Progetto DSBD 2020/2021

Sviluppo di un sistema "e-commerce" distribuito.

Note generali

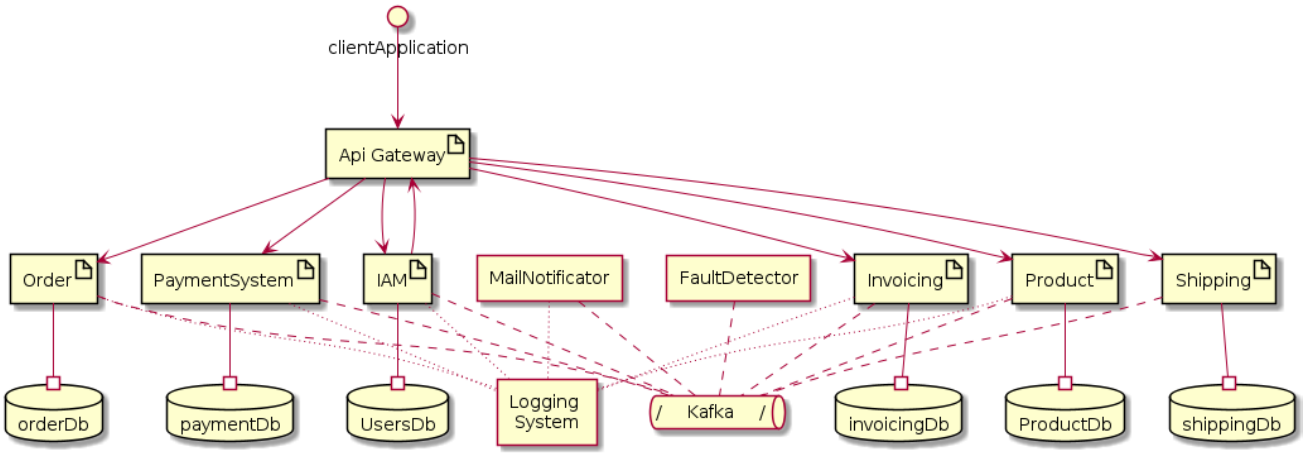
- Tutti i progetti vanno sviluppati usando Git (e una piattaforma pubblica a scelta tra Github e Gitlab)
- Ogni componente dei progetti va sviluppato in un container docker separato
- Sviluppare le interfacce REST e le altre specifiche fornite di seguito rimanendo piu' fedeli possibile a quanto richiesto
- A tutti i progetti va allegata una relazione dettagliata delle scelte implementative e progettuali: puo' essere fornita in pdf o in markdown (README.MD nel repository git)
- Ognuno dei progetti e' disponibile in 4 varianti (A meno dei progetti 5 - Fault detector, e 6 - Notifiche email):
  - Variante A: sviluppo con Java Spring MVC, JPA e MySql
  - Variante B: sviluppo con Java Spring MVC, Spring Data MongoDB e MongoDB
  - Variante C [Opzionale]: sviluppo con Java Spring WebFlux, Spring Data Reactive MongoDB e MongoDB
  - Variante D [Opzionale]: sviluppo con Java Spring WebFlux, Spring RDBC2 Reactive e MySql
- Ogni coppia (nProgetto, Variante) puo' essere sviluppata al piu' da due gruppi/studenti
- Ogni progetto (nProgetto, variante esclusa) va sviluppato da almeno un gruppo
- Ogni gruppo scelga una strategia di health-check, come da successivo paragrafo: ogni strategia di ping deve essere sviluppata in almeno il 30% dei progetti

Ricevimento:

Si puo' richiedere un ricevimento online scrivendo a:

- antdis@gmail.com
- a.dimaria@studium.unict.it
- alessandro.distefano@phd.unict.it

Artchitettura generica di riferimento



Kafka

Nei progetti si dovra' fare uso di Kafka come sistema di messaging asincrono.

Il deploy di Kafka puo' essere ottenuto attraverso i due seguenti services da aggiungere al docker-compose.yml:

```
x-xxx-common-services-config: &common-services-config
  restart: always

x-kafka-env: &kafka-env
  KAFKA_BROKER_ID: 1
  KAFKA_ADVERTISED_PORT: 9092
  BROKER_ID_COMMAND: "hostname | cut -d'-' -f2"
  KAFKA_ZOOKEEPER_CONNECT: "zoo:2181"
  KAFKA_CREATE_TOPICS: "logging:20:1,pushnotifications:10:1,invoicing:10:1,mailing:10:1,userupdates:10:1,orderupdates:10:1"
  KAFKA_LISTENERS: "PLAINTEXT://:9092"

services:
  zoo:
    <<: *common-services-config
    image: library/zookeeper:3.4.13
    environment:
      ZOO_MY_ID: 1

  kafka:
    <<: *common-services-config
    environment: *kafka-env
    image: wurstmeister/kafka:2.11-2.0.0
```

Fake producer

Per necessita' di debug e testing sara' necessario avere a disposizione un producer Kafka fake.

Di seguito uno snippet che lo implementa in python facendo uso della python interactive shell bpython.

```
# File/Module name: kafka_producer_interactive.py
# Install dependencies with: pip3 install kafka-python [bpython (optional)]
from kafka import KafkaProducer
import json
"""
Just a KafkaProducer for interactive testing purposes

Usage example:

    /code # bpython
        bpython version 0.20.1 on top of Python 3.9.0 /usr/local/bin/python
        >>> import kafka_producer_interactive as kpi
        >>> send = kpi.producer(topic='mailing')
        >>> send('myKey', value={'myKeyStr': 'asd'})
"""


def producer(broker='kafka:9092', topic='my-topic'):
    p = KafkaProducer(bootstrap_servers=[broker],
                              value_serializer=lambda x: json.dumps(x).encode('utf-8'))

    def send_and_flush(key: str, value: dict, *args, **kwargs):
        p.send(topic, key=key.encode('utf-8'), value=value, *args, **kwargs)
        p.flush()

    return send_and_flush
```

Fake consumer

Anolagamente, per leggere i messaggi ricevuti su Kafka in uno specifico topic si puo' usare uno script gia' disponibile all'interno del container Kafka usato nel docker-compose sopra:

```
kafka-console-consumer.sh --topic my-topic --bootstrap-server localhost:9092

## Help
kafka-console-consumer.sh --help
```

Further information

https://kafka.apache.org/quickstart

Specifiche comuni a tutti i progetti

Ping endpoints

Ogni micro-servizio dovra' sviluppare una tra le seguenti strategie di ping.

Ping endpoint 1 (ping-ack mode)

Ogni micro-servizio espone un endpoint `GET /ping` che risponde con:

```
{
    "serviceStatus": "up|down",
    "dbStatus": "up|down"
}
```

Le due key, rappresentano, rispettivamente, uno stato generale del servizio e lo stato di connessione al database.

Rispondere, per semplicita', sempre con up.

Ping endpoint 2 (heart-beat mode)

Ogni micro-servizio periodicamente (periodo definita con variabile d'ambiente) esegue una richiesta `POST /ping` con body:

```
{
    "service": "serviceName",
    "serviceStatus": "up|down",
    "dbStatus": "up|down"
}
```

Le due key, rappresentano, rispettivamente, uno stato generale del servizio e lo stato di connessione al database. La key service, invece, riporta il nome del micro-service.

Messaggi di errore

Ogni micro servizio, al fallimento di una richiesta HTTP, deve inviare il seguente messaggio sul topic `logging` :

```
key = http_errors
value = {
    timestamp: UnixTimestamp,
    sourceIp: sourceIp
    service: products,
    request: path + method
    error: error
}
```

Se l'errore e' di tipo 50x, error e' una la stringa con lo stack trace della exception generata.

Se l'errore e' di tipo 40x, error e' lo status code in formato numerico.

SourceIp e' l'indirizzo IP del client da cui proviene la richiesta.

UserID: dove prenderli

Dove specificato che si utilizza uno userId nella richiesta, e.g., POST /orders, si consideri che questo venga fornito come un id numerico tramite l'Header HTTP "X-User-ID".

```
curl -H "X-User-ID: 0" http://myservice:8080/my-fake-authenticated-request
```

Lo userId 0 rappresentera' un admin, ogni altro userId, invece, rappresentera' un generico utente.

Progetto 1: Gestione prodotti

Sviluppare un micro-servizio per la gestione dei prodotti.

Dovra' offrire le seguenti API:

1. `POST /products` : Aggiunge un prodotto al DB
2. `PUT /products/{id}` : Modifica generica di un prodotto

3. `GET /products/{id}` : Risponde con la rappresentazione Json del prodotto con id `{id}`
4. `GET /products/` : Risponde con la rappresentazione Json dell'array dei prodotti. Gestire anche la "paginazione". Ad esempio aggiungendo due parametri "per_page" e "page" cosi che si risponda con "per_page" prodotti a partire da quello con `id = per_page * (page - 1)`. `GET /products?per_page_10&page=2` indica la pag. 2 dei prodotti a 10 a 10, ovvero l'array dei prodotti con id da 10 a 20. Hint: su Spring potrebbero essere gia' disponibili delle 'utilities' per gestire correttamente la paginazione (google keywords: "Spring Rest Pagination")
5. Le stesse CRUD call per la gestione delle categorie su `/categories`

Gestire gli errori HTTP in caso di richieste errate, i.e., 404 se il prodotto in GET non esiste, 400 se qualcuno dei dati nei body o nel path delle richieste non e' valido.

Inoltre il sistema dovra' essere produttore e consumatore dei seguenti messaggi su Kafka:

1. Consumatore su topic `orders` del messaggio "order_completed":

```
key = order_completed
value = {
    orderId: id,
    products: [ { product_id: quantity }, "..." ],
    total: amount,
    shippingAddress: {...},
    billingAddress: {...},
    userId: :id
    extraArgs: {.... // Ignorare }
}
```

alla ricezione del messaggio esegue:

1. la verifica di disponibilita' (quantita') dei prodotti). Se tutti sono disponibili nelle quantita' richieste, vengono ridotte le quantita' dei prodotti.
2. la verifica del totale pagato con la somma dei prezzi dei prodotti (per la quantita' di ognuno)

Alla fine della validazione, il micro-servizio produce il messaggio sui topic `orders` e `notifications` :

```
key = order_validation
value = {
    timestamp: UnixTimestamp
    status: status_code,
    orderId: id
    extraArgs: {} // Vedi sotto
}
```

Lo status_code puo' essere:

- `0` : validazione ok (quantita' dei prodotti verificata e totale corretto).
- `-1` : qualche prodotto non e' disponibile, aggiungere agli extraArgs una lista dei prodotti non disponibili `extraArgs: { products: [ {id: q}, ... ]}`
- `-2` : totale errato
- `-3` : entrambe le verifiche sono fallite

Se lo status code e' -2 o -3, il messaggio va inviato anche sul topic `logging`

Progetto 2: Gestione ordini

Sviluppare un micro-servizio per la gestione degli ordini.

Dovra' offrire le seguenti API:

1. `POST /orders` : Aggiunge un ordine al DB (deve includere la lista dei prodotti, l'indirizzo di spedizione , l'indirizzo di fatturazione e lo user id preso dall'header X-User-ID);
2. `GET /orders/{id}` : Risponde con la rappresentazione Json dell'ordine con id `{id}` se lo user id dell'ordine e' uguale a quello fornito o e' 0. Altrimenti 404 (se l'ordine non esiste o non e' associato allo user id fornito)
3. `GET /orders/` : Risponde con la rappresentazione Json dell'array degli ordini associati allo user id fornito o di tutti gli utenti se lo user id e' 0. Gestire anche la "paginazione". Ad esempio aggiungendo due parametri "per_page" e "page" cosi che si risponda con "per_page" ordini a partire da quello con `id = per_page * (page - 1)`. `GET /orders?per_page_10&page=2` indica la pag. 2 degli ordini a 10 a 10, ovvero l'array degli ordini con id da 10 a 20. Hint: su Spring potrebbero essere gia' disponibili delle 'utilities' per gestire correttamente la paginazione (google keywords: "Spring Rest Pagination")

Inoltre il sistema dovra' essere produttore e consumatore dei seguenti messaggi su Kafka:

1. Se la post di cui sopra ha successo si invia sui topic `orders` e `notifications` i messaggi:

```
key = order_completed
value = {
    orderId: id,
    products: [ { product_id: quantity }, "..." ],
    total: amount,
    shippingAddress: {...},
    billingAddress: {...},
    userId: :id
    extraArgs: {// Ignorare }
}
```

2. Consumatore del seguente messaggio sul topic `orders` :

```
    key = order_validation
    value = {
        timestamp: UnixTimestamp
        status: status_code,
        orderId: id
        extraArgs: {// Ignorare }
}
```

Se lo status code ricevuto non e' 0, si imposta l'ordine nello stato `Abort` .

2. Consumatore del seguente messaggio sul topic `orders` :

```
key = order_paid
value = {
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
    extraArgs: {...} // Vedi sotto
}
```

Si verifichi che la tripla orderId, userId, amountPaid esista nel database. Se esiste, lo stato dell'ordine viene impostato su `Paid` e si inoltra lo stesso messaggio sui topic `notifications` , `invoicing` . Se non esiste, lo stato dell'ordine viene impostato su `Abort` , si inoltra lo stesso messaggio sul topic `logging` , con key `order_paid_validation_failure` e, tra gli extraArgs, si aggiunge:

1. Ordine non esistente: `extraArgs: {error: "ORDER_NOT_FOUND"}` ;
2. Amount errato: `extraArgs: {error: "WRONG_AMOUNT_PAID"}` .

Progetto 3: Gestione pagamenti

Espone un endpoint HTTP che simula la notifica di pagamento avvenuto con sistema Paypal:

- `POST /ipn`

Espone inoltre un endpoint che permette di fare retrieve delle transazioni effettuate:

- `GET /transactions?fromTimestamp=unixTimestamp1&endTimestamp=unixTimestamp2`

Ritorna tutte le transazioni effettuate nell'intervallo di tempo compreso tra `fromTimestamp` e `endTimestamp` solo se lo user id fornito come a inizio documento tramite header e' uguale a 0

1. Verifica validita' della richiesta (che venga da paypal e che rispecchi la transazione): aggiungere solo un finto "if" per la fase di sviluppo che porta sempre al successo del check. Vedi follow-up sotto.

2. [Punto precedente e' ok] Si verifica che il campo "business" del body della richiesta riporta una email uguale a quella disponibile in una variabile d'ambiente (ad esempio MY_PAYPAL_ACCOUNT="mybusiness@mycompany.tld")

si invia su Kafka, nel topic `orders` il messaggio:

```
key = order_paid
value = {
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
```

```
    extraArgs: {...} // ignorare
}
```

1. Si risponde con 200, comunque sia andato il processo e se ne salva su Db quanto possa risultare utile: e.g., tempo, messaggio che si invia su kafka, contenuto della IPN...

Casi di errore

Si inviano su `logging` tutti i parametri e il body della richiesta ricevuta:

```
    key = order_paid_failure
    value = {
        timestamp: UnixTimestamp
        request params and body  {dict-like/map<string,string>}
    }
```

Se si fallisce al punto (1), usare la key="bad_ipn_error".

Se si fallisce al punto (2), usare la key="received_wrong_business_paypal_payment"

Esempio di body della richiesta ricevuta dal broker di pagamento 3rd party (i.e., Paypal)

```
{
    "invoice": // Corrisponde all'order id
    "item_id": // corrisponde allo user id
    "mc_gross": // Ammontare pagato
    "business": // Email dell'account su cui e' stato ricevuto il pagamento
    ... Possibili altri field da ignorare ...
}
```

Follow-up: full-integration con paypal [opzionale]

Il punto 1 precedente prevede che all'arrivo della richiesta se ne faccia un'altra sincrona alle API di Paypal.

Per poter integrarsi in modalita' test con Paypal e' necessario creare un account Sandbox e utilizzare l'IPN Simulator (vedi link sotto).

L'IPN Simulator inviera' una richiesta alla URI del micro-servizio.

Il microservizio al punto 1, dovra' inviare, a sua volta, una richiesta POST con content type `application/x-www-form-urlencoded` all'indirizzo `https://ipnpb.sandbox.paypal.com/cgi-bin/webscr` che includa, come parametri, tutti quelli inclusi nella richiesta ricevuta (attenzione, la richiesta con content-type application/x-www-form-urlencoded). Come ultimo parametro aggiungere `&cmd=_notify-validate` .

Se la risposta contiene il testo `VERFIFIED` la verifica ha successo, altrimenti no.

Esempio di verifica in "pseudo" python

```python
class IpnView(APIView):
    # @PostMapping("/ipn")
    def post(self, request, format=None):
        try:
            self.__serve_ipn(request.POST)
        except Exception as e:
            logging.error(e)
        return Response()

    def __serve_ipn(self, data):
        response = self._verify_request(data)
        if not response == 'VERIFIED':
            send_to_kafka("logging", $$$bad_ipn_error$$$)
            return
        if not data.get("business") == os.environ.get("MY_PAYPAL_ACCOUNT")
            send_to_kafka("logging", $$$received_wrong_business_account_payment$$$)
            return
        send_to_kafka("orders", $$$order_paid$$$)

    def _verify_request(self, data):
        params = urlencode(data) + '&cmd=_notify-validate'
        headers = {'content-type': 'application/x-www-form-urlencoded'}
        r = requests.post(settings.PAYPAL_VERIFY_URL, params=params, headers=headers, verify=True)
        r.raise_for_status()
```

```
        return r.text
```

Link utili

- https://developer.paypal.com/docs/api-basics/notifications/ipn/IPNandPDTVariables/
- https://developer.paypal.com/developer/ipnsimulator/
- https://www.paypal.com/gp/smarthelp/article/how-do-i-add-paypal-checkout-to-my-custom-shopping-cart-ts1200

Progetto 4: Gestione fatturazione

Sviluppare un micro-servizio per la gestione delle fatture.

Dovra' offrire le seguenti API:

1. `GET /invoices/{id}` : Risponde con la rappresentazione Json della fattura con id `{id}` se lo user id della fattura e' uguale a quello fornito o e' 0. Altrimenti 404 (se la fattura non esiste o non e' associata allo user id fornito)
2. `GET /invoices/` : Risponde con la rappresentazione Json dell'array delle fatture associate allo user id fornito o di tutti gli utenti se lo user id e' 0. Gestire anche la "paginazione". Ad esempio aggiungendo due parametri "per_page" e "page" cosi che si risponda con "per_page" fatture a partire da quello con `id = per_page * (page - 1)`. `GET /invoices?per_page_10&page=2` indica la pag. 2 delle fatture a 10 a 10, ovvero l'array delle fatture con id da 10 a 20. Hint: su Spring potrebbero essere gia' disponibili delle 'utilities' per gestire correttamente la paginazione (google keywords: "Spring Rest Pagination")

Inoltre il sistema dovra' essere produttore e consumatore dei seguenti messaggi su Kafka:

- Sottoscrittore del seguente messaggio sul topic `orders` :

```
key = order_completed
value = {
    orderId: id,
    products: [ { product_id: quantity }, "..." ],
    total: amount,
    shippingAddress: {...},
    billingAddress: {...},
    userId: :id
    extraArgs: {.... // Ignorare }
}
```

Alla ricezione genera una fattura in DB che riporti order_id, billing address, prodotti (solo id/quantita') e totale.

- Consumatore del seguente messaggio sul topic `orders` :

```
key = order_validation
value = {
    timestamp: UnixTimestamp
    status: status_code,
    orderId: id
    extraArgs: {...} // ignorare
}
```

Se lo status code ricevuto non e' 0, si imposta la fattura nello stato `Abort` .

- Consumatore del seguente messaggio sul topic `invoicing` (attenzione, non nel topic `orders` , se necessario vedi progetto gestione ordini):

```
key = order_paid
value = {
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
    extraArgs: ... // vedi sotto
}
```

Cercare la fattura associata a (orderId, userId, amountPaid). Se esiste, lo stato della fattura viene impostato su `Paid` e si genera un numero fattura incrementale (non e' la eventuale primary key in db).

Il numero fattura deve ricominciare ad ogni 1 gennaio. Quindi ad ogni generazione: si prende il max dei numeri tra le fatture dell'anno (se nessuna fattura e' ancora stata emessa sara' 0) e si incrementa di uno.

Se non esiste la fattura si invia nel topic `logging` lo stesso messaggio con key `invoice_unavailable` .

In questo caso, aggiungere il timestamp unix nel value: `value = {..., timestamp: unixTimestamp,....}`.

Progetto 5: Ping-Ack Fault detector

Scrivere un micro-servizio che periodicamente (es, 30s, usare env var per gestire il periodo) esegua la richiesta GET /ping su una lista di host.

Varianti

- Variante A: usare Spring MVC
- Variante B [Opzionale]: usare WebFlux

Specifiche

Esempio host:

```
["orders", "products", "invoicing", ...]
```

Scrivere un micro-servizio separato che sappia rispondere a GET /ping genericamente come da paragrafo 'Specifiche comuni'.

Ovvero, un servizio che alla GET /ping risponda con:

```
{
    "serviceStatus": "up|down",
    "dbStatus": "up|down"
}
```

Scegliere up o down in maniera casuale: up scelto se `rand.uniform(0, 1) < 0.7 (70 % dei casi)`.

Se la risposta da uno degli host riporta almeno un "down", mandare su Kafka, nel topic `logging` la risposta ricevuta con il messaggio:

```
key: service_down
value: {
    time: UnixTimeStamp
    status: JsonResponse
    service: hostname
}
```

Se il service non e' proprio raggiungibile lo status diventa:

```
{
    "serverUnavailable": // messaggio di errore come "Connection refused",
      // "No route to host", "Timeout"...
}
```

Nota bene: la lista degli host deve essere configurata da variabile d'ambiente.

Progetto 6: Heart-Beat fault detector

Varianti

- Variante A: usare Spring MVC
- Variante B [Opzionale]: usare WebFlux

Specifiche

Scrivere un micro-servizio che espone un endpoint POST /ping.

Il micro-servizio deve tenere in memoria una map<Host, Object>.

- Host rappresenta l'hostname del servizio che chiama l'endpoint
- L'Object associato al generico Host e' la rappresentazione Json dell'ultimo body ricevuto servendo la richiesta POST /ping, con l'aggiunta del timestamp unix in cui la richiesta e' ricevuta.
- Se uno tra i field `serviceStatus` e `dbStatus` e' diverso da "up" (i.e., e' down), il service manda sul topic `logging` il messaggio:

```
key: service_down
value: {
    time: UnixTimeStamp
    status: {...}
    service: hostname
```

```
}
```

dove `status` e' il body stesso in formato Json.

Periodicamente, il micro-service, itera sulla map e verifica che

```
time.Now().Unix() - value.Timestamp < $PERIOD
```

dove $TPERIOD rappresenta una threshold (env var) in secondi oltre la quale il service si ritiene unavailable.

Se la verifica fallisce, si invia sul topic `logging` il messaggio:

```
key: service_down
value: {
    time: UnixTimeStamp
    status: { serverUnavailable: 'No heart-beat received' }
    service: hostname
}
```

Body della richiesta

Il body, fornito in formato Json, riportera':

```
{
 "serviceName": "hostname",
 "serviceStatus": "up|down",
 "dbStatus": "up|down"
}
```

Progetto 7: Notifiche Mail

Scrivere un micro-servizio che sia sottoscrittore della coda `notifications` e in funzione dei messaggi ricevuti crei una notifica da inviare via mail.

Varianti

- Variante A: utilizzare Spring MVC e un server SMTP com gmail
- Variante B: utilizzare Spring MVC e un Api di terze parti, e.g., sendgrid.com
- Variante C [Opzionale]: utilizzare WebFlux e un API di terze parti, e.g., sendgrid.com
- Variante D [Opzionale]: utilizzare WebFlux un server SMTP come gmail

Specifiche

Nel seguito si utilizzeranno messaggi dei micro-servizi precedenti, che non sono disponibili durante lo sviluppo.

Saranno, pero', necessarie, altre informazioni, dell'utente, degli ordini ecc... Queste potrebbero essere richieste ad un altro micro-servizio. Per lo scopo del progetto usare il pattern Proxy per nascondere la non-disponibilita' dei micro-servizi Users e Orders, i.e., UsersProxy e OrdersProxy. Usare quindi i loro metodi per richiedere le informazioni necessarie per la mail (E.g., Nome dell'utente, email e lista prodotti). Nota: nel possibile servizio completo, si aggiungerebbe una implementazione delle interfacce UsersProxy e OrdersProxy che esegua realmente le richieste HTTP.

I messaggi ricevuti possono essere:

1.

```
key = order_validation
value = {
    status: status_code,
    orderId: id
    extraArgs: {...} // vedi sotto
}
```

lo status_code puo' essere:

- 0: validazione ok (quantita' dei prodotti verificata e totale corretto).
- -1: qualche prodotto non e' disponibile, gli id di questi prodotti sono disponibili su `['extraArgs']['products']`
- -2: totale errato
- -3: entrambe le verifiche sono fallite

In funzione dello status code, generare una mail da inviare ad un utente.

2.

```
    key = order_completed
    value = {
    orderId: id,
    products: [ { product_id: quantity }, "..." ],
    total: amount,
    shippingAddress: "...",
    billingAddress: "...",
    userId: "..."
    extraArgs: {.... // Ignorare }
    }
```

Inviare una mail che segnali di 'aver ricevuto l'ordine'.

3.

```
key = order_paid
value = {
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
    extraArgs: ... // vedi sotto
}
```

Inviare un mail che segnali di 'aver ricevuto il pagamento'.

Progetto 8: Logging System

Scrivere un micro-servizio che sia sottoscrittore kafka del topic `logging`.

Alla ricezione di ognuno dei possibili messaggi, salva sul db il loro contenuto. Trovare un modello dati comune per i messaggi. Nota: se non e' disponibile un timestamp aggiungerlo con quello attuale (e.g., `time.Now().Unix()`)

I log sono disponibili via endpoints HTTP:

- `GET /keys/{key}?from=unixTimestampStart&end=unixTimestampEnd` retrieve di tutti i log message con key = {key} che abbiano tempo compreso fra i due forniti nei parametri della GET (obbligatori)
- `GET /http_errors/services/{service}?from=unixTimestampStart&end=unixTimestampEnd` : retrieve di tutti i log message di tipo http_errors associati al service {service}
- `GET /uptime/services/{service}?from=unixTimeStampStart&end=unixTimestampEnd` risponde con informazioni sull'availability del service {service}. Ovvero, ricerca tutti i messaggi service_down associati al service {service} li considera validi per 30 secondi, li conta, raggrouppati per status e ritorn un json con le availability: `(specific_status_count * 30 / deltaTime)`

Riepilogo messaggi di logging

Genericamente da tutti i micro-servizi

```
    key = http_errors
    value = {
        timestamp: UnixTimestamp,
        sourceIp: sourceIp
        service: products,
        request: path + method
        error: error
    }
```

Da un micro-service di nome 'products'

```
    key = order_validation
    value = {
        timestamp: UnixTimestamp
        status: status_code,
        orderId: id
        extraArgs: {...}
```

Da un microservice di nome 'orders'

```
key = order_paid_validation_failure
value = {
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
    extraArgs: {
        error: "ORDER_NOT_FOUND"
        error(alternativamente):"WRONG_AMOUNT_PAID"
    }
}
```

Da un micro-service di nome 'payment'

```
    key="bad_ipn_error" oppure
    key="received_wrong_business_paypal_payment"
    value = {
        timestamp: UnixTimestamp
        request params and body {dict-like/map<string,string>}
    }
```

Da un micro-servizio di nome 'invoicing'

```
key = invoice_unavailable
value = {
    timestamp: UnixTimestamp
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
    extraArgs: {...}
}
```

Da un micro-servizio di nome 'shipping'

```
key = shipping_unavailable
value = {
    timestamp: UnixTimestamp
    orderId: orderId,
    userId: userId,
    amountPaid: amountPaid
    extraArgs: {...}
}
```

Da un micro-servizio di nome 'fault-detector'

```
key: service_down
value: {
    time: UnixTimeStamp
    status: JsonResponse
    service: hostname
}
```

status

Gli status possono essere:

```
{
    "serverUnavailable": // messaggio di errore come "Connection refused",
      // "No route to host", "Timeout"...
}
```

o

```
{
    "serviceStatus": "down",
    "dbStatus": "down"
}
```

l'aggregazione da realizzare dovra' ottenere un oggetto dict-like, come:

```
{
    "serviceDown": 36, // => 36 * 30 secondi
    "dbStatus": 80, // => 80 * 30 secondi
    "serverUnavailable": 100 // => 100 * 30 secondi
}
```

Quindi le availability restituite saranno del tipo:

```
{
  "serviceAvailability": 1 - 36 * 30 / 86400
  "serverAvailability":  1 - 100 * 30  / 86400,
  "dbAvailability":  1 - 80 * 30 / 86400,
}
```

Dove 86400 rappresenta un delta di tempo come esempio, i.e, usando una richiesta che abbia start ed end timestamp distanti esattamente di un giorno (86400 secondi)

Progetto 9: Shipping system

Sviluppare un micro-servizio per la gestione delle spedizioni.

Dovra' offrire le seguenti API:

1. `GET /shipping/{id}` : Risponde con la rappresentazione Json della spedizione con id `{id}` se lo user id della spedizione e' uguale a quello fornito o e' 0. Altrimenti 404 (se la spedizione non esiste o non e' associata allo user id fornito)
2. `GET /shippings/` : Risponde con la rappresentazione Json dell'array delle spedizioni associate allo user id fornito o di tutti gli utenti se lo user id e' 0. Gestire anche la "paginazione". Ad esempio aggiungendo due parametri "per_page" e "page" cosi che si risponda con "per_page" fatture a partire da quello con `id = per_page * (page - 1)`. `GET /shippings?per_page_10&page=2` indica la pag. 2 delle fatture a 10 a 10, ovvero l'array delle spedizioni con id da 10 a 20. Hint: su Spring potrebbero essere gia' disponibili delle 'utilities' per gestire correttamente la paginazione (google keywords: "Spring Rest Pagination")

Inoltre il sistema dovra' essere produttore e consumatore dei seguenti messaggi su Kafka:

- Sottoscrittore del seguente messaggio sul topic `orders` :

```
key = order_completed
value = {
    orderId: id,
    products: [ { product_id: quantity }, "..." ],
    total: amount,
    shippingAddress: {...},
    billingAddress: {...},
    userId: :id
    extraArgs: {.... // Ignorare }
}
```

Alla ricezione genera una spedizione in DB che riporti order_id, shpping address, prodotti (solo id/quantita').

- Consumatore del seguente messaggio sul topic `orders` :

```
key = order_validation
value = {
    timestamp: UnixTimestamp
    status: status_code,
    orderId: id
    extraArgs: {...} // ignorare
}
```

Se lo status code ricevuto non e' 0, si imposta la spedizione nello stato `Abort` .

- Consumatore del seguente messaggio sul topic `invoicing` (attenzione, non nel topic `orders` , se necessario vedi progetto gestione ordini):

```
key = order_paid
value = {
    orderId: orderId,
    userId: userId,
```

```
    amountPaid: amountPaid
    extraArgs: ... // vedi sotto
}
```

Cercare la spedizione associata a (orderId, userId). Se esiste, lo stato della spedizione viene impostato su `TODO` e si genera un numero DDT incrementale (non e' la eventuale primary key in db).

Se non esiste la spedizione si invia nel topic `logging` lo stesso messaggio con key `shipping_unavailable`.

In questo caso, aggiungere il timestamp unix nel value: `value = {..., timestamp: unixTimestamp,....}`.