Progetto DSBD

[3A]

Prof. Antonella Di Stefano

Prof. Alessandro Di Stefano

Prof. Andrea Di Maria

Anno Accademico: 2020/2021

Corso di Laurea Magistrale – Ingegneria Informatica

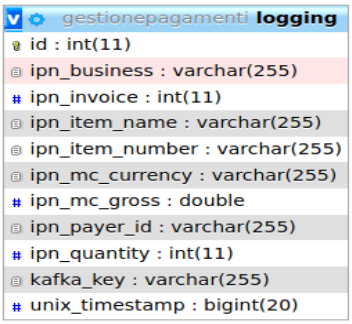
Impalà Antonio – Matricola 1000012375

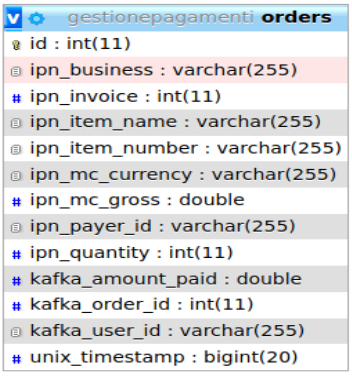
Di Blasi Kristian – Matricola 1000012372

Introduzione

Il progetto scelto dal gruppo è il 3A: il microservizio relativo alla gestione dei pagamenti. La strategia di helth-check scelta è il Ping-Ack mode. Tale microservizio utilizzerà Sping MVC, JPA e MySql ed implementerà i requisiti richiesti.

Il database Mysql si è scelto implementarlo come di seguito riportato:





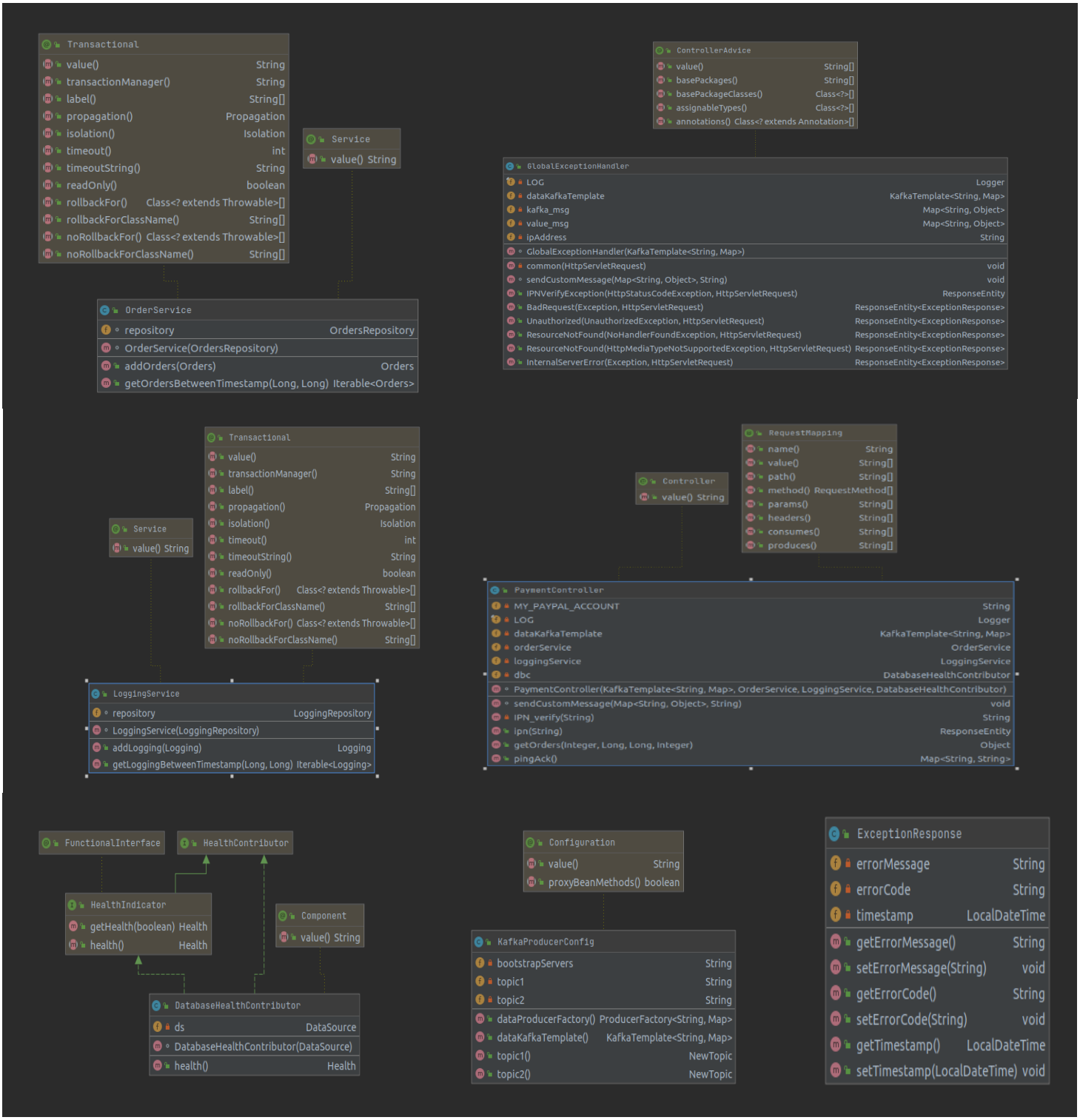
Sono stati implementati due model corrispondenti ai topics di Kafka.

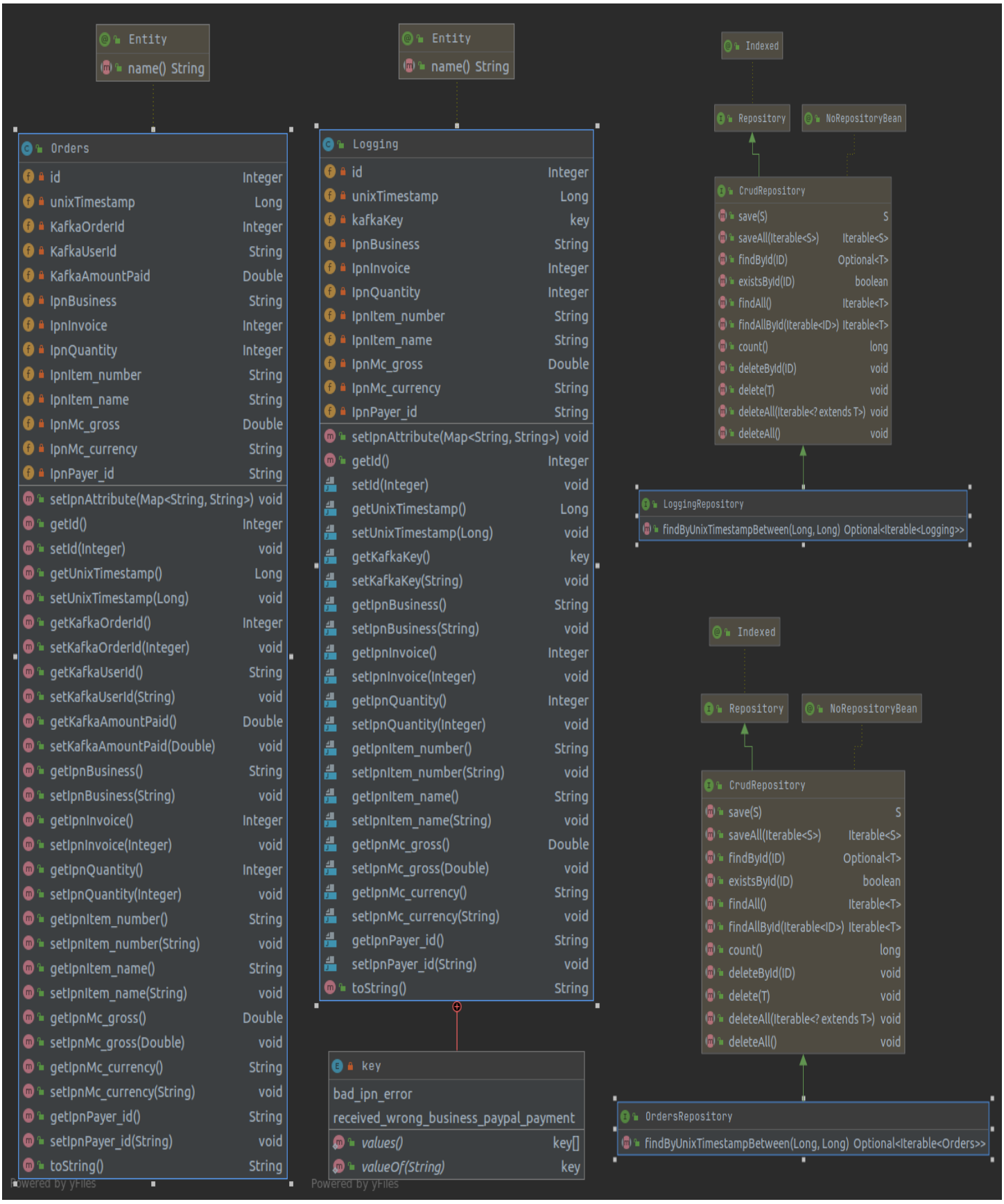
All'interno di "***Logging***" verranno inseriti i messaggi inviati a Kafka sul topic "***logging***" con le possibili key: "***bad_ipn_error***" e "***received_wrong_business_paypal_payment***".

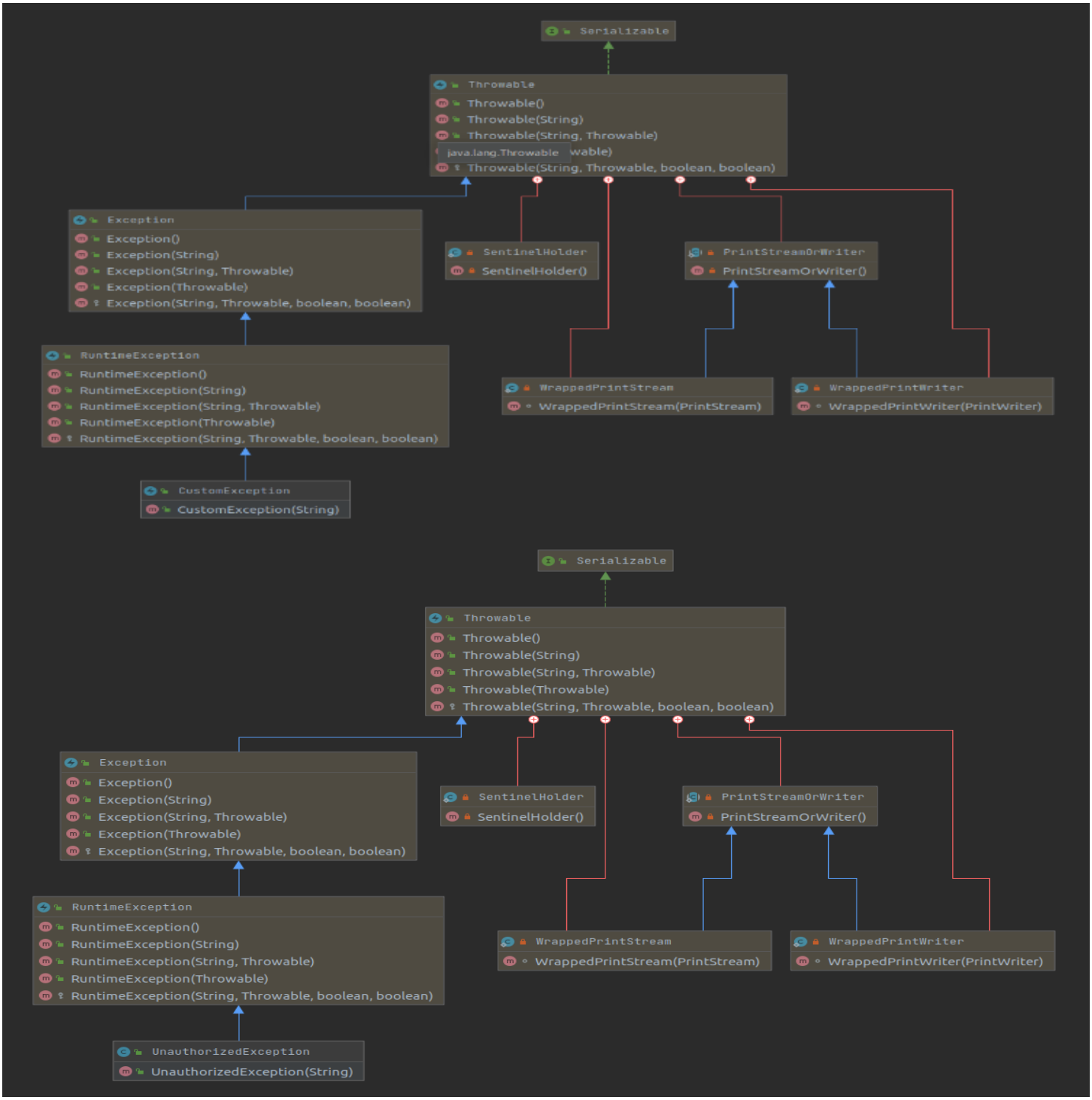
All'interno di "***Orders***" invece verranno inseriti i messaggi inviati a Kafka sul topic "***orders***" con la key : "***oder_paid***" .

In aggiunta ai messaggi sopra indicati, verranno inserite all'interno delle tabelle anche le informazioni contenute nel body della richiesta POST ricevuta dall'endpoint ***ipn*** ("*invoice", "item_name", "item_number", "quantity", "mc_currency", "payer_id", "mc_gross", "business*").

Diagramma delle Classi




Entity
name() String
Entity
name() String
Indexed
Repository
NoRepositoryBean
Orders
id Integer
unixTimestamp Long
KafkaOrderId Integer
KafkaUserId String
KafkaAmountPaid Double
IpnBusiness String
IpnInvoice Integer
IpnQuantity Integer
IpnItem_number String
IpnItem_name String
IpnMc_gross Double
IpnMc_currency String
IpnPayer_id String
setIpnAttribute(Map<String, String>) void
getId() Integer
setId(Integer) void
getUnixTimestamp() Long
setUnixTimestamp(Long) void
getKafkaOrderId() Integer
setKafkaOrderId(Integer) void
getKafkaUserId() String
setKafkaUserId(String) void
getKafkaAmountPaid() Double
setKafkaAmountPaid(Double) void
getIpnBusiness() String
setIpnBusiness(String) void
getIpnInvoice() Integer
setIpnInvoice(Integer) void
getIpnQuantity() Integer
setIpnQuantity(Integer) void
getIpnItem_number() String
setIpnItem_number(String) void
getIpnItem_name() String
setIpnItem_name(String) void
getIpnMc_gross() Double
setIpnMc_gross(Double) void
getIpnMc_currency() String
setIpnMc_currency(String) void
getIpnPayer_id() String
setIpnPayer_id(String) void
toString() String
Logging
id Integer
unixTimestamp Long
kafkaKey key
IpnBusiness String
IpnInvoice Integer
IpnQuantity Integer
IpnItem_number String
IpnItem_name String
IpnMc_gross Double
IpnMc_currency String
IpnPayer_id String
setIpnAttribute(Map<String, String>) void
getId() Integer
setId(Integer) void
getUnixTimestamp() Long
setUnixTimestamp(Long) void
getKafkaKey() key
setKafkaKey(String) void
getIpnBusiness() String
setIpnBusiness(String) void
getIpnInvoice() Integer
setIpnInvoice(Integer) void
getIpnQuantity() Integer
setIpnQuantity(Integer) void
getIpnItem_number() String
setIpnItem_number(String) void
getIpnItem_name() String
setIpnItem_name(String) void
getIpnMc_gross() Double
setIpnMc_gross(Double) void
getIpnMc_currency() String
setIpnMc_currency(String) void
getIpnPayer_id() String
setIpnPayer_id(String) void
toString() String
CrudRepository
save(S) S
saveAll(Iterable<S>) Iterable<S>
findById(ID) Optional<T>
existsById(ID) boolean
findAll() Iterable<T>
findAllById(Iterable<ID>) Iterable<T>
count() long
deleteById(ID) void
delete(T) void
deleteAll(Iterable<? extends T>) void
deleteAll() void
LoggingRepository
findByUnixTimestampBetween(Long, Long) Optional<Iterable<Logging>>
key
bad_ipn_error
received_wrong_business_paypal_payment
values() key[]
valueOf(String) key
Indexed
Repository
NoRepositoryBean
CrudRepository
save(S) S
saveAll(Iterable<S>) Iterable<S>
findById(ID) Optional<T>
existsById(ID) boolean
findAll() Iterable<T>
findAllById(Iterable<ID>) Iterable<T>
count() long
deleteById(ID) void
delete(T) void
deleteAll(Iterable<? extends T>) void
deleteAll() void
OrdersRepository
findByUnixTimestampBetween(Long, Long) Optional<Iterable<Orders>>
Powered by yFiles
Powered by yFiles


Serializable
Throwable
Throwable()
Throwable(String)
Throwable(String, Throwable)
java.lang.Throwable
wable)
Throwable(String, Throwable, boolean, boolean)
Exception
Exception()
Exception(String)
Exception(String, Throwable)
Exception(Throwable)
Exception(String, Throwable, boolean, boolean)
SentinelHolder
SentinelHolder()
PrintStreamOrWriter
PrintStreamOrWriter()
RuntimeException
RuntimeException()
RuntimeException(String)
RuntimeException(String, Throwable)
RuntimeException(Throwable)
RuntimeException(String, Throwable, boolean, boolean)
WrappedPrintStream
WrappedPrintStream(PrintStream)
WrappedPrintWriter
WrappedPrintWriter(PrintWriter)
CustomException
CustomException(String)
Serializable
Throwable
Throwable()
Throwable(String)
Throwable(String, Throwable)
Throwable(Throwable)
Throwable(String, Throwable, boolean, boolean)
Exception
Exception()
Exception(String)
Exception(String, Throwable)
Exception(Throwable)
Exception(String, Throwable, boolean, boolean)
SentinelHolder
SentinelHolder()
PrintStreamOrWriter
PrintStreamOrWriter()
RuntimeException
RuntimeException()
RuntimeException(String)
RuntimeException(String, Throwable)
RuntimeException(Throwable)
RuntimeException(String, Throwable, boolean, boolean)
WrappedPrintStream
WrappedPrintStream(PrintStream)
WrappedPrintWriter
WrappedPrintWriter(PrintWriter)
UnauthorizedException
UnauthorizedException(String)

1. Requisito: Ping endpoint (ping-ack mode)

Tale endpoint si occupa di comunicare lo stato del sistema a seguito di una richiesta GET alla route ***"/ping***". In questa fase il microservizio risponderà con un messaggio così formato:

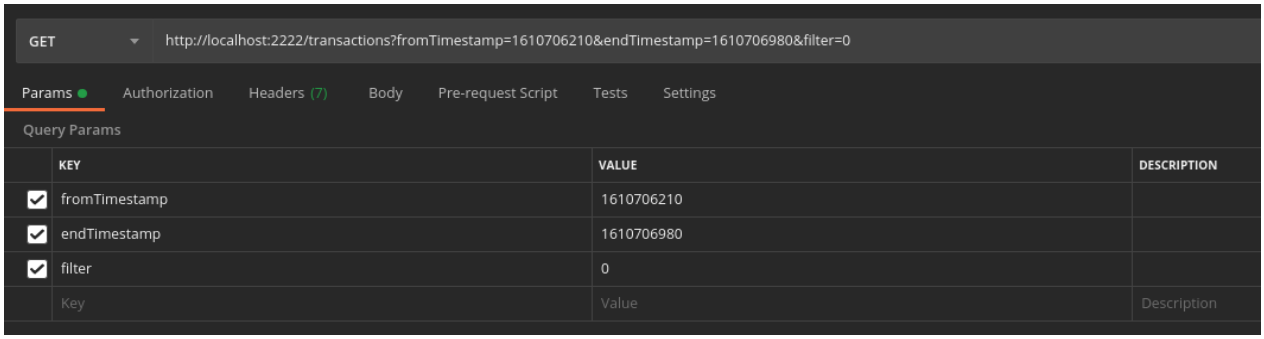
Il "***serviceStatus***" sarà sempre "***UP***", in quanto il "***DOWN***" sarà rappresentato da un mancata risposta.

Il "***dbStatus***" invece sfrutta la dipendenza "***SpringBootActuator***" per implementare un controllo sull'effettivo stato del DB. E' stato creato appositamente un Component database che implementa un collegamento al data source ed effettua una query di prova. Se scatenerà un'eccezione allora lo status "***dbStatus***" sarà "***DOWN***", altrimenti "***UP***".

2. Requisito: Retrieve delle transazioni effettuate.

Questo endpoint si occupa di restituire le transazioni effettuate a seguito di una richiesta GET alla route "***/transactions***". Al fine di garantire solo all'amministratore l'accesso alle transazioni, viene prelevato il parametro X-USER-ID contenuto all'interno dell'header della richiesta. L'assenza di tale parametro genererà un "***HttpError : 400***". Se l'utente non è l'amministratore verrà generato un "***httpError : 401***". Inoltre, si è ritenuto opportuno applicare dei filtri alla selezione delle transazioni. Tale filtro è stato implementato mediante il passaggio di un nuovo parametro "**filter**". A quest'ultimo sarà possibile assegnare i seguenti valori:

- Filter = 0 : ritorna le transazioni relative ad ordini validi (Orders).
- Filter = 1 : ritorna le transazioni relative ad ordini non validi (Logging).
- Filter = -1 : ritorna tutte le transazioni di ordini validi e non.



Se non verrà specificato nessun valore per il parametro "**filter**" l'endpoint restituirà tutte le transazioni riferenti ad ordini validi e non.

Nel caso in cui venisse passato qualsiasi altro valore, verrà generato un "***HttpError : 400***".

3. Requisito: Endpoint notifica pagamento PayPal

Questo endpoint si occupa, a seguito di una richiesta POST alla route "**/ipn**", di verificare la sua validità inviando, a sua volta, una richiesta POST all'indirizzo PayPal: "*https://www.sandbox.paypal.com/cgi-bin/webscr*". Esso risponderà affermando la validità o meno della notifica di pagamento. Per effettuare il test di pagamento si è fatto utilizzo di uno script in python per l'assemblamento della richiesta POST e relativi parametri da passare allegati all'URL; il file in questione è il file "***ipn.py***" Per effettuare la richiesta POST si è fatto uso delle librerie RestTemplate di Spring. Una volta effettuata la richiesta si possono verificare varie casistiche.

Se la richiesta non va a buon fine verrà inviato sul topic "***logging***" di Kafka un messaggio come:

```
{
    key = http_errors
    value = {
        timestamp: UnixTimestamp,
        sourceIp: Indirizzo ip microservizio (autore richiesta),
        service: payment manager,
        request: path + "|POST",
        error: codice errore
    }
}
```

Se invece la richiesta va a buon fine:

- Il valore di ritorno è “**INVALID**”. La key, il timestamp e i dati più rilevanti (descritti nell’introduzione) contenuti nella richiesta POST verranno salvati nella tabella “***logging***” del database e si invierà sul topic “***logging***” di Kafka

```
{
    key = bad_ipn_error
    value = {
        timestamp: UnixTimestamp,
        Tutti i dati contenuti nella richiesta ricevuta
    }
}
```

un messaggio come:

- Il valore di ritorno è “**VERIFIED**”, a questo punto verrà controllato che il campo “business” della richiesta sia uguale alla variabile di ambiente “**MY_PAYPAL_ACCOUNT**”. In caso negativo la key, il timestamp e i dati più rilevanti (descritti nell’introduzione) contenuti nella richiesta POST verranno salvati nella tabella “***logging***” del database e si invierà sul topic “***logging***” di Kafka un messaggio come:

```
{
    key = received_wrong_business_paypal_payment
    value = {
        timestamp: UnixTimestamp,
        Tutti i dati contenuti nella richiesta ricevuta
    }
}
```

- Il valore di ritorno è “**VERIFIED**” e il campo “*business*” coincide con “**MY_PAYPAL_ACCOUNT**”. Allora salverà nella tabella “***orders***” del database i campi più rilevanti (descritti nell’introduzione) della POST ricevuta, i dati inviati su Kafka ed un Timestamp corrispondente al

salavataggio. A questo punto si invierà sul topic "***orders***" di Kafka un messaggio come:

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

A questo punto l'endpoint ritornerà sempre il codice "***200 OK***".

4. Requisito: Invio messaggi su topic Kafka

Questa specifica di cui è stato parlato in precedenza è stata realizzata aggiungendo alla sezione controller una classe di configurazione per il produttore dei messaggi che si occupi anche di creare i ***Topic*** se questi non esistono già.

5. Gestione Errori: Exception handler

Per la gestione delle eccezioni si è scelto di utilizzare il componente "***Controller Advise***" attraverso l'omonima annotazione sulla classe *handler*. In particolare si occuperà di gestire tutti gli errori (exception) di rilevanza per il nostro scenario, alcuni dei quali sono stati già descritti durante la specifica delle **API REST**. Altri sono: in caso di endpoint errato il controller scriverà sul ***topic logging*** i dati relativi all'errore e risponderà con una ***Exception Response*** contenente l'***HttpError*** "***404 NOT FOUND***"; nel caso in cui il database va in crash e arriva una richiesta il controller scriverà sul ***topic logging*** i dati relativi all'errore(compreso di stack trace) e risponderà con una ***Exception Response*** contenente l'***HttpError*** "***500 INTERNAL SERVER ERROR***".

I messaggi inviati su Kafka in caso di errori presentano la seguente forma:

```
{
    key = http_errors
    value = {
        timestamp: UnixTimestamp,
        sourceIp: sourceIp
        service: products,
        request: path + method
        error: error //codice errore se si tratta di errori 40x, stack trace se si tratta di errori 50x
    }
}
```

L' ***Exception Response*** inviata dall'endpoint in caso di errore presenta la seguente forma:

```
{
    errorMessage: messaggio contenuto nell'eccezione,
    errorCode: codice errore,
    timestamp: timestamp errore
}
```