API Gateway – Roberto Antoniello 47891A

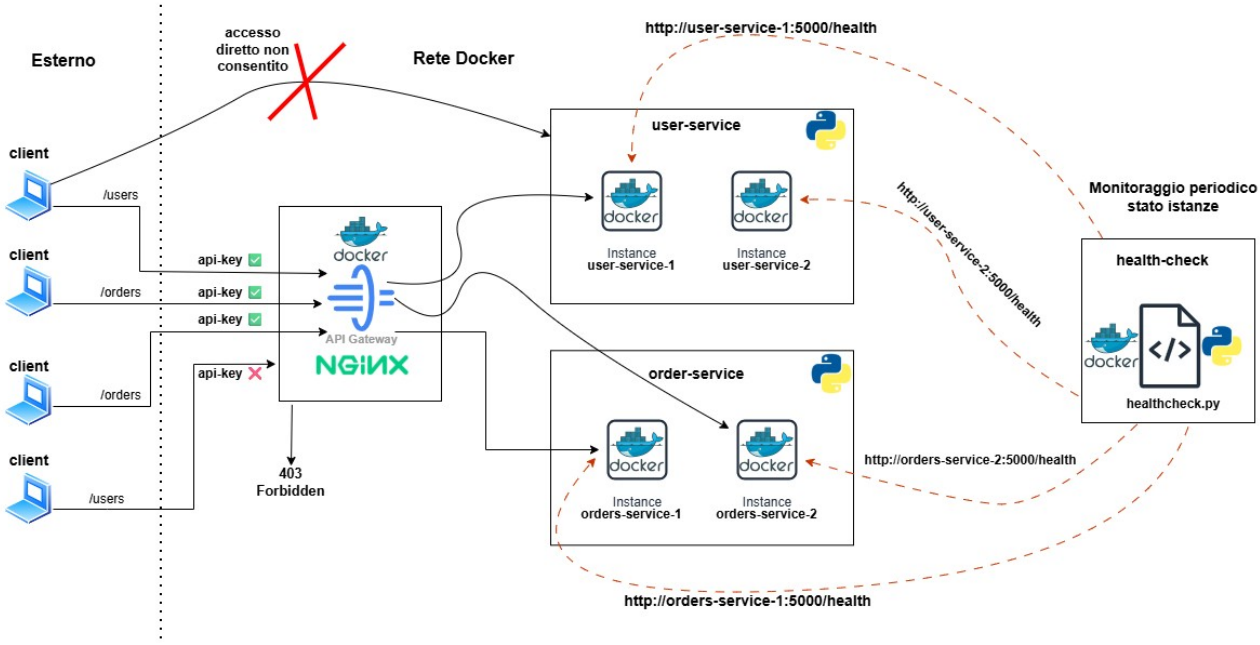
1. Introduzione

Lo scopo del progetto è la realizzazione di un'applicazione a microservizi basata su un'architettura API Gateway. I microservizi sono di semplice complessità a scopo dimostrativo i cui endpoint saranno illustrati a seguire.
È stato tutto progettato in modo tale che vi sia evidenza della presenza di funzionalità di Load balancing, Fault Tolerance e security.

2. Architettura del sistema

È stato scelto NGINX come API Gateway, il quale si occupa di esporre un punto di accesso unificato per i client che inviano richieste dall'esterno, instradare le stesse verso i microservizi e bilanciare il carico tra le diverse istanze disponibili.
Il gateway applica inoltre controlli di base sulle richieste di ingresso, contribuendo alla parte di sicurezza del sistema.
Sono presenti un paio di microservizi REST sviluppati sfruttando la libreria Flask presente in Python.
Ciascun microservizio è eseguito su più istanze, garantendo una migliore affidabilità e bilanciamento del carico.
L'isolamento delle componenti è realizzato tramite Docker e Docker compose, che permettono l'esecuzione di ciasun microservizio all'interno di container separati.

Disegno architetturale del progetto



3. API Gateway Design

L'API Gateway ha lo scopo di essere il componente dedicato alla gestione centralizzata delle funzionalità dell'architettura a microservizi.
Questa scelta consente di separare le responsabilità infrastrutturali dalla logica applicativa dei

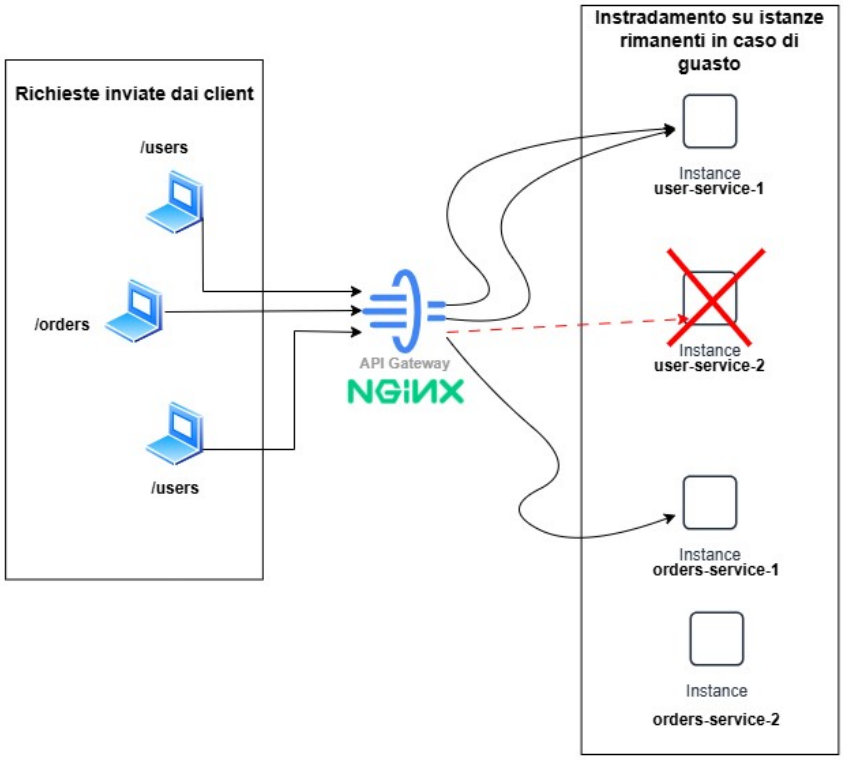
servizi, migliorando la manutenibilità e la resilienza complessiva del sistema.
Il gateway funge inoltre da livello di astrazione tra i client e i servizi.
Il client parla solo con l'API Gateway, il quale decide a quale microservizio inoltrare la richiesta ricevuta. Inoltre Il client non conosce quante istanze del microservizio esistono, come si chiamano o dove sono state allocate. Questo approccio permette di gestire in modo trasparente guasti o modifiche all'infrastruttura senza che i client subiscano alcun impatto visivo.
I controlli sulle richieste in ingresso citati nel capitolo precedente sono impleementati tramite l'introduzione di un api-key che se non utilizzata o inserita con valore errato, fa si che NGINX rifiuti le richieste.

Di seguito gli endpoint disponibili con esempio di chiamata GET:

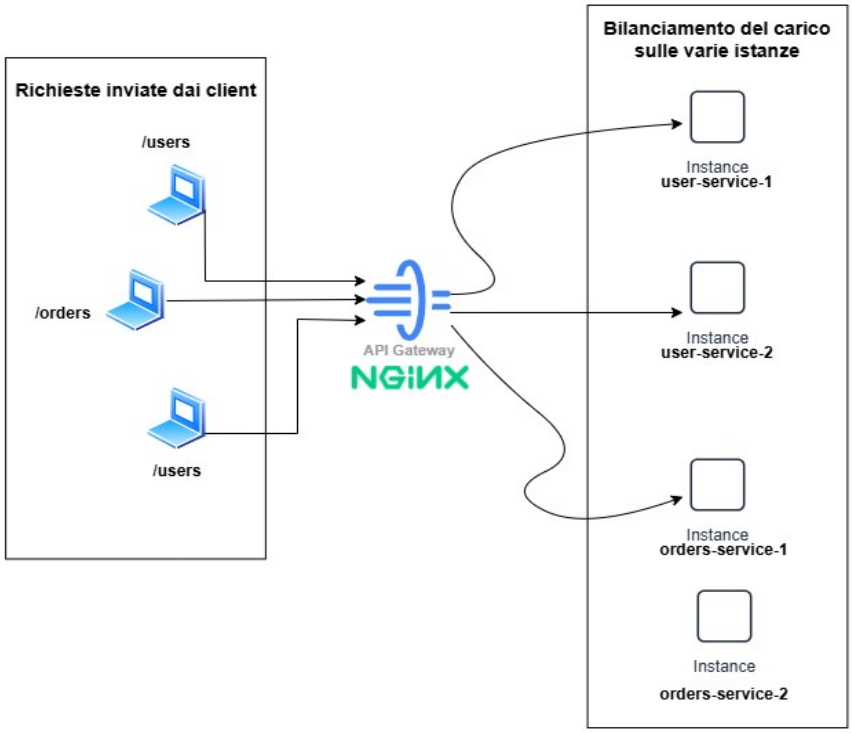
- /users → http://localhost:8080/users?api_key=password01*
- /orders → http://localhost:8080/orders?api_key=password01*

4. Proprietà non funzionali

La **fault tolerance** è ottenuta grazie alla replica di più istanze sui due microservizi con l'utilizzo di NGINX come API Gateway.
Ogni istanza è indipendente dall'altra e sono isolate all'interno di container Docker separati.
In caso di malfunzionamento della singola istanza, il servizio continua ad essere erogato correttamente grazie all'instradamento automatico verso le altre istanze ancora operative.
Ogni microservizio inoltre dispone di una rotta /health che ci consente di verificare lo stato della singola istanza. Nel caso in cui tutte le istanze di un servizio risultassero indisponibili, otterremo di conseguenza errore 502 da parte di NGINX.
Questa rotta di monitoraggio dello stato delle istanze è utilizzata da un componente esterno ai microservizi, implementato come container dedicato ed eseguito all'interno della rete Docker, in modo da non esporre questa funzionalità di servizio verso l'esterno e preservare l'isolamento dell'infrastruttura.



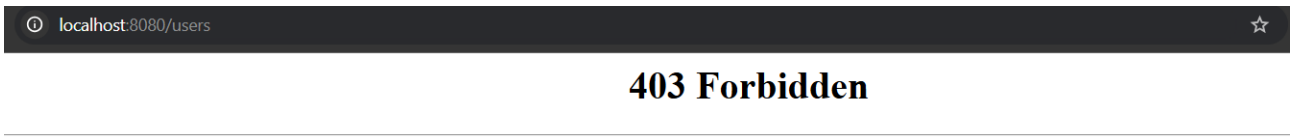
Il **load balancing** è garantito da NGINX, il quale distribuisce le richieste HTTP in ingresso tra le diverse istanze dei microservizi. In questo modo si consente una distribuzione uniforme del carico evitando di sovraccaricare una singola istanza.
A scopo dimostrativo, l'id dell'istanza è presente nel response del microservizio, così che si possa verificare l'alternanza tra le istanze a seguito di più richieste in modo diretto e trasparente. Questa proprietà contribuisce al miglioramento delle prestazioni del sistema e alla sua affidabilità.



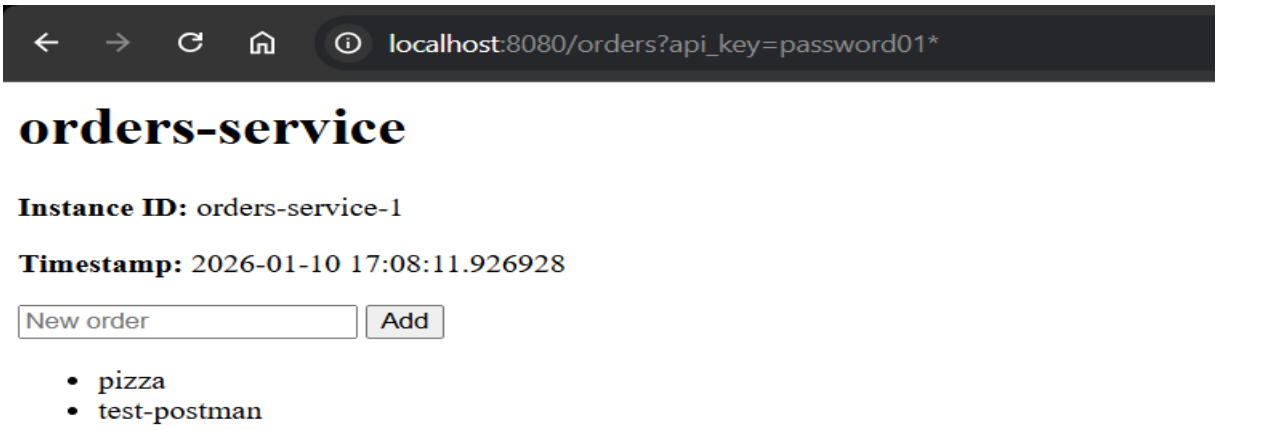
L'aspetto **security** è coperto sempre da NGINX che centralizza l'accesso principale ai microservizi e consente di applicare controlli uniformi alle richieste in ingresso.
I microservizi interni non sono direttamente esposti all'esterno, ogni richiesta deve passare attraverso l'API Gateway. L'istanza del servizio con la sua porta interna è raggiungibile solamente da dentro la rete Docker, preservando l'isolamento.
Inoltre, come già citato è stato implementato un semplice meccanismo di autenticazione introducendo un controllo tramite API Key.
Tutte le richieste verso i microservizi devono includere una chiave valida nell'header *x-api-key.*
Le richieste prive di chiave o con chiave errata ricevono come risposta un codice 403 Forbidden.

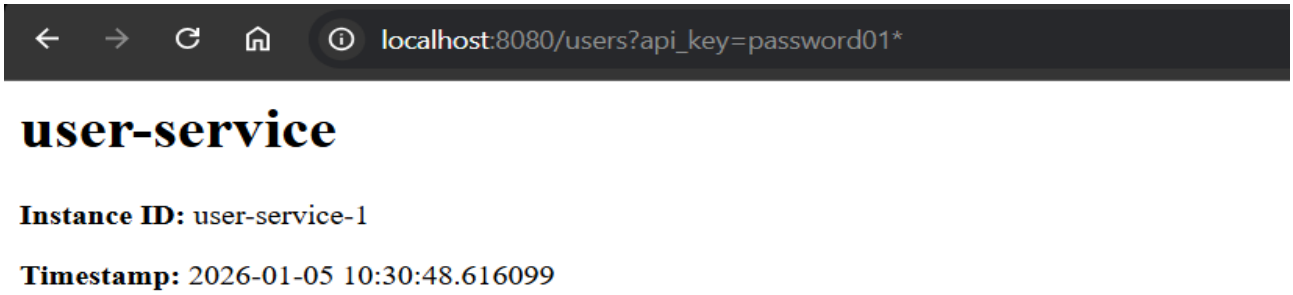
5. Scenari dimostrativi

Scenario 1: in questo primo scenario andiamo a dimostrare che l'accesso ai microservizi è protetto da una chiave e il criterio di accettazione o rifiuto delle richieste passa tramite l'API Gateway NGINX. A seguito del comando *docker-compose up* per avviare l'applicazione effettueremo richieste senza utilizzo di API Key e poi aggiungendola, osservando il comportamento.



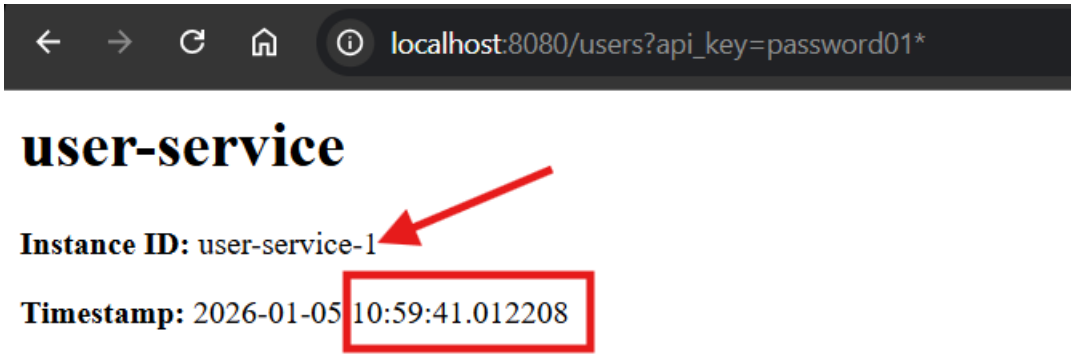
Come previsto, se arrivano richieste sprovviste di api-key o con api-key errata, vengono rifiutate.
http://localhost:8080/users (senza key) → 403 forbidden
http://localhost:8080/users?api_key=password01 (api-key errata) → 403 forbidden

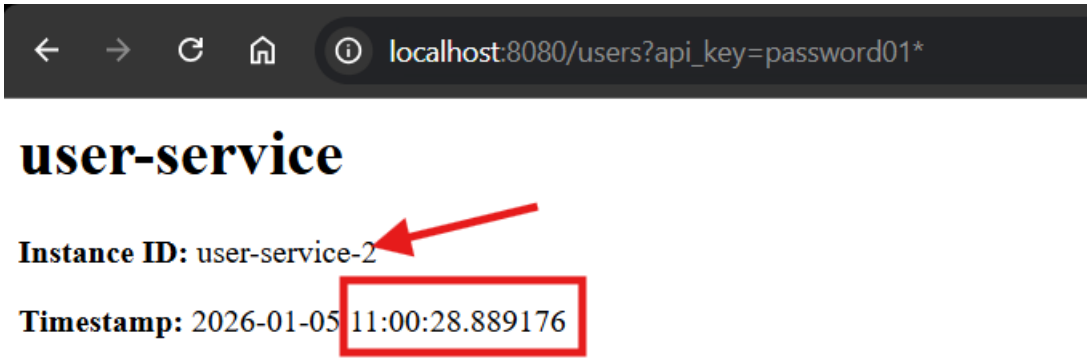




Mentre se arrivano richieste provviste di api-key corretta, come risultato avremo la risposta corretta da parte del microservizio.
http://localhost:8080/users?api_key=password01* → risposta corretta del microservizio
curl.exe -H "x-api-key: password01" http://localhost:8080/users* → risposta corretta tramite curl da terminale

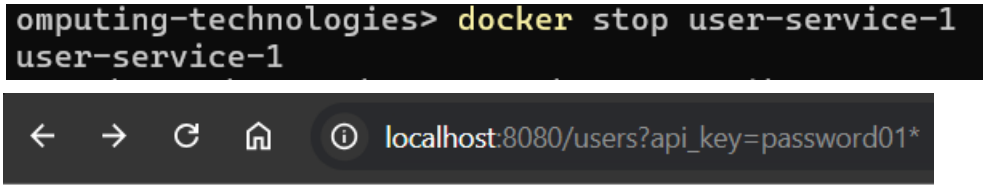
Scenario 2: in questo scenario andiamo a dimostrare la proprietà del load balancing, quindi il bilanciamento del carico. Verrà mostrato che l'API Gateway è in grado di distribuire le richieste in ingresso tra le varie istanze dello stesso microservizio. Andremo a effettuare richieste ripetute allo stesso servizio osservando come varia il campo instance_id. Il risultato atteso è l'alternanza tra le istanze.





Il client di norma non è a conoscenza del numero o dell'identità delle istanze, le quali sono mostrate in questo scenario a puro scopo dimostrativo. NGINX utilizza un meccanismo di round-robin per alternare da un'istanza all'altra durante la ricezione delle richieste. In sostanza distribuisce a turno le richieste tra le istanze disponibili.

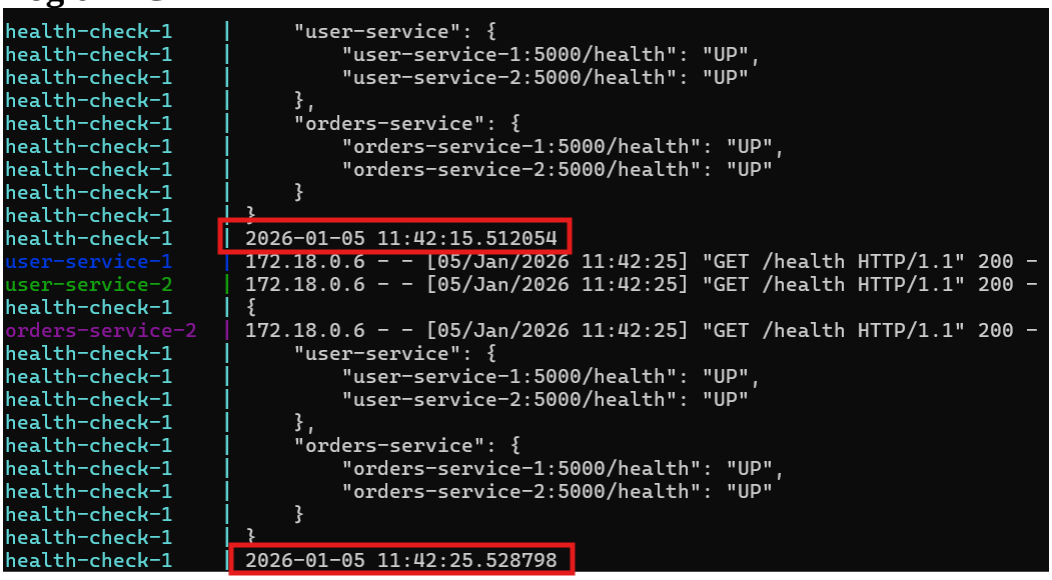
Scenario 3: in questo scenario dimostriamo la fault tolerance e quindi la tolleranza ai guasti nel caso in cui vi sia l'arresto imprevisto di un'istanza di un microservizio. Il risultato atteso è che il servizio continui a rispondere, essendo erogato dalle istanze rimanenti ancora attive.
Per eseguire questo test andiamo a lanciare il comando ***docker stop user-service-1***



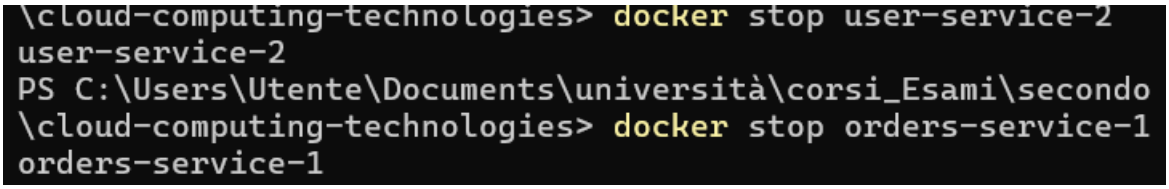
Come previsto, il servizio users risponderà esclusivamente dall'istanza con id 2, poiché l'istanza con id 1 attualmente è fuori servizio.
Possiamo confermare che il guasto di una singola istanza non compromette la disponibilità del servizio. NGINX instrada in automatico verso le istanze rimanenti attive.

Scenario 4: in questo scenario viene mostrato il funzionamento del semplice sistema di monitoraggio interno citato nei capitoli precedenti, il quale verifica periodicamente lo stato delle istanze dei microservizi.
Lo script quindi interroga i microservizi sulle loro porte interne comunicando direttamente tramite la rete Docker. Lo stato delle istanze è consultabile tramite i log di NGINX o lanciando manualmente il comando ***docker compose logs health-check***(il quale mostrerà tutti i log tracciati fino a quel momento).

Log di NGINX

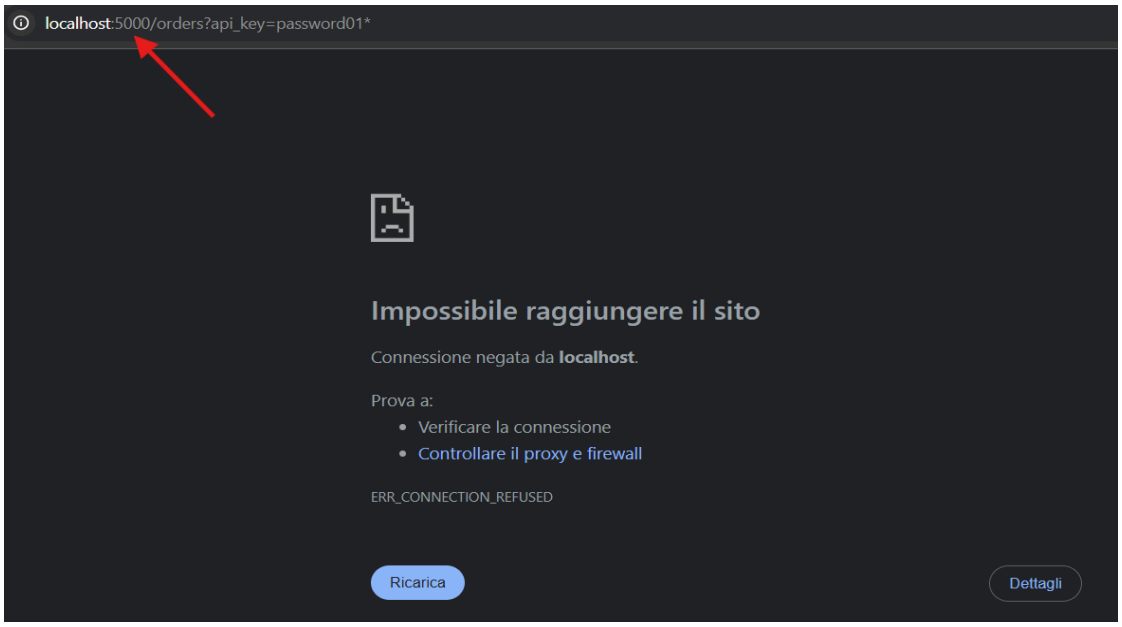


Adesso ad esempio arrestiamo due istanze e osserviamo la variazione dell'output nel sistema di monitoraggio.



Possiamo osservare come l'output dello stato della singola istanza si aggiorni in base alla reale disponibilità da parte di essa.

Scenario 5: in questo ultimo scenario mostriamo l'isolamento dei microservizi.
Il gateway resta in ascolto di richieste dall'esterno sulla porta 8080 e i microservizi internamente sono esposti sulla porta 5000.
Il tentativo di accedere direttamente al microservizio sulla porta interna non andrà mai a buon fine poiché non è stata esposta sull'host. Il servizio dall'esterno è raggiungibile esclusivamente passando tramite il gateway NGINX. Un esempio di raggiungibilità interna solo tramite la rete docker non è altro che il sistema di monitoraggio mostrato nello scenario precedente.



6. Confronto con architetture prive di API-Gateway

In assenza di API Gateway, i client dovrebbero interagire direttamente con i singoli microservizi e sarebbe necessario che conoscano indirizzi, porte e numero di istanze.
In questo tipo di architettura priva di un punto unico di accesso centralizzato, ogni microservizio dovrebbe implementare per sé le proprietà non funzionali quali controllo degli accessi, bilanciamento del carico e tolleranza ai guasti.
Questo porta chiaramente ad avere una complessità maggiore, quindi a essere meno facilmente mantenibile, oltre ad aumentare la probabilità di avere configurazioni incoerenti tra i microservizi.
Dal punto di vista della sicurezza ogni microservizio diventerebbe un punto di accesso, mentre con un API Gateway vi è un unico punto di accesso all'applicazione.

7. Possibili miglioramenti

un primo miglioramento può riguardare l'aspetto della sicurezza. Al momento come mostrato vi è un controllo degli accessi mediante un API key. Questo approccio potrebbe essere sostituito con un sistema di autenticazione più avanzato basato su token o con servizi esterni.
Il sistema di monitoraggio, al momento implementato con uno script dedicato, potrebbe evolvere non mostrando solo lo stato dell'istanza ma anche altre metriche come il consumo di risorse o i tempi di risposta, integrando anche con strumenti dedicati.
Per quanto riguarda la scalabilità, il sistema attuale prevede un numero di istanze definito staticamente. Un miglioramento potrebbe implementare dei meccanismi di scalabilità dinamica basati sul carico.

Infine una possibile evoluzione del progetto potrebbe essere il suo deploy su un cloud provider come Azure o AWS, così da gestire in modo più efficiente la resilienza, scalabilità e il ciclo di vita dei servizi. Successivamente si potrebbe adottare kubernetes come orchestratore dei container.
Inoltre le macchine virtuali e le conseguenti configurazioni per gestire quanto appena detto potrebbe essere implementato tramite una soluzione di tipo IaC(Infrastructure as Code), la quale permetterebbe di descrivere e gestire l'infrastruttura tramite codice, migliorando la manutenibilità e riproducibilità.