S.D.C.C. - Shopping List 2021-2022

Smart Distributed Capable Consumption-Aware Shopping List

Diana Pasquali
Università di Roma Tor Vergata
Roma, Italia
diana.pasquali@alumni.uniroma2.eu

Giacomo Lorenzo Rossi
Università di Roma Tor Vergata
Roma, Italia
giacomolorenzo.rossi@alumni.uniroma2.eu

I. Introduzione

L'obiettivo del progetto è costruire un'applicazione distribuita con una architettura a **Microservizi**. L'applicazione è una lista della spesa, con backend implementato suddividendo le funzionalità dell'applicazione in microservizi, ognuno istanziato su un container per ottenere un maggiore isolamento tra essi. I microservizi sono stati implementati in diversi linguaggi, adattandoli a seconda della funzionalità implementate. Il deployment è stato effettuato sulle nostre macchine tramite docker-compose e su delle istanze EC2 tramite Terraform, Ansible e Docker Swarm.

II. Architettura

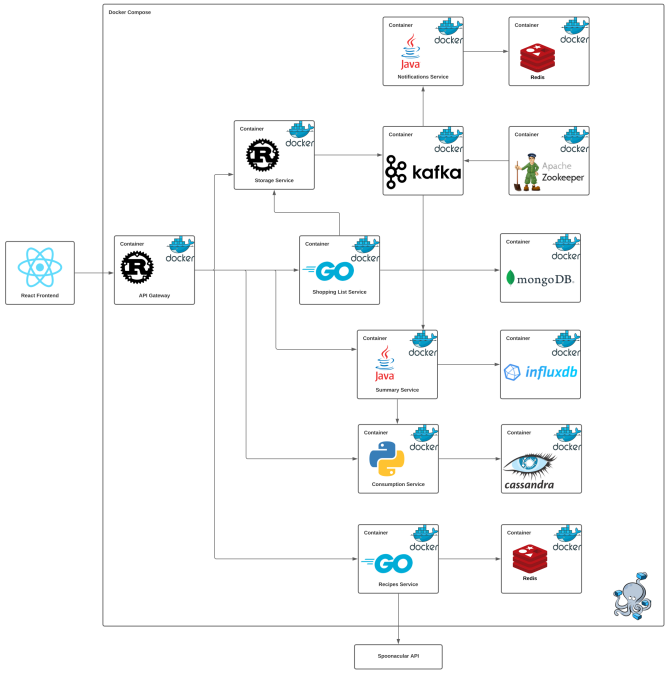

Fig. 1. Architettura a microservizi dell'intera applicazione.

Ai fini del progetto, sono stati implementati i seguenti microservizi, ognuno nel proprio container Docker:

- **Shopping List**: si occupa della lista della spesa vera e propria. Implementato in **Go**, comunica via gRPC con Product Storage per aggiungere nuovi prodotti alla dispensa. Per memorizzare le informazioni dei prodotti in lista, utilizza **MongoDB**, su un container esterno.
- **Product Storage**: implementa la dispensa che mantiene i dettagli dei prodotti acquistati, aggiungendo alle informazioni base del prodotto le seguenti altre informazioni:
  - il numero di volte che il prodotto è stato acquistato.
  - il numero di volte che il prodotto è stato usato.
  - la data di scadenza più prossima.

  Questo microservizio, implementato in **Rust**, è stateful perché contiene all'interno un database **SQLite** su cui vengono memorizzate in maniera persistente le informazioni sui prodotti. Comunica in maniera asincrona e periodica sfruttando il framework Kafka con Notification e Summary, inviando rispettivamente la lista dei prodotti scaduti/esauriti e i logs delle transazioni eseguite sulla dispensa, utili per calcolare le statistiche in Summary. Inoltre, comunica in maniera sincrona via gRPC con Recipes, inviando l'intero contenuto della dispensa nel momento in cui viene richiesta la lista di ricette disponibili.
- **Recipes**: questo microservizio ha il compito di fornire consigli sulle ricette in base ai prodotti che si hanno nel Product Storage. E' implementato in **Go** e comunica con le API REST esterne fornite da Spoonacular, che fornisce dati sulle ricette a partire da specifici ingredienti. Per ridurre il numero di richieste, si sfrutta il data store NoSQL **Redis**, istanziato in un container a parte, per memorizzare il risultato della richiesta all'API esterna per una certa lista di ingredienti specificata. In questo modo, se il contenuto della dispensa non cambia, allora la lista di ricette viene recuperata direttamente da .
- **Notification**: si occupa di notificare al frontend la lista di eventuali prodotti in scadenza e terminati. Comunica tramite **Kafka** con ProductStorage, per recuperare in maniera asincrona la lista di tali prodotti e poi il frontend, richiede in maniera sincrona le notifiche disponibili, salvate temporaneamente da Notification su **Redis**. Redis mantiene i prodotti in scadenza e terminati, per evitare l'invio di notifiche duplicate. Questo microservizio è scritto in **Java**.
- **Consumption**: predice i consumi dei singoli prodotti nella settimana successiva, sfruttando un algoritmo di

regressione incrementale. Questo microservizio è scritto in **Python**, sfruttando le librerie per il machine learning e per l'analisi dei dati come **numpy** e **sklearn**. Comunica via gRPC con Summary, recuperando da questo microservizio le informazioni necessarie a calcolare le features del dataset utilizzato per calcolare i consumi. Il dataset e le ultime predizioni effettuate sono memorizzati nel datastore NoSQL **Cassandra**, istanziato in un altro container.

- **Summary**: fornisce il resoconto sui prodotti più acquistati, scaduti e utilizzati nell'ultima settimana, mese o da sempre. Scritto in **Java**, comunica tramite Kafka con ProductStorage per recuperare i logs e tramite gRPC con Consumptions per inviare la lista dei prodotti usati, comprati e scaduti nell'ultima settimana. I dati ricevuti da ProductStorage vengono salvati nel time series database **InfluxDB**.

Questi microservizi sono accessibili al frontend tramite l'utilizzo di un **API Gateway**, scritto in **Rust**, che fornisce delle API REST che a sua volta richiamano le funzionalità implementate dai vari microservizi. Il frontend è stato sviluppato con **React** e **Typescript**.

III. Microservizi

A. Shopping List

Questo primo microservizio implementa in Go diverse funzionalità di base dell'applicazione:

1) Aggiunta di un prodotto alla lista. Se il prodotto è già in lista, aggiorna solo la quantità da comprare.
2) Modifica di un prodotto.
3) Rimozione di un prodotto dalla lista.
4) Restituzione di tutta la lista
5) Aggiunta di un prodotto al carrello.
6) Rimozione di un prodotto dal carrello.
7) Acquisto di tutti i prodotti nel carrello: i prodotti vengono inviati al microservizio Product Storage con gRPC. In questo caso è stata utilizzata la libreria **Hystrix** per implementare il pattern circuit-breaker. In questo modo è possibile gestire eventuali fallimenti temporanei o crash del servizio Product Storage, per evitare lunghe attese in Shopping List per il trasferimento dei prodotti in dispensa.

Tutte queste funzionalità vanno a modificare la collezione in MongoDB, e per essere richiamate è necessario effettuare una chiamata REST all'API Gateway, il quale a sua volta comunicherà con il microservizio via gRPC. Il microservizio è stateless in quanto non è necessario mantenere informazioni nel microservizio di cui si deve mantenere una consistenza se fossero presenti più repliche.

B. Product Storage

Il Product Storage è un servizio **stateful**, in quanto al suo interno è presente un database SQLite per mantenere i prodotti correntemente in dispensa ed in caso di replicazione del microservizio sarebbe necessario mantenere la consistenza del database tra le varie repliche. Le funzionalità gRPC disponibili, sviluppate in Rust, sono le seguenti:

1) ricezione di tutti i prodotti acquistati provenienti da Shopping List.
2) aggiunta di un prodotto alla dispensa.
3) rimozione di un prodotto dalla dispensa.
4) aggiornamento delle informazioni su un prodotto nella dispensa.
5) utilizzo di una certa quantità di un prodotto.
6) restituzione di tutti i prodotti in dispensa. Oltre che dal frontend via API Gateway, questa funzionalità è anche utilizzata da Recipes, per recuperare i prodotti disponibili con cui si può fare una ricetta.

Le funzionalità (1), (2) e (5) producono anche dei messaggi di log che vengono inviati in maniera asincrona su Kafka. Tali messaggi servono a registrare tutte le transazioni eseguite sulla dispensa ed ogni messaggio di log è associato alle informazioni del prodotto coinvolto nella transazione. Tali transazioni rientrano in queste tre tipologie:

- Prodotto aggiunto in dispensa (`"add_product_to_pantry"`)
- Prodotto comprato aggiunto in dispensa (`"add_bought_product_to_pantry"`)
- Prodotto usato in dispensa (`"used_product_in_pantry"`)

Tali messaggi di log saranno ricevuti dal microservizio Summary, per calcolare il resoconto settimanale, mensile o totale dei prodotti acquistati ed usati. Infine il Product Storage si occupa anche di inviare periodicamente a Kafka dei messaggi sui prodotti scaduti o terminati, che vengono recuperati da Notification. Per far sì che Product Storage sia allo stesso tempo un server gRPC che un producer Kafka, utilizziamo le funzioni asincrone e il paradigma async-await offerto dalla libreria **tokio** di Rust, che semplifica molto il multithreading.

C. Recipes

Recipes fornisce dei consigli sulle ricette che è possibile fare con gli ingredienti in dispensa. E' implementato in Go e ottiene le ricette da una API REST esterna, offerta da Spoonacular. Poiché è una API a pagamento ed esistono dei limiti sia alla frequenza delle chiamate, sia al numero di richieste, si è pensato di sfruttare il datastore in-memory key-value Redis per salvare in cache le ricette richieste a Spoonacular, in modo da poterle recuperare più rapidamente e ridurre il numero di chiamate all' API esterna. Perciò, i risultati delle richieste vengono memorizzati come coppie chiave-valore, aventi come chiave una stringa contenente la concatenazione dei vari prodotti presenti nella dispensa (e.g: "prodotto1-prodotto2-prodotto3-...") e come valore la risposta HTTP ottenuta a partire dalla particolare combinazione di ingredienti. Ogni ricetta recuperata è caratterizzata da diverse informazioni, che includono:

- titolo della ricetta.
- lista degli ingredienti necessari **presenti** nella dispensa.
- lista degli ingredienti necessari **non presenti** nella dispensa.

- URL della pagina del sito Spoonacular relativa alla ricetta.
- URL dell'immagine della ricetta.

D. Notification

Questo microservizio si produce le notifiche per il client sui prodotti scaduti e terminati. E' stato sviluppato in Java e comunica in modo asincrono con il microservizio ProductStorage, tramite il middleware publish/subscribe Kafka. In particolare Notification è il consumer dei messaggi che vengono prodotti dal Product Storage, in due topic differenti: `"finished"` e `"expired"`. Il frontend verifica se esistono delle notifiche da visualizzare, facendo una chiamata Rest all'API Gateway, il quale a sua volta chiama la funzione gRPC eseguita sul microservizio Notification per recuperare le *nuove* notifiche eventualmente presenti ed infine reinoltrarle al frontend. Le notifiche vengono salvate in un Set Redis, con 2 chiavi "expired" o "finished" e come membri di ciascun set i prodotti che sono appartenenti al topic corrispondente. In questo modo, se Product Storage invia un prodotto già presente, non viene duplicato perché viene aggiunto nel Set corrispondente solo se non è già un membro. Dopo un certo TTL i set vengono eliminati automaticamente da Redis.

E. Consumption

Il microservizio Consumption, in Python, esegue training e predizioni su un modello di regressione che stima i consumi dei singoli prodotti ogni settimana. Il modello è stato realizzato con le librerie **scikit-learn**, **pandas** e **numpy**. In particolare il modello predittivo utilizzato è SGDRegressor, un regressore on-line che da un peso maggiore alle ultime osservazioni e usa l'algoritmo di ottimizzazione **Stochastic Gradient Descent** per l'aggiornamento periodico dei parametri di regressione. Il training dei modelli è stato implementato col metodo Walk-Forward: per ogni settimana, tutte le osservazioni delle settimane precedenti sono il training set e la settimana corrente è il testing set. Per ogni prodotto si addestra un modello differente, per predire i consumi settimanali in modo più accurato rispetto a un modello addestrato su tutti i prodotti. L'addestramento viene effettuato solo per i prodotti che sono stati aggiunti, usati o scaduti nell'ultima settimana all'interno della dispensa, ricevendo queste informazioni dal microservizio Summary. Inoltre, dopo la 20-esima settimana, è possibile effettuare un addestramento parziale e on-line del modello tramite la funzione `partial_fit`, senza perdere accuratezza nella predizione. Infine, il microservizio Consumption salva il dataset e le ultime predizioni ottenute all'interno del datastore NoSQL a famiglia di colonne Cassandra, scelto poiché semplice da utilizzare grazie al linguaggio CQL, molto simile a SQL e con una rappresentazione dei dati adatta al salvataggio di dataset. Il training viene effettuato su richiesta del microservizio Summary, il quale ogni settimana esegue una chiamata gRPC a Consumption per aggiornare i modelli, mentre le predizioni vengono richieste direttamente dal client tramite l'API Gateway.

F. Summary

Infine, il microservizio Summary, sviluppato in Java, fornisce tramite gRPC tre funzionalità molto simili: produce le informazioni sui prodotti acquistati, usati e scaduti nella settimana, nel mese o da quando è stata installata l'applicazione. Il riassunto viene generato su richiesta del frontend, che fa una chiamata REST ad API Gateway, che a sua volta invia una richiesta gRPC a Summary. Quindi viene interrogato il database InfluxDB, su cui vengono salvati i log con i prodotti acquistati ed usati, assegnando come timestamp del dato quello associato allo stesso log. Poi, seguendo il percorso a ritroso, viene fornita al frontend la risposta contenente il resoconto per il periodo scelto. Oltre a ciò, il microservizio Summary esegue altre due operazioni periodiche:

- periodicamente Summary controlla se su Kafka sono stati pubblicati dal Product Storage dei messaggi per prodotti scaduti o terminati. In caso positivo, questi prodotti vengono salvati sul database a serie temporali InfluxDB. La scelta di InfluxDB è stata influenzata dal fatto che i log inviati da Product Storage contengono un timestamp, e poiché summary si occupa di ricavare dati aggregati sui prodotti acquistati ed usati in un particolare periodo temporale, la scelta di un database time-series è naturale, in quanto ottimizzato per questo tipo di query.
- tramite un thread **chron-job**, ogni settimana recupera dal InfluxDB i prodotti acquistati ed usati fin'ora nell'ultima settimana, effettuando una richiesta di training a Consumptions, per aggiornare i modelli dei prodotti e quindi i consumi della settimana successiva.

IV. Pattern Implementati

A. API Gateway

L'API Gateway è implementato in Rust utilizzando diverse librerie (`crates`). La scelta del linguaggio e' dovuta al fatto che essendo l'API gateway l'unico punto di accesso a tutti i servizi, sono necessarie performance elevate. Inoltre, i crates esistenti si integrano molto bene tra loro e permettono facilmente di implementare le API Rest e gRPC in modo asincrono, con bassissimo consumo di risorse.

- **tokio**: un runtime che permette la programmazione asincrona in Rust
- **tonic**: crate che permette di generare e compilare i file grpc, in automatico durante la compilazione del codice. Inoltre permette di definire le funzioni grpc (asincrone).
- **prost**: implementazione di protocol buffer per Rust.
- **actix-web**: framework che permette di implementare API Rest in modo asincrono (basato su tokio).
- **failsafe**: libreria che implementa il pattern **circuit-breaker**, compatibile con tokio.

La nostra implementazione di API Gateway fa da proxy tra il frontend e il backend a microservizi: per ogni funzionalità dei microservizi che può essere eseguita su richiesta, è stata creata una API REST, che a sua volta comunica con il microservizio richiesto tramite gRPC. Se la richiesta ha successo, viene inviato il risultato al frontend in formato **json**, altrimenti viene

inviato un messaggio di errore. Se il microservizio sottostante è down, il crate failsafe permette di evitare che il client attenda un tempo troppo lungo, restituendo immediatamente il messaggio di errore. Perciò, in caso di crash o fallimenti intermittenti dei vari microservizi, tramite il circuit-breaker è possibile interrompere temporaneamente la comunicazione con uno o più microservizi, evitando in questo modo errori di comunicazione o attese troppo lunghe per ricevere una risposta dal microservizio guasto.

B. Database Per Service

Per implementare questo pattern si è scelto di usare un database diverso per ogni microservizio. In questo modo si riesce a sviluppare in modo indipendente i microservizi e si riesce a soddisfare al meglio i requisiti di ciascuno adattando le diverse tecnologie selezionate allo specifico caso d'uso. Di seguito le motivazioni sulla scelta dei datastore/database.

- **Shopping List**: in questo caso abbiamo utilizzato il datastore NoSQL **MongoDB**, in quanto i prodotti aggiunti potrebbero avere dei campi opzionali, come la data di scadenza e la quantità
- **Product Storage**: questo microservizio è stateful, perciò abbiamo preferito usare un database file-based come **SQLite**, in modo che il microservizio potesse avere uno stato interno. Tuttavia la presenza di un database relazionale rende più complessa un'eventuale replicazione del microservizio, in quanto è più difficile gestire la consistenza tra le differenti repliche.
- **Recipes**: poiché gli ingredienti in dispensa potrebbero avere una elevata variabilità, non è conveniente salvare le ricette consigliate in modo persistente. Perciò abbiamo usato il datastore key-value **Redis** come cache in modo da ridurre al minimo le richieste alle API Spoonacular e di recuperare velocemente i dati poiché memorizzati in RAM.
- **Notification**: anche in questo microservizio utilizziamo **Redis**, per mantenere temporaneamente le notifiche prodotte da Notification e che devono essere inviate al frontend. Anche in questo caso non è di interesse salvare le notifiche in modo persistente.
- **Consumptions**: per salvare il dataset su cui fare training abbiamo usato un datastor NoSQL a famiglia di colonne a consistenza tunable: **Cassandra**. Il fatto che permette di usare il linguaggio CQL ha permesso di semplificare lo sviluppo delle query.
- **Summary**: questo microservizio riceve i log da kafka dei prodotti nella dispensa comprati, aggiunti manualmente e usati. Poiché Summary fa dei calcoli su dati di serie temporali in base al tempo, utilizzare un database time-series come **InfluxDB** permette di ottimizzare il recupero di tali dati dalla persistenza, ottimizzando i calcoli.

C. Circuit Breaker

Il Pattern Circuit breaker permette di evitare che i client attendino troppo a lungo quando fanno una richiesta a un servizio non disponibile, ad esempio a causa di interruzioni di rete o crash. E' stato implementato il pattern in tutti quei casi in cui è stato usato gRPC come client, ovvero:

1) Lato API Gateway, quando fa le richieste ai singoli microservizi. Per ciascuna API Rest è stato usato il crate **failsafe** Rust per evitare di attendere troppo sia in fase di connessione al microservizio, sia in fase di chiamata.
2) Lato Shopping List, quando viene richiesto dall'API Gateway di inviare tutti i prodotti nel carrello nel Product Storage, è stata usata la libreria **Hystrix** in Go.
3) Lato Summary, quando si deve connettere al microservizio Consumpion, è stata utilizzata la libreria **Resilience4j**.

Sia failsafe che Resilience4j prendono ispirazione dalla libreria Hystrix. Ciascuna di esse implementa degli health-check (ritrasmissioni, timer) e permettono di definire dei fallback nei casi in cui il servizio viene rilevato come down. In Go sono stati usati i *channel* per poter inviare tra il main thread e la goroutine di hystrix il risultato della connessione o della funzione grpc. In Rust, è stata utilizzata la versione *asincrona* del circuit breaker di failsafe per mantenere la compatibilità con il resto delle librerie usate. In Java invece abbiamo usato le API fornite dalla libreria Resilience4j, in quanto è stato utilizzato un *thread* chron-job che periodicamente comunica con Consumptions tramite una funzione gRPC bloccante.

V. Frontend

Il frontend è stato realizzato in Typescript e con la libreria React. Nel frontend sono presenti 4 pagine fondamentali:

1) **Home**: è la pagina che mostra la lista della spesa e in cui è possibile aggiungere e togliere prodotti dalla stessa ed aggiungere prodotti al carrello. Inoltre tramite l'apposito bottone, è possibile acquistare tutti i prodotti nel carrello, aggiungendoli allo storage.

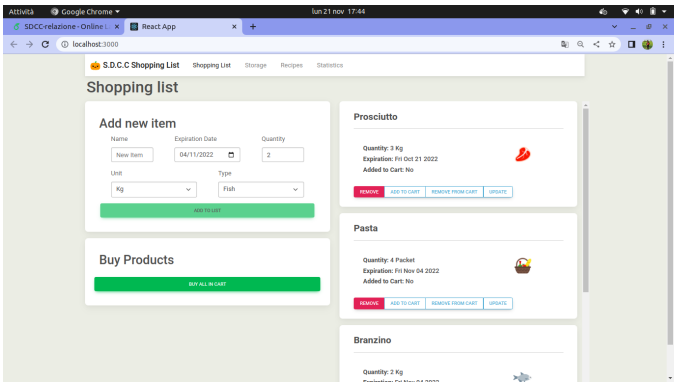

Fig. 2. Home page.

2) **Storage**: in questa pagina è possibile visualizzare gli elementi attualmente nello storage e lo stato (Disponibile, Terminato o Scaduto). Inoltre tramite appositi form, è possibile aggiungere e utilizzare elementi nello storage. Infine è possibile rimuovere o aggiornare i prodotti.
3) **Recipes**: In questa pagina è possibile visualizzare la lista di ricette disponibili ricavate a partire dall'intero contenuto dello storage allo stato attuale.

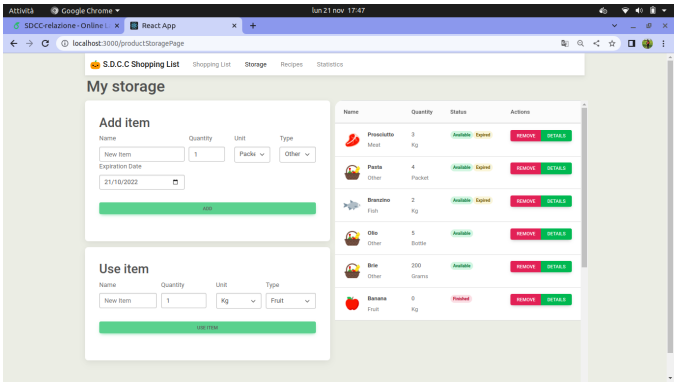

Fig. 3. Storage page.

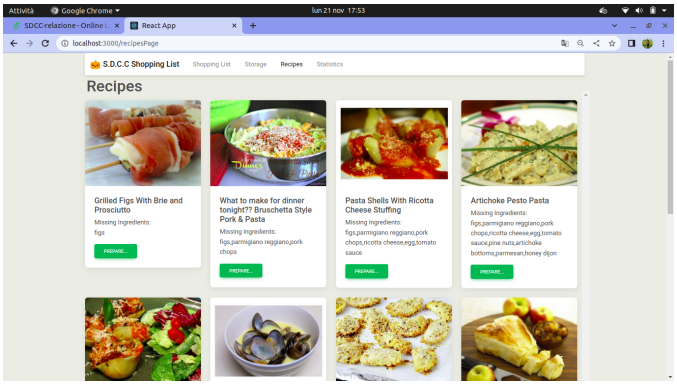

Fig. 4. Recipes page.

4) **Statistics**: In questa pagina è possibile visualizzare la raccolta di statistiche provenienti dai servizi Summary e Consumption, e inoltre è possibile visualizzare eventuali notifiche presenti sulla disponibilità o scadenza di prodotti nello storage.

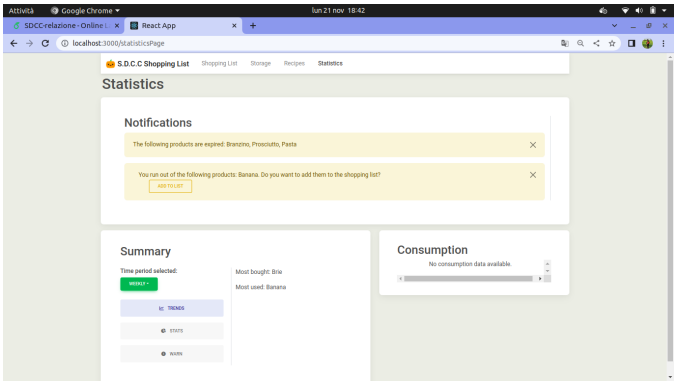

Fig. 5. Statistics page.

VI. Deployment

Il deployment è stato effettuato in due diverse modalità:

1) Con **Docker Compose** sulla singola macchina locale. Sono stati inizialmente definiti i 7 Dockerfile per i microservizi e l'API gateway, nella stesura dei quali si è cercato di sfruttare al massimo il caching dei layer, in modo da ridurre al minimo i tempi di build e le dimensioni delle immagini, ad esempio eseguendo prima il download delle dipendenze e poi compilando il codice e sfruttando le *multi-stage build*. Dopodiché tutti i container vengono avviati con docker compose. I parametri di configurazione vengono passati a tutti i container tramite un *bind-mount* del file *config.properties*.
2) Con **Docker Swarm** su delle macchine **EC2** istanziate con **Terraform**, il tool open-source per gestire l'infrastruttura da codice. Per prima cosa abbiamo eseguito il push delle immagini dei 7 container (microservizi e api gateway) su Docker Hub. Dopodiché con Terraform abbiamo avviato N macchine EC2 (N configurabile nel file main.tf) sfruttando le credenziali e i crediti forniti dal Lab AWS. Poi copiamo i file config.properties e docker-stack.yml sulle istanze EC2, avviamo uno swarm formato da N nodi e lanciamo lo Stack di Docker Service che rappresenta l'intera applicazione. Per poter comunicare tra loro, sono state aperte sulle istanze EC2 tramite Terraform le porte in ingresso e in uscita necessarie per far funzionare Docker swarm, nonché quelle definite nel file config.properties. Sfruttando il framework Ansible, è stato possibile automatizzare tutto il processo di provisioning, evitando di eseguire manualmente la copia dei file sulle istanze EC2 e la creazione automatica dello swarm.

VII. Limitazioni e sviluppi futuri

Le limitazioni riscontrate nel progetto sono la mancata replicazione del microservizio stateful, Product Storage, che potrebbe essere un single point of failure. Inoltre abbiamo notato come i microservizi in Java e Python siano molto più resource-consuming di quelli in Go e Rust, mentre lo sviluppo dei microservizi in Java e Rust non è rapido come quello in Go e Python, perché i tempi di compilazione sono non trascurabili. Consigliamo quindi Go per avere un buon compromesso tra velocità di esecuzione e di compilazione e Rust per i microservizi che necessitano il massimo delle performance. Come sviluppo futuro, si è pensato di sostituire Kubernetes a Swarm, possibilmente usando un servizio cloud come Amazon EKS, che permettebbe di semplificare il deploy soprattuto dei servizi stateful, in quanto docker swarm non li supporta nativamente. Abbiamo in mente di aggiungere diverse funzionalità: un microservizio "catalogo", per semplificare l'inserimento di nuovi prodotti in lista (ad esempio per evitare di inserire tipo e unità), la traduzione in altre lingue e la ricerca dei prodotti. . Altrettanto utili ma secondarie sono la gestione di liste multiple per ciascun utente, la condivisione delle liste e il salvataggio di ricette preferite. Lato database, il datastore Cassandra è risultato molto lento all'avvio, mentre SQLite è file-based quindi non proprio adatto a richieste concorrenti. Abbiamo quindi esplorato diverse possibilità alternative come ad esempio il database NewSQL SurrealDB, che è leggero, serverless, multi-model e scalabile.

REFERENCES

- Documentazione Docker:https://docs.docker.com/
- Documentazione Kafka https://kafka.apache.org/documentation/
- Documentazione Rust: https://doc.rust-lang.org/stable/book/
- Actix-Web: https://actix.rs/
- API Spoonacular: https://spoonacular.com/food-api