Metricio

Installazione e avvio

Aprire una shell nella cartella contenente i sorgenti ed eseguire:

```
npm install
# successivamente
npm run production
```

All'avvio, Metricio provvederà a creare (se non presente) un utente admin, una sua dashsuite e una sua dashboard completa di widgets.

Se al primo avvio sulla dashboard non viene visualizzato alcun valore, niente panico! Metricio genera una piccola cache ogni volta che vengono eseguiti i vari jobs e vi è almeno 1 client in ascolto. I jobs vengono eseguiti ogni minuto quindi entro i prossimi 60 secondi qualcosa verrà mostrato.

Docker

E' presente anche un `Dockerfile` nel caso in cui sia necessario mettere Metricio all'interno di un container.
Per un ambiente docker preparato al volo è presente anche un file `docker-compose.yaml`

Introduzione

La struttura base di Metricio si componeva di una o più dashboards descritte a codice le quali erano raggiungibili indicandone il nome nell'url. I dati dei singoli widget erano inoltre ottenuti da dei 'Jobs'.

Ogni Job era un insieme di una o più task che venivano schedulate atomicamente ed eseguite a periodi regolari. Ogni Job ritornava dunque un hashmap con come chiavi i valori delle tasks e valori i valori ritornati dall'esecuzione delle tasks: al widget era dunque sufficiente indicare il nome della task da cui prendere il valore da visualizzare.

I vari jobs e le annesse tasks erano descritte anch'esse a codice. I limiti di utilizzo di tale organizzazione erano evidenti. Questa struttura è stata dunque profondamente rivista.

Allo stato attuale in Metricio la definizione di dashboards e jobs è stata resa totalmente user-friendly, ma la struttura risultante non è più così elementare.

Funzionamento

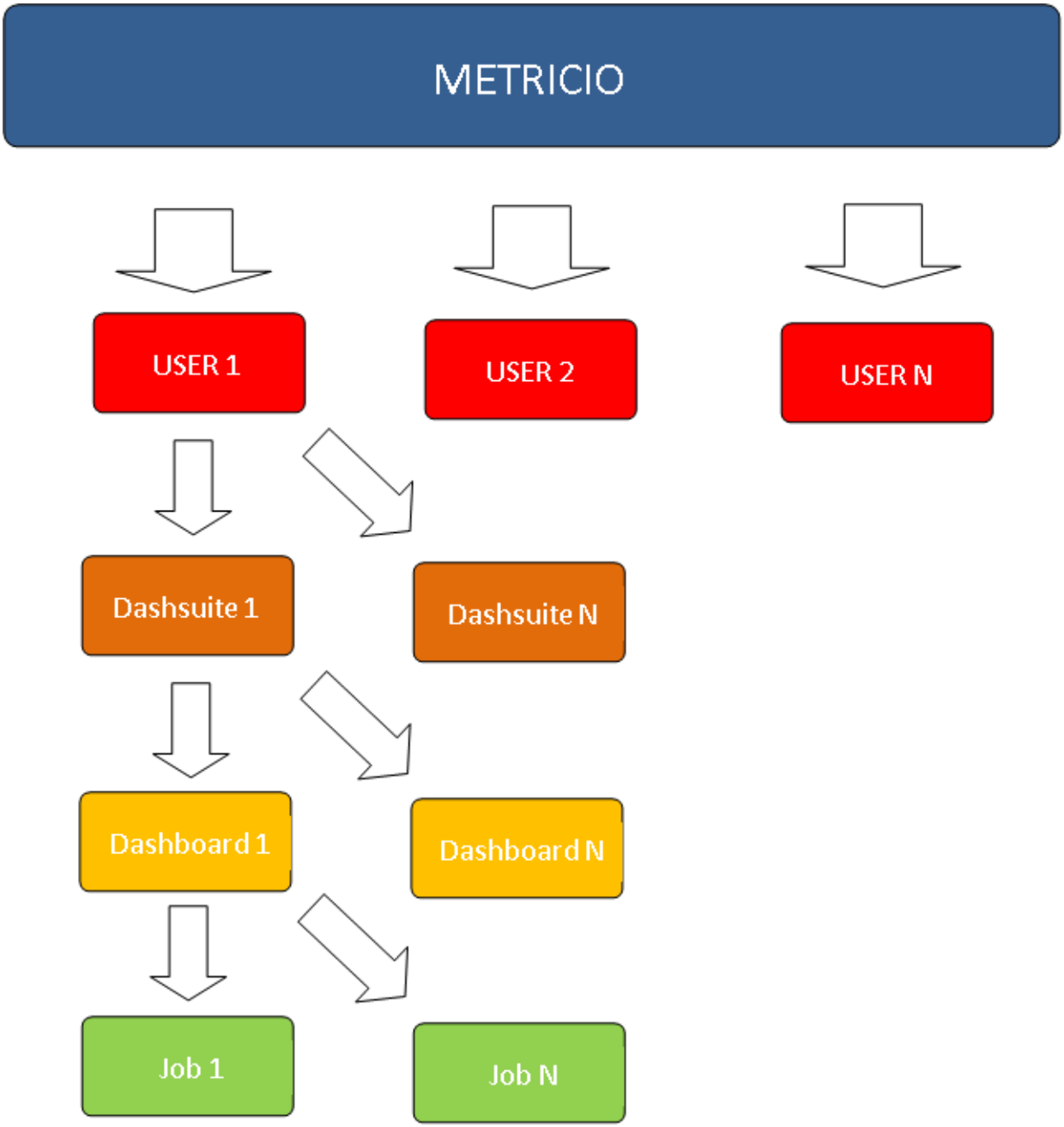
In Metricio ci sono 4 principali entità distinte di seguito elencate in ordine di gerarchia:

1. **User**
2. **Dashsuite**
3. **Dashboard**
4. **Job**

Ogni entità che sta più in alto nella gerarchia vede e include una o più istanze delle entità inferiori.

In parole più semplici: Metricio può gestire uno o più utenti (User); ogni user può avere una o più dashsuite; ogni dashsuite possiede una o più dashboards; ogni dashboard possiede uno o più jobs.

Nel caso non fosse ancora chiaro si osservi la seguente figura.



Di seguito il significato delle varie entità:

- **User**: un utente generico.
- **Dashsuite**: una collezione di Dashboards appartenenti ad uno stesso contesto funzionale.
- **Dashboard**: un contenitore di widgets che visualizzano i dati ottenendoli da una task di un Job. Widgets diversi possono ottenere dati da tasks di Jobs diversi.
- **Job**: un contenitore di tasks eseguito ad un intervallo regolare definito appartenente ad una ed una sola Dashboard.

L'intero progetto è stato strutturato inoltre per supportare ulteriori espansioni di tipologie di widgets e jobs senza (quasi) alcun code-refactoring. Questo sviluppo

“generalizzato” è stato reso necessario dalla poca chiarezza del formato e delle metodologie di accesso ai dati da visualizzare.

Ciò nonostante il risultato di questa strada di sviluppo ha portato ad uno scheletro ben strutturato espandibile per ambiti di visualizzazione diversi dalla esclusiva visualizzazione per il progetto Disloman.

Architettura

Dal punto di vista architetturale, Metricio si appoggia a due databases:

- **Redis**: un database NoSQL in-memory di tipo key-value, utilizzato come cache per lo scheduling dei jobs e la memorizzazione dei risultati degli stessi.
- **MongoDB**: un database NoSQL document-oriented, utilizzato per la memorizzazione persistente dello stato di tutte le entità sopracitate.

Il server è sviluppato in Node.Js sfruttando principalmente il modulo Express.Js, che fornisce efficaci capacità di routing delle richieste all’interno del server.

Sono state così definite 4 principali routes:

- `/users`
- `/dashsuites`
- `/dashboard`
- `/jobs`

La route base / porta alla home.

Ogni endpoint di ogni route sfrutta un “manager”. Un manager è un modulo che si occupa di svolgere tutte le operazioni di modifica della entità a cui corrisponde e di scriverla su un database.

Vi sono dunque 4 managers:

- usersManager
- dashsuitesManager
- dashboardManager
- jobsManager

Ogni manager non ha dipendenze verso altri managers. Ogni manager notifica attraverso un bus interno ogni operazione di **scrittura** su db e può consumare notifiche

provenienti da altri managers.

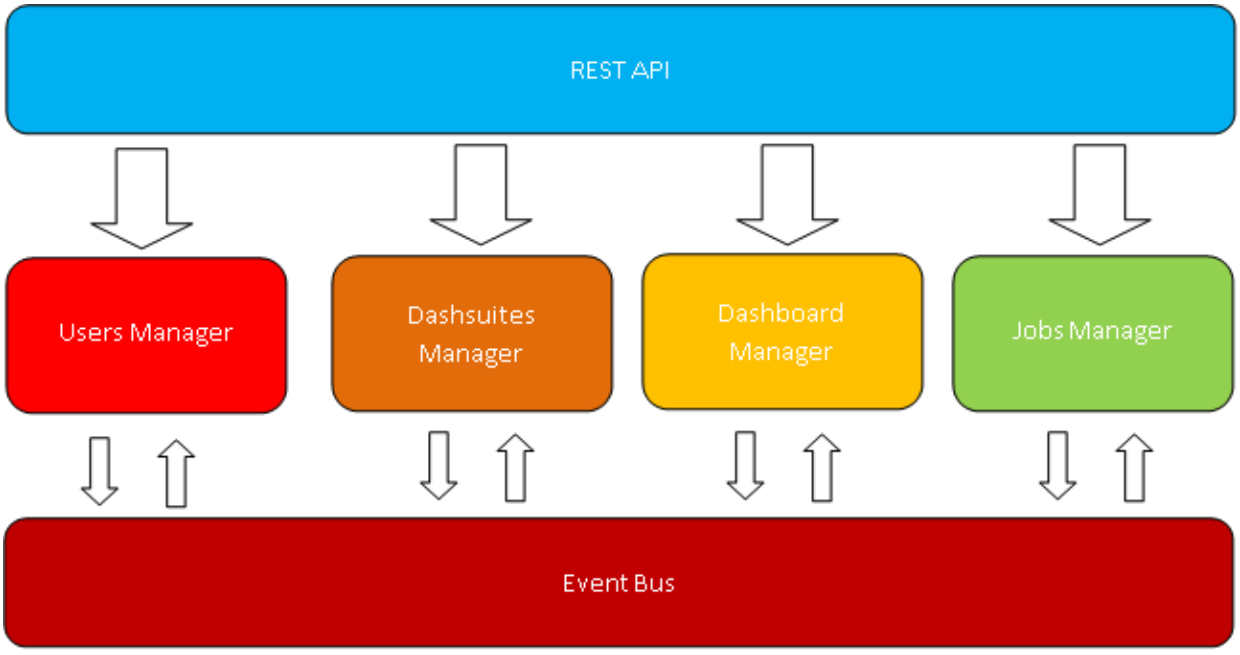
Questo comporta così ad avere una struttura di entità sviluppate in modo indipendente tra loro dove dunque la modifica della logica di una operazione di scrittura non impatta sul funzionamento delle altre entità fintanto che queste continuano a venire notificate.

Il lato negativo è che un eventuale operazione di eliminazione di un entità madre, può non avvenire a cascata anche sulle entità figlie in caso di collasso del server per motivi non noti.

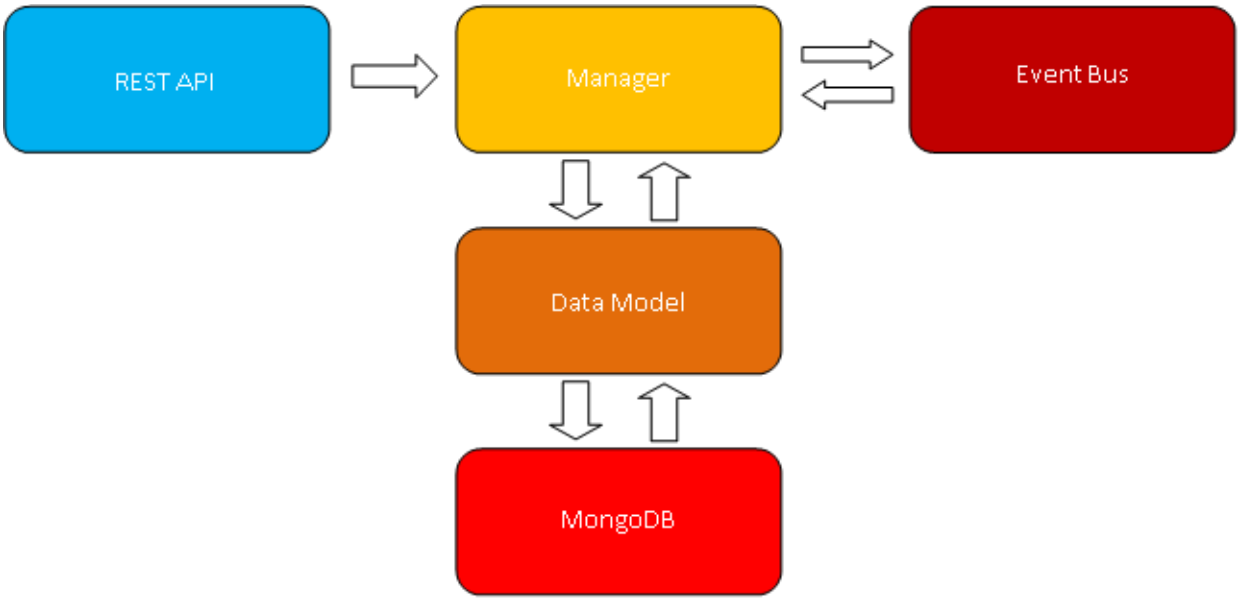
Tuttavia, a livello visivo questo non dovrebbe essere un problema. La consistenza può essere riottenuta eliminando ogni entità figlia non referenziante un entità madre (non esiste un documento di un entità madre con ObjectId pari a quello indicato dal documento dell'entità figlia).

Nel caso in cui la transazionalità si ritiene sia necessaria 3 differenti strategie sono elencate sotto il capitolo **Transazionalità**.

L'architettura risultate attualmente è dunque la seguente:



Ogni manager riceverà comandi dal REST API layer, effettuerà query/updates su MongoDB attraverso un data model (in particolare messo a disposizione da Mongoose.Js) e pubblicherà eventi nell' Event Bus:



Organizzazione del progetto

Il progetto è stato organizzato per mantenere la più netta separazione possibile tra client e server.

Tutto ciò che si trova sotto la cartella `/src` appartiene al client e non ha (quasi) dipendenze con quello che sta fuori che appartiene invece al server.

Client

In particolare sotto `/src` si trovano:

- `/dashboard`: contenente il React Router, unico punto di ingresso nel client.
- `/jobs`: contenente i jobs originari definiti in javascript. Il supporto a questa tipologia di job è stato mantenuto.
- `/lib`: contiene tutte le librerie di funzionalità utilizzate dal client
- `/react-elements`: contenente elementi di React
- `/react-views`: contenente viste utilizzate dal React router per la navigazione
- `/styles`: alcuni stylesheets in scss
- `/views`: contiene la pagina html di base che viene renderizzata e inviata a chi fa richiesta di utilizzare il client.

- `/widgets` : contiene tutti i widgets e la struttura della dashboard di base.

Server

Fuori da `/src` si trovano:

- `/lib` : contenente tutte le librerie di funzionalità utilizzate dal server.
- `/listeners` : contenente il modulo dell'EventBus.
- `/managers` : contenente i moduli dei vari managers.
- `/model` : contenente i moduli dei data models di Mongoose.Js
- `/routes` : contenente i moduli Express Routes utilizzati come middlewares da Express in /app.js

Espansioni dei widgets e dei jobs

Definizione di un nuovo widget

Per la definizione di un nuovo widget è necessario:

- Definire una nuova classe che estenda `/src/widgets/base.jsx`
- Aggiungere il modulo come dipendenza in `/src/widgets/Widgets.js`

E' richiesto che venga implementato il getter `static get className()` il quale deve ritornare il nome letterale della classe.

Esempio:

`/src/widgets/newWidget.jsx`

```
import React from 'react';
import BaseWidget from './base.jsx';

export default class MyNewWidget extends BaseWidget {
    static get className() {
        return 'MyNewWidget';
    }
```

```
    render() {
        return (
            <div className="helloWord">
                <h1>Hello world!</h1>
            </div>
        );
    }
}
```

`/src/widgets/Widgets.js`

```
import MyNewWidget from './newWidget';
// ...
const Widgets = {};
// ...
Widgets[MyNewWidget.className] = MyNewWidget;
```

Definizione di un nuovo Job

Per la definizione di un nuovo Job è necessario prima di tutto definire una nuova tipologia di Job.

Dunque sotto `/lib/jobs` (a partire dalla cartella contenente l'intero progetto) creare un nuovo modulo che esporti una classe che estenda `/lib/jobs/util/baseJob.js`.
Il costuttore di baseJob è il seguente: `constructor(jobName, interval)`

In particolare è richiesto che la classe implementi i seguenti metodi:

- `static get className()`
- `static fromObject(job, vars)`

Il primo dovrà ritornare il nome esatto della classe come stringa.

Il secondo dovra ritornare una nuova istanza della classe a partire dall'oggetto job e dall'oggetto vars.

`job` è una struttura rappresentate il job, e `vars` racchiude i valori dei parametri delle tasks parametrizzate. Ogni task parametrizzata avrà una o più proprietà di tipo string con la quale è possibile ottenerla valorizzata sfruttanto il metodo `this.getValueAll(string, vars)`.

Dopo aver definito la nuova classe di Job è necessario aggiungere la dipendenza a `/lib/jobs/index.js`.

Esempio:

`/lib/jobs/myNewJob.js`

```
import BaseJob from './util/baseJob';
import Task from './util/task';

export class MyNewJob extends BaseJob {
    // Custom constructor
    constructor (jobName, interval, moreParams) {
        super(jobName, interval);
        // More stuff...
        // Otherwise constructor is optional.
    }

    static get className() {
        return 'MyNewJob';
    }

    static fromObject(job, vars) {
        const newJob = new MyNewJob(job.jobName, job.interval);
        job.tasks.forEach(t => newJob.addTask(t));
        return newJob;
    }

    addTask(t) {
        this.tasks.push(new Task(t.taskName, () => console.log('NEW TASK!')
    }
}
```

`/lib/jobs/index.js`

```
// ...
export * from './myNewJob';
```

La logica di esecuzione del job e quindi delle rispettive tasks è già inclusa in `/lib/jobs/utils/baseJob.js`.

Estendendo `BaseJob` ogni nuova istanza della nostra classe avrà le seguenti proprietà:

- `jobName: string`
- `interval: string`
- `tasks: Array`
- `perform: Function`

Le prime due sono le stesse passate come parametri nel costruttore.
`tasks` è un array di `Task` (classe importabile da `/lib/jobs/util/task`).
`perform` invece è la funzione eseguita periodicamente dallo scheduler: è **caldamente** consigliato non modificarne il valore, a meno che si voglia creare qualcosa di più complesso. In quest'ultimo caso si consiglia di guardare come avviene l'interazione tra `baseJob.js` e `task.js` e come il modulo npm `node-resque` funziona.

Ogni `Task` richiede come parametri nel costruttore il nome della task (`taskName`) e una funzione rappresentante il compito della task da eseguire (`execute`) che ritorni un qualsiasi valore rappresentante il risultato dell'esecuzione di tale task.

Nota: per l'export è **necessario** utilizzare `export` anziché `module.exports` o `export default`

Transazionalità

La nuova architettura di Metricio pecca della mancata transazionalità in caso di updates in cascata (es. eliminazione di una dashsuite che deve corrispondere alla conseguente eliminazione di tutte le sue dashboards e all'eliminazione di ogni dashboard deve corrispondere l'eliminazione di tutti i jobs correlati).
Questo significa che nel caso di collasso del server per motivi non noti, l'update in cascata si verificherà parzialmente (es. viene eliminata la dashsuite, tutte le dashboards, ma non i jobs).

Nel caso in cui la transazionalità sia un requisito esseziale di seguito sono riportate 3 strategie per ottenerla.

Multi-document transaction

La transazionalità può essere ottenuta attraverso una transazione multi-documento disponibile a partire da MongoDB 4.0 (disponibile solo per server db in replica set, non per db standalone. Sarà funzionante anche su un db sharded a partire dalla 4.2).

La strategia di esecuzione sarebbe dunque la seguente:

- Richiesta di eliminazione
- Presa in carico dal manager adeguato che:
    - Inizia una transazione
    - Emette un evento di notifica sul bus

- Ogni altro manager che si sottoscrive a tale evento dovrà farlo in maniera **sincrona** e aggiungere all'oggetto "transazione" ricevuto dalla notifica le operazioni da fare.
- Il manager notificante effettua il commit.

Ogni passaggio riportato deve avvenire necessariamente in maniera **sincrona**. Diversamente il commit avverrà prima dell'aggiunta delle operazioni da svolgere da parte dei manager subscribers.

Ciò comporterebbe un minimo cambiamento nei moduli dei managers (e quindi dell'application logic), ma una possibile completa revisione dei moduli appartenenti al layer di interfacciamento al db.

Single-document transaction

MongoDB garantisce come atomiche tutte le operazioni effettuate sul medesimo documento.

Nel caso quindi in cui un MongoDB server in replica set non sia un opzione fattibile, è necessario creare un solo documento per ogni utente contenente tutti i sottodocumenti necessari per ogni sotto entità.

In tal caso, il procedimento per la modifica è:

- Presa in carico del manager notifier che:
  - Ottiene il documento e applica le sue modifiche
  - Pubblica un evento nel bus passando l'oggetto documento come riferimento.
- Ogni manager subscriber applica le sue modifiche all'oggetto in maniera **sincrona**.
- Il manager notifier avvia l'update sul db.

Ogni passaggio riportato deve avvenire necessariamente in maniera **sincrona**. Anche in questo caso il layer di interfacciamento a MongoDB sarebbe totalmente rivisto, e l'implementazione della logica di reazione agli eventi potrebbe non essere banale.

Cambio di database

Un'ultima opzione è abbandonare MongoDB per un database più consono (eventualmente anche relazionale) con la possibile revisione parziale dell'architettura generale.