



Università
di Catania

Corso di *Distributed Systems and Big Data*

Gabriele Vitali (Matricola: 1000010255)

Relazione Progetto Finale:

Piattaforma per la prenotazione di cibo d'asporto

Sommario

1. Descrizione generale dell'applicazione	3
2. Schema architetturale e componenti	4
3. Elenco dei microservizi e delle API implementate	7
3.1 Authentication Service	7
3.1.1 Signup [POST]	7
3.1.2 Login [POST]	8
3.2 Food Service	9
3.2.1 Ricerca piatto [POST]	9
3.2.2 Gestione dei piatti (POST, DELETE)	10
3.3 Order Service	10
3.3.1 Creazione nuovo ordine [POST]	10
3.3.2 Verifica dello stato di un ordine [GET]	11
3.4 Payment Service	11
3.4.1 Ricarica del credito [POST]	11
3.4.2 Verifica della cifra totale spesa da un utente nel mese specificato [GET]	12
4. Gestione della transazione distribuita	13
5. Kubernetes	15
6. Black-box and White-box Monitoring	18
7. Indicazioni per build & deploy;	20
Istruzioni per l'esecuzione	20

1. Descrizione generale dell'applicazione

Il progetto in questione consiste in una piattaforma per la prenotazione di cibo d'asporto, strutturata come sistema distribuito con **architettura a microservizi**.

In particolare, sono stati definiti i seguenti microservizi:

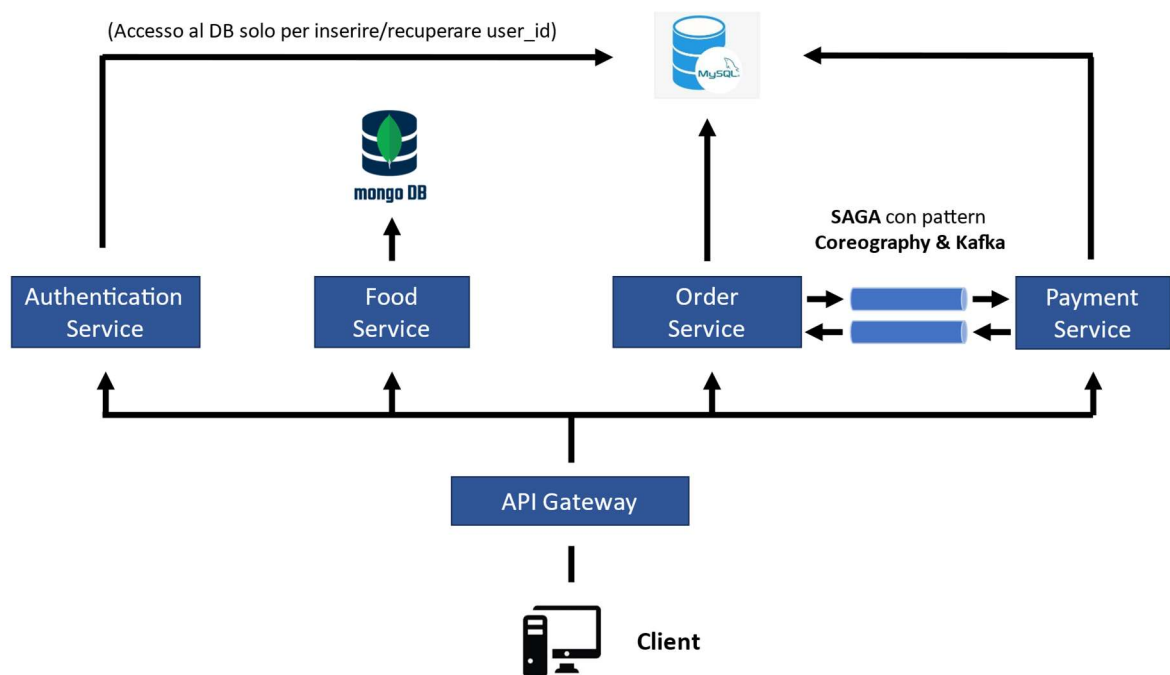
- **Authentication Service:** fornisce le funzionalità di registrazione e autenticazione dell'utente, discriminando fra user e admin; inoltre, completata l'autenticazione, si occupa di generare e fornire un token necessario all'utente per accedere alle altre funzionalità offerte dalla piattaforma.
- **Food Service:** fornisce agli utenti funzionalità di ricerca dei piatti disponibili per la prenotazione, sulla base del nome e della categoria specificati. Inoltre, consente all'admin di gestire i piatti già esistenti. Food Service recupera e modifica i dati presenti nel database MongoDB.
- **Order Service:** fornisce il meccanismo di creazione di un nuovo ordine, sulla base del cibo selezionato dall'utente, così come quello di monitoraggio dello stato corrente dell'ordine stesso.
- **Payment Service:** gestisce il pagamento dell'ordine, verificando che l'utente abbia credito sufficiente per completare la transazione; fornisce, inoltre, funzionalità di ricarica del credito dell'utente e consente a quest'ultimo di visualizzare la statistica relativa all'importo totale speso nel mese specificato.

Il flusso principale di utilizzo prevede la registrazione e l'autenticazione dell'utente, seguite dalla ricerca del cibo da prenotare. Dopo aver trovato il piatto da prenotare, si potrà confermare la scelta e procedere ad effettuare l'ordine: verrà creato un ordine con stato di pagamento pendente. Il pagamento prevede la decurtazione dell'importo totale dell'ordine dal credito associato all'utente. L'ordine potrà essere effettuato con successo solo qualora il credito associato all'utente sia sufficiente (pagamento completato), altrimenti nessuna cifra verrà decurtata e la procedura fallirà (pagamento fallito).

L'utente potrà monitorare lo stato dell'ordine effettuato ed eventualmente ricaricare il proprio credito, così come visualizzare l'ammontare della propria spesa in un dato mese.

L'altro flusso di utilizzo è quello relativo alla gestione dei piatti ad opera dell'admin.

2. Schema architetturale e componenti



L'applicazione utilizza un API Gateway Nginx come unico punto di accesso, seguendo l'*API Gateway Pattern*. Il client può inoltrare tutte le richieste verso un unico endpoint, specificando di volta in volta la risorsa necessaria. L'API Gateway si occupa di instradare le richieste verso i rispettivi microservizi, semplificando l'interazione lato client ed eliminando la necessità di conoscere gli URL specifici di ciascun microservizio. Grazie a questa forma di decoupling e location transparency, i microservizi possono evolversi, essere sostituiti o riorganizzati senza avere un impatto diretto sul client, rendendo l'architettura più flessibile e scalabile.

Questo approccio garantisce, infine, una gestione centralizzata del controllo degli accessi ai servizi back-end, migliorando la sicurezza e la manutenibilità.

Per quanto concerne il meccanismo di autenticazione, si è preferito creare un microservizio ad hoc (Authentication Service) anziché sfruttare l'API Gateway già presente. Ciò consente, infatti, di ottenere un controllo più fine sulla logica di autenticazione e autorizzazione, potendo implementare strategie personalizzate e più articolate, oltre al fatto che in questo modo il processo di autenticazione è isolato rispetto agli altri microservizi, favorendone la manutenibilità e l'evoluzione nel tempo.

A livello di comunicazione fra i microservizi, invece, si segnala che la comunicazione fra Order Service e Payment Service è basata su *Apache Kafka*, che funge da sistema di messaggistica distribuito. Questa scelta offre molteplici vantaggi:








- **decoupling** tra i microservizi: Kafka consente ai microservizi di comunicare in modo asincrono, eliminando la dipendenza diretta e favorendo il disaccoppiamento temporale. Ciò implica che Order Service può continuare a operare senza dover attendere né che Payment Service sia disponibile né che esso restituisca una risposta, e viceversa;
- favorisce la **scalabilità** del sistema, all'aumentare delle comunicazioni fra i due microservizi, ovvero all'aumentare della richiesta di nuovi ordini, nel caso della piattaforma in questione;
- **affidabilità e durability**: Kafka è in grado di garantire che nessun evento venga perso anche in caso di interruzioni o errori temporanei nei servizi;
- **supporto** all'implementazione di pattern **Saga Choreography**, adottato per gestire in modo robusto le transazioni distribuite fra i due microservizi previste nell'applicazione in questione.

Un approccio analogo si sarebbe potuto adottare per la comunicazione fra Food Service e Order Service una volta che il cliente ha confermato la scelta del piatto da prenotare. Ciò non è stato fatto, però, in quanto la ricerca del piatto e la conferma della scelta da parte dell'utente prevede una comunicazione fra Food Service e il client. Ad esempio, si pensi al caso in cui il piatto cercato dall'utente non sia disponibile: Food Service cerca un piatto alternativo della stessa categoria da proporre all'utente e, solo dopo che quest'ultimo avrà confermato la scelta del piatto alternativo si potrà passare alla creazione dell'ordine, con la trasmissione dei dati a Order Service.

Per quanto concerne, invece, la scelta dei database utilizzati dai diversi microservizi, essa è stata motivata dalle seguenti ragioni:

- Food Service utilizza un database **MongoDB** in quanto particolarmente adatto alla natura dei dati manipolati dal microservizio, i quali hanno una struttura particolarmente semplice e flessibile. Difatti, si pensi che non necessariamente per tutti i piatti debba essere definito ogni attributo (si pensi, ad esempio, al campo “descrizione”). Per il database MongoDB è stata utilizzata un'immagine ufficiale.

Database MongoDB: *meals_db* ; collection: *meals*

_id	name	description	category	price
  6751a89f2dfa11f2ca7545fe	hot roll	roll fritti con philadelphia e fragole	sushi	14
  6751a8ac2dfa11f2ca7545ff	margherita	mozzarella di bufala DOP, pomodoro San Marzano, o...	pizza	9
  6751a8b72dfa11f2ca754600	gyoza	ravioli grigliati di verdure	sushi	8
  6751a8c02dfa11f2ca754601	sweet dreams	provola affumicata di Napoli, bacon, patate dolci...	pizza	12

- Authentication, Order e Payment Service utilizzano invece un database **MySQL**: infatti, data la natura dei dati manipolati dai microservizi in questione e dal momento che essi sono sia articolati sia fra loro correlati, si rende necessaria una gestione più rigida e precisa. Ogni campo ha, difatti, un preciso significato e una sua importanza nel flusso di funzionamento dell'applicazione. Per il database MySQL è stata utilizzata un'immagine custom, ottenuta a partire da un'immagine ufficiale cui è stato aggiunto uno script *init.sql* personalizzato, così da ottenere un database già inizializzato e *ready-to-use*.

Database MySQL: mysql-db ; tabelle: Utenti, Ordini, Pagamenti

username	password	email	admin	credito
admin	\$2b\$12\$cmqAR4MH0k4dB8fInx1i3uI/Cx6GB08Pv0F.FTHFzr2GoYikmDJHO	admin@foodies.com	1	0.00
user	\$2b\$12\$3kW9ABGJDeZJa9wb9okBbug/pcD3Gh2d.RIO2.0ZJvkJskjUkdGEI	user@foodies.com	0	364.00

order_id	username	status	created_at	updated_at	amount	meal
86326	user	CONFIRMED	2024-12-09 15:15:05	2024-12-09 15:15:05	36.00	hot roll

payment_id	order_id	username	payment_timestamp	amount	status
33377	86326	user	2024-12-09 15:15:05	36.00	SUCCESS

3. Elenco dei microservizi e delle API implementate

Complessivamente, l'applicazione prevede l'implementazione dei seguenti microservizi, ciascuno dei quali fornisce specifiche API:

- Authentication Service:
 - *signup*
 - *login*
- Food Service:
 - *search meal*
 - *manage meals (create & delete)*
- Order Service:
 - *create order*
 - *check order status*
- Payment Service:
 - *add credit*
 - *get total month payment*

Nelle pagine che seguono, viene fornita una descrizione dettagliata delle API fornite da ciascuno dei microservizi previsti nell'ambito dell'applicazione in questione.

3.1 Authentication Service

Consente al client di effettuare:

- registrazione (*signup*)
- accesso (*login*)

3.1.1 Signup [POST]

Il client invia username, password e email. La richiesta include, inoltre, un campo *admin* che, a seconda del valore, determina il *role* dell'utente registrato (0: *user*, 1: *admin*). L'Authentication Service elabora la richiesta e verifica se nel database MySQL sia già presente un utente avente l'username specificato. In caso negativo, il servizio aggiunge nel database MySQL le informazioni relative al nuovo utente.

In caso di informazioni mancanti o username già presente in db, l'errore viene opportunamente gestito e la richiesta viene rigettata.

Nel caso di registrazione di un admin, un ulteriore campo "credito" verrà impostato a 0; nel caso di un user generico, invece, per convenzione verrà settato ad un valore di default. Si tratta di una semplificazione, legata al fatto che in questa versione del progetto non è stato previsto l'inserimento di un apposito service volto a consentire all'user di ricaricare il proprio credito.

A completamento di quanto detto, si segnala che è stato gestito l'aspetto relativo alla security: la password viene inviata in chiaro dal client al server, il quale la cifra e memorizza cifrata in database. Ipotizzando che si utilizzi un protocollo HTTPS, ciò consente di evitare potenziali *replay attack*.

3.1.2 Login [POST]

- nel caso in cui il client non sia ancora in possesso di un token valido, invia username e password all'Authentication Service. Quest'ultimo elabora la richiesta e verifica se nel database MySQL sia presente una coppia username-password corrispondente a quella specificata dal client. In caso affermativo, il servizio genera un **token JWT (JSON Web Token)**, che il client utilizzerà successivamente per effettuare le richieste a Food Service;
- nel caso in cui il client sia già in possesso di un token valido, invece, non sarà necessario contattare l'Authentication Service e potrà effettuare direttamente una richiesta al Food Service, specificando **token e piatto richiesto**;
- vengono gestiti correttamente casi di username non trovato o password errata.
- sicurezza: per le stesse ragioni già specificate per il *signup*, il server riceve la password in chiaro dal client, la cifra e la confronta con quella memorizzata in database (già cifrata).

In generale, un token *JWT* viene usato per l'autenticazione e la gestione delle autorizzazioni. Contiene tre parti: header, payload, e signature, ognuna delle quali avente un ruolo specifico. In particolare, il token generato dall'Auth Service contiene informazioni essenziali sull'utente e di tipo temporale:

- **sub** (*subject*): identità del soggetto (esempio: username)
- **role**: consente all'applicazione di differenziare i permessi tra utenti
- **iat** (*issued at*): timestamp dell'emissione
- **exp** (*expiration*): scadenza del token (esempio: 1 ora)

Un approccio di questo tipo introduce molteplici **vantaggi**:

- **efficienza**: si riduce la necessità di effettuare costanti richieste all'Authentication Service e query al database MySQL per recuperare le informazioni dell'utente, riducendo la pressione su di essi;
- **sicurezza**: la firma digitale del token garantisce l'integrità e l'autenticità delle informazioni contenute;
- **decoupling**: consente di disaccoppiare il Food Service (e altri potenziali microservizi introdotti in una fase successiva) dall'Auth Service, rendendo più indipendente e scalabile l'architettura;

- **gestione delle autorizzazioni e personalizzazione:** il campo *role* del token permette allo stesso tempo sia di verificare che un certo client possa accedere a determinate funzionalità sia di personalizzare l'esperienza dell'utente.

3.2 Food Service

Consente al client di effettuare:

- ricerca di un pasto da acquistare (*search meal*) [user]
- gestione (POST, DELETE) dei piatti memorizzati in db (*manage meals*) [admin]

3.2.1 Ricerca piatto [POST]

Il client invia il token come header *Authorization* della richiesta al Food Service.

Oltre al token, vengono specificati dall'utente e inseriti nella richiesta il nome del piatto cercato e la sua categoria di appartenenza.

Qualora il token sia scaduto, prima di contattare il Food Service il client dovrà ripetere la procedura di *login* tramite l'Authentication Service.

Quando Food Service riceve una richiesta dal client, sia esso un generico user oppure l'admin, effettua la validazione del token. Essa è fondamentale affinché si possa garantire la sicurezza dell'applicazione, consentendo l'accesso solo a utenti autorizzati. In particolare, essa si basa sui seguenti aspetti:

- verifica della **firma digitale**: garantisce integrità, ovvero assicura che il token non sia stato manomesso
- controllo della **data di scadenza**: impedisce l'utilizzo di token scaduti
- verifica dell'**issuer**: garantisce autenticità, ovvero che il token sia stato emesso dalla fonte autorevole attesa (nel caso del corrente progetto, l'*auth service*)
- verifica del **role**: permette di consentire l'accesso a certe funzionalità solo agli effettivi destinatari. Ad esempio, solo un client con *role user* potrà effettuare la ricerca dei piatti da acquistare e solo un client con *role admin* potrà gestire i piatti in vendita.

In caso di token non valido (ad esempio, in caso di contenuto alterato oppure di *issuer* o *role* diversi da quelli attesi), la richiesta verrà rigettata.

In caso contrario, si procederà con la ricerca del piatto specificato dall'utente, in MongoDB. In caso di piatto non trovato, Food Service cerca un piatto alternativo della stessa categoria, da proporre all'utente. Qualora, anche in questo secondo caso, nessun piatto idoneo venga trovato, la richiesta verrà rigettata. Viceversa, in caso di esito positivo della ricerca, verranno restituite al client le informazioni relative al prodotto richiesto. A questo punto, il client potrà o meno confermare di voler procedere con l'ordine.

3.2.2 Gestione dei piatti (POST, DELETE)

La funzionalità di gestione dei piatti consente all'admin sia di crearne di nuovi sia di eliminarne uno già esistente.

In entrambi casi, il client invia il token come header *Authorization* della richiesta al Food Service. Anche per la gestione dei piatti, valgono le stesse considerazioni precedentemente riportate in merito al token, al controllo della sua validità e alla gestione delle autorizzazioni (vedi API "Ricerca piatto").

Nel caso della creazione di un nuovo piatto, oltre al token vengono specificati dall'admin e inseriti nella richiesta i seguenti campi: nome del piatto, descrizione, categoria di appartenenza e prezzo.

In particolare, la categoria viene sfruttata in fase di ricerca di un piatto (*search meal*): qualora il piatto cercato non sia disponibile, verrà proposto un piatto alternativo della stessa categoria.

Food Service elabora la richiesta, verifica che in database (MongoDB) non sia già presente un piatto avente lo stesso nome e memorizza le informazioni relative al nuovo piatto. In caso di esito positivo, invia al client un messaggio di conferma, altrimenti segnala un opportuno messaggio di errore.

Nel caso della cancellazione di un piatto già esistente, invece, oltre al token viene specificato, nel path della richiesta, il nome del piatto che si vuole cancellare. Se il piatto specificato è effettivamente presente in database, verrà cancellato, inviando al client un messaggio di conferma, altrimenti segnalerà un opportuno messaggio di errore.

3.3 Order Service

Consente di:

- avviare la procedura di creazione di un nuovo ordine (*create order*) [user]
- verificare lo stato di un ordine (*check order status*) [user]

3.3.1 Creazione nuovo ordine [POST]

Il client invia il token come header *Authorization* della richiesta all'Order Service.

Oltre al token, vengono specificati dall'utente e inseriti nella richiesta i dettagli relativi al pasto che s'intende acquistare (nome, categoria, prezzo).

Relativamente al token, le stesse considerazioni fatte per Food Service valgono anche per Order Service.

Dopo aver verificato la validità del token, Order Service:

- genera e memorizza nel database MySQL le informazioni sul nuovo ordine;

- pubblica un nuovo messaggio associato al topic “order-event”, specificando al suo interno le seguenti informazioni: il codice univoco dell’ordine, l’username dell’utente che ha richiesto la creazione del nuovo ordine, il nome del cibo richiesto e l’importo totale dell’ordine.

La pubblicazione del nuovo messaggio dà inizio alla **transazione distribuita** fra Order Service e Payment Service, gestita secondo un pattern **Saga** di tipo **Choreography** (vedi apposito paragrafo più avanti): il messaggio associato al topic *order-event* verrà consumato da Payment Service, il quale si occuperà sia di verificare che l’utente avente l’username specificato abbia un credito sufficiente sia di memorizzare in database le informazioni sul pagamento, dopo di che creerà un nuovo messaggio associato al topic *payment-event*. Quest’ultimo verrà a sua volta consumato da Order Service per aggiornare lo stato dell’ordine in database (da *pending* a *confirmed/cancelled* a seconda che il campo *status* del messaggio associato a *payment-event* abbia valore *success/failed*).

3.3.2 Verifica dello stato di un ordine [GET]

L’utente specifica l’*order id* dell’ordine di cui vuole verificare lo stato (possibili valori: *pending, failed, confirmed*).

Il client invia il token come header *Authorization* della richiesta all’Order Service.

Relativamente al token, valgono le considerazioni già fatte precedentemente.

Dopo aver verificato la validità del token, Order Service elabora la richiesta, procedendo con la ricerca dell’ordine avente l’*order id* specificato, con particolare riferimento al campo *status*. Se presente, restituisce al client l’informazione richiesta, altrimenti segnala un errore.

L’API in questione consente all’utente di verificare lo stato di un certo ordine dopo averlo creato, tramite l’apposita API *create order*, sopra descritta.

3.4 Payment Service

Consente di:

- ricaricare il credito di un certo utente (*add credit*) [user]
- visualizzare la cifra totale spesa da un certo utente nel mese specificato (*get total month payment*) [user]

3.4.1 Ricarica del credito [POST]

Il client invia il token come header *Authorization* della richiesta al Payment Service.

Oltre al token, viene inserito nella richiesta la cifra specificata dall’utente per ricaricare il proprio credito.

Relativamente al token, valgono le considerazioni già fatte precedentemente. Dopo aver verificato la validità del token, il Payment Service elabora e gestisce la richiesta, ricaricando il credito del corrispondente user (se presente) memorizzato nel database MySQL (tabella “Utenti”).

3.4.2 Verifica della cifra totale spesa da un utente nel mese specificato [GET]

L'utente specifica il mese di interesse.

Il client invia il token come header *Authorization* della richiesta al Payment Service.

Relativamente al token, valgono le considerazioni già fatte precedentemente.

Dopo aver verificato la validità del token, Payment Service elabora la richiesta: viene eseguita la ricerca di tutti gli ordini effettuati dall'utente in questione nel mese specificato e contestualmente si calcola la somma totale dei singoli importi degli ordini effettuati. Il valore calcolato viene, infine, restituito al client.

4. Gestione della transazione distribuita

Come già anticipato nel paragrafo relativo a Order Service, è stato adottato un pattern di tipo *SAGA* con approccio *Choreography*. Quest'ultimo è stato preferito al pattern Orchestrator dal momento che in questo caso solo due entità sono coinvolte nello scambio e l'altro approccio avrebbe introdotto una complessità aggiuntiva in questa fase eccessiva, con un'ulteriore service Orchestrator e maggior scambio di messaggi). In un'ottica futura, invece, implementare un Orchestrator avrebbe favorito una maggiore scalabilità del sistema.

Il pattern Saga Choreography è stato implementato mediante una comunicazione basata su Kafka Apache, con pubblicazione di messaggi/event associati ai topic *order-event*, *payment-event* e *payment-compensation-event*.

Di seguito si riporta il flusso di gestione della transazione distribuita fra i servizi Order Service e Payment Service (incluso il rollback in caso di un eventuale malfunzionamento).

In caso di credito **sufficiente**:

1. **Order** Service crea un ordine in stato *pending* e pubblica un evento in *order-event*, contenente i dettagli dell'ordine;
2. **Payment** Service consuma l'evento, scala al credito dell'utente associato all'ordine l'importo di quest'ultimo e pubblica un *payment-event* con status *success*;
3. **Order** Service consuma *payment-event* e aggiorna lo stato dell'ordine da *pending* a *confirmed*.

In caso di credito **insufficiente**:

1. **Order** Service crea un ordine in stato *pending* e pubblica un evento in *order-event*, contenente i dettagli dell'ordine;
2. **Payment** Service rileva che il credito è insufficiente e pubblica *payment-event* con status *failed*;
3. **Order** Service consuma *payment-event* e aggiorna lo stato dell'ordine da *pending* a *cancelled*.

Nel caso in cui **Payment** Service **fallisca**, dopo aver già scalato il credito dell'utente, viene prodotto un *compensation-event*:

1. **Order** Service crea un ordine in stato *pending* e pubblica un evento in *order-event*, contenente i dettagli dell'ordine;
2. **Payment** Service consuma l'evento, scala al credito dell'utente associato all'ordine l'importo di quest'ultimo e, prima di pubblicare un *payment-event* con status *success*, s'interrompe per un malfunzionamento;
3. **Order** Service consuma *compensation-event* per:
 - ripristinare il credito dell'utente;
 - aggiornare lo stato dell'ordine da *pending* a *cancelled*.

A completamento di quanto riportato sopra, si riportano di seguito alcuni screenshot di esempio relativi sia ai topic Kafka *order-event* e *payment-event* sia ad alcuni messaggi ad essi associati. Nell'esempio, è stato effettuato un primo ordine andato a buon fine (*confirmed*) e un secondo ordine non completato (*failed*), a causa del credito insufficiente dell'utente.

Nell'esempio non figura, inoltre, alcun topic *compensation-event* perché nel caso specifico nessun malfunzionamento si è verificato in Payment Service dopo esser intervenuto sul credito dell'utente e prima di pubblicare un *payment-event*.

```
sh-4.4$ kafka-topics --bootstrap-server kafka:9092 --list
__consumer_offsets
order-event
payment-event
```

```
sh-4.4$ kafka-console-consumer --bootstrap-server kafka:9092 --topic order-event --from-beginning
{"order_id": "56716", "username": "user", "meal": "hot roll", "price": 78}
{"order_id": "18837", "username": "user", "meal": "hot roll", "price": 78}
{"order_id": "82745", "username": "user", "meal": "hot roll", "price": 78}
```

```
sh-4.4$ kafka-console-consumer --bootstrap-server kafka:9092 --topic payment-event --from-beginning
{"order_id": "18837", "status": "success"}
{"order_id": "82745", "status": "failed", "reason": "Insufficient credit"}
^CProcessed a total of 2 messages
```

5. Kubernetes

A partire dal *docker-compose.yml* utilizzato per il deployment Docker, è possibile utilizzare il tool Kompose per generare dei file *.yml* di partenza. È utile utilizzarli come base di partenza, ma si rende necessario apportare alcune opportune modifiche.

Si riportano di seguito alcuni dettagli relativi agli oggetti utilizzati e alle loro proprietà:

- **Cluster:** consente di configurare un cluster Kind (*Kubernetes IN Docker*), il quale è uno strumento per creare cluster Kubernetes locali.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1beta1
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 80    # Porta del container (nel nodo)
        hostPort: 80        # Porta dell'host (esterna)
        protocol: TCP
      - containerPort: 443  # Porta del container (nel nodo)
        hostPort: 443      # Porta dell'host (esterna)
        protocol: TCP
```

È stato definito, tramite la proprietà *nodes* un singolo nodo con il ruolo di control-plane (master node) che gestisce le operazioni principali del cluster.

Extra Port Mappings consente, invece, di mappare le porte del container al sistema host per consentire l'accesso dall'esterno al cluster.

Ciò risulta necessario per far funzionare correttamente il controller NGINX Ingress che si è deciso di utilizzare all'interno del cluster Kind. Esso richiede che le porte HTTP (80) e HTTPS (443) siano accessibili dal sistema host per gestire il traffico verso i servizi interni al cluster.

In questo modo, vien reso possibile accedere ai servizi esposti tramite l'Ingress.

Difatti, come da specifiche, l'API Gateway (Nginx) utilizzato in docker-compose è stato sostituito in Kubernetes con un oggetto Ingress.

- **Ingress:** consente di configurare regole di routing HTTP/HTTPS per indirizzare il traffico esterno ai servizi interni del cluster

ingressClassName: "nginx": specifica che questo Ingress deve essere gestito dal controller Nginx.

rules → *host* consente di specificare la regola principale; per ogni path specificato al di sotto di *rules*, il traffico viene inoltrato al service backend

appropriato. Ad esempio, per il path “/auth/” il traffico viene indirizzato verso il servizio *auth* sulla porta 5001.

```
rules:
- host: foodies
  http:
    paths:
    - path: /auth/
      pathType: Prefix
      backend:
        service:
          name: auth    # nome del Service
          port:
            number: 5001    # numero della porta del Service
```

- **Deployment:** definisce come distribuire una certa applicazione all'interno di un cluster Kubernetes.

spec → *replicas* consente di definire il numero di pod da deployare

selector → *matchLabels* → *app*: specifica la label nei pod creati

spec → *containers*: contiene proprietà relative ai container eseguiti nei pod

spec → *containers* → *image*: tutte le immagini utilizzate nel progetto vengono prelevate da repository pubblici.

spec → *containers* → *imagePullPolicy: Always*: specifica di prelevare sempre l'immagine quando il container viene avviato o riavviato

spec → *containers* → *ports* → *containerPorts*: specifica la porta del container

spec → *containers* → *resources*: specifica le risorse minime richieste e le risorse massime disponibili

spec → *strategy* → *type: Recreate*: fa in modo che Kubernetes elimini tutti i Pod esistenti prima di crearne di nuovi. Utilizzato per MySQL e MongoDB

spec → *volumes*: definisce i dettagli dei volume associati al pod

spec → *volumes* → *persistentVolumeClaim* → *claimName: mysql-data*: associa il volume a un *PersistentVolumeClaim* (PVC) con un certo nome associato. Quest'ultimo dev'essere creato separatamente e consente di garantire lo storage persistente per i dati del volume

- **Service:** consente la comunicazione tra i Pod o con l'esterno del cluster Kubernetes.

spec → *selector* → *app*: costituisce un selettore di label attraverso cui identificare i Pod a cui il Service deve inoltrare il traffico. Occorre che vi sia una corrispondenza fra questa label e quelle definite nei pod degli oggetti Deployment associati all'oggetto Service considerato

ports: definisce il mapping delle porte tra il Service e i Pod

- **ConfigMap:** è stato utilizzato per gestire configurazioni o parametri che possono essere forniti ai container come variabili di ambiente.

data: contiene le chiavi e i valori che rappresentano i parametri di configurazione (ad esempio, per i database MySQL utilizzati oppure per i microservizi *auth/food/order/payment service*)

- **Persistent volume:** richiede uno spazio di archiviazione persistente nel cluster Kubernetes. Ad esso sono associati i volumi. È stato utilizzato questo tipo di oggetto per entrambi i database.

A tutti gli oggetti utilizzati è stato associato un namespace “foodies” comune (tramite proprietà *namespace*).

6. Black-box and White-box Monitoring

In merito al monitoring basato su Prometheus, in fase di invio della proposta del corrente progetto, era stato specificato quanto segue:

- **White-box** monitoring: numero di richieste ed error 500 in un certo intervallo di tempo, tempo medio di risposta;
- **Black-box** monitoring: monitoraggio delle risorse computazionali (CPU, spazio di archiviazione) utilizzate dagli ambienti di esecuzione.

Purtroppo, per motivi di tempo, non si è riusciti a implementare quest'ultima parte richiesta. Tuttavia, si riportano di seguito alcune considerazioni di carattere generale relative al modo in cui si sarebbe approcciato il problema.

- **White-box** monitoring

Tenuto conto dell'architettura definita per l'applicazione, in particolare dell'uso già fatto di *Apache Kafka* nei microservizi Order Service e Payment Service, avrei introdotto un ulteriore microservizio dedicato al monitoraggio delle performance, con la raccolta delle metriche misurate dai diversi microservizi.

Secondo un approccio di questo tipo, ogni microservizio (Order Service, Payment Service ed eventualmente anche gli altri) raccoglierebbe le metriche relative ai tempi di risposta alle richieste e il numero di errori 500 (*server error*), come da specifica. Queste metriche verrebbero inviate tramite Kafka, utilizzando i producer Kafka (già presenti in Order e Payment Service, da creare negli altri due microservizi) per inviare eventi contenenti le informazioni di monitoraggio.

Ogni microservizio agirebbe come producer Kafka, inviando le metriche al microservizio di monitoring che funge da consumer Kafka. Quest'ultimo si occuperebbe di aggregare i dati provenienti da tutti i microservizi.

I dati acquisiti verrebbero poi inviati a Prometheus per la raccolta, mentre verrebbe utilizzato Grafana per creare dashboard, finalizzate al monitoraggio in tempo reale sia dello stato del sistema sia delle performance dei vari microservizi, evidenziando eventuali anomalie.

Per quanto concerne, invece, la predizione sulle performance future, le metriche raccolte potrebbero essere analizzate tramite il metodo Holt-Winters, basato sull'analisi delle serie temporali: i dati raccolti verrebbero trasformati in time series tramite libreria Pandas, sulle quali sarebbe possibile applicare il modello Holt-Winters, una volta formulato, a partire dalle componenti di media, trend e seasonability.

Un sistema di monitoraggio di questo tipo si integrerebbe facilmente con l'architettura già esistente e consentirebbe di adottare un approccio scalabile e flessibile.

- **Black-box monitoring**

In sintesi, l'idea sarebbe stata quella di sfruttare cAdvisor per acquisire le metriche in tempo reale relativamente alle risorse (CPU, spazio di archiviazione) utilizzate dai container Docker e, a partire da queste, si sarebbe proceduto effettuando una predizione basata sul metodo Holt-Winters, allo stesso modo di come brevemente indicato nella pagina precedente. Si sarebbe cercato un metodo analogo per quanto concerne i pod in Kubernetes.

7. Indicazioni per build & deploy;

Deployment tramite Docker Compose: da terminale, posizionarsi nella cartella del progetto ed eseguire il comando

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

Istruzioni per l'esecuzione

Dopo aver fatto il deployment, allo scopo di semplificare la verifica di ciascuna delle funzionalità implementate, è stato realizzato un client python (con interfaccia da linea di comando) per comunicare con l'applicazione distribuita, mediante REST API. Per utilizzarlo è sufficiente eseguire *main.py*, presente all'interno della cartella "client", ed eseguire le seguenti operazioni (da terminale):

- specificare "1" per effettuare la registrazione oppure "2" per effettuare il login. A tal proposito, si precisa che nell'immagine MySQL utilizzata per il database sono già stati inizializzati due utenti:

username: **admin** password: **qwerty** [admin]

username: **user** password: **qwerty** [user generico]

Link del repository contenente l'immagine MySQL custom utilizzata:

<https://hub.docker.com/r/gabrielevitali/foodies-mysql>

Alternativamente, si può procedere creando un nuovo utente.

Si presti attenzione al fatto che l'utente con cui ci si sta loggando sia o meno un admin: in base a questo potrà accedere a set di funzionalità distinti, essendo stata implementata una gestione delle autorizzazioni basata su token JWT.

- Dopo aver effettuato il login, sarà stato ottenuto un token, utilizzato in automatico per le successive richieste.
- Sia per procedere alla ricerca di un piatto da prenotare ed eventualmente creare un nuovo ordine sia per cancellare un piatto già esistente, occorre che in MongoDB vengano inizializzati (ad esempio, sfruttando Mongo Express) database (*meals_db*), collection (*meals*) e almeno un document.

Si riporta, di seguito, un esempio di document:

```
{
  "name": "hot roll",
  "description": "roll fritti con philadelphia e fragole",
  "category": "sushi",
  "price": 14
}
```

In *meals.json*, sono riportati altri esempi di document da poter inserire in MongoDB.

- Per provare le restanti funzionalità, seguire le indicazioni fornite nel menù principale e nel menù dedicato all'admin.