NotesHub

Servizio di appunti cloud

RELAZIONE PROGETTO CLOUD COMPUTING

Bacciottini Leonardo

Barsellotti Luca

Guggino Filippo

Paolini Emilio

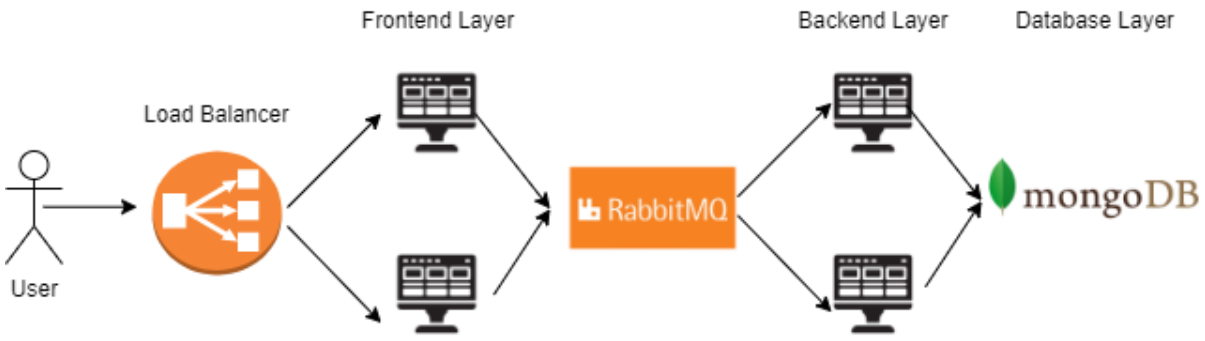
Sommario

Introduzione

Scopo del progetto è la creazione di un'applicazione che gestisca gli appunti/note di vari utenti. In particolare, è possibile creare un nuovo appunto, modificare i suoi campi, eliminare una nota giá inserita. È inoltre possibile cercare le note per materia, per autore o per data. L'applicazione è stata sviluppata utilizzando un approccio multi-tier, dove ogni livello offre delle funzionalitá specifiche. Nel seguito verrá descritta in dettaglio l'architettura dell'applicazione in ogni sua componente e come l'utente puo' interagire con essa.

Architettura

Un primo sguardo all'architettura dell'applicazione puo' essere visto nella seguente figura:

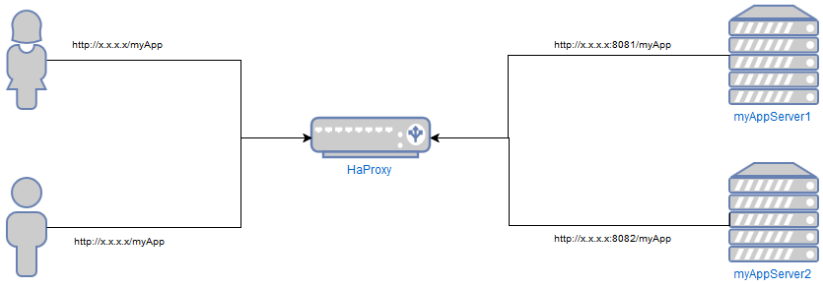


L'utente puo' interagire con l'applicazione semplicemente mandando delle richieste REST tramite una qualsiasi interfaccia che lo permetta (ad esempio tramite un plugin installato sul browser o

usando la riga di comando tramite comando CURL). A quel punto il load balancer, implementato utilizzando Haproxy, gestisce la richiesta inoltrandola ad uno dei due frontend. In quest'ultimo, la richiesta viene preparata per essere mandata al backend. Per garantire la scalabilitá nella comunicazione tra backend e frontend è stato utilizzato RabbitMQ. Successivamente la richiesta viene prelevata da uno dei due backend, i quali interagiscono con il database, implementato in MongoDB, e restituiscono la risposta al frontend. Infine la risposta viene restituita all'utente.

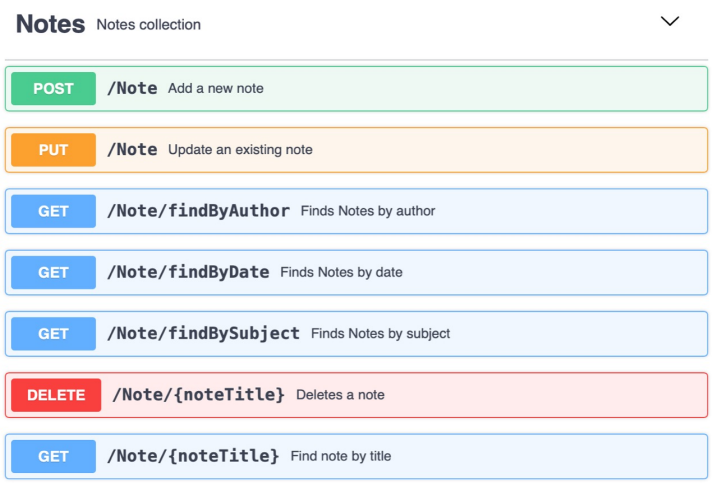
Load Balancer

Come già accennato in precedenza, il load balancer è stato implementato utilizzando HAProxy. Per quanto riguarda la sua configurazione, la scelta è stata quella di utilizzare round robin come algoritmo di load balancing. Utilizzando questo algoritmo, il load balancer considera i server disponibili come una lista circolare e le richieste vengono assegnate in modo sequenziale ai server. In particolare, nel nostro caso i frontend su cui smistare le richieste erano due. HAproxy è stato fatto eseguire all'interno di un container, presente nella seguente macchina: 172.16.1.136, in ascolto sulla porta 8081.

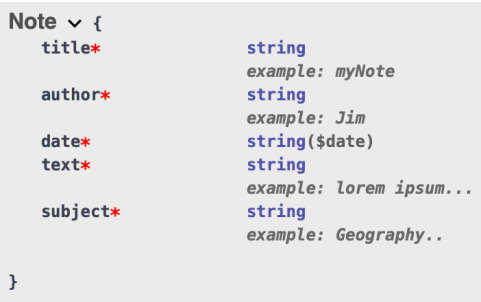


Frontend

L'interfaccia disponibile all'utente è stata implementata utilizzando swagger e Flask. In particlare, l'interfaccia REST mette a disposizione dell'utente le seguenti funzioni:



È quindi possibile aggiungere una nuova nota, aggiornare una nota esistente, trovare una nota per titolo o eliminare una nota. Inoltre è possibile ottenere piú note contemporaneamente cercandole per autore, data o materia. Una nota è caratterizzata dai seguenti campi:



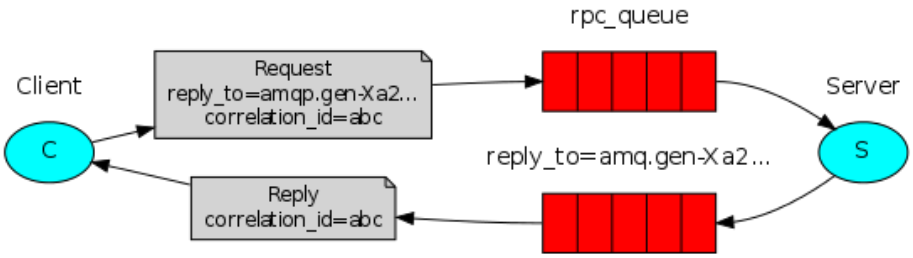
Quando un frontend invia una richiesta, essa è codificata come un dictionary (quindi una struttura costituita da coppie chiave-valore). All'interno di tale struttura è presente un campo "requestType" per specificare il tipo di richiesta e in base ad esso saranno determinati gli altri campi presenti. Nella seguente tabella è sintetizzato il comportamento di tale struttura:

requestType	**Campi struttura**
newNote	*title, author, date, text, subject*
deleteNote	*title*
findNotesByAuthor	*author*
findNotesByDate	*date (format of date is yyyy-mm-dd)*
findNotesBySubject	*subject*
getNoteByTitle	*title*
updateNote	*title, author, date, text, subject*

Ogni frontend è stato fatto eseguire all'interno di un container. In particolare, i due container si trovano sulle seguenti macchine: 172.16.2.40, 172.16.1.191.

RabbitMQ

RabbitMQ è stato utilizzato per realizzare una comunicazione scalabile tra frontend e backend. Per gestire in modo appropriato le richieste da parte dei frontend nei backend, la configurazione di rabbitMQ utilizzata è stata quella a Remote Procedure Call (RPC), che utilizza comunquedelle working queue per fare in modo che le richieste dei frontend vengano smistate tra i vari backend disponibili. La seguente immagine spiega il suo funzionamento:



Il client (nel nostro caso il frontend) manda un messaggio contenente la richiesta al server (nel nostro caso il backend) attraverso una coda. Il server esegue l'elaborazione della richiesta e pubblica il risultato della computazione in una coda, specificando a quale client appartiene tale risposta. Così facendo, il client può recuperare la risposta a lui appartenente. Anche rabbitMQ è stato eseguito all'interno di un container, presente sulla macchina: 172.16.3.34.

Backend

Per quanto riguarda il backend, esso non è altro che una semplice applicazione python che utilizza rabbitMQ per ottenere le richieste, le processa a seconda del valore di requestType, interagisce con il database MongoDB effettuando query e infine restituisce il risultato sempre utilizzando rabbitMQ. In particolare, sono presenti due backend, ognuno su un container, nelle seguenti macchine: 172.16.2.40, 172.16.1.191.

MongoDB

Il database è stato implementato utlizzando MongoDB, questo per offrire risposte più veloci e interagire in modo più rapido con le richieste, che sono sotto forma di dictionary python. Il database contiene una solo collection, Notes. Per mostrare l'immediatezza di MongoDB rispetto ad altre soluzioni, come MySQL, si riporta un codice che gestisce l'inserimento di una nuova nuota nel database:

```
def new_note(request):
    title = request['title']
    author = request['author']
    date = request['date']
    text = request['text']
    subject = request['subject']
    dict = {"title" : title, "author" : author, "date" : date, "text" : text, "subject" : subject}
    query = {"title" : title}
    if collection.count_documents(query) != 0:
      return "This title already exists"
    collection.insert_one(dict)
    return "The note has been inserted"
```

ZooKeeper

ZooKeeper è un server che permette la condivisione di parametri tra le varie componenti dell'applicazione. Il suo utilizzo è molto semplice: uno script python inizializza ZooKeeper, inserendo al suo interno delle variabili, come è mostrato dal seguente codice:

```python
from kazoo.client import KazooClient

zk = KazooClient(hosts='172.16.3.34:2181', read_only=True)
zk.start()
#Store the data
zk.ensure_path("/credentials/rabbitmq")
zk.create("/credentials/rabbitmq/username", b"guest")
zk.create("/credentials/rabbitmq/password", b"guest")
```

In questo caso stiamo salvando dentro ZooKeeper, al percorso "/credentials/rabbitmq", username e password per accedere a RabbitMQ (nell'esempio "guest", "guest").

I parametri che vogliamo salvare su ZooKeeper sono i seguenti: credenziali, indirizzo e porta di RabbitMQ, credenziali, indirizzo e porta di MongoDB, la lunghezza massimo del testo delle note, e la coda di rabbitMQ per far comunicare frontend e backend. Una volta salvati, ogni modulo dell'applicazione che necessita di tali parametri ci puo' facilmente accedere, utilizzando un client ZooKeeper. Per fare ciò abbiamo definito una classe ZooClient, i cui metodi ci permettono di ottenere i parametri salvati sul server ZooKeeper. Uno snapshot del codice della classe client è il seguente, in cui viene mostrato il costruttore e il metodo per ottenere le credenziali di rabbitMQ.

```python
from kazoo.client import KazooClient

class ZooClient:

    def __init__(self, ipAddress):
        self.client = KazooClient(hosts=ipAddress, read_only=True)

    def getRabbitCredentials(self):
        self.client.start()
        username, stats= self.client.get("/credentials/rabbitmq/username")
        psw, stats = self.client.get("/credentials/rabbitmq/password")
        username = username.decode("utf-8")
        psw = psw.decode("utf-8")
        self.client.stop()
        return username, psw
```

Infine, per quanto riguarda la configurazione di ZooKeeper, esso è stato clusterizzato: ci sono quindi due istanze del server ZooKeeper in esecuzione, ognuna su un container diverso. Le macchine su cui sono presenti questi due container sono: 172.16.1.191 e 172.16.2.40.