Stateful serverless application for data pipeline processing

SDCC project a.y. 2023/2024

Stefano Belli, matricola 0350116

Università degli Studi di Roma "Tor Vergata"

Agenda

1. Serverless SAGA pattern
2. Servizi AWS utilizzati
3. Tabelle DynamoDB
4. Lambda di autorizzazione
5. Lambda di validazione
6. Lambda di trasformazione
7. Lambda di store
8. Lambda di flagging
9. State machine

Serverless SAGA pattern

Adattiamo il pattern SAGA al serverless computing[1]: i **microservizi** diventano le **funzioni serverless** e l'**orchestratore** è il coordinatore delle funzioni: in base al **valore di ritorno** o a eventuali **errori** delle funzioni orchestrate, il coordinatore decide se proseguire con la transazione o avviare un rollback.

[1] Tabby Ward (AWS), Rohan Mehta (AWS), and Rimpy Tewani (AWS) - "Implement the serverless saga pattern by using AWS Step Functions" - `https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html`

Servizi AWS utilizzati

Per effettuare il deployment dell'applicazione è stato utilizzato AWS, e in particolare i seguenti servizi:

- **API Gateway**: configurazione dell'endpoint e route HTTP
- **Lambda**: per il serverless computing
- **Step Functions**: come coordinatore delle funzioni lambda
- **DynamoDB**: database serverless NoSQL di tipo chiave-valore
- **Secrets Manager**: storage crittografico gestito da AWS per memorizzare segreti

Tabelle DynamoDB

Le tabelle necessarie a supportare la transazione sono 3:

- `validationStatus`: tabella di appoggio alla lambda `validate`
- `transformationStatus`: tabella di appoggio alla lambda `transform`
- `storeStatus`: tabella di appoggio alla lambda `store`

Tutte e 3 memorizzano la tupla ricevuta in input dalla lambda (`RawTuple`), l'ID della transazione (`StoreRequestId`, uguale tra tutte le tabelle per la stessa transazione, *partition key numerica*, assimilabile a una primary key di un RDBMS) e lo stato della transazione (`StatusReason`)

La tabella finale è `nycYellowTaxis` ed è consultabile dall'applicativo che effettua analisi sui dati (processamento vero e proprio). E' composta da item che hanno attributi distinti per tutti gli elementi separati dal carattere di separazione della CSV entry, e non più quindi una "`RawTuple`". Sono presenti in questa tabella solo i dati processati **correttamente**. Viene mantenuto lo `StoreRequestId` (che rimane uguale a quello delle tabelle di appoggio, per la stessa transazione).

Tabelle DynamoDB: stato in base all'esito della transazione

Transazione terminata con successo

Transazione (intesa come entry composta dagli attributi `StoreRequestId`, `RawTuple` e `StatusReason`) presente in tutte le tabelle d'appoggio con `StatusReason` $= 0$ per ognuna di esse.
E' presente l'item corrispondente nella tabella finale `nycYellowTaxis`

Transazione fallita alla fase *i*-esima di preprocessing

L'entry corrispondente alla transazione è presente al più nella tabella d'appoggio della fase *i*- esima fino a quella della prima fase, con `StatusReason` $\neq 0$ che ha stesso valore tra tutti i db di appoggio.
NON è presente l'item corrispondente nella tabella finale `nycYellowTaxis`

Lambda di autorizzazione

1. La lambda `authorizer`, viene invocata dall'API gateway se viene agganciato a quest'ultimo un'*Authorizer*: gli viene passato un JSON object contenente un array che ha come chiave `"identitySource"`, che ha in prima posizione la chiave di autenticazione inserita dal client HTTP nell'header della richiesta - ovvero l'entry che ha come chiave `"Authorization"`.
2. `authorizer` confronta quindi la chiave di autenticazione fornita dal client con quella impostata in fase di deployment e memorizzata in uno storage crittografico gestito da AWS Secrets Manager (chiedendone quindi la decrittazione).
3. Ritornando un JSON object { `"isAuthorized": true` } oppure { `"isAuthorized": false` }, questa lambda permette all'API gateway di decidere se far "passare" o meno la richiesta.

Vantaggi

Si mitiga l'effetto di attacchi di tipo denial of service, che a causa del run delle lambda, diventano attacchi di tipo monetario, e proprio per quest'ultima ragione sarebbero solamente mitigati, e non risolti del tutto: comunque viene eseguita la lambda authorizer. Ma si elimina del tutto il problema di inserimenti non autorizzati

(Il suo utilizzo è opzionale)

Lambda di validazione

Alla lambda di validazione, la prima nella catena di preprocessing, viene passata la tupla in "formato raw", ovvero una CSV entry (stringa): è la lambda che apre la transazione.

1. Calcola `StoreRequestId`, un ID univoco della transazione - a partire dalla tupla in input stessa (una "weak" hash function), il tempo UNIX al momento dell'invocazione della lambda e il valore che restituisce un PRNG
2. Inserisce nel proprio db di appoggio la tupla ricevuta in input, con l'ID di transazione e lo stato posto inizialmente a 0 (successo)
3. Effettua validazione sui vari campi in base alle dizionario fornito dal sito della città di New York in merito dal dataset dei taxi gialli, controlla che le date siano corrette. Se la colonna non è ritenuta importante e il check fallisce, comunque "lascia correre" e la sostituisce con stringa vuota.
4. Effettua validazione inter-colonna: constraint su più colonne.
   - Check data "pickup" $<$ "dropoff"
   - Check colonna somma di altre
5. Ritorna un JSON object che segnala la riuscita o meno dell'operazione, l'ID di transazione, la tupla stessa (validata, trimmed + elim. campi) e il reason code (si vedano le lambda di flagging in seguito)

Lambda di trasformazione

Riceve dalla lambda precedente, la tupla validata con l'ID univoco della transazione (è ovvio che a questo punto, la parte di JSON object ricevuta in input rimanente che indica fallimenti in parte precedente è:
{ `"success": true, "reason": 0, ...` }), quindi:

1. Inserisce l'entry nel suo db di appoggio composto da tupla ricevuta, ID univoco della transazione e stato della transazione
2. Effettua trasformazioni sulla tupla:
   - Convertire un'enumerazione da intero a stringa secondo il dizionario del dataset
   - Convertire da numero decimale a intero la colonna 4 (numeri erano espressi come 10.0, 9.0, 8.0, ...)
   - Convertire data e ora in un altro formato
   - Cambio carattere di separazione da ',' a '\t' per la tupla (entry CSV)
3. Ritorna { `"success": true, "reason": 0, ...` } se la trasformazione è andata a buon fine, { `"success": false, "reason": 2, ...` } altrimenti. E' presente nel JSON object ritornato, anche la tupla trasformata e l'ID di transazione

Lambda di store

L'ultima lambda da eseguire è quella che colleziona la tupla preprocessata nella tabella finale:
Riceve lo stesso JSON object che riceveva `transform` da `validate`, solo con la tupla trasformata secondo le regole scritte per la trasformazione.

1. Inserisce la tupla trasformata ricevuta nella propria tabella di appoggio, insieme, come al solito, a ID univoco di transazione e stato della transazione.

A che serve avere un'ulteriore tabella di appoggio?

Se dovesse fallire l'inserimento nella tabella finale, non viene persa la tupla raw trasformata!

2. Costruisce l'entry per la tabella finale, separando opportunamente gli attributi (non è più una stringa di una riga CSV). Nell'entry della tabella finale viene mantenuto l'ID della transazione eseguita.
3. Inserisce l'entry appena costruita nella tabella finale
4. Ritorna un JSON object { `"success" : true` }

Vale la pena notare che la lambda di store non **fallisce** *mai per* **valore di ritorno** *"unexpected", ma solamente per un* **errore** *(inserimento non riuscito in tabella)*

Lambda di flagging

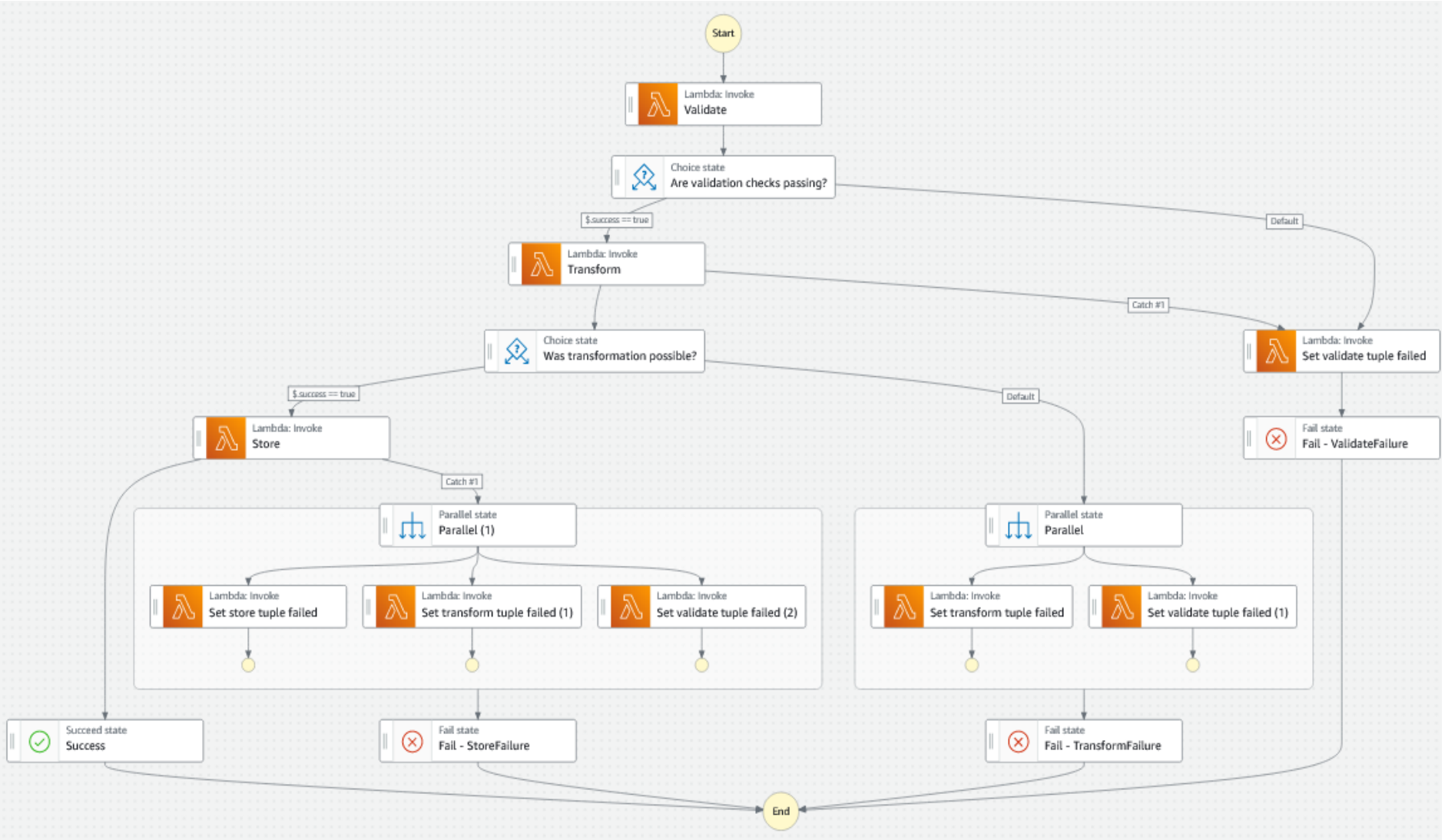
Le lambda di flagging `flagValidateFailed`, `flagTransformFailed` e `flagStoreFailed` condividono lo stesso identico codice: è stato realizzato un modulo Go `flagPhaseFailed` che generalizza queste lambda che modificano l'entry nella propria tabella di competenza al verificarsi di un fallimento di transazione.

Il modulo, in base all'input fornito dalla state machine:

- **E' già in grado di recuperare il codice d'errore reason** - la transazione è fallita perchè la lambda corrispondente a una certa fase ha **ritornato** un JSON object diverso da quanto ci si sarebbe aspettati (il codice d'errore è indicato direttamente dalla lambda fallita)
- **Deve dedurre il codice d'errore reason** - lo fa dalle informazioni sull'**errore**, che ha causato fallimento della lambda corrispondente a una certa fase, incluse dalla state machine nell'input alla lambda di flagging

Una volta dedotto, viene aggiornata l'entry corrispondente alla tabella di appoggio (in base alla partition key `StoreRequestId`) di competenza, con il codice di stato corrispondente alla fase di preprocessing che ha causato fallimento

State machine



Grazie per l'attenzione!