UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Ingegneria Elettrica Elettronica Informatica

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

DSBD – Relazione finale

Musify

Alessandro Rausa: 1000008707

Docente: Antonella Di Stefano

Anno accademico [2021-2022]

INDICE

INTRODUZIONE

Il progetto sviluppato, denominato Musify, consiste in tre microservizi utili per la gestione di CD e un eventuale loro acquisto da parte degli utenti. L'utente che, dopo aver avuto la possibilità di registrarsi, intende effettuare l'acquisto di un CD potrà visualizzare tutti i CD disponibili e, dopo aver scelto quello di suo interesse, potrà procedere con l'acquisto. In particolare, l'utente potrà, inoltre, annullare un acquisto effettuato in precedenza. Il sistema sarà in grado di verificare vari parametri, quali l'effettiva esistenza del CD e la disponibilità del numero di CD richiesti. Se la verifica andrà a buon fine, il sistema potrà procedere con la registrazione dell'acquisto e confermare se esso sia andato a buon fine.

Il set di progettazione utilizzato consiste in:

- 3 microservizi
- Docker
- Kafka
- Saga
- Kubernetes
- API Gateway
- Whitebox Monitoring (Prometheus)

Come già accennato, è stata utilizzata una logica a microservizi, i quali permettono molti vantaggi come, ad esempio, scalabilità, resilienza, maggiore facilità nella gestione di guasti e possibilità di aggiornamento.

Si utilizza, inoltre, la piattaforma Docker che permette di distribuire facilmente le applicazioni all'interno di un container, per automatizzare il deployment delle varie applicazioni utilizzando un unico sistema operativo. Kubernetes ci permette, invece, in maniera agevole di gestire i container e di implementare un meccanismo di orchestrazione.

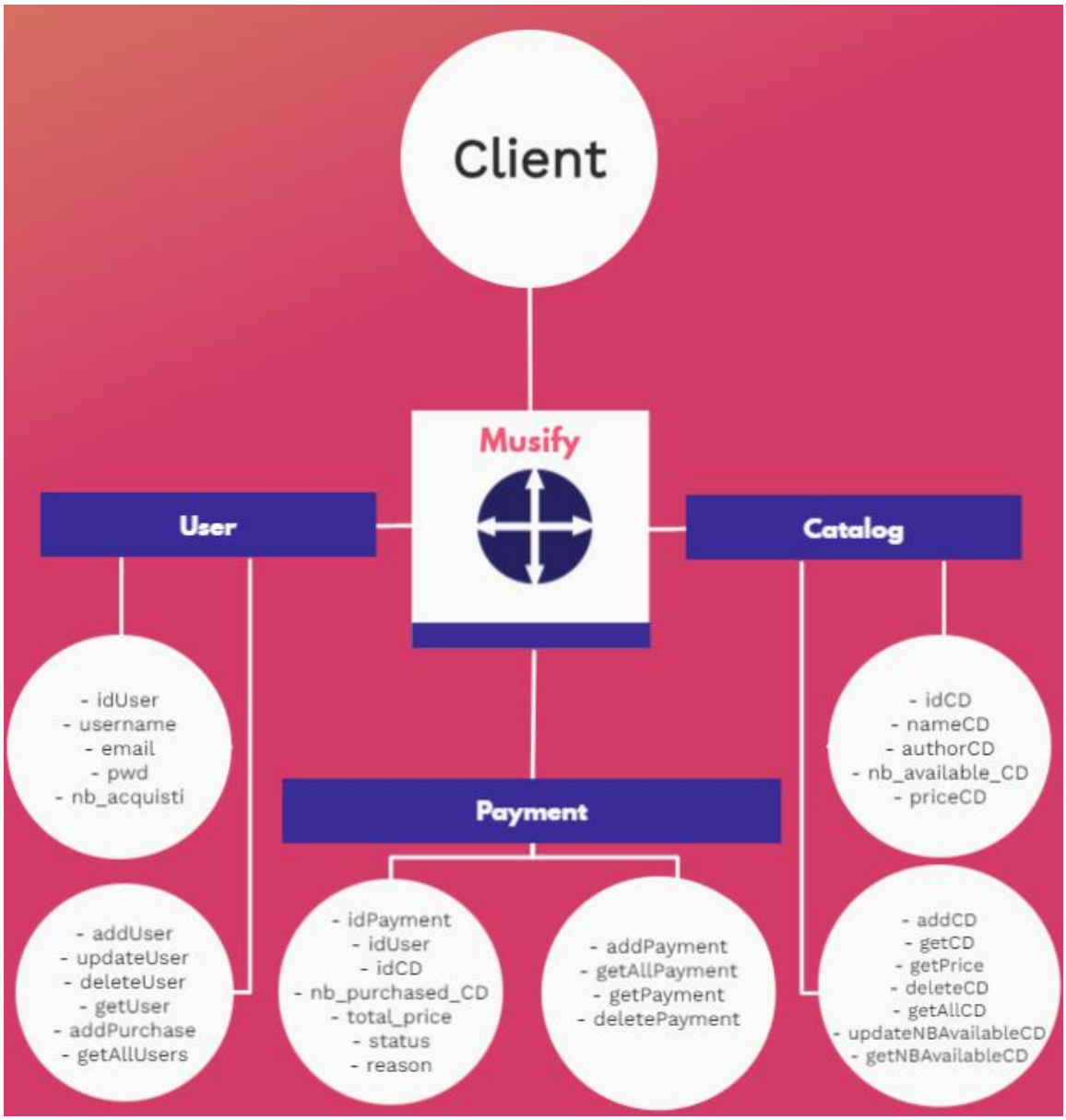
La consistenza dei dati è stata gestita tramite il pattern di progettazione Saga. La comunicazione è stata gestita tramite message broker Kafka. Inoltre, è stato applicato il pattern API Gateway, per gestire le API e aggregare in unico endpoint i vari servizi richiesti.

Infine, è stato utilizzato Prometheus con approccio whitebox monitoring: in particolare, vengono monitorate le metriche relative all'applicazione.

MICROSERVIZI

I microservizi realizzati sono:

- UserManager (gestore degli user)
- CatalogManager (gestore dei CD)
- PaymentManager (gestore dei pagamenti



Il primo microservizio è il gestore degli utenti. Esso offre l'opportunità ad un utente di registrarsi chiedendo informazioni quali username, email, psw. In particolare, l'username e l'email risultano essere univoci. In caso di registrazione con stessa email e/o username, l'applicazione restituirà un avviso di impossibilità di creazione dell'account. All'utente verrà inoltre associato un id e un contatore del totale degli acquisti effettuati.

Tale microservizio offre inoltre la possibilità di visualizzare tutti gli utenti registrati sino al momento della richiesta, la possibilità di aggiornare la password e, infine, la possibilità di eliminare la registrazione. In particolare, si è deciso di non cancellare eventuali pagamenti effettuati da un utente che ha deciso la propria cancellazione in quanto non segue una filosofia degli eCommerce standard.

Il secondo microservizio, il gestore dei CD, offre l'opportunità di registrare un CD all'interno di una collezione salvando informazioni quali il nome, l'autore, il prezzo e le quantità disponibile nella collezione. Registrato un CD viene offerta la possibilità di ricavare il prezzo di un determinato CD o visualizzare l'intera collezione disponibile. Si potrà inoltre aggiornare la quantità di un CD in collezione e, eventualmente, aggiornarne il prezzo. Infine, si offre la possibilità di poter eliminare un CD dalla collezione.

Il terzo microservizio è il gestore dei pagamenti. Tale microservizio offre la possibilità di effettuare un acquisto da parte di un utente per un relativo CD. L'aggiunta di un pagamento comporta la verifica da parte del server di vari parametri quali l'esistenza dell'utente nel gestore degli utenti, l'esistenza del CD nel gestore dei dischi e, durante la stessa chiamata, il controllo se il numero di CD richiesti è minore o uguale alla quantità disponibile nella collezione. In caso di acquisto con successo verrà informato lo UserManager che un dato utente ha effettuato un acquisto; verrà inoltre informato il CatalogManager che un determinato numero di CD è stato acquistato; deve essere, quindi, scalato dal totale dei CD disponibili. Infine, il microservizio offre la possibilità di cancellare un pagamento, informando dunque UserManager e CatalogManager di aggiornare rispettivamente il totale degli acquisti effettuati da un utente e la quantità disponibile in magazzino di un determinato CD.

API REQUESTS

Esempi di API implementate per l'UserManager sono:

- **POST** user/addUser: richiede come body l'inserimento dei parametri che caratterizzano l'utente:
  Es. body:

```
{
    "email": “ale@email.com",
    "username": "ale2",
    "psw": "ale@@”
}
```

  Tale richiesta fallirà in caso della presenza della stessa email o dello stesso username all'interno del database; in questo caso verrà visualizzato un messaggio personalizzato che informerà della già presenza dell'utente.

- **DELETE** user/deleteUser/{username}/{psw}:
  elimina l'utente, restituisce un errore 404 not found nel caso in cui l'utente o non è presente o i dati inseriti risultano essere errati.
- **GET** user/getAllUsers
  Ritorna la lista degli utenti con le relative informazioni. Le password hanno la priorità READ_ONLY che permette solamente di inserirle durante la creazione ma non di visualizzarle con una eventuale GET.

Esempi di API implementate per il CatalogManager sono:

- **POST** catalog/addCD: richiede come body l'inserimento dei parametri che caratterizzano il CD:
  Es. body:

```
{
   "nameCD" :  "Disumano",
   "authorCD" : "Fedez",
   "nbavailableCD" : "6",
   "priceCD" : "18"
}
```

Tale richiesta fallirà in caso della presenza di un CD con lo stesso nome all'interno del database; in questo caso verrà visualizzato un messaggio personalizzato che informerà della già presenza del CD.

- **DELETE** catalog/deleteCD/{name}:
  elimina un CD, restituisce un errore 404 not found nel caso in cui il CD o non è presente o i dati inseriti risultano essere errati.
- **GET** catalog/getAllCD
  Ritorna la lista dei CD con le relative informazioni.
- **GET** catalog/getCD
  Ritorna le informazioni di un determinato CD.

Esempi di API implementate per il PaymentManager sono:

- **POST** payment/addPayment/{idUser}/{nbCD}/{idCD}
  Il pagamento viene aggiunto a prescindere da eventuali controlli con Status posto a Pending. Tale richiesta, con l'utilizzo del message broker kafka, comunica con entrambi gli altri microservizi. La prima comunicazione avviene con lo UserManager che controllerà e risponderà l'eventuale presenza nel database dell'idUser. In caso di fallimento, il PaymentManager aggiornerà lo stato a Failed e la reason a "User not found". La seconda comunicazione avviene con il CatalogManager che controllerà la presenza del CD nel database e l'eventuale disponibilità del numero di CD richiesti. In caso di fallimento verrà posto a Failed lo status del pagamento e la reason a, rispettivamente, "CD does not exists" e "Available CD less then the requested". In caso tutto vada a buon fine, sempre tramite kafka verrà segnalato allo UserManager di incrementare di uno il numero di acquisti per l'utente e al CatalogManager di decrementare il numero di CD disponibili con quelli appena acquistati.
- **DELETE** payment/deletePayment/{idPayment}:
  elimina un pagamento, comunicando allo UserManager e al CatalogManager rispettivamente di decrementare di uno il numero di acquisti per l'utente e di aumentare il numero di CD disponibili con quello salvato dal pagamento eliminato.
- **GET** payment/getAllPayments
  Ritorna la lista dei pagamenti con le relative informazioni.

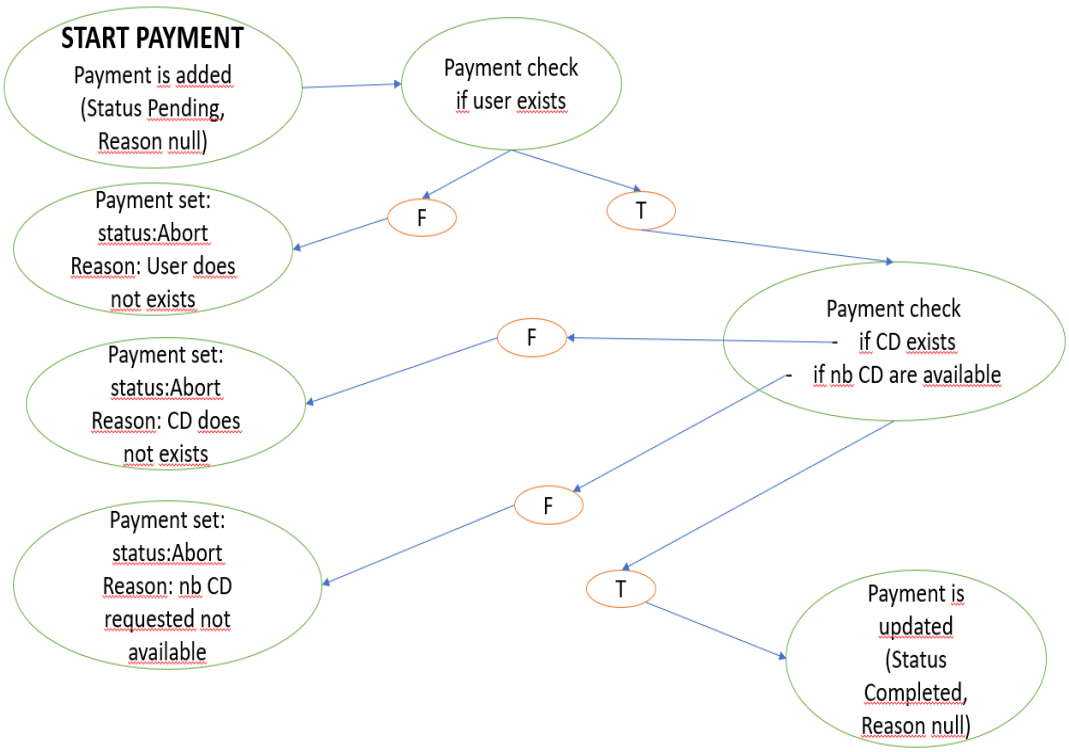
KAFKA E SAGA

Saga è un pattern il cui scopo è garantire l'ordine di esecuzione delle transazioni che fanno parte della stessa saga, ovvero che devono essere eseguite in un ordine prestabilito e il cui effetto non è compromesso da eventuali transazioni intermedie appartenenti a saghe diverse. Questo pattern aiuta quindi a gestire la consistenza dei dati nell'esecuzione di transazioni distribuite tra microservizi diversi; coinvolge diversi attori (servizi) che vanno ad agire su una stessa entità tramite singole transazioni atte all'aggiornamento di un dato comune. L'obiettivo è duplice: mantenere l'integrità della transazione ed effettuare azioni di compensazione per ripristinarla in caso di errore.

Per questo progetto è stato utilizzato un approccio events/choreography: esso prevede che tutti i microservizi coinvolti lavorino innescati da eventi specifici (ascoltando quindi su un kafka topic), come in una vera e propria coreografia. Non esiste infatti, in questo approccio, un controllore che si occupi di amministrare tutto, ma ogni servizio sa esattamente cosa fare e quando farlo.

Con l'utilizzo del message broker kafka, il PaymentManager comunica con entrambi gli altri microservizi.

1. La prima comunicazione avviene con lo UserManager che controllerà e risponderà l'eventuale presenza nel database dell'idUser.
2. In caso di fallimento, il PaymentManager aggiornerà lo stato a Failed e la reason a "User not found".
3. La seconda comunicazione avviene con il CatalogManager che controllerà la presenza del CD nel database e l'eventuale disponibilità del numero di CD richiesti.
4. In caso di fallimento verrà posto a Failed lo status del pagamento e la reason a, rispettivamente, "CD does not exists" e "Available CD less then the requested".
5. In caso tutto vada a buon fine, sempre tramite kafka verrà segnalato allo UserManager di incrementare di uno il numero di acquisti per l'utente e al CatalogManager di decrementare il numero di CD disponibili con quelli appena acquistati e il pagamento verrà aggiornato con status Completed.


START PAYMENT
Payment is added (Status Pending, Reason null)
Payment check if user exists
F
T
Payment set: status:Abort Reason: User does not exists
Payment check if CD exists if nb CD are available
F
Payment set: status:Abort Reason: CD does not exists
F
Payment set: status:Abort Reason: nb CD requested not available
T
Payment is updated (Status Completed, Reason null)

API GATEWAY E INGRESS

Per accedere alle risorse dei microservizi da un dominio comune è stato implementato un file yml con i vari uri da raggiungere.

```yaml
spring:
  application:
    name: api-gateway

  cloud:
    gateway:
      routes:
        - id: usermanager
          uri: http://usermanager:1111
          predicates:
            - Path=/user/**
        - id: catalogmanager
          uri: http://catalogmanager:2222
          predicates:
            - Path=/catalog/**
        - id: paymanager
          uri: http://paymanager:3333
          predicates:
            - Path=/payment/**
```

Su Kubernetes per gestire le API e aggregare i vari servizi è stato utilizzato Ingress. Un Ingress è una regola che definisce come un servizio, presente all'interno di un cluster, possa essere reso disponibile all'esterno del cluster.

```yaml
spec:
  rules:
  - host: musify
    http:
      paths:
      - path: /user
        pathType: ImplementationSpecific
        backend:
          service:
            name: user-service
            port:
              number: 1111
      - path: /catalog
        pathType: ImplementationSpecific
        backend:
          service:
            name: catalog-service
            port:
              number: 2222
      - path: /payment
        pathType: ImplementationSpecific
        backend:
          service:
            name: pay-service
            port:
              number: 3333
```

MONITORING E PREDIZIONI

Per effettuare uno screening delle performance è stato utilizzato un monitoraggio di tipo white box. Un tale tipo di monitoraggio è utilizzato nel caso in cui si ha la piena visibilità dell'infrastruttura. La struttura di tale progetto e la sua implementazione sono note al tester, ciò significa che un monitoraggio di tale tipo può essere effettuato.

Per la suddetta analisi è stato utilizzato Micrometer, un set di librerie che interfaccia l'applicazione e il tool di monitoraggio. In questo progetto sono state generate metriche di tipo counter, che, dopo essere state create sono state estratte da Prometheus.

In particolare, le metriche studiate sono:

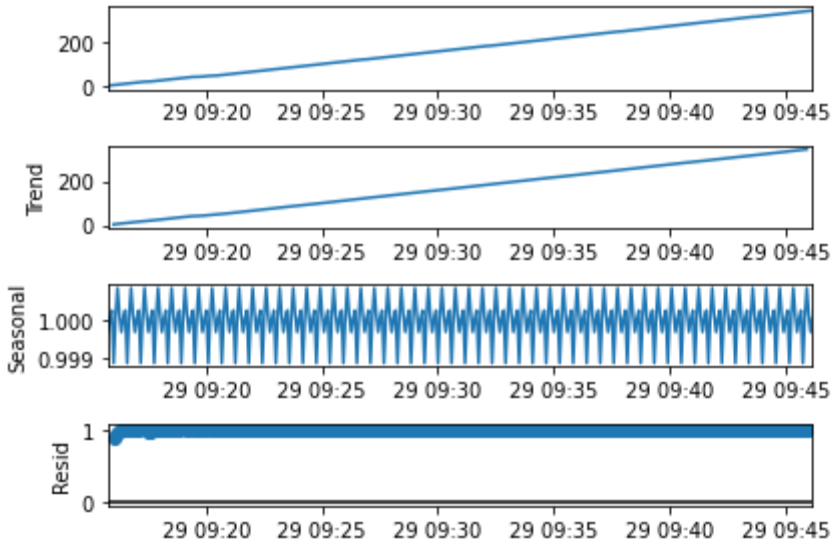
- Contatore di richieste POST al microservizio User
- Contatore di richieste POST al microservizio Catalog
- Contatore di richieste POST al microservizio Payment

I dati sono stati estratti da Prometheus tramite script Python e memorizzati in dei file CSV.

Successivamente questi dati sono stati caricati su Jupyter e analizzati inserendoli all'interno di DataFrame, che sono stati poi ricampionati (resample a 0.16T) e indicizzati per ottenere delle TimeSeries. Queste ultime sono poi state analizzate tramite la funzione seasonal_decompose, e, dopo aver suddiviso i dati tra dati di training e dati di test, abbiamo potuto studiare il trend, la seasonal e il Resid.

La scelta del parametro model va fatta sulla base dell'osservazione del grafico della serie. Generalmente, quando la serie ha un andamento esponenziale o logaritmico il parametro model va settato a 'mul'. Fatto ciò, risulta opportuno tracciare un grafico del risultato e verificare che l'andamento del trend sia corretto e la media dell'errore sia circa uguale a 1 (o uguale a 0 nel caso in cui viene settato il model con il valore 'add').

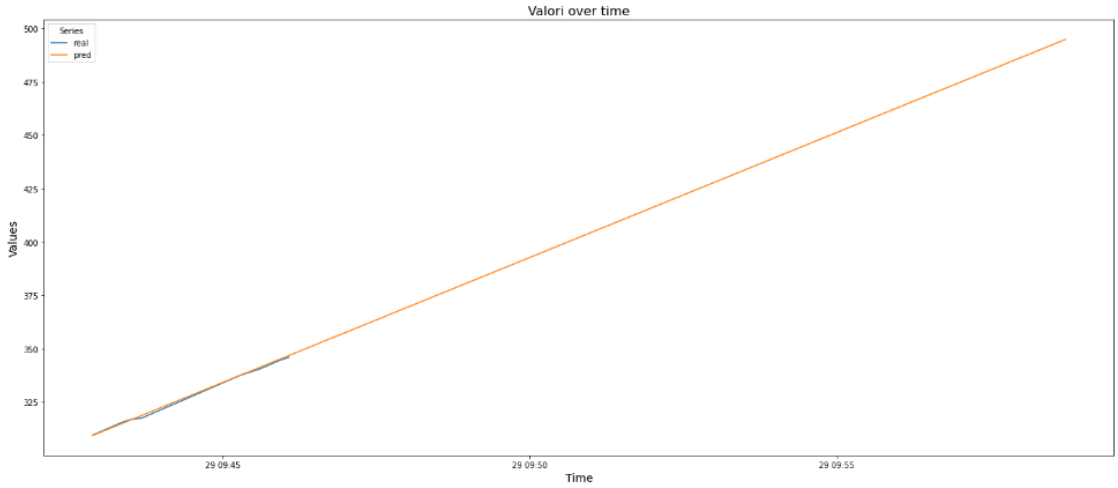
Nel progetto è stato scelto di usare model = "mul" e il risultato ottenuto è il seguente (es. UserManager):



Si nota che il residuo, come ci si aspetta, è pari a 1.

Infine, è stata effettuata una predizione; in questo progetto è stato scelto di usare una predizione di tipo Holt-Winters.

Il risultato di tale predizione è il seguente (es. UserManager):



Suppongo, infine, l'andamento non sia del tutto veritiero in quanto la porzione di dati su cui ci si è basati è esigua.