuth inoltre gli utenti non devono condividere la propria password con diversi sistemi, bensì un server centrale di autorizzazione, funge da garante per l'identificazione delle informazioni di accesso e per la fruizione di specifici dati utente. Dal punto di vista tecnico, il protocollo è implementato utilizzando un sistema di redirezioni HTTP e di controlli da parte del server che funge da garante.

Mole.io si configura al pari di altri servizi presenti sul web e, in maniera simile, sfrutta il protocollo OAuth 2.0 per l'identificazione e l'autenticazione degli utenti. In questo caso il server scelto come garante delle identità utente

è quello messo a disposizione da Twitter.

Nello schema 5.10 è possibile vedere le diverse fasi del processo di autenticazione di un utente che richiede l'accesso a Mole.io.

1. l'utente richiede l'accesso a Mole.io;
2. l'applicazione esegue un *redirect* HTTP verso la pagina di autenticazione fornita da Twitter, passandole un *token* per l'identificazione dell'utente;
3. il server di autenticazione di Twitter esegue la verifica di identità dell'utente e, in caso di esito positivo, esegue una redirezione verso un URL fornito dall'applicazione restituendo il token identificativo e un codice di verifica dell'utente;
4. a questo punto l'applicazione ottiene i dati dell'utente e può ritenere l'operazione di accesso effettuata con successo.

L'implementazione dell'autenticazione OAuth 2.0 all'interno di mole-suit è stata realizzata utilizzando il modulo Passport per Node.js, introdotto nella sezione 4.1.3. Passport sfrutta il sistema di middleware offerto da Express per garantire un controllo di accesso a livello di singola rotta, fornendo quindi una granularità molto fine.

Il listato seguente è la porzione di codice, estratta da mole-suit, che implementa la procedura di autenticazione utilizzando Passport.

```
var passport = require('passport'),  
    TwitterStrategy = require('passport-twitter').Strategy;  
  
passport.use(new TwitterStrategy({  
    consumerKey: TWITTER_CONSUMER_KEY,  
    consumerSecret: TWITTER_CONSUMER_SECRET,  
    callbackURL: 'http://mole.io/auth/twitter/callback'  
});
```

```
consumerSecret: TWITTER_CONSUMER_SECRET,  
callbackURL: "/auth/twitter/callback"  
,  
function(token, tokenSecret, profile, done) {  
  User.findOrCreate(..., function(err, user) {  
    if (err) { return done(err); }  
    done(null, user);  
  });  
}  
));  
  
[...]  
  
app.get('/auth/twitter',  
  passport.authenticate('twitter')  
);  
  
app.get('/auth/twitter/callback',  
  passport.authenticate('twitter', {  
    successRedirect: '/',  
    failureRedirect: '/login'  
  })  
);
```

Dopo aver importato i moduli richiesti, si configura la *TwitterStrategy* di Passport, cioè la modalità di autenticazione che permette di interagire con i server di Twitter. La configurazione prevede l'inserimento di due dati: TWITTER_CONSUMER_KEY e TWITTER_CONSUMER_SECRET. Questi valori

sono forniti da Twitter e identificano univocamente Mole.io. Sono necessari in quanto il processo di validazione prevede che Twitter sia a conoscenza delle informazioni dell'utente e di quelle dell'applicazione di terze parti. Per completare la configurazione è richiesta infine una funzione che permette di fornire a mole-suit le informazioni relative all'utente appena autenticato.

Le successive istruzioni riguardano la configurazione delle rotte che faranno uso dell'autenticazione con Twitter. La prima dichiara che, a fronte della richiesta dell'endpoint `/auth/twitter`, è necessario eseguire la procedura di autenticazione. La seconda è necessaria per validare una autenticazione, in caso di successo, l'utente viene rediretto verso la pagina principale di mole-suit, in caso di insuccesso, viene riproposta la pagina di *login*.

A fronte di un processo di autenticazione, Passport fornisce un oggetto contenente alcune informazioni relative all'utente. Alcuni dei campi forniti sono:

- **provider**, il garante della procedura di autenticazione, Twitter;
- **id**, l'identificativo utente all'interno dei server di Twitter;
- **displayName**, il nome dell'utente da mostrare a video;
- **photos**, un elenco di *avatar* dell'utente;

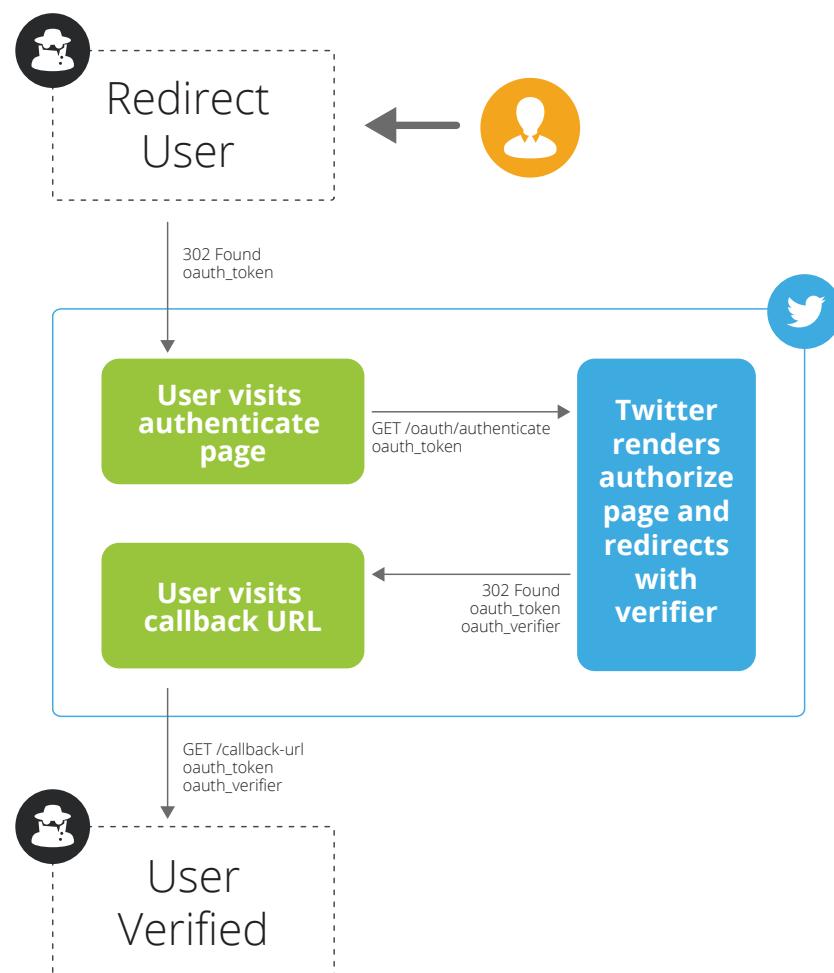


Figura 5.10: Schema di autenticazione con protocollo OAuth 2.0

5.3 Scalabilità e Affidabilità

Mole.io utilizza una configurazione di MongoDB in grado di garantire un alto grado di scalabilità del sistema e, contemporaneamente, un ottimo livello di affidabilità. La configurazione utilizzata prevede l'uso di due funzionalità fornite da MongoDB: lo *sharding* ed i *replica-set*.

Sharding

Lo sharding è una funzionalità che ideata per garantire una *scalatura orizzontale* del database. Essa consiste nella suddivisione dei dati su diversi server, detti *shard*. MongoDB utilizza questa funzionalità per distribuire grandi *data-set* su un numero variabile di macchine e garantire un alto *throughput* del database.

Ogni shard è un database indipendente, che, in collaborazione con altri shard all'interno della stessa configurazione, concorre a formare un'unica istanza del database: un *cluster*.

L'obiettivo dello *sharding* è la riduzione del carico di lavoro su ogni database. Al crescere del numero di shard presenti all'interno del medesimo cluster, infatti, il numero di operazioni a carico di ogni singolo server, si riduce. Come conseguenza, si ottengono un incremento della capacità totale del database e un incremento del throughput del sistema.

Al fine di distribuire i dati sui diversi server, è necessaria una politica di scelta dello shard di destinazione. MongoDB offre la possibilità di definire una *shard key*, cioè una chiave identificativa di un singolo documento, che permette al database di identificare lo shard che dovrà contenere il dato.

MongoDB suddivide i valori delle chiavi di shard in *chunk* e distribuisce i chunk sui diversi server. La suddivisione è operata esclusivamente creando

partizioni sull’insieme di valori delle chiavi di shard e assegnando a ciascuna partizione un server ospitante.

La distribuzione dei chunk all’interno dei diversi shard, assicura un bilanciamento del carico di lavoro di ogni server all’interno del cluster. MongoDB utilizza due processi in background per garantire il bilanciamento del cluster:

splitting è un processo che evita la crescita eccessiva dei chunk. Si occupa, infatti, di tenere costantemente monitorata la dimensione di ogni chunk, quando uno di essi cresce oltre un limite preconfigurato, lo suddivide in due chunk più piccoli. Le eventuali suddivisioni avvengono dopo operazioni di *insert* e *update* e non implicano alcuno spostamento di chunk.

balancer ha il compito di eseguire la migrazione dei chunk da uno shard ad un altro. Quando la distribuzione dei chunk nei diversi server è sbilanciata, questo processo sposta i chunk contenuti in shard più popolati verso shard con *slot* liberi, mantenendo il più possibile costante il numero di chunk per shard.

Tra i vantaggi forniti dallo sharding, è possibile annoverare anche un miglioramento delle performance in scrittura. L’aumento della velocità di scrittura si ottiene come conseguenza della politica di *locking* utilizzata dal database, introdotta nella sezione 5.1.1. I write-lock di MongoDB permettono di eseguire una operazione di scrittura per volta, accodando tutte le altre richieste di inserimento o aggiornamento dei dati. In caso di configurazione cluster, il write-lock avviene a livello di singolo shard. I diversi server possono quindi eseguire una reale parallelizzazione delle operazioni, ottenendo un incremento della velocità di scrittura globale all’interno del cluster.

Replica Set

L'alta affidabilità di MongoDB è garantita ridondando i dati, mantenendo, quindi, più copie dello stesso documento all'interno di server differenti. Nel caso di guasto ad uno dei server e di perdita del dato originale, il sistema è in grado di fornire una copia valida dell'informazione richiesta, ottenendola da un server attivo.

Un Replica Set è un insieme di istanze di MongoDB contenenti gli stessi dati. Uno dei server è definito *primary* e riceve dai client le richieste di scrittura. Il primary esegue l'aggiornamento del suo *data-set* locale e, successivamente, inoltra le richieste verso i *secondary* server, in modo che anch'essi possano mantenere una copia aggiornata dei dati.

In caso di interruzione del servizio di un server primary, i secondary restanti indicono una votazione per l'elezione automatica di un nuovo primary.

In alcuni casi, la configurazione con Replica Set, può essere utilizzata per incrementare la capacità di lettura dei dati. I client, infatti, possono inviare richieste di lettura dei dati a server differenti, ottenendo così una parallelizzazione delle operazioni. Nel caso di cluster distribuiti a livello *world-wide* è possibile, inoltre, configurare il Replica Set in modo da *avvicinare* le copie di dati alla zona di maggiore fruizione, ottenendo, di conseguenza, un incremento delle performance.

MongoDB in Produzione

L'utilizzo di MongoDB in produzione, presuppone una combinazione delle funzionalità di Sharding e Replica Set. In configurazioni di produzione, infatti ogni shard è un Replica Set, questo approccio permette di ottenere

un cluster in grado di fornire consistenza dei dati e, al tempo stesso, alta affidabilità.

Uno sharded cluster MongoDB possiede le seguenti componenti:

Shard rappresentano i server di *storage* veri e propri. Ogni shard è un Replica Set gestito da processi chiamati `mongod`.

Query Router sono implementati da un *demone* di sistema chiamato `mongos` e si occupano di interfacciare i client con l'intero cluster. A fronte di ogni richiesta di scrittura o lettura, redirigono le operazioni al cluster di competenza, eseguendo, di fatto, il bilanciamento del carico dell'intero sistema. I Query Router sono a loro volta configurati come Replica Set, in modo da garantire alta affidabilità e non divenire un *single point of failure* dell'intero sistema.

Config Server possiedono i *metadati* di configurazione dell'intero cluster, cioè i riferimenti ai dati contenuti in ogni shard. I Query Router interrogano i Config Server per identificare il server contenente i dati richiesti dai client. I Config Server sono processi `mongod` equivalenti a quelli utilizzati per gli shard, ma eseguiti con parametri di configurazione specifici, che ne determinano il comportamento.

La figura 5.11 mostra schematicamente una possibile configurazione di MongoDB per il deploy in produzione. I server mole e mole-suit contattano i Query Router e interagiscono esclusivamente con i server `mongos`. I Query Router utilizzano quindi le informazioni contenute nei Config Server per identificare uno specifico shard da contattare al fine di completare la richiesta in arrivo dai server di Mole.io

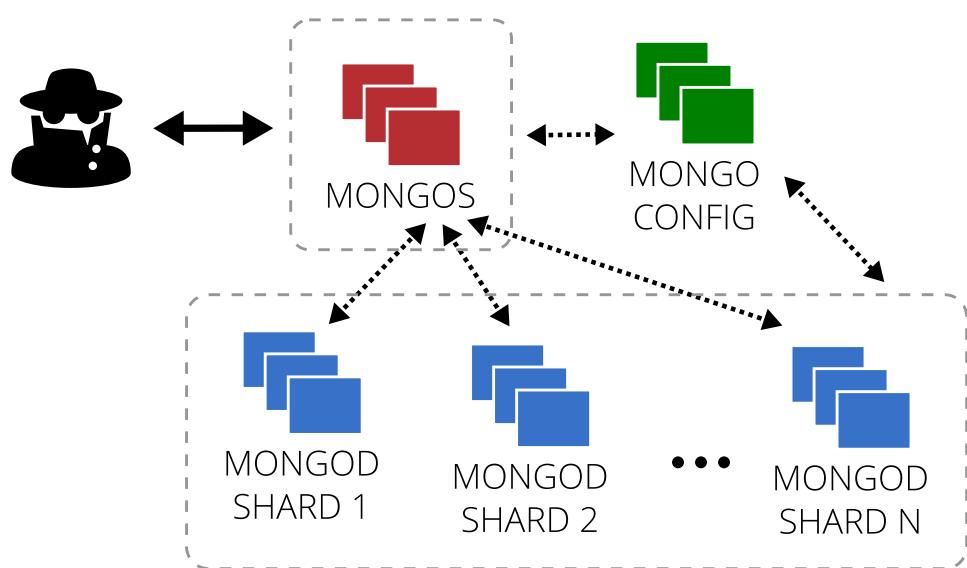


Figura 5.11: Configurazione di MongoDB per Mole.io

Capitolo 6

Configurazioni e Benchmark

In un sistema che si occupa di raccogliere dati, la porzione di sistema più delicata è proprio il salvataggio delle informazioni.

Si è quindi scelto di testare il sistema cercando di mettere stress, procedure di scrittura dati nel sistema

Per il *testing* è stata quindi posta l'attenzione al
il punto critico del sistema è il db e principalmente le op di scrittura
abbiamo fatto uno script per testare le performance in scrittura e abbia-
mo cercato di capire come ottimizzare il db per le scritture

tre config di sharding 1 shard 2 shard id 2 shard hashed
parlare del balancer dei lock di scrittura
come abbiamo eseguito i test:

3 VM 512mb ram, 1,2 ghz 1. mongos+config 2. mongod s1 3. mongod

s2

media su 3 run

| Client | 1 Shard | 2 Shard (id) | 2 Shard (hashed id) |
|--------|---------|--------------|---------------------|
| 1 | 0.239 | 0.172 | 0.285 |
| 10 | 0.451 | 0.442 | 0.448 |
| 20 | 0.884 | 0.909 | 0.983 |
| 30 | 1.310 | 1.395 | 1.460 |
| 40 | 1.776 | 1.859 | 1.960 |
| 50 | 2.281 | 2.319 | 2.525 |
| 60 | 2.751 | 2.862 | 3.112 |
| 70 | 3.275 | 3.484 | 3.925 |
| 80 | 3.798 | 3.910 | 4.525 |
| 90 | 4.201 | 4.358 | 4.856 |
| 100 | 4.784 | 4.967 | 5.604 |

Tabella 6.1: Tabella small

MongoDB Write Test with Small Docs (250 b)

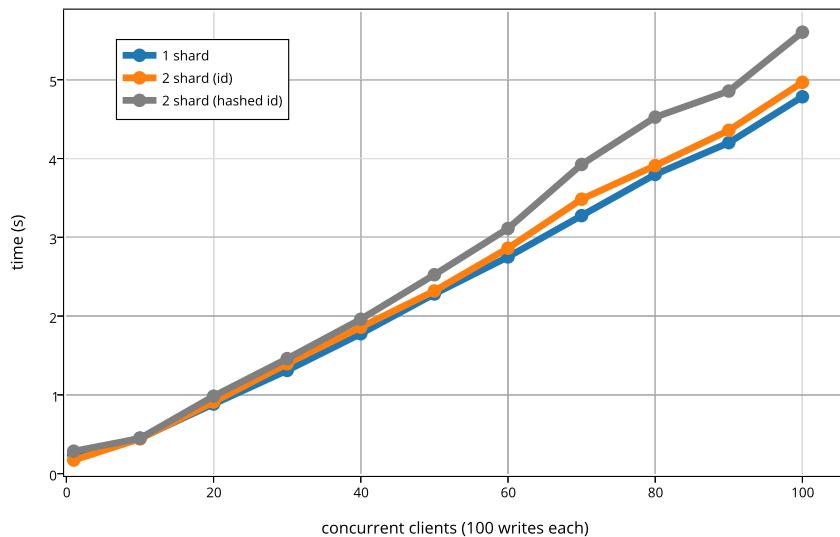


Figura 6.1: small

| Client | 1 Shard | 2 Shard (id) | 2 Shard (hashed id) |
|--------|---------|--------------|---------------------|
| 1 | 0.682 | 0.828 | 0.692 |
| 10 | 2.165 | 2.853 | 2.095 |
| 20 | 7.009 | 7.243 | 4.290 |
| 30 | 8.753 | 10.141 | 6.335 |
| 40 | 14.535 | 15.912 | 9.089 |
| 50 | 31.931 | 43.645 | 12.949 |
| 60 | 30.808 | 24.503 | 15.000 |
| 70 | 46.350 | 59.534 | 19.179 |
| 80 | 32.045 | 43.703 | 28.898 |
| 90 | 114.287 | 97.787 | 25.843 |
| 100 | 127.658 | 88.600 | 32.171 |

Tabella 6.2: Tabella big

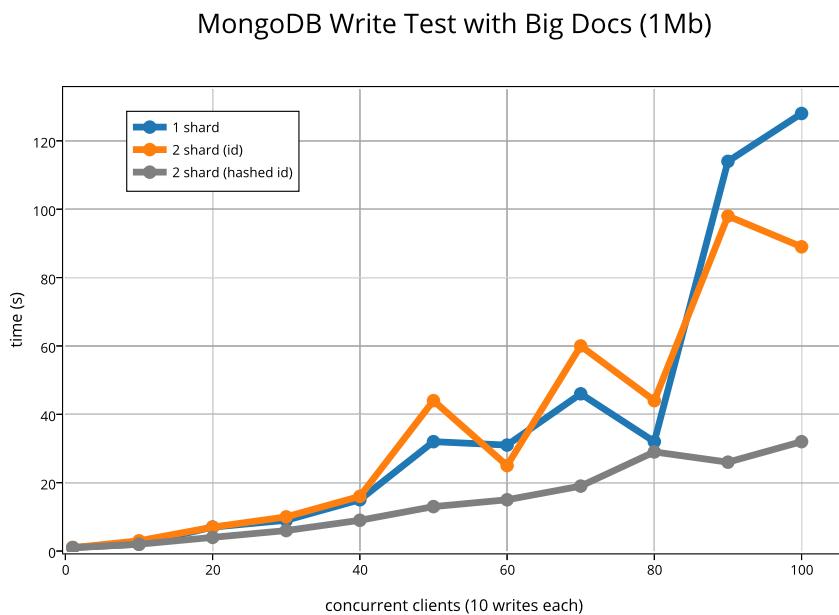


Figura 6.2: big

Conclusioni e Sviluppi Futuri

testere il sistema in piccolo è difficile, provare test sul sistema deployato

Bibliografia

- [1] Emanuele DelBono. Codiceplastico website, 2011.
- [2] Engineering Ingegneria Informatica S.p.A. Spagobi website, 2006.
- [3] Inc Rackspace US. Airbrake website.
- [4] Michael Smathers. Log.io website, 2011.
- [5] Brian Rue. Rollbar website, 2004.
- [6] Papertrail Inc. Papertrail website, 2010.
- [7] Inc. Treasure Data. Fluentd website, 2011.
- [8] Elasticsearch. Elasticsearch website, 2009.
- [9] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.
- [10] Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011.
- [11] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

- [12] Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [13] A. Osmani. *Learning JavaScript Design Patterns.* JavaScript and jQuery developer's guide. O'Reilly Media, Incorporated, 2012.
- [14] M. Fugus. *Functional JavaScript: Introducing Functional Programming with Underscore.js.* O'Reilly Media, 2013.
- [15] Inc Joyent. Node.js website, 2009.
- [16] 10Gen. Mongodb website, 2008.
- [17] Rabbit Technologies Limited. Rabbitmq website, 2006.
- [18] David Liu. Bootstrap website, 2010.
- [19] Sindre Sorhus. Bower website, 2012.
- [20] Jeff Lindsay. Dokku project on github, 2013.
- [21] Inc Joyent. Node package manager (npm) website, 2010.
- [22] Google. Angularjs website, 2010.
- [23] D. Crockford. *JavaScript: The Good Parts: The Good Parts.* O'Reilly Media, 2008.
- [24] Jason Sanford. Leaflet website, 2012.
- [25] The New York Times. Docker website, 2011.
- [26] Google Inc. Yeoman website, 2012.
- [27] Ben Alman. Grunt website, 2012.

- [28] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Series. O'Reilly Media, Incorporated, 2010.
- [29] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, New York, 1988.