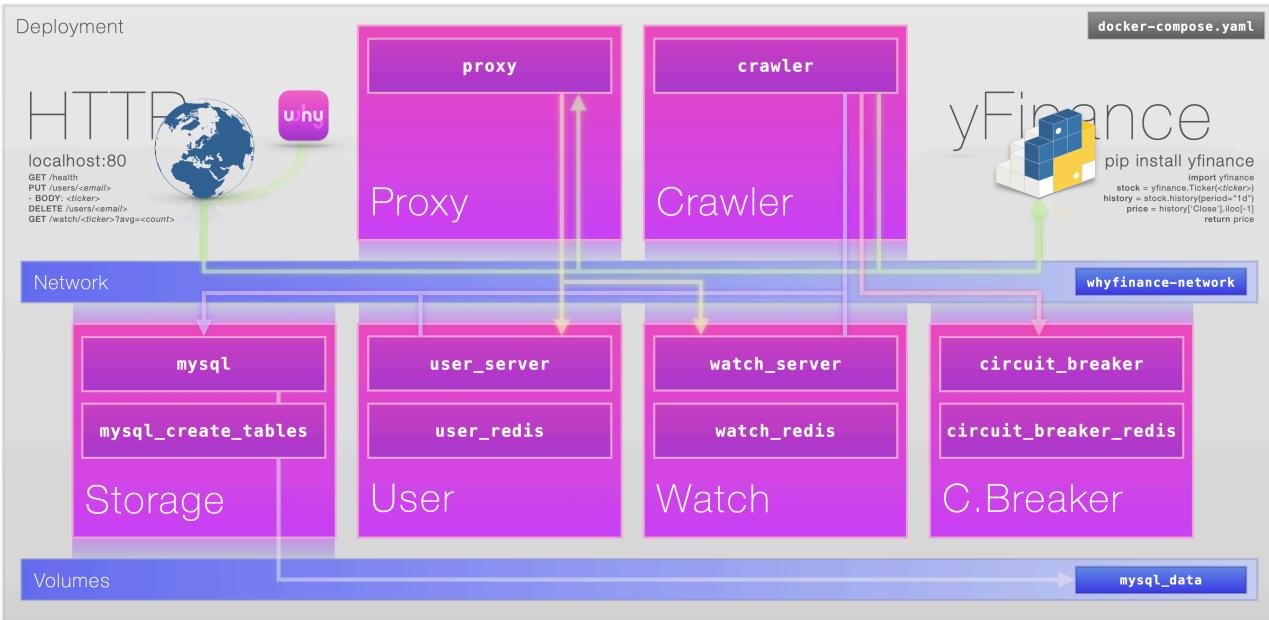


Why Finance

Homework 1

L'idea



Abbiamo suddiviso il sistema in dieci micro-servizi, che possono essere raggruppati in cinque unità di deployment

Questo progetto sviluppa i requisiti del sistema distribuito per il monitoraggio di dati finanziari tramite la libreria **yFinance** descritto nella consegna di **Homework 1**, fra cui l'uso di gRPC nella comunicazione con i server per la gestione dei dati utente e finanziari, protetta internamente con un riconoscimento e gestione opportuna delle richieste duplicate (politica *at-most-once*) ed esternamente con conteggio degli errori avvenuti e temporanea limitazione delle richieste (il *circuit breaker*).

Ogni modulo è stato posto in un suo specifico Docker *container*, garantendo la portabilità del sistema e la facilità di distribuzione su diverse piattaforme. Inoltre abbiamo utilizzato volumi Docker per consentire la persistenza dei dati del database.

gRPC è stato usato come tecnologia di comunicazione verso i micro-servizi User (per l'inserimento e la rimozione dei dati utente, cioè *email* e *ticker*, nella relativa tabella del database) e Watch (per il monitoraggio dei dati finanziari relativi a uno specifico *ticker symbol*, con o senza media degli ultimi valori), consentendoci così di definire un'interfaccia capace di fornire prestazioni elevate rispetto ad HTTP grazie alla serializzazione dei dati con *Protocol Buffer*, che come visto a lezione consente di ridurre il consumo di banda.

Ai due micro-servizi che svolgono il ruolo di server gRPC sono associati rispettivamente due container separati dove è in funzione il database *on-*

memory Redis, utilizzato per conservare i valori serializzati dei messaggi di risposta (secondo definizione nei relativi *protocol buffer*) e consentire l'implementazione della politica *at-most-once* in maniera più robusta che usando un semplice dizionario Python, nonché aprendo la possibilità di mantenere queste risposte persistenti ai riavvii del container qualora si introduca l'uso di volumi anche per questi micro-servizi.

Complessivamente abbiamo suddiviso il sistema in dieci micro-servizi, che possono essere raggruppati in cinque unità di *deployment*, ma che per semplicità in questa fase convivono in un singolo `docker-compose.yaml` e sono connessi alla stessa rete virtuale in bridging `whyfinance-network`: connettendo tutti i container fra loro, rispetto il fare affidamento alla rete bridge predefinita cambia solo il fatto di esplicitare questo fatto nel file `compose` e null'altro.

La scelta di suddividere quanto più possibile i componenti è fatta in accordo all'architettura esagonale, in ottica di consentire una migliore scalabilità del sistema lungo i tre assi visti a lezione.

Per gestire le dipendenze tra i micro-servizi all'interno del `docker-compose.yaml`, si è usata sia la tecnica più avanzata di *health-check* mediante script accennata a lezione (per il database), che il più semplice `depends_on-restart` (per tutto il resto): nello specifico `proxy` dipende da `user_server`, `watch_server`, e `crawler` e indirettamente tramite di essi dal database `mysql`; `user_server`, `watch_server` dipendono anche dai rispettivi `*_redis`; infine il `crawler` dipende dal `circuit_breaker`; tutti e tre dipendono dal database attraverso `mysql_create_tables` (quest'ultimo dipende anche da `mysql`), per garantire che le tabelle del database siano create prima che vi si possa accedere.

Il database relazionale

Una nota sul micro-servizio `mysql` per il database, che era stato inizialmente implementato con un alto accoppiamento fra i dati utente nella tabella 'users' e dati finanziari 'stock_data', con questi ultimi associati tramite *foreign key* all'indirizzo email di ciascun utente (causando quindi la duplicazione dello stesso titolo azionario qualora scelto da diversi utenti, e l'eliminazione di questi dati contestualmente alla cancellazione dell'utente stesso): abbiamo scelto per la consegna finale di separare queste tabelle, con i dati finanziari che sono condivisi fra i vari utenti e prelevati da yFinance una sola volta, e richiesti direttamente con il *ticker symbol* al servizio di monitoraggio.

Ciò nonostante, al contrario dei micro-servizi Redis, condividiamo l'istanza di MySQL per entrambe le tabelle anche in questa nuova organizzazione, per consentirci più agevolmente di configurare le tabelle opportunamente mediante il micro-servizio `mysql_create_tables`, ma siamo ben consapevoli che sarebbe cosa opportuna tenere i database di ciascun micro-servizio completamente separati e, se le specifiche aggiuntive dei successivi *homework* lo dovessero consentire, intendiamo separarli completamente in futuro.

Raccolta dei dati finanziari

Il micro-servizio `crawlers` si interfaccia e comunica direttamente, attraverso la libreria `yFinance`, con Yahoo Finance per il recupero dei dati relativi ai titoli finanziari gestendo il processo di raccolta, di controllo e salvataggio nel database MySQL. Per proteggere il sistema da eventuali errori dovuti alla libreria esterna il *crawler* utilizza un *circuit breaker*, che abbiamo sviluppato come micro-servizio separato.

In sostanza, prima di inviare una richiesta a Yahoo Finance, il Crawler verifica lo stato del circuito, operando come segue: se è chiuso o semi-aperto, la richiesta viene inoltrata, altrimenti il ciclo di aggiornamento è interrotto immediatamente, e ripreso più in avanti in accordo con l'intervallo di tempo in secondi impostato come variabile di ambiente.

Differenze rispetto la bozza: Proxy e Circuit Breaker

Nella bozza di Homework 1 che vi è stata consegnata, si faceva riferimento alla possibile centralizzazione della politica *at-most-once* all'interno di `proxy` e del *circuit breaker* in un micro-servizio separato: nella consegna effettiva la prima è stata abbandonata, dal momento che `proxy` svolge il ruolo di *API Gateway*, lavorando da client gRPC nei confronti del resto del sistema, convertendo richieste HTTP intrinsecamente idempotenti come PUT nelle corrispondenti chiamate gRPC, le quali sono poi gestite con *retry* e *metadata* in maniera analoga a quanto visto a lezione.

Il proxy fa uso della libreria Python `Flask` e per impostazione predefinita è in ascolto per richieste sulla porta `:80`, scelta che può essere sovrascritta nel port mapping effettuato nel `docker-compose.yaml` (cogliamo l'opportunità per sottolineare che tutte le porte di tutti i container sono mappate 1:1 tra container e host per semplificare il *debugging*, ma ad eccezione fatta per il `proxy` non sono ovviamente necessarie al funzionamento del sistema, ma che anzi in un reale sistema in produzione sarebbe l'ideale bloccare).

Per quanto concerne il micro-servizio `circuit_breaker`, nell'idea della bozza sarebbe dovuto essere un servizio cui delegare le richieste HTTP, internamente gestite tramite libreria `Requests` e gestendo la macchina a stati finiti che rappresenta il *circuit breaker* sulla base dei parametri ricevuti tramite gRPC circa il *timeout*, la soglia dei fallimenti, e l'intervallo per la semi-chiusura del circuito (non è previsto un parametro per specificare il numero di richieste di successo per portare alla chiusura completa del circuito, che è sempre 1 (questa scelta è stata fatta per semplificare la persistenza dello stato del circuito per ogni identificativo - più dettagli in avanti)).

All'atto di integrare la parte client del `circuit_breaker.proto` si è però realizzato come `yFinance` non sia un wrapper Python intorno ad una *API* pubblica per Yahoo Finance, ma che questa è stata rimossa dal mercato da anni e che la libreria effettivamente faccia scraping (in violazione dei termini di servizio di Yahoo Finance, come ricorda la documentazione stessa di `yFinance` qualora si utilizzi per qualunque fine non strettamente personale): pur usando al suo interno la stessa libreria `Requests` usata in `circuit_breaker`, la natura stessa di uno *scraper* rende la prospettiva di un suo *fork* per usare il nostro `circuit_breaker` invece di `Requests` una soluzione talmente fragile da non riuscire potenzialmente a sopravvivere al semestre; anche la possibilità di integrare direttamente `yFinance` dentro `circuit_breaker` a formare una sorta di "servizio `yFinance`" distinto dal processo *worker* che periodicamente se ne serve è stata scartata in quanto non è chiara l'utilità di accedere al servizio al di fuori del *worker*.

Piuttosto che abbandonare la nostra implementazione e far ciò che senza dubbio la maggioranza degli studenti avrà scelto di fare, cioè usare come libreria direttamente l'implementazione di esempio che ci è stata caricata su Studium, abbiamo aggiunto interfacce aggiuntive al *protocol buffer* per ottenere lo stato del circuito per un dato *host* e segnalarne successo o fallimento: a tutti gli effetti stiamo centralizzando lo stato dei servizi esterni usati dall'intero sistema distribuito, al prezzo di una minore efficienza di esecuzione rispetto una libreria integrata nello stesso processo.

Il primo modo di operare continua ad essere presente, e può essere testato attraverso il `test.py` presente all'interno dello stesso micro-servizio `circuit_breaker`.

Internamente `circuit_breaker`, dipendendo da `circuit_breaker_redis` in maniera analoga a quanto fatto per i micro-servizi `user_server` e `watch_server`, conserva in questo attraverso operazioni di inserimento e lettura chiave-valore (con la stringa *host* come chiave) un intero che corrisponde al conteggio dei fallimenti consecutivi, quando il valore è

inesistente (si assume zero) o minore del numero massimo di fallimenti accettabili, oppure al valore di *unix timestamp* oltre il quale l'intervallo di tempo di recupero dallo stato di circuito aperto è concluso (la possibilità che si desideri specificare un numero di fallimenti pari o superiore al timestamp corrente e che quindi questa rappresentazione non funzioni è irragionevole).

Secondo questa rappresentazione, il circuito è da considerarsi **CLOSED** per un dato *host* quando il valore è minore del numero di fallimenti consentito (valore che è contenuto in ogni singola richiesta verso il *circuit breaker*), **OPEN** quando è maggiore di questo, ma il valore di *unix timestamp* corrente è minore della scadenza dell'intervallo di recupero, altrimenti **HALF_OPEN**.

Si noti che Auth è stato rinominato in User nel progetto finito, per sottolineare il fatto che non sia più previsto alcun meccanismo di autenticazione.

Guida di configurazione

Clonare il progetto

```
git clone <https://github.com/alessiogiordano/why-finance.git>
```

Configurazione delle Variabili d'ambiente

il file `/Containers/.env` consente di configurare in maniera centralizzata variabili d'ambiente. Ad esempio, è possibile cambiare l'intervallo di tempo tra un ciclo di aggiornamento dei dati finanziari con `CRAWLER_TIME_INTERVAL`.

Qualora queste variabili non siano definite, il sistema assegna valori predefiniti presenti nel file `docker-compose.yaml`.

```
DB_HOST=mysql
DB_PORT=3306
DB_USER=root
DB_PASSWORD=root
DB_NAME=whyfinance_hw1

PROXY_PORT=80
REDIS_PORT=6379
WATCH_SERVER_PORT=50052
USER_SERVER_PORT=50051
CIRCUIT_BREAKER_PORT=30112

CRAWLER_TIME_INTERVAL=15 # Secondi tra un ciclo e l'altro
```

Build dei Container

Per costruire i container, eseguire il comando nella cartella `/Containers`:

```
docker compose up --build
```

Non appena saranno visualizzati a schermo i messaggi di log del server web Flask, l'ultimo dei container a partire, il sistema sarà pronto all'uso.

Uso del client gRPC tramite HTTP o app

Come descritto nella bozza del progetto, è stato previsto sia un client HTTP che una applicazione nativa macOS per testare le funzionalità del sistema.

Il server web si mette in ascolto per impostazione predefinita alla porta 80 della macchina sulla quale è in esecuzione il servizio Docker, quindi è sufficiente un client HTTP come cURL o Postman per testare gli endpoint HTTP:

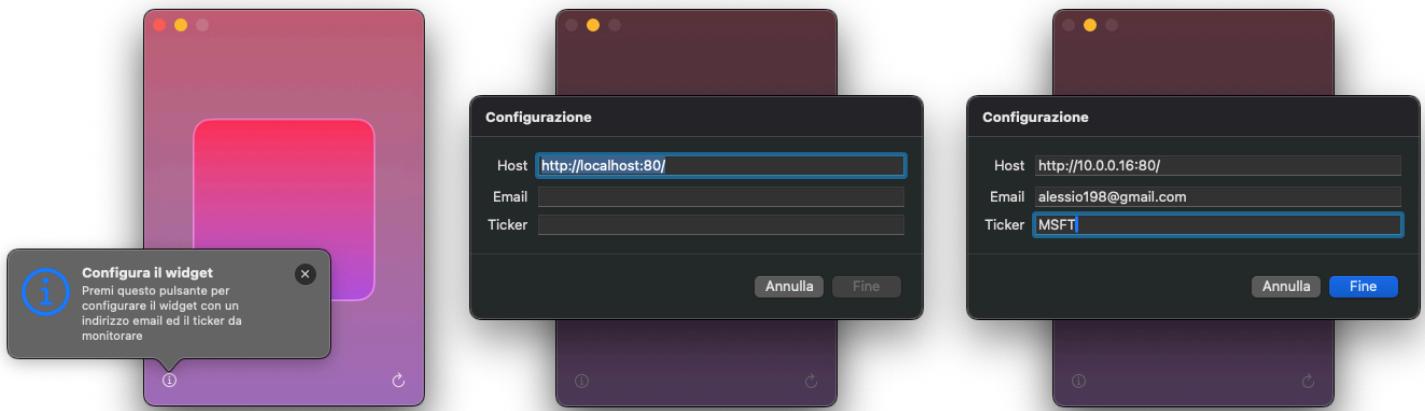
```

# Restituisce 204 No Content se il sistema è in funzione
GET /health

# Prova ad effettuare l'inserimento dell'utente e se l'operazione fallisce (l'utente esiste già) riprova aggiornando invece l'utente, restituendo 204 No Content
PUT /users/<email>
- BODY: <ticker> # Content-Type: text/plain
# Elimina dal database tutti i record aventi quel dato indirizzo email (per vincolo di unicità sarà al più uno solo, ma se l'indirizzo non è presente non sarà restituito alcun errore, ma sempre 204 No Content)
DELETE /users/<email>
# Restituisce l'ultimo valore presente nella tabella 'stock_data' per quel dato ticker, o sarà lanciata una eccezione se nessun dato è presente
GET /watch/<ticker>
# Restituisce la media degli ultimi count valori per quel dato ticker, o sarà lanciata una eccezione se nessun dato è presente
GET /watch/<ticker>?avg=<count>

```

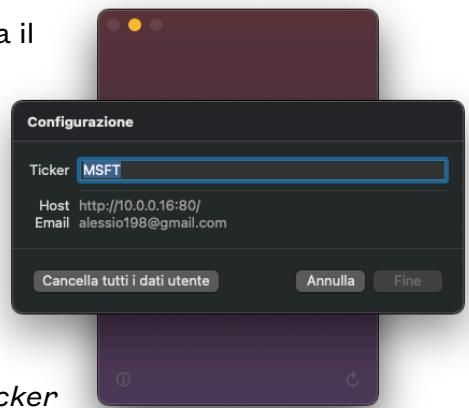
L'applicazione macOS Why Finance effettua queste richieste per conto dell'utente, fornendo una interfaccia sicuramente più facile da usare.



Estraendo la cartella compressa '/Apps/Why Finance.app.zip' e eseguendo l'applicazione (essendo *notarizzata* con un Apple Developer Account, non sarà necessario fare null'altro), si aprirà una finestra che suggerirà di aprire il pannello di configurazione, che richiederà i tre dati necessari alla chiamata PUT discussa poc'anzi.

Se l'operazione ha buon fine (*204 No Content*) allora il pannello si chiuderà, ma potrà essere sempre riaperto alla stessa maniera, rivelando la possibilità di modificare il *ticker symbol* (ma la chiamata PUT sarà analoga) oppure di effettuare l'operazione **DELETE**, premendo il pulsante in basso a sinistra nel pannello, prima non presente.

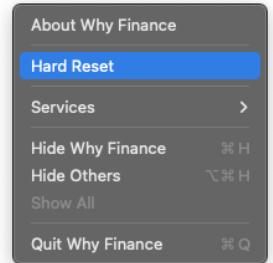
Non appena il pannello è chiuso verrà mostrato il *ticker symbol* scelto in alto a sinistra nel *widget*, ma potenzialmente nessun dato se il ticker in questione non è già presente nel



database: la politica di aggiornamento automatica ogni minuto viene in tal caso velocizzata ad una volta al secondo, e fino a quando un valore non viene recuperato sarà visualizzato un segnaposto in basso a sinistra nel widget dove normalmente è visualizzato il valore.

In ogni momento è possibile richiedere manualmente l'aggiornamento del valore, ovvero la chiamata GET /watch/<ticker>, premendo il pulsante di ricaricamento in basso a destra nella finestra.

Poiché il pannello di configurazione non permette la modifica di host o email senza passare prima da una operazione di DELETE, qualora non dovesse essere più disponibile il precedente host è possibile superare questo inconveniente scegliendo 'Why Finance > Hard Reset' dalla barra dei menù.



È consigliato usare l'applicazione accanto al terminale dal quale si è fatto partire il sistema per poter visualizzare il suo comportamento in risposta alle azioni dell'utente.

```

Distributed Systems and Big Data
proxy-1 | 2024-11-30 14:54:28,150 - INFO - DELETE alessio198@gmail.com
proxy-1 | 2024-11-30 14:54:28,152 - INFO - 10.0.0.7 - - [30/Nov/2024 14:54:28] "DELETE /users/alessio198@gmail.com HTTP/1.1" 204 -
watch_server-1 | 2024-11-30 14:54:59,593 - INFO - Received GetLastStockValue request for ticker: MSFT
watch_server-1 | 2024-11-30 14:54:59,642 - INFO - Stock value retrieved for MSFT: 423.46
proxy-1 | 2024-11-30 14:54:59,645 - INFO - WATCH MSFT=423.46
proxy-1 | 2024-11-30 14:54:59,651 - INFO - 10.0.0.7 - - [30/Nov/2024 14:54:59] "GET /watch/MSFT"
crawler-1 | 2024-11-30 14:55:02,848 - INFO - Recuperato stock: AAPL ---- Prezzo corrente: 237.3300
crawler-1 | 2024-11-30 14:55:02,884 - INFO - Dati salvati per AAPL: 237.3300018310547 a 2024-11-30 3
crawler-1 | 2024-11-30 14:55:02,885 - INFO - Ciclo di inserimento n. 2520
watch_server-1 | 2024-11-30 14:55:59,727 - INFO - Received GetLastStockValue request for ticker: MSFT
watch_server-1 | 2024-11-30 14:55:59,782 - INFO - Stock value retrieved for MSFT: 423.46
proxy-1 | 2024-11-30 14:55:59,786 - INFO - WATCH MSFT=423.46
proxy-1 | 2024-11-30 14:55:59,790 - INFO - 10.0.0.7 - - [30/Nov/2024 14:55:59] "GET /watch/MSFT"
crawler-1 | 2024-11-30 14:56:03,105 - INFO - Recuperato stock: AAPL ---- Prezzo corrente: 237.3300
crawler-1 | 2024-11-30 14:56:03,145 - INFO - Dati salvati per AAPL: 237.3300018310547 a 2024-11-30 4
crawler-1 | 2024-11-30 14:56:03,145 - INFO - Ciclo di inserimento n. 2521
crawler-1 | 2024-11-30 14:57:03,389 - INFO - Recuperato stock: AAPL ---- Prezzo corrente: 237.3300
crawler-1 | 2024-11-30 14:57:03,432 - INFO - Dati salvati per AAPL: 237.3300018310547 a 2024-11-30 4
crawler-1 | 2024-11-30 14:57:03,433 - INFO - Ciclo di inserimento n. 2522
user_server-1 | 2024-11-30 14:57:56,266 - INFO - New user registered: alessio198@gmail.com for ticker
user_server-1 | 2024-11-30 14:57:56,303 - INFO - Il server gRPC è avviato con successo.
user_server-1 | 2024-11-30 14:57:56,304 - INFO -
user_server-1 | =====
user_server-1 | 2024-11-30 14:57:56,304 - INFO - Database Statistics:
user_server-1 | 2024-11-30 14:57:56,304 - INFO - Total Users: 2
user_server-1 | 2024-11-30 14:57:56,304 - INFO - Total Stock Data Records: 5241
user_server-1 | 2024-11-30 14:57:56,304 - INFO - Last Stock Update: AAPL - Value: 237.33 at 2024-11-30
user_server-1 | 2024-11-30 14:57:56,304 - INFO - =====
user_server-1 |
proxy-1 | 2024-11-30 14:57:56,308 - INFO - PUT alessio198@gmail.com for MSFT
proxy-1 | 2024-11-30 14:57:56,312 - INFO - 10.0.0.7 - - [30/Nov/2024 14:57:56] "PUT /users/alessio198@gmail.com HTTP/1.1" 204 -
watch_server-1 | 2024-11-30 14:57:56,650 - INFO - Received GetLastStockValue request for ticker: MSFT
watch_server-1 | 2024-11-30 14:57:56,698 - INFO - Stock value retrieved for MSFT: 423.46
proxy-1 | 2024-11-30 14:57:56,700 - INFO - WATCH MSFT=423.46
proxy-1 | 2024-11-30 14:57:56,702 - INFO - 10.0.0.7 - - [30/Nov/2024 14:57:56] "GET /watch/MSFT HTTP/1.1" 200 -
crawler-1 | 2024-11-30 14:58:03,696 - INFO - Recuperato stock: MSFT ---- Prezzo corrente: 423.4599914550781
crawler-1 | 2024-11-30 14:58:03,732 - INFO - Dati salvati per MSFT: 423.4599914550781 a 2024-11-30 14:58:03.69770
0
crawler-1 | 2024-11-30 14:58:03,880 - INFO - Recuperato stock: AAPL ---- Prezzo corrente: 237.3300018310547
crawler-1 | 2024-11-30 14:58:03,915 - INFO - Dati salvati per AAPL: 237.3300018310547 a 2024-11-30 14:58:03.88070
1
crawler-1 | 2024-11-30 14:58:03,916 - INFO - Ciclo di inserimento n. 2523

```