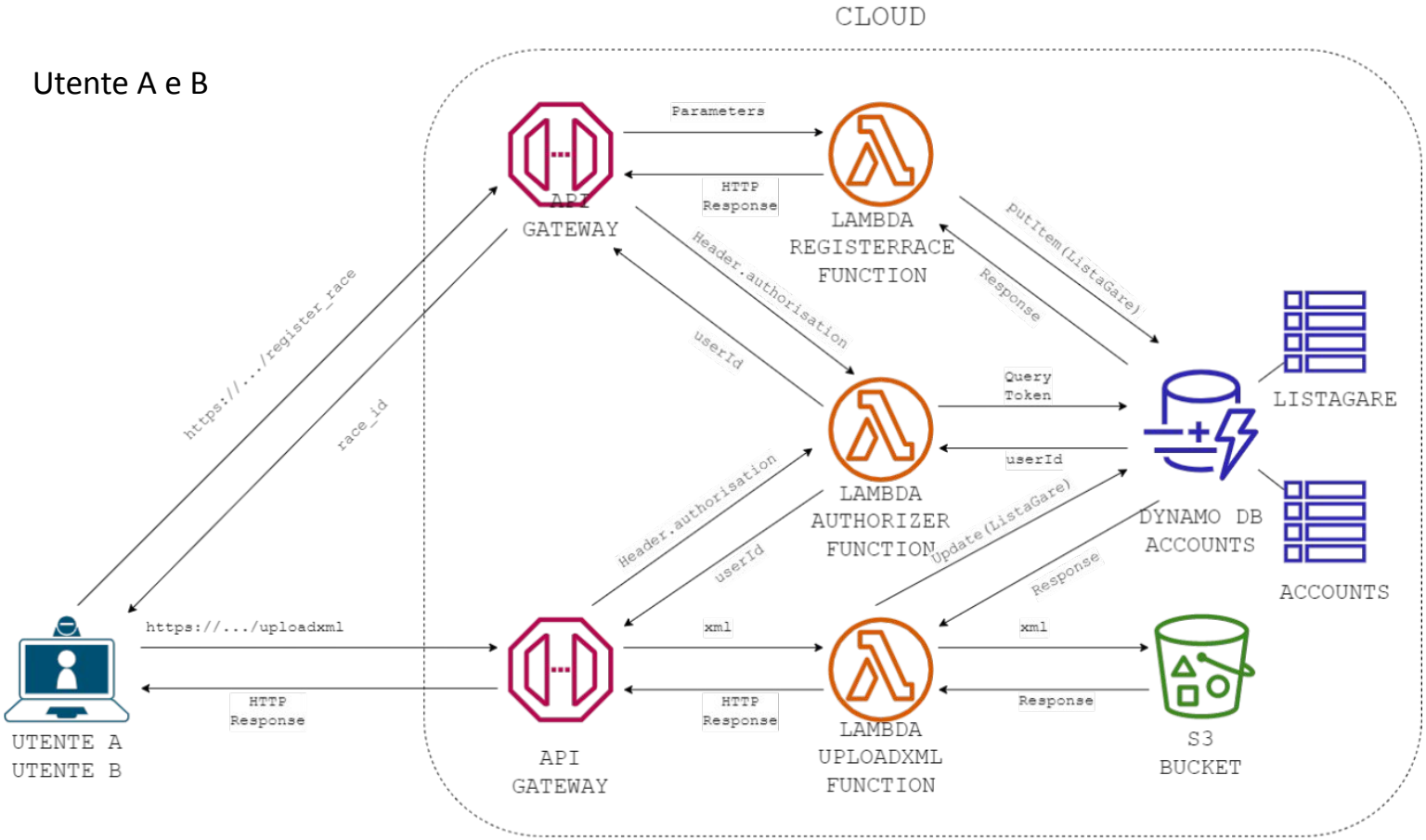
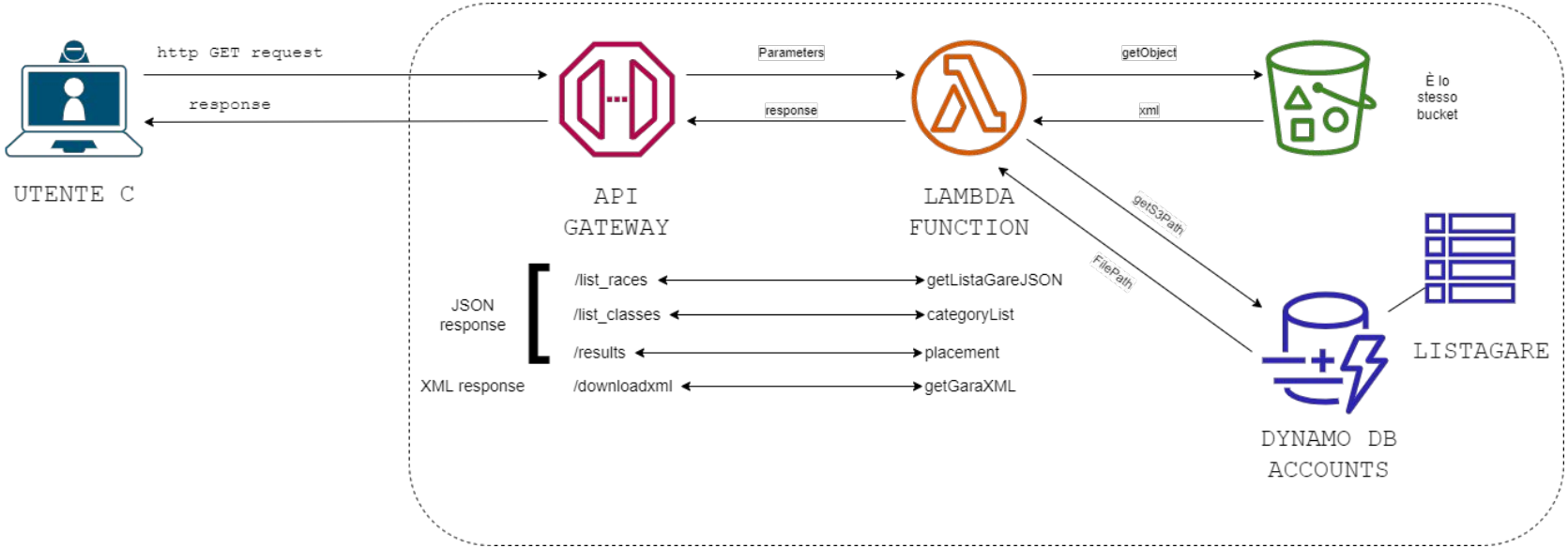
JQuelli:
Capelli Luca 1064893
Ragosta Stefano 1064527
Vedovati Matteo 1064586

PRESENTAZIONE SECONDA PARTE DELLE ESERCITAZIONI AWS

Utente A e B


CLOUD
Parameters
HTTP Response
API GATEWAY
LAMBDA REGISTERRACE FUNCTION
PutItem(ListaGare)
Response
Header.authorisation
userId
Query Token
userId
LAMBDA AUTHORIZER FUNCTION
DYNAMO DB ACCOUNTS
LISTAGARE
ACCOUNTS
Update(ListaGare)
Response
https://.../register_race
race_id
https://.../uploadxml
HTTP Response
Header.authorisation
userId
xml
HTTP Response
xml
Response
UTENTE A
UTENTE B
API GATEWAY
LAMBDA UPLOADXML FUNCTION
S3 BUCKET

Utente C (Visualizzatore)



Register Race

Per poter registrare una corsa bisogna fare una richiesta POST all'endpoint /register_race fornendo: l'autenticazione relativa ad un account personale (Basic Auth), il nome della gara ed la data di svolgimento.

La funzione Authorizer si occuperà di verificare la validità del token di autenticazione, verificando la presenza dell'account nella tabella Accounts del DB, e passerà alla funzione lambda registerRace l'ID utente corrispondente.

La funzione registerRace si occuperà invece di creare una gara all'interno del DB salvando al suo interno: l'ID gara univoco generato all'interno della funzione stessa, il nome della gara, la data di svolgimento e l'ID utente che ha richiesto di creare la gara (sarà quindi quest'ultimo utente che avrà i permessi per l'upload del XML della gara). La funzione ritorna infine l'ID della gara nella risposta HTTP.

(Non è necessario ritornare un token per fare l'upload dei risultati poiché l'unico utente abilitato a fare ciò è colui che l'ha creato la gara)

Modifiche ad UploadXML

Le uniche modifiche apportate ad uploadXML sono:

- ID della gara ora si passa come parametro
- Si controlla che l'utente che ha richiesto l'upload sia il gestore della gara. In particolare si controlla che l'ID ritornato dall'authorizer sia lo stesso dell'ID utente della gara, salvato all'interno del DB.

(Le altre funzionalità richieste erano invece già presenti all'interno della funzione [punti: 2, 3, 4])

Lambda function chiamate dall'utente C

(in base alla numerazione degli homework)

5: alla richiesta GET con "/getListaGareJSON" la funzione si connette a DynamoDB e facendosi restituire tutti gli elementi all' interno della tabella "ListaGare". Se ci sono gare, le ritornerà in formato JSON, altrimenti ritornerà che la tabella della "ListaGare" è vuota. (Per ogni gara i valori ritornati saranno: ID, NomeGara, DataInizio, OraInizio e OraFine.)

6: alla richiesta GET su /list_classes?id=X la funzione, prima prende il file da S3 dopo aver chiesto il path al dynamoDB (ListaGare), poi per ogni ClassResult nell'xml, cerca Class prendendo id e nome e ritorna un JSON con quelle informazioni

7: alla richiesta GET su /results?id=X&class=Y la funzione, prima prende il file da S3 dopo aver chiesto il path al dynamoDB (ListaGare), cerca la Class con quell'id, appena la trova prende id, nome, cognome, tempo, posizione e status di ogni atleta ritornando tutte queste informazioni in formato JSON (scelto perché più facile da gestire universalmente, utile se si prevede di fare un client. Inoltre come detto a lezione comunica bene con Flutter che useremo in futuro)

8: alla richiesta GET con "/getGaraXML?id=X" la funzione si connette al DynamoDB e cerca il percorso del file corrispondente all'id cercato. Se la ricerca va a buon fine, prende il link del file con il quale si connette al bucket S3 e restituisce in formato XML il contenuto del file, altrimenti ritorna che l'id inserito non è presente.

9: alla richiesta GET su /results?id=X&organisation=Z la funzione, prima prende il file da S3 dopo aver chiesto il path al dynamoDB (ListaGare), per ogni ClassResult, cerca tutte le persone con organizzazione = Z e si segna id, nome e cognome delle persone, ritornandole poi in formato JSON