OVERVIEW PAYMENT MICROSERVICE

DSBD2020/2021_SANTONOCITO_PALUMBO

PROF.SSA ANTONELLA DI STEFANO

INTRODUZIONE

- Di seguito verranno brevemente esposte e commentate alcune delle funzionalità più significative del microservizio di gestione dei pagamenti implementato.
- La gestione del nostro lavoro ha previsto la suddivisione delle classi in package distinti, cercando di separare le varie responsabilità dei moduli in accordo a quanto fatto durante le esercitazioni del corso.
- I commenti che riguardano le classi ed i metodi implementati si possono trovare anche all'interno del codice con l'aggiunta di maggiori dettagli, in modo tale da semplificarne la sua leggibilità.
- Le tre funzionalità più significative del microservizio (Heartbeating, gestione dell'IPN e retrieve delle transazioni) sono descritte in maniera dettagliata all'interno dei diagrammi di sequenza UML generati tramite il software Astah.

PACKAGE «ADAPTER»: REST-ADAPTER

- Adapter REST Reactive per l'implementazione del nostro servizio:

a. public Mono<ServerResponse> savePayment(ServerRequest serverRequest) ➔ Metodo usato per il caricamento nel DB di Payments per la fase di testing. È stato utilizzato per riempire la collezione di pagamenti durante le fasi iniziali del progetto.

b. public Mono<ServerResponse> ipn(ServerRequest serverRequest) ➔ Metodo per la gestione dell'Instant Payment Notification di Paypal (nel nostro caso si è fatto uso dell'IPN Simulator messo a disposizione da PayPal utilizzando un account sandbox).

c. public Mono<ServerResponse> getTransactions(ServerRequest serverRequest) ➔ Permette all'utente ADMIN (UserID=0) di ottenere le transazioni che sono state correttamente registrate nel DB in uno specifico intervallo di tempo, quest'ultimo specificato mediante timestamps in formato UNIX.

PACKAGE «ENTITIES»: ACK | BEAT | LOG

- Classi di utility per la nostra applicazione:

➔ Classe che modella il concetto di ACK in relazione al servizio di Heart-Beat.

➔ Classe che modella il concetto di Beat (come da specifica del progetto) in relazione al servizio di Heart-Beat.

➔ Classe che modella il concetto di Log (come da specifica del progetto) che viene utilizzato sistematicamente per la pubblicazione sul topic Kafka "logging".

PACKAGE «KAFKA»: KAFKA-PRODUCER-CONFIG

- Classe di configurazione del produttore Kafka (utilizzato per la pubblicazione sui vari topic dalla nostra applicazione):

a. public Map<String, Object> producerConfigs() ➔ Bean per la creazione della configurazione del produttore.

b. public ReactiveKafkaProducerTemplate<String, String> reactiveKafkaTemplate() ➔ Bean per la creazione del Kafka template da utilizzare (nel nostro caso reactive) per la pubblicazione reattiva dei messaggi su Kafka.

c. public NewTopic ordersTopic() ➔ Bean per la creazione (se non esiste) del topic "orders" su Kafka.

d. public NewTopic loggingTopic() ➔ Bean per la creazione (se non esiste) del topic "logging" su Kafka.

PACKAGE «MODEL»: PAYMENT

- Classe che modella il concetto di pagamento da rendere persistente all'interno del database NoSQL MongoDB.
- Implementa l'interfaccia *Serializable* in modo tale che le istanze di tale classe possano essere serializzate/deserializzate in/dal formato JSON.
- Il costruttore della classe è annotato come *@JsonCreator* in modo tale che esso consenta la costruzione di un oggetto JSON a partire dall'oggetto Java associato.

(1) PACKAGE «MONGO»: MONGO-CONFIGURATION

- Classe di configurazione per Mongo annotata con *@EnableReactiveMongoRepositories* in modo tale da abilitare i repository reactive di Mongo. Nello specifico definiremo un *@Bean* necessario per fornire all'ambiente un'istanza specifica di TransactionManager (nel nostro caso reattiva) da utilizzare per la nostra applicazione.

a. ReactiveMongoTransactionManager transactionManager(ReactiveMongoDatabaseFactory rmdf) ➔ Definendo un Bean per l'istanziazione del TransactionManager da utilizzare, stiamo stabilendo che tipologie di transazioni dovranno essere gestite (nel nostro caso transazioni basate su esiti di operazioni legate alla pubblicazione di uno specifico publisher immerso in pipe reattive). Le nostre transazioni copriranno tutte le operazioni reattive, a patto di seguire uno dei seguenti 3 approcci:

(2) PACKAGE «MONGO»: MONGO-CONFIGURATION

1) NON DICHIARATIVO:

- Si utilizza un'istanza di un TransactionalOperator creato a partire da un Reactive Transaction Manager in modo esplicito usando il metodo: *(publisher sotto controllo della transazione) transactionalOperator.execute(publisher da considerare nella transazione);*
- Si utilizza un'istanza di un TransactionalOperator creato a partire da un Reactive Transaction Manager in modo esplicito usando il metodo: *(publisher da considerare nella transazione).as(transactionalOperator::Transactional)*.

2) DICHIARATIVO (QUELLO SEGUITO IN QUESTA APPLICAZIONE):

- Non si usa esplicitamente un Transactional Operator ma si annota con *@Transactional* direttamente il metodo (o la classe come insieme di metodi) da considerare transazionali (è stato scelto poiché più compatto, veloce e simile a quanto fatto negli esempi in aula).

PACKAGE «REPOSITORY»: PAYMENT-REPOSITORY

- Creazione dell'interfaccia per avere a disposizione un Reactive CRUD Repository, ossia un repository che effettua operazioni CRUD sul DB in modo reattivo (l'esito dell'operazione sul DB è il valore pubblicato da un publisher di tipo Mono o Flux in reazione alla sottoscrizione di un subscriber) ➔ ATTENZIONE: QUANDO SI USANO SUBSCRIBER PERSONALIZZATI OCCORRE TENERE IN CONSIDERAZIONE CHE SE IL PUBLISHER FA PARTE DI UNA PIPE REATTIVA A CUI SI SOTTOSCRIVERA' UN SOTTOSCRITTORE DI DEFAULT (AD ESEMPIO QUANDO FA PARTE DI UNA PIPE CHE PORTA ALLA CREAZIONE DI UNA MONO<SERVER-RESPONSE>) TALI SUBSCRIBER IMPORRANNO UN'OPERAZIONE SUL DB CIASCUNO (QUINDI SE L'OPERAZIONE E' UN SAVEAD ESEMPIO, SUL DB OTTERREMO DUE SALVATAGGI DISTINTI). IN QUESTI CASI QUINDI NON SI FA USO DI UN SOTTOSCRITTORE CUSTOM.

a. Flux<Payment> findPaymentsByUnixTimestampBetween(long from, long to) ➔ Permettiamo l'autogenerazione di un metodo reattivo per il recupero dell'elenco di pagamenti tra un UnixTimestamp di "from" ed uno di "to".

PACKAGE «ROUTER»: PAYMENT-ROUTER

- Classe di configurazione per un router che instrada al nostro ADAPTER Reactive REST le richieste HTTP ai vari ENDPOINT esposti dal nostro microservizio, come richiesto dalle specifiche.
- In tale router troviamo anche l'instradamento verso un ENDPOINT di test (per il caricamento del DB) e un ENDPOINT per gestire il servizio, simulato localmente, di Heart-Beat Acknowledgment (il cui adapter però è differente rispetto a quello che si occupa dei servizi propri dell'applicazione).

PACKAGE «SERVICE»: (1) HEART-BEAT-SERVICE | (2) PAYMENT-SERVICE

- (1) Componente che contiene del codice che implementa il servizio di Heart-Beat su un Thread separato sganciato dagli altri (implementa l'interfaccia *Runnable*).
- (2) Classe che modella il servizio di pagamento (la nostra core business logic). ATTENZIONE: tutti i metodi che restituiscono Mono<Void> sono metodi in cui il chiamante non è interessato ad un valore specifico prodotto ma soltanto all'evento di corretta (o meno) pubblicazione da parte del publisher.
- Per ulteriori informazioni consultare i diagrammi di sequenza UML.

PACKAGE «SIMULATOR/ADAPTER» & PACKAGE «SIMULATOR/SERVICE»

- Simulator/Adapter/RESTAdapterHeartBeatSink: Adapter Reactive REST per la gestione dell'Heart-Beat ACK: il nostro servizio funge anche da Heart-Beat Monitor (oltre ad essere normalmente un Heart-Beat Source).
- Simulator/Service/BeatResponseService: Servizio che simula l'Heart-Beat Monitor. Contiene un metodo che verifica la validità del messaggio di heartbeat e restituisce un Mono<Ack> contenente un Ack impostato sulla base dello stato del microservizio e del DB.

PACKAGE «SUBSCRIBER»: SERVER-RESPONSE-SUBSCRIBER

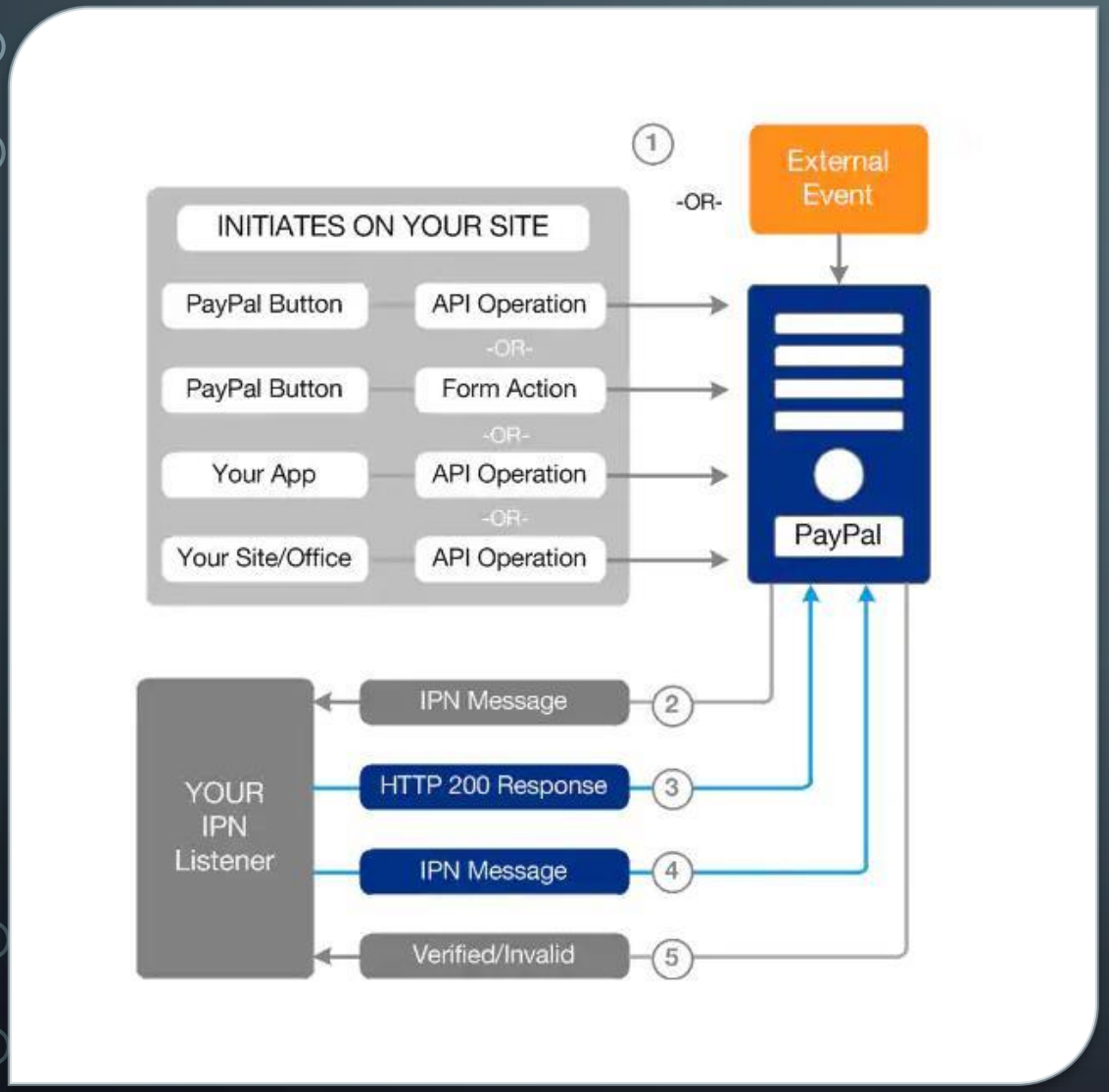
- Classe che modella un Subscriber personalizzato. Nel nostro caso lo useremo per sottoscriverci e gestire la pubblicazione della ServerResponse associata all'Happy-Path del metodo «ipn()» all'interno del REST Reactive Adapter.

a. public void hookOnNext(T value) ➔ Metodo eseguito all'atto della pubblicazione del publisher Mono<Void> a cui questo subscriber sarà sottoscritto.

PACKAGE «TEST»: TRANSACTION-TEST

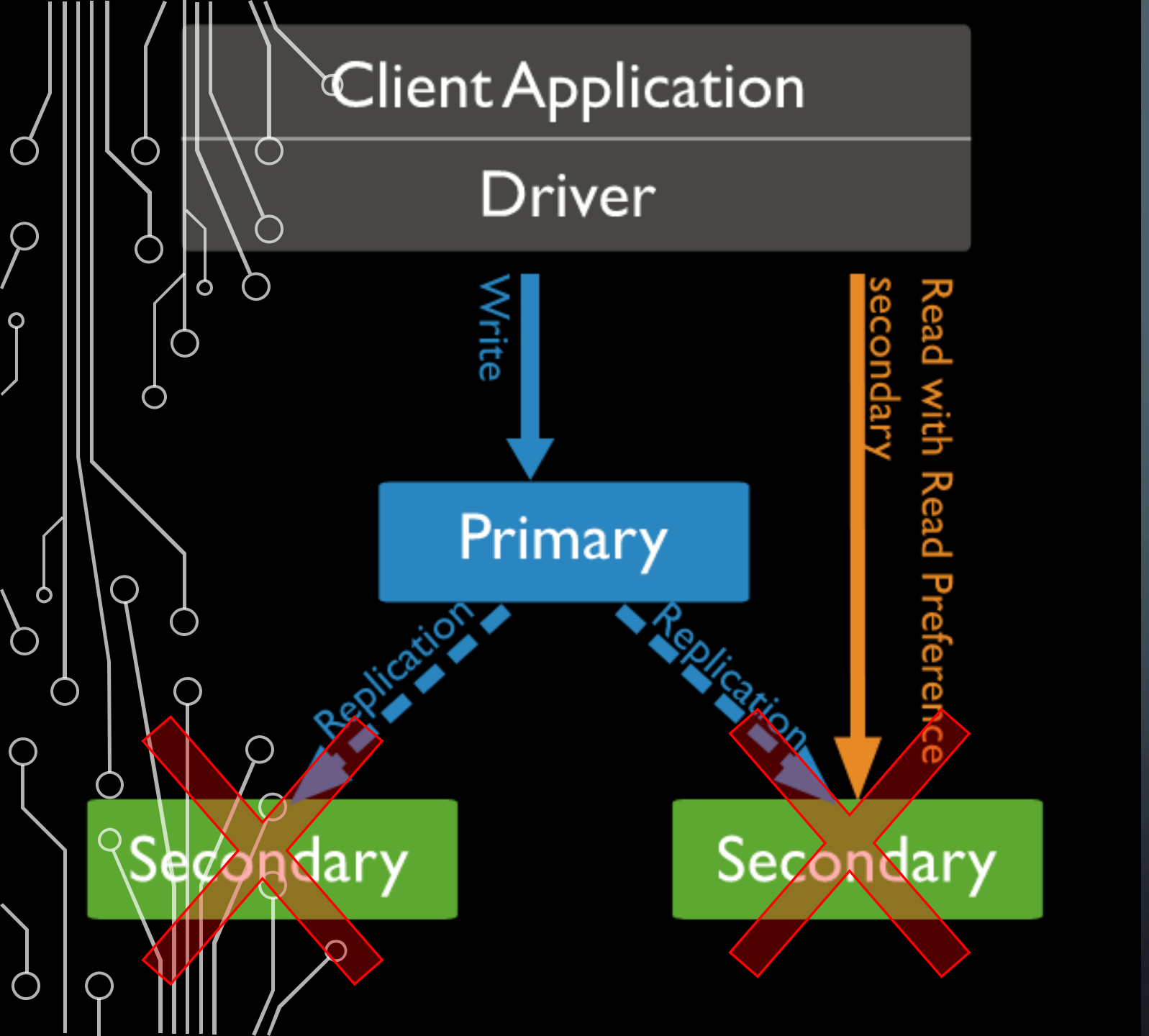
- Classe per il testing dell'effettiva transazionalità di una sequenza di operazioni sul DB.

a. public void savePayments() ➔ Metodo che sfrutta l'omonimo metodo di test contenuto in PaymentService.class per verificare il corretto funzionamento delle transazioni sul database.



NOTA IMPLEMENTATIVA 1

- Modello di interazione a 4 vie adoperato per la collaborazione con il servizio IPN (Instant Payment Notification) di PayPal.
- Nel nostro caso il passo numero 1 della figura è stato eseguito mediante l'IPN Simulator messo a disposizione dallo stesso servizio.



NOTA IMPLEMENTATIVA 2

- Modello di utilizzo del database MongoDB attraverso il meccanismo del REPLICA-SET.
- Si rende necessario l'uso di tale meccanismo poiché è l'unico modo per consentire l'impiego di transazioni ACID con questa tipologia di database NoSQL.
- Nel caso dell'applicazione, è stato utilizzato un approccio a replica set sigle node, dove questo nodo è il container «mongo» e rappresenta il nodo PRIMARY del replica set creato (potremmo definirlo «degenere»).

END