Relazione Homework
Progettazione Sistemi Distribuiti

Alessandro Messina, matricola O55000354

Orazio Scavo, matricola O55000414

Homework 1: homework 12

Homework 2: variante 3

ANNO ACCADEMICO 2018/2019

Sommario

1. Introduzione

L'elaborato prevedeva l'implementazione di un sistema di File Storage Management come descritto nell'Homework12 (per quanto riguarda la prima parte) e della relativa distribuzione con persistenza mediante 5 repliche attive (per quanto riguarda la seconda parte).

1.1 Scelte architetturali

L'applicazione si compone di diversi moduli comunicanti tramite REST/HTTP, così come mostrato nel *deployment diagram* a seguire (Fig. 1).

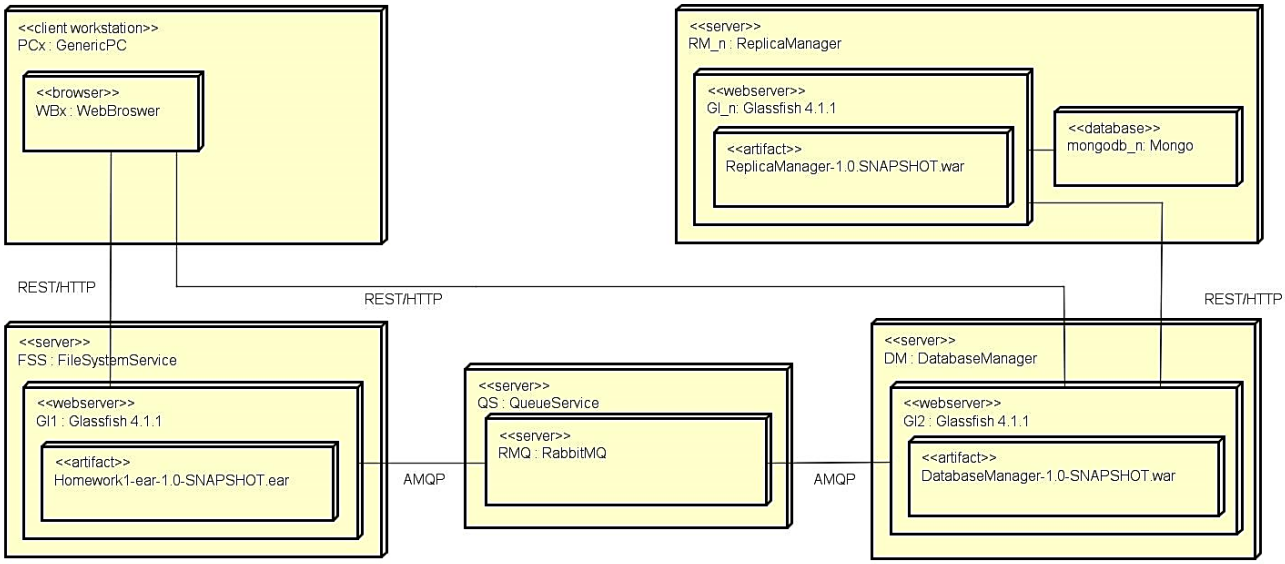

Figura 1. Architettura del sistema

- ***FileSystemService***. Gestisce il servizio di File System e dei relativi test (Load Generator).
  - La realizzazione del modulo è stata effettuata utilizzando un progetto *Maven Application* nel quale sono presenti: la Web Application (.war) contente una servlet per il Load Generator e un REST web service, e un EJB (.jar), gestiti dal .ear che viene deployato su container Docker.
  - È stato utilizzato Glassfish 4.1.1 come web server per eseguire il modulo.
  - Il modulo offre un'interfaccia grafica per l'avvio del test e per l'utilizzo del File System.
- ***RabbitMQ***. RabbitMQ è stato utilizzato come servizio di broking tra FileSystemService (che invia i risultati dei test) e DatabaseManager (che rappresenta il writer/reader FE verso il sistema di repliche per la persistenza dei dati) in modalità *Basic Queue* (una semplice coda che disaccoppia le due parti).
- ***DatabaseManager***. Il DatabaseManager è il writer/reader FE verso il sistema di repliche per la persistenza dei dati. Il modulo raccoglie i dati dalla coda di RabbitMQ per salvarli sul sistema di repliche e offre 3 richieste REST per lavorare su tale sistema.
  - La realizzazione del modulo è stata effettuata utilizzando un progetto *Java Web Application*. Il file .war generato dal build dell'applicazione è deployato su container Docker.
  - Il modulo offre anche un'interfaccia grafica mediante la quale è possibile avviare le REST.
- ***ReplicaManager***. Questo modulo rappresenta una singola replica del sistema di storage. La replica gestisce l'accesso ad un database Mongo (uno per replica) offrendo delle REST CRUD su documenti e collezioni (REST utilizzate dal DatabaseManager).
  - Anche questo modulo è stato realizzato utilizzando un progetto *Java Web Application* il cui .war è deployato su container Docker (ogni replica è deployata in un container diverso, ha il proprio database e il proprio volume).

2. FileSystemService

2.1 Servizio di File System

Un generico client (browser web nel diagramma di deployment del capitolo 1) può usufruire dell'applicazione collegandosi all'indirizzo del FileSystemService. Questo modulo consente all'utente, tramite un'interfaccia grafica web user-friendly, di effettuare le azioni di creazione, lettura, modifica e cancellazione di directory e file del file system memorizzato nella macchina su cui il modulo stesso è in esecuzione. Sempre tramite questo modulo un client può avviare il test automatico che consente di rilevare le statistiche di deviazione standard e media relativa alle azioni di creazione e download di file. I risultati di tale test vengono inviati su una coda RabbitMQ mediante protocollo AMQP.

2.2 REST e bean

Nel file *FileSystemResource* sono state implementate le chiamate REST del servizio. Le REST esposte sono delle semplici CRUD su file e directory e utilizzano l'EJB *DirectoryBean* per quanto riguarda la logica di business. All'interno del bean è stata effettuata la manipolazione dei file e delle directory del file system utilizzando le opportune librerie Java.

2.2 Load Generator

La classe *LoadGeneratorServlet* è la servlet che implementa il load generator per il test dell'applicazione e la raccolta delle statistiche. La servlet può essere raggiunta dall'interfaccia grafica utilizzando l'apposito pulsante (posto in alto a destra) e ad ogni richiesta risponderà eseguendo un test completo così come specificato nei requisiti di progetto. La LoadGeneratorServlet si appoggia ad un altro servizio, il *RequestSenderService* per l'invio effettivo delle richieste al backend dell'applicazione. Per ottenere dei risultati significativi le richieste del tester (richieste di add e download) sono parallelizzate (inviate tramite thread): il servizio, una volta inviati tutti i thread, aspetta la risposta da ognuno di essi e la restituisce al LoadGeneratorServlet che li esporrà su console e su interfaccia grafica (Fig. 2) in modo da poter essere facilmente letti dall'utente. La LoadGeneratorServlet invierà inoltre i risultati del test alla coda RabbitMQ.

La LoadGeneratorServlet è stata inserita all'interno del modulo FileSystemService poiché i test da essi effettuati sono mirati alla valutazione delle performance delle operazioni sul file system e non sulla rete. Non è stato quindi necessario collocare la servlet in un diverso componente.

Test Results

Directory	Cycle	Add Mean	Download Mean	Add Standard Deviation	Download Standard Deviation	Cycle state
Directory_0	1	17.4	8.3	4.827007354458868	4.967673276069772	✔
Directory_0	2	76.6	11.8	73.17308248256322	4.802776974487434	✔
Directory_0	3	14.8	6.7	9.011104260855047	3.4334951418181574	✔
Directory_1	1	29.8	11.4	9.57601169589929	6.752777206453652	✔
Directory_1	2	21.2	13.4	5.761944116355173	9.582391258043174	✔
Directory_1	3	28.2	4.1	16.02186006679624	1.1972189997378646	✔
Directory_2	1	13.2	6.7	8.012490249604053	3.8311588035185618	✔
Directory_2	2	28.2	5.6	39.83340306827927	3.3065591380365986	✔
Directory_2	3	41.8	5.7	33.83341543503996	2.540778533354601	✔

Figura 2. Risultati di un test

2.3 Interfaccia grafica

Per esporre al meglio i servizi offerti dall'applicazione è stata realizzata una semplice interfaccia grafica[1] (Fig. 3) mediante angularjs, javascript, HTML5 e css3. Dall'interfaccia è possibile utilizzare tutte le REST del servizio, verificare i path delle varie cartelle, lanciare il tester e andare alla repository ufficiale di git. L'interfaccia ha consentito inoltre di velocizzare la fase di test dell'applicazione.

Una nota importante deve essere fatta per quanto riguarda la manipolazione dei path del file system. Per poter gestire i path nelle chiamate REST (specialmente nelle GET) è stato necessario lavorare con path contenenti asterischi, *, invece che slash, /. Ciò è necessario per la corretta interpretazione degli url per accedere alle REST offerte dal backend dell'applicazione. Sono quindi state create e usate delle opportune funzioni di conversione degli url.

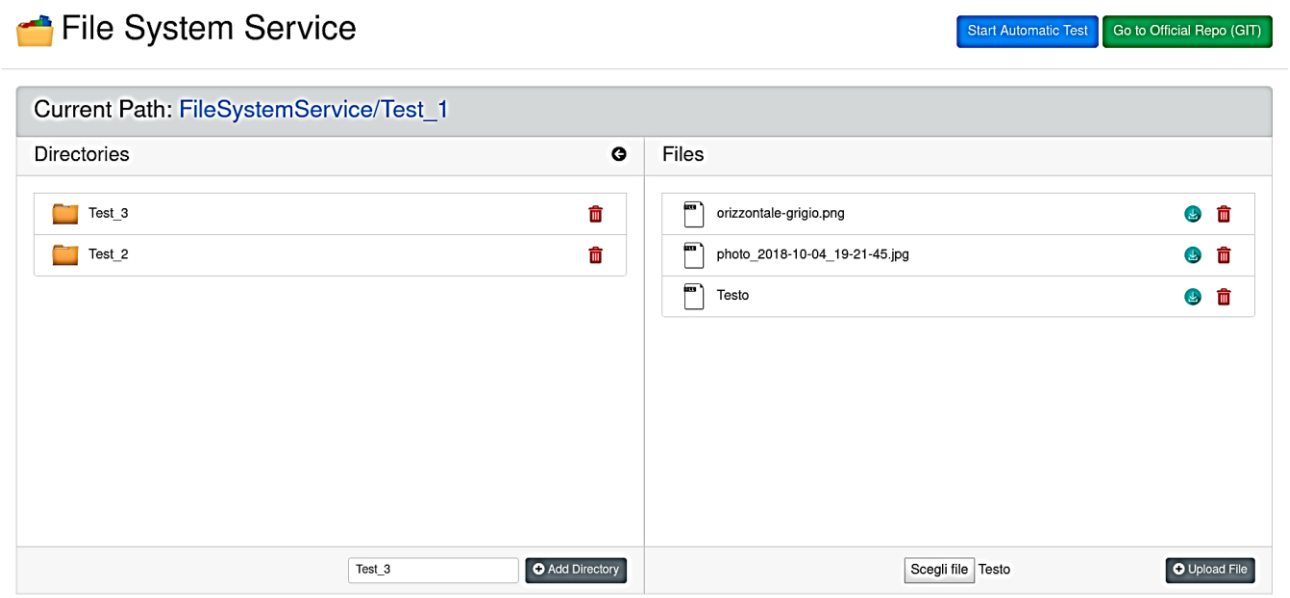

Figura 3. Interfaccia grafica del File System Service

[1] Le interfacce dell'applicazione sono state realizzate per funzionare in maniera ottimale (responsive) su Google Chrome.

3. Database Manager

3.1 Gestore delle repliche

Il database manager rappresenta ciò che nei requisiti di progetto è stato indicato con il termine di *reader/writer FE*. Questo modulo è una web application java che si compone di 3 elementi principali:

- Gestione della coda RabbitMQ (*QueueListener*)
- Gestione delle transazioni sul sistema di repliche (*Transaction Manager*)
- Chiamate REST e interfaccia grafica.

3.1 Queue Listener

Il database manager contiene una classe *QueueListener* che lancia un listener (*ResultReceiver*) costantemente in ascolto sulla coda RabbitMQ in attesa di ricevere dati da parte del FileSystemService. Quando vengono ricevuti dei dati, questi vengono inviati al sistema di repliche per la memorizzazione tramite quorum. Ogni transazione viene gestita dal *TransactionManager* così come descritto in seguito.

3.2 Transaction Manager

Il TransactionManager è il gestore delle transazioni sul sistema di repliche (di cui ne mantiene una lista). Ogni transazione (lettura/scrittura) è atomica e gestita così come indicato nei requisiti del progetto.

- Il TransactionManager è stato implementato come un singleton in modo da poter accedere sempre alla stessa istanza ad ogni transazione e quindi tenere traccia dei sequence number utilizzati.
- Il TransactionManager offre diversi metodi che consentono di effettuare: prima fase del 2PC, seconda fase del 2PC, decisione di scrittura con quorum, decisione di lettura con quorum.
- Le richieste inviate dal TransactionManager sono effettuate tramite thread (PostThread e GetThread) verso le repliche del sistema in modo da parallelizzare l'esecuzione. Il TransactionManager, una volta inviate tutte le richieste per una delle funzioni sopra citate, attende che tutti i thread rispondano in modo da poterne elaborare le risposte in maniera opportuna.

3.3 REST e interfaccia grafica

Il DatabaseManager offre 3 REST all'utente. Queste consentono di: inserire un nuovo documento[2], leggere tutti i documenti di una collezione e leggere l'ultimo documento committato all'interno di una collezione (tutto seguendo la politica del quorum descritta nella consegna dell'elaborato). Le REST fanno uso del TransactionManager e ne richiamano le funzioni che sono state descritte nel paragrafo precedente.

Per poter interagire meglio con il database manager è stata creata un'apposita interfaccia grafica (Fig. 4). Questa consente semplicemente di utilizzare le 3 REST prima descritte e ne visualizza la risposta in formato JSON opportunamente formattato.

[2] È stato utilizzato MongoDB come database, dunque parleremo di *documenti* e di *collezioni*.

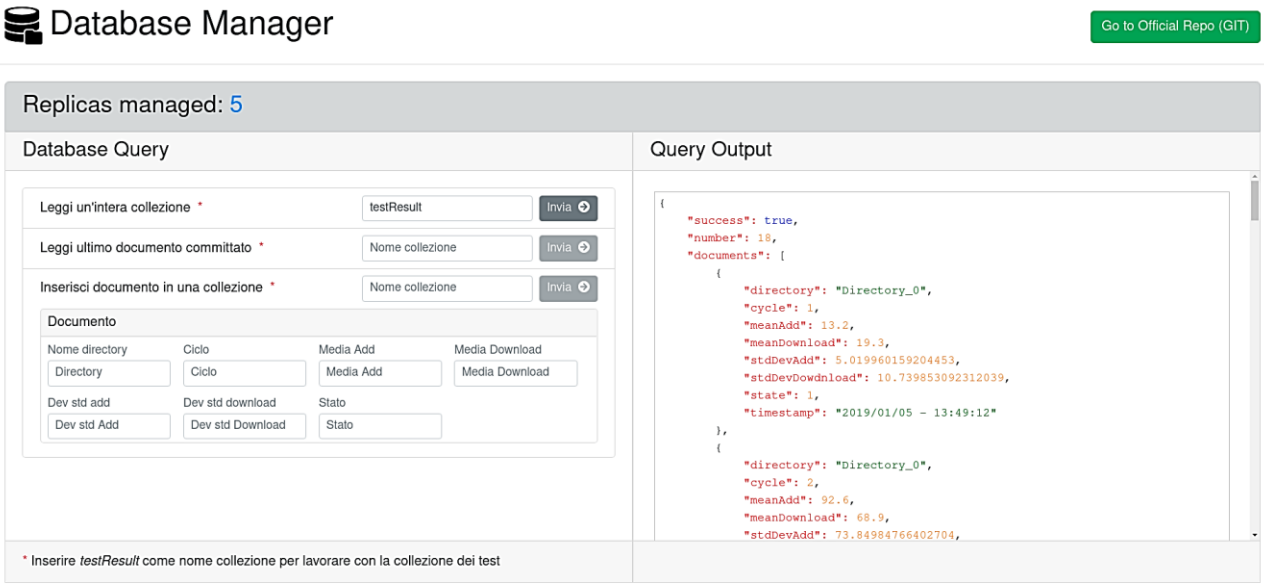

Figura 4. Interfaccia grafica del Database Manager

4. ReplicaManager

4.1 Replica

Questo componente si occupa della gestione di una specifica replica del database, comprende un database Mongodb ed un servizio REST che offre un'interfaccia per le operazioni sul database. Il ReplicaManager non dispone di una interfaccia grafica interattiva (a meno del link al repository Git) poiché i suoi servizi sono gestiti dal DatabaseManager.

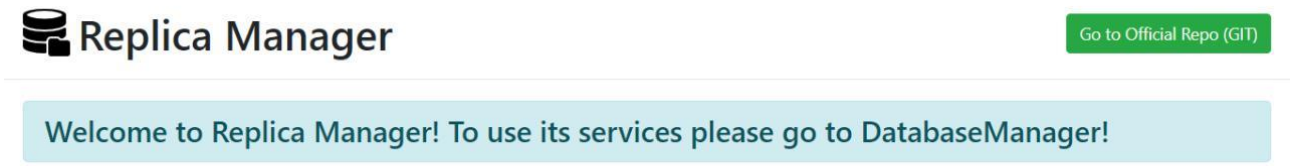

Figura 5. Interfaccia grafica minimale (senza alcuna interazione) del Replica Manager

4.2 ReplicaResource

ReplicaResource è la web application Java che espone il servizio REST per le operazioni sul database Mongodb. Essa interagisce con il database Mongodb tramite una apposita libreria ed istanzia un LogManager per gestire il file di log, necessario in quanto viene implementato un protocollo di 2-phase commit per le scritture.

4.3 Log Manager

LogManager è il componente responsabile della gestione del file di log, utilizzato per implementare il two-phase commit nelle scritture. Esso utilizza le apposite API di Java per leggere e scrivere sul file di log le entry relative alle varie operazioni di scritture effettuate sul database.
Nella prima fase di ogni operazione di scrittura viene aggiunta una riga all'interno di questo file, successivamente rimossa una volta completata la seconda fase. Poiché il servizio stateless, all'interno di ogni entry del log file (entry che sarà relativa ad una specifica operazione) vengono salvati due elementi:

1) il dato che deve essere inserito nel database;
2) un numero di sequenza, che consente di identificare l'operazione nella seconda fase del protocollo.

4.4 MongoDB

Si è deciso di utilizzare il database non relazione MongoDb poiché ritenuto più adatto alle caratteristiche dei dati che si intendono memorizzare (lista di elementi omogenei) rispetto ad altri relazionali.

5. Docker

5.1 Dockerizzazione

Parte integrante del progetto è stata la fase di containerizzazione dei servizi sviluppati, in particolare è stato utilizzato *Docker* per assicurare la corretta esecuzione del sistema in qualsiasi host a prescindere dalle differenze legate all'ambiente o ad altri fattori esterni.

La *dockerizzazione* di un sistema distribuito come quello presentato si articola in due fasi principali:

1) *Containerizzazione dei singoli componenti*, ossia la creazione di un'immagine Docker per ogni tipo di elemento presente nell'architettura. Questa operazione corrisponde alla realizzazione di un Dockerfile in cui vengono specificate tutte le operazioni necessarie alla preparazione dell'ambiente per lo specifico componente, a partire da questo Dockerfile è possibile costruire un'immagine istanziabile in uno o più container.
2) *Composizione delle immagini realizzate in un sistema multi-container*, questa operazione corrisponde alla realizzazione di un Docker-compose file, al cui interno vengono definiti i vari servizi che compongono il sistema, insieme ad altri importanti dettagli architetturali.

5.1.1 Fase 1: containerizzazione dei singoli componenti

Inizialmente è stata stabilita la logica di collocazione dei servizi all'interno dei diversi container, una scelta significativa in questo senso è stata inserire il ReplicaManager ed il database Mongodb all'interno di una singola immagine, in modo da formare un unico container responsabile della gestione di una replica del database.

Dopo delle accurate valutazioni di questa natura si è deciso di procedere alla creazione di 4 differenti immagini da cui potrà essere effettuata l'istanziazione dei container per il deploy.

1) *filesystemservice*. La configurazione di questo container usa come immagine di partenza quella disponibile pubblicamente su “DockerHub” per l'istanziazione di un container con Glassfish server installato e configurato. Su questa immagine di partenza viene copiato l'archivio (.ear) ottenuto eseguendo un build del progetto JEE ed un file (start.sh) che è stato indicato come “entry point”. Il file start.sh viene quindi eseguito al momento della costruzione del container ed esegue il deploy dell'archivio sul server glassfish.
2) dbmanager. Il processo di configurazione di questo container è analogo a quello precedente. Il servizio deployato viene ovviamente ottenuto da un archivio (.war) differente generato dal build della relativa Java Web Application.
3) rabbitmq. Per questo servizio non è stato necessario definire alcun Dockerfile, in quanto il container può essere direttamente istanziato dall'immagine di rabbitmq disponibile su “DockeHub”.
4) replicamanager. Per la definizione di questo componente è stato necessario definire un'immagine personalizzata di un ambiente avente installati sia Glassfish-server che Mongodb. Per fare ciò è stato necessario scrivere un Dockerfile di appoggio in cui veniva usata la stessa immagine di partenza utilizzata per i componenti (1) e (2) e venivano lanciati i comandi necessari all’installazione di Mongodb. Lanciando un build di questo Dockerfile è stata ottenuta una nuova immagine con le caratteristiche desiderate (Glassfish-server + Mongodb). Tale immagine è stata committata in una apposita repository su “DockerHub” (orazioscavo13/glassfish_mongo:1.0), diventando anch'essa pubblicamente disponibile ed utilizzabile in qualsiasi host. A questo punto è stato possibile utilizzare l'immagine ottenuta come immagine di partenza per un nuovo Dockerfile, che prevede come negli

altri casi la copia dell'archivio (.war) e di un file (start.sh) indicato come entry point. Il file, oltre al deploy della web application java, lancia il database Mongodb tramite un apposito comando.

5.1.1 Fase 2: composizione dei componenti con docker compose

Una volta completata la prima fase è stato possibile cominciare la definizione del file docker-compose. La scrittura di questo file consiste nella definizione dei vari servizi e delle rispettive immagini da cui istanziare i container.

Per ogni servizio sono state indicate le porte su cui questo viene messo in ascolto, i servizi da cui esso dipende e i volumi che si desidera rendere persistenti (come i database o lo storage del filesystem).

L'uso di questo strumento ha permesso di definire implicitamente una overlay network a cui tutti i componenti del sistema sono collegati. All'interno di tale rete è possibile fare riferimento ai vari container tramite il nome specificato nella definizione del servizio.

La scelta architetturale più rilevante in questa fase riguarda la modalità adottata per l'istanziazione multipla del ReplicaManager, infatti l'architettura prevede 5 istanze di tale componente.

Nel file docker compose è possibile definire il numero di istanze da lanciare per ogni servizio tramite l'uso di un semplice parametro, ma dopo delle accurate valutazioni si è giunti alla conclusione che tale soluzione non è adatta in questo caso. Il servizio di replica offerto da Docker (utilizzabile solo in modalità swarm), infatti, prevede l'istanziazione di più repliche totalmente intercambiabili su cui viene applicato in maniera trasparente anche un servizio di *load balancing*, non è possibile dunque con questa tecnica fare riferimento ad una specifica replica per una interazione di rete ma semplicemente al servizio, senza avere certezze su quale istanza servirebbe la richiesta, ovviamente questo non è accettabile per un cluster di database basato su un protocollo di quorum come quello presentato.

Considerato ciò è stato deciso di definire cinque servizi logicamente distinti (replicamanager_1, ..., replicamanager_5), e dunque ad uno ad uno direttamente referenziabili, istanziati utilizzando la stessa immagine, quella personalizzata che contiene Glassfish-server e Mongodb.

5. Git

L'intera attività di analisi, progettazione, sviluppo, test (non automatico) e containerizzazione è stata effettuata mantenendo il progetto e tutti i relativi file versionati tramite Git. Il progetto è infatti disponibile al repository pubblico GitHub accessibile dai seguenti link: https://github.com/orazioscavo13/FileSystemService o https://github.com/taletex/FileSystemService.

Nel repository è possibile trovare l'intero progetto e documentazione (JavaDoc, questo file e file README.md).

Il repository si compone di 2 branch (come buona prassi):

1) Branch *master*; usato per il deploy in produzione. Il branch master è la versione del progetto dockerizzata, sarà dunque necessario usare Docker per poterlo utilizzare.
2) Branch *deploy*; usato per lo sviluppo. Questo branch non è dockerizzato e mantiene quindi tutti i riferimenti agli elementi locali non containerizzati.

La guida all'utilizzo del progetto è disponibile nel file README.md di cui sopra.