
# Specifica Architetturale

**Informazioni sul Documento**

| | |
|---|---|
| **Versione** | 2.0.0 |
| **Data di approvazione** | 2022-06-18 |
| **Approvatori** | Luciano Wu |
| **Redattori** | Francesco Protopapa<br>Greta Cavedon<br>Matteo Basso |
| **Verificatori** | Michele Gatto<br>Pietro Villatora |
| **Uso** | Esterno |
| **Distribuzione** | Prof. Cardin Riccardo<br>Gruppo *DreamTeam* |

*e-mail: dreamteam.unipd@gmail.com*

# Registro delle Modifiche

| Versione | Data | Nominativo | Ruolo | Descrizione |
|---|---|---|---|---|
| v2.0.0 | 2022-06-18 | Luciano Wu | Responsabile | Approvazione per il rilascio. |
| v1.1.0 | 2022-06-14 | Francesco Protopapa | Progettista | Verifica complessiva di coesione e consistenza. (Verificatore: *Pietro Villatora*) |
| v1.0.3 | 2022-06-14 | Francesco Protopapa | Progettista | Modifica §2.3 (Verificatore: *Pietro Villatora*) |
| v1.0.2 | 2022-06-12 | Francesco Protopapa | Progettista | Modifica §2.2 (Verificatore: *Pietro Villatora*) |
| v1.0.1 | 2022-06-11 | Greta Cavedon | Progettista | Modifica §2.4.1 (Verificatore: *Pietro Villatora*) |
| v1.0.0 | 2022-05-14 | Luciano Wu | Responsabile | Approvazione per il rilascio. |
| v0.2.0 | 2022-05-13 | Greta Cavedon | Progettista | Verifica complessiva di coesione e consistenza. (Verificatore: *Pietro Villatora*) |
| v0.1.2 | 2022-05-12 | Greta Cavedon | Progettista | Stesura §§2.4.1-2 (Verificatore: *Pietro Villatora*) |
| v0.1.1 | 2022-05-09 | Greta Cavedon | Progettista | Stesura §2.1.2, §2.4 e §3 (Verificatore: *Pietro Villatora*) |
| v0.1.0 | 2022-05-07 | Francesco Protopapa | Progettista | Verifica complessiva di coesione e consistenza (Verificatore: *Michele Gatto*) |
| v0.0.4 | 2022-05-05 | Matteo Basso | Progettista | Stesura §2.3 (Verificatore: *Michele Gatto*) |
| v0.0.3 | 2022-05-05 | Francesco Protopapa | Progettista | Stesura §2.2 (Verificatore: *Michele Gatto*) |
| v0.0.2 | 2022-05-04 | Francesco Protopapa | Progettista | Stesura §2.1 (Verificatore: *Michele Gatto*) |
| v0.0.1 | 2022-04-30 | Francesco Protopapa | Amministratore | Creazione scheletro documento e stesura §1 (Verificatore: *Michele Gatto*) |

# Indice

**1 Introduzione** **4**
1.1 Scopo del Documento . . . 4
1.2 Scopo del Prodotto . . . 4
1.3 Glossario . . . 4
1.4 Riferimenti . . . 4
1.4.1 Normativi . . . 4
1.4.2 Informativi . . . 4

**2 Architettura del Prodotto** **5**
2.1 Architettura generale . . . 5
2.1.1 Schema . . . 5
2.1.2 Descrizione . . . 5
2.1.3 Comunicazione tra servizi . . . 6
2.1.3.1 Impostazioni coda SQS . . . 6
2.1.3.2 Dead letter queue . . . 6
2.2 Architettura del Crawling Service . . . 8
2.2.1 Descrizione . . . 8
2.2.2 Diagrammi delle classi . . . 8
2.2.3 Diagrammi di sequenza . . . 8
2.2.4 Struttura messaggio SQS . . . 10
2.2.5 Design pattern notevoli utilizzati . . . 11
2.2.6 Schema del database . . . 11
2.3 Architettura del Ranking Service . . . 12
2.3.1 Descrizione . . . 12
2.3.2 Diagrammi delle classi . . . 12
2.3.3 Diagrammi di sequenza . . . 12
2.3.4 Note sul processo di analisi . . . 14
2.3.5 Design pattern notevoli utilizzati . . . 15
2.3.6 Schema del database . . . 15
2.4 Architettura del FrontEnd . . . 16
2.4.1 Diagramma delle classi . . . 17
2.4.2 Diagramma di sequenza . . . 17

# Elenco delle figure

1 Architettura generale . . . 5
2 Crawling Service - Diagramma delle classi . . . 8
3 Crawling Service - Diagramma di sequenza - 1 . . . 9
4 Crawling Service - Diagramma di sequenza - 2 . . . 9
5 Crawling Service - Diagramma di sequenza - 3 . . . 10
6 Crawling Service - Esempio di un messaggio SQS . . . 10
7 Crawling Service - Schema ER del database . . . 11
8 Ranking Service - Diagramma delle classi . . . 12
9 Ranking Service - Diagramma di sequenza - 1 . . . 13
10 Ranking Service - Diagramma di sequenza - 2 . . . 14
11 Ranking Service - Schema ER del database . . . 15
12 Schema MVVM . . . 16
13 Architettura Frontend - Diagramma delle classi . . . 17

# 1 Introduzione

## 1.1 Scopo del Documento

Lo scopo del presente documento è quello di descrivere in maniera coesa, coerente ed esaustiva le caratteristiche architetturali del prodotto *Sweeat* sviluppato dal gruppo *DreamTeam*.

## 1.2 Scopo del Prodotto

L'obiettivo di Sweeat e dell'azienda Zero12 è la creazione di un sistema software costituito da una Webapp. Lo scopo del prodotto è di fornire all'utente una guida dei locali gastronomici sfruttando i numerosi contenuti digitali creati dagli utenti sulle principali piattaforme social (Instagram e TikTok). In questo modo, è possibile realizzare una classifica basata sulle impressioni e reazioni di chiunque usufruisca dei servizi dei locali, non solo da professionisti ed esperti del settore.

## 1.3 Glossario

Per evitare ambiguità relative alle terminologie utilizzate è stato creato un documento denominato "*Glossario*". Questo documento comprende tutti i termini tecnici scelti dai membri del gruppo e utilizzati nei vari documenti con le relative definizioni. Tutti i termini inclusi in questo glossario, vengono segnalati all'interno del documento con l'apice $^{G}$ accanto alla parola.

## 1.4 Riferimenti

### 1.4.1 Normativi

- *Analisi dei Requisiti v2.0.0*
- Presentazione del capitolato - Zero12 Progettazione e sviluppo di una Social guida Michelin: https://www.math.unipd.it/~tullio/IS-1/2021/Progetto/C4p.pdf

### 1.4.2 Informativi

- Regolamento del progetto didattico - Materiale didattico del corso di Ingegneria del Software: https://www.math.unipd.it/~tullio/IS-1/2021/Dispense/PD2.pdf
  - *Slides 12, 17.*
- Diagrammi delle classi - Materiale didattico del corso di Ingegneria del Software: https://www.math.unipd.it/%7Ercardin/swea/2021/Diagrammi%20delle%20Classi_4x4.pdf
- Design Pattern Strutturali - Materiale didattico del corso di Ingegneria del Software: https://www.math.unipd.it/%7Ercardin/swea/2021/Design%20Pattern%20Strutturali_4x4.pdf
  - *Slides 4-13, 25-34.*
- Design Pattern Comportamentali - Materiale didattico del corso di Ingegneria del Software: https://www.math.unipd.it/%7Ercardin/swea/2021/Design%20Pattern%20Comportamentali_4x4.pdf
  - *Slides 32-40.*
- Model-View Patterns - Materiale didattico del corso di Ingegneria del Software: https://www.math.unipd.it/~rcardin/sweb/2022/L02.pdf
  - *Slides 31-40.*
- Static Factory - Cleaner Code with Static Factory Methods: https://stackify.com/static-factory-methods/

# 2 Architettura del Prodotto

## 2.1 Architettura generale

### 2.1.1 Schema

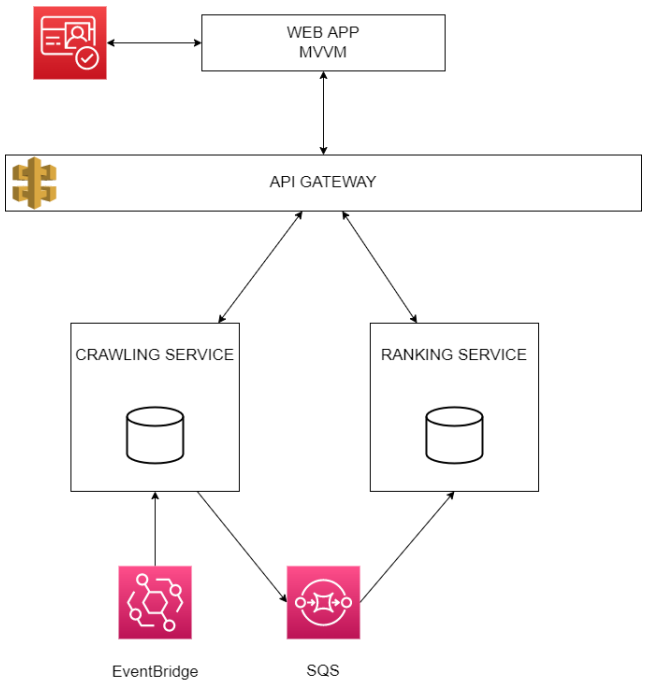


Figura 1: Architettura generale

### 2.1.2 Descrizione

Come richiesto dal capitolato si è deciso di utilizzare un'architettura a microservizi, i quali comunicano con il Frontend tramite API Gateway[G]. In particolare sono stati individuati i seguenti microservizi:

- *Crawling Service*: questo microservizio si occupa di tutto ciò che riguarda il crawling dei dati da Instagram. Il processo di crawling viene innescato da un servizio di AWS chiamato EventBridge[G]che si occupa dello scheduling del crawling. Ogni volte che viene trovato dal crawler un post relativo ad un ristorante, questo viene inviato ad una coda SQS[G]dalla quale andrà a leggere il servizio di ranking. Infine il Crawling Service espone una API al Frontend per permettere di suggerire profili Instagram da aggiungere alla lista di quelli osservati dal crawler.
- *Ranking Service*: questo microservizio invece si occupa dell'analisi dei contenuti estratti dal crawler e della realizzazione di una classifica di ristoranti. Il processo di analisi di un post viene fatto partire dalla ricezione di un messaggio sulla coda SQS, una volta letto il messaggio esso viene rimosso dalla coda ed analizzato. Infine il Ranking Service espone molteplici API al Frontend in grado di fornire tutte le informazioni necessarie per poter visualizzare la classifica, i dettagli di un locale e la gestione dei preferiti.

Invece, per il Frontend è stato scelto di realizzare la struttura sfruttando il pattern architetturale *Model-View-ViewModel* (MVVM), il quale comunica con il Backend esclusivamente tramite API Gateway.

Possiamo riassumere le motivazioni per cui abbiamo scelto il pattern architetturale MVVM per la parte Frontend come segue:

- La parte di Front-end è stata realizzata sfruttando la libreria React[G]che si integra particolarmente bene con il pattern MVVM,
- Permette di riutilizzare i vari componenti in diversi contesti senza dover effettuare modifiche; un esempio è il modello che sfruttiamo per estrarre dati dal database, i quali (tramite la stessa chiamata) vengono usati in diverse pagine della WebApp. Lo stesso vale per la vista (perché usiamo un componente in diverse pagine della WebApp),

- Permette di disaccoppiare la parte di business logic dalla presentation logic, aspetto che rende più semplice anche i test di unità,
- Maggior semplicità di sviluppo in team: in questo modo, ogni singolo componente del gruppo può occuparsi di una sola parte della WebApp,
- Manutenibilità più semplice, per via del disaccoppiamento del codice.

Inoltre, per la parte di autenticazione è stato utilizzato Cognito[G], un servizio di AWS consigliatoci dall'azienda *Zero12*, che permette di creare e gestire bacini d'utenza, oltre al framework AWS Amplify[G]per la parte di hosting ed autenticazione lato Frontend.

### 2.1.3 Comunicazione tra servizi

Per implementare la comunicazione tra il Crawling Service e Ranking Service viene utilizzata una coda SQS. La scelta di utilizzare il servizio SQS deriva da due motivi principali:

- Il servizio è completamente gestito da AWS, con un uptime del 99 % ;
- Si integra nativamente con il servizio Lambda[G];
- Costi ridotti

#### 2.1.3.1 Impostazioni coda SQS

É stata scelta una coda di tipo **FIFO** poiché garantisce l'elaborazione unica di ogni messaggio e gestisce internamente i duplicati, cancellandoli. Questo viene a costo delle performance, che non sono una criticità poiché il servizio di analisi di Ranking Service lavora in background, consumando i messaggi ad una velocità ridotta. Di seguito alcuni parametri impostati:

- **Visibility Timeout**: il tempo che un messaggio rimane nascosto dopo essere ricevuto da un consumatore. Impostato a 20 minuti poiché deve essere maggiore del tempo di esecuzione della Lambda che effettua l'analisi, che è di 15 minuti;
- **Message retention period**: il tempo massimo di permanenza dei messaggi nella coda. Impostato a 2 giorni;
- **Maximum message size**: La dimensione massima del messaggio. Impostata a 256 KB, che è la massima dimensione e quella consigliata da AWS.
- **Receive message wait time**: Il tempo massimo che il consumatore aspetta per i messaggi che diventino disponibili. Impostata a 20 secondi, che è il tempo massimo. Questo riduce il consumo di risorse, perché aspetta 20 secondi prima di ritornare una risposta vuota e permette al consumatore di lavorare sempre a pieno regime, in quanto aspettando 20 secondi la coda generalmente fa in tempo a riempirsi di almeno un batch completo;
- **Dimensione batch**: Il massimo numero di messaggi che un consumatore riceve per volta. Impostato a 10 che è il massimo per le code FIFO. In questo modo vengono ridotte le chiamate al servizio di analisi.

#### 2.1.3.2 Dead letter queue

Una dead letter queue (DLQ) è una coda agguntiva utilizzata per gestire i messaggi che non vengono elaborati con successo. Il suo utilizzo si è ritenuto necessario per spostare della coda pricipale i messaggi che generavano errori, permettendo a quelli corretti di essere elaborati, e quindi sbloccando la coda. Quindi i messaggi non corretti vengono spostati nella DLQ e possono essere analizzati in sede separata. I motivi principali per cui è stato scelto di usare una DLQ sono la semplificazione del testing e debug, permettendo di mantenere il servizio in funzione e la possibilità di reindirizzare nella coda principale i messaggi che generavano errore (dopo che il problema è stato individuato e risolto) evitando di perderli. Di seguito alcuni dei parametri impostati:

- **Message retention period**: Impostato al suo valore massimo, 14 giorni, così da far pemanere più tempo possibile i messaggi che generano errore;

- **Maximum Receives**: Il numero massimo di volte che il messaggio viene ricevuto ed elaborato senza successo. Impostato a 3, quindi dopo la seconda elaborazione senza successo viene messo nella DLQ. Il valore è stato deciso empiricamente, in questo modo i messaggi vengono ritentati per 3 volte nel giro di un'ora (20 minuti di visibility timeout). L'idea è che in generale un eventuale problema di downtime di qualche servizio possa essere mitigato e al tempo stesso si evita di riprovare troppe volte un messaggio che genera errore nel codice delle procedure, poiché finché non viene corretto, verrà continuamente generato lo stesso errore, sprecando risorse ed impedendo ad eventuali altri messaggi corretti di venire elaborati.

In generale il resto delle impostazioni resta identico a quello della coda principale.

## 2.2 Architettura del Crawling Service

### 2.2.1 Descrizione

Il microservizio denominato *Crawling Service* si occupa principalmente di due funzionalità:

- Crawling dei dati: questo processo avviene tramite l'utilizzo della libreria Instagrapi. Ogni volta che viene chiamato il metodo `start_crawling` della classe FacadeCrawling, il sistema sceglie dal proprio database il profilo Instagram che non viene guardato da più tempo e ne effettua il crawling dei dati. Per ogni media[G]prelevato dal profilo, viene analizzata la sua location[G]e nel caso in cui essa sia quella di un ristorante, il media viene salvato nel database ed inviato alla coda SQS in modo che possa essere analizzato dal servizio *Ranking Service*;
- Suggerimento dei profili: questa funzionalità viene esposta tramite una API al frontend al fine di fornire all'utente la possibilità di suggerire un profilo Instagram sul quale successivamente il sistema andrà ad effettuare il crawling dei dati. Prima di essere aggiunto al database, viene controllato che il profilo non sia già presente, esista e sia pubblico. In base a ciascun esito verrà restituito il corrispettivo ritorno.

### 2.2.2 Diagrammi delle classi

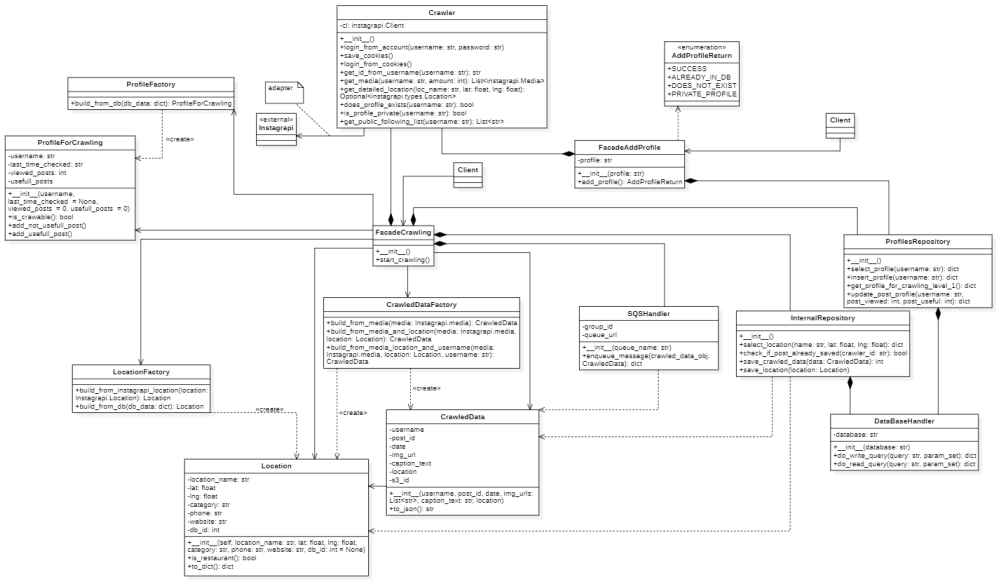


Figura 2: Crawling Service - Diagramma delle classi

### 2.2.3 Diagrammi di sequenza

In questa sezione vengono presentati i diagrammi di sequenza che modellano le operazioni principali del Crawling Service:

- il suggerimento di un profilo Instagram da aggiungere alla lista dei profili su cui viene effettuato il crawling dei dati, nel caso in cui il profilo non sia già presente e sia pubblico;
- il processo di crawling dei dati;
- la formattazione di un singolo media ottenuto tramite crawling

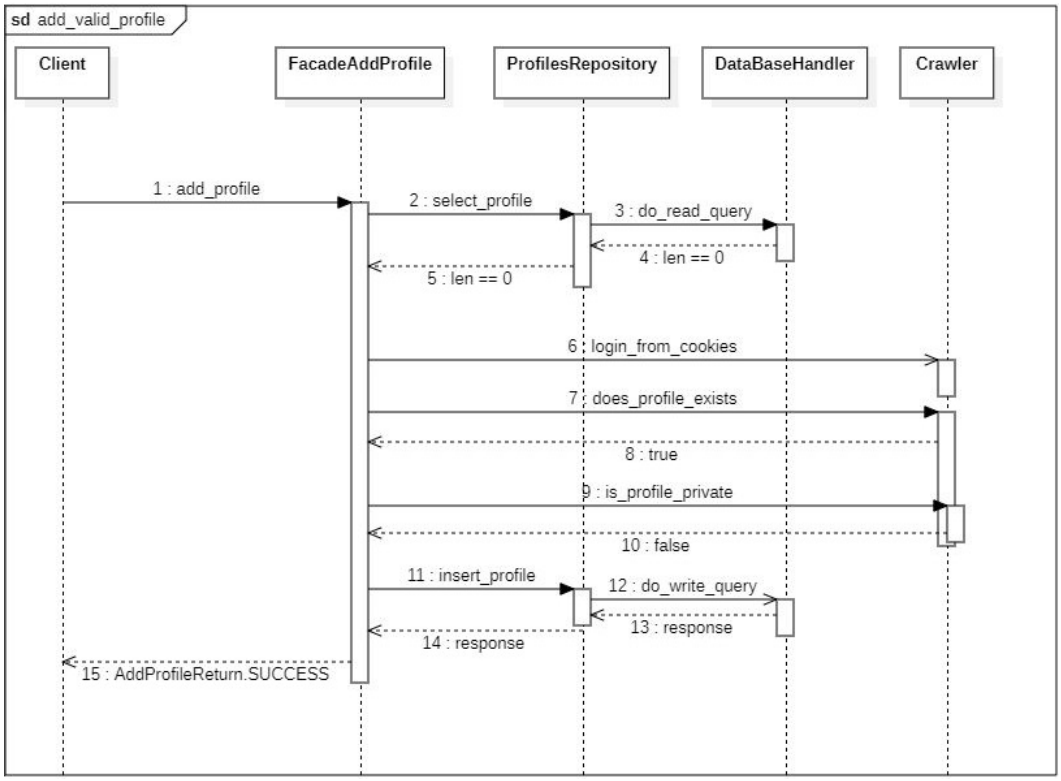


Figura 3: Crawling Service - Diagramma di sequenza - 1

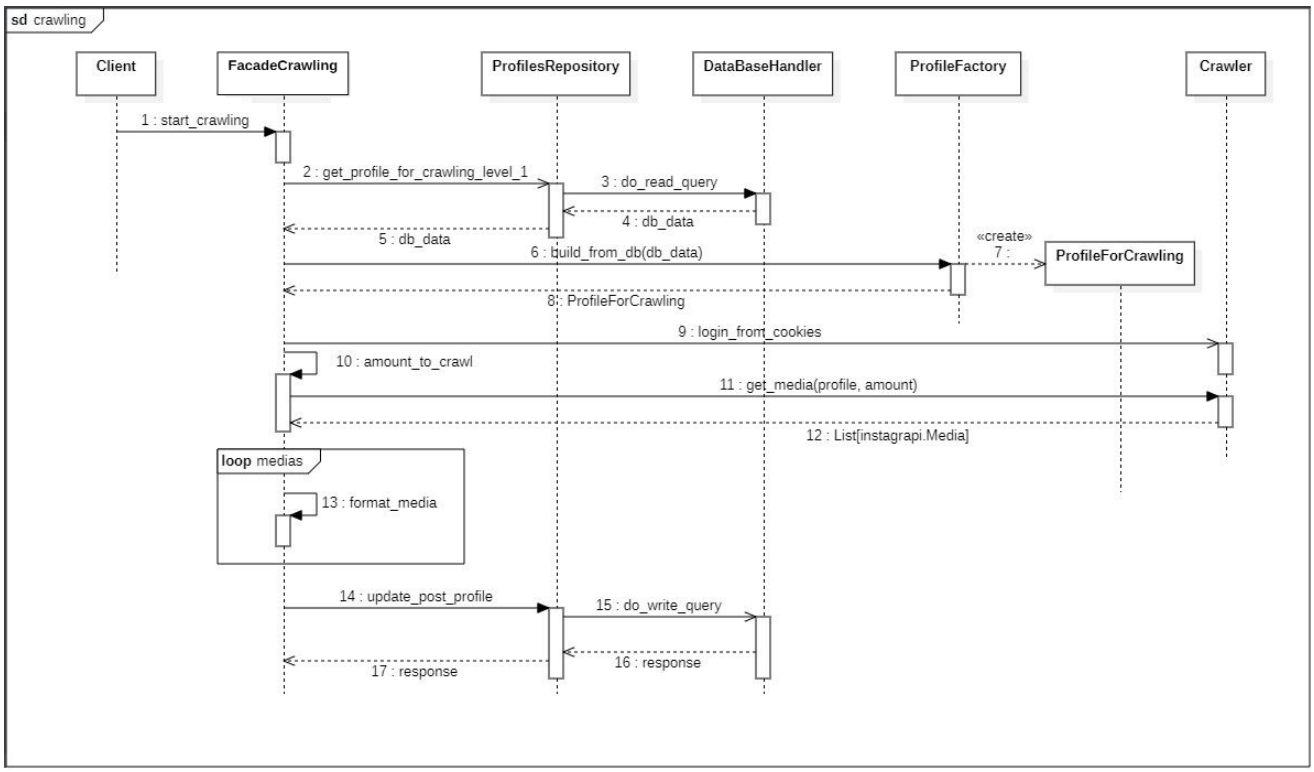


Figura 4: Crawling Service - Diagramma di sequenza - 2

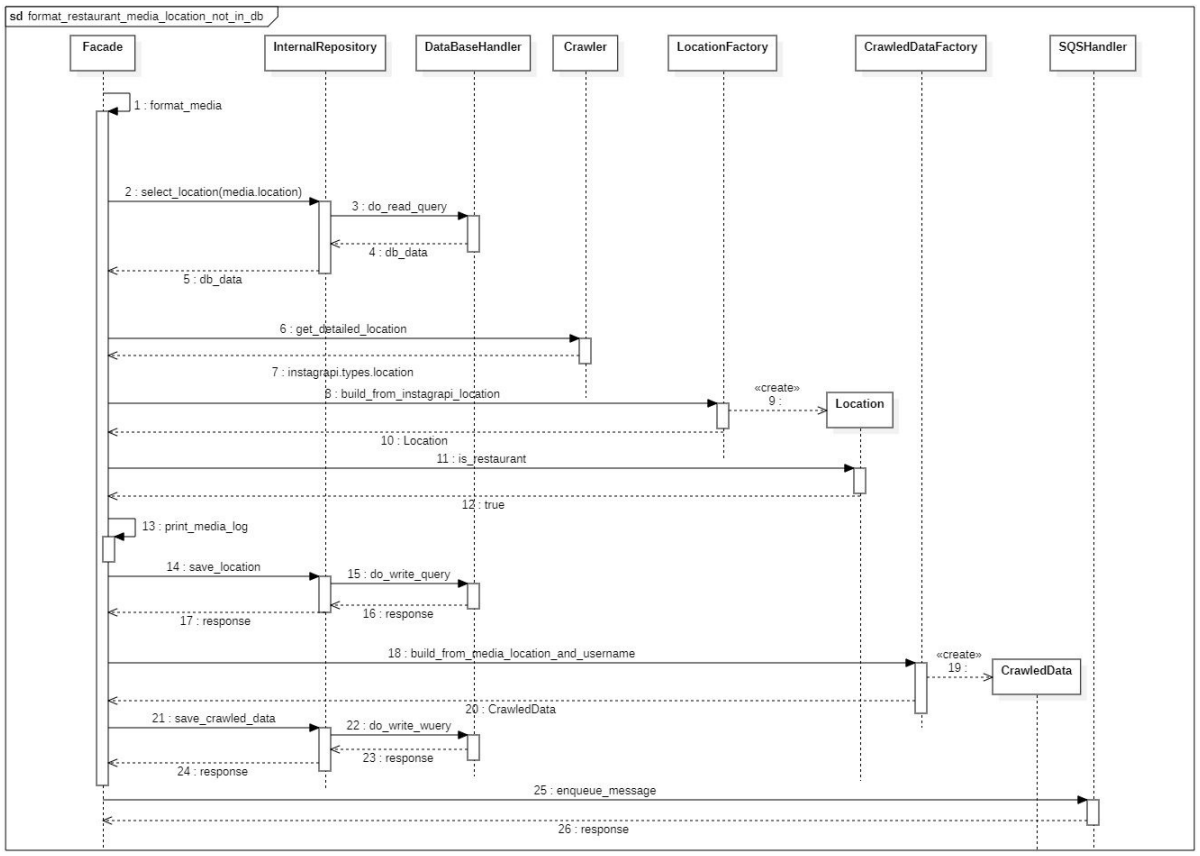


Figura 5: Crawling Service - Diagramma di sequenza - 3

### 2.2.4 Struttura messaggio SQS


```
{
  "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
  "receiptHandle": "MessageReceiptHandle",
  "body": {
    "username": "lorenzolinguini",
    "post_id": "2806725869999084478_241913061",
    "date": "2022-04-01 10:10:18",
    "img_url": [
      44,
      123
    ],
    "caption_text": "“Ask The Angels”👼\n#PattiSmith 🇺🇸🎙\n:\nRiassunto langarolo
      (potenziato) in sfoglia d’agnolotti al sugo d’arrosto🥟👔🤍🟦\nPoi Russa; Tonnato,
      Cruda & ogni sorta di beatitudine territoriale - confezionata al cesello - tra i
      capisaldi de @lapiolaalba 🥩🍷🍒\nPerché la tradizione, realizzata a queste altezze
      ascetiche, non riserva mai ‘pesci d’Aprile’🙏🤍🤖🟦\nBuon Pranzo & Buon WeekEnd
      Splendori♥",
    "location": {
      "location_name": "La Piola Alba",
      "lat": 44.7005,
      "lng": 8.036,
      "category": "Piedmont Restaurant",
      "phone": "0173442800",
      "website": "http://www.lapiola-alba.it/",
      "db_id": 8
    }
  },
  "attributes": {
    "ApproximateReceiveCount": "1",
    "SentTimestamp": "1523232000000",
    "SenderId": "123456789012",
    "ApproximateFirstReceiveTimestamp": "1523232000001"
  },
  "messageAttributes": {},
  "md5OfBody": "{{{md5_of_body}}}",
  "eventSource": "aws:sqs",
  "eventSourceARN": "arn:aws:sqs:us-east-1:123456789012:MyQueue",
  "awsRegion": "us-east-1"
}
```


Figura 6: Crawling Service - Esempio di un messaggio SQS

### 2.2.5 Design pattern notevoli utilizzati

Per La realizzazione del Crawling Service sono stati utilizzati i seguenti design pattern:

- **Facade:** Utilizzato per la realizzazione delle classi FacadeCrawling e FacadeAddProfile, in modo da fornire ai client un'interfaccia semplice ad un sottosistema molto complesso e disaccoppiando la logica di implementazione del sistema dal client.
- **Adapter:** Utilizzato dalla classe Crawler per disaccoppiare il resto del sistema dai metodi di Instagrapi, rendendo disponibili solo quelli necessari tramite un'interfaccia nota al sistema.

### 2.2.6 Schema del database

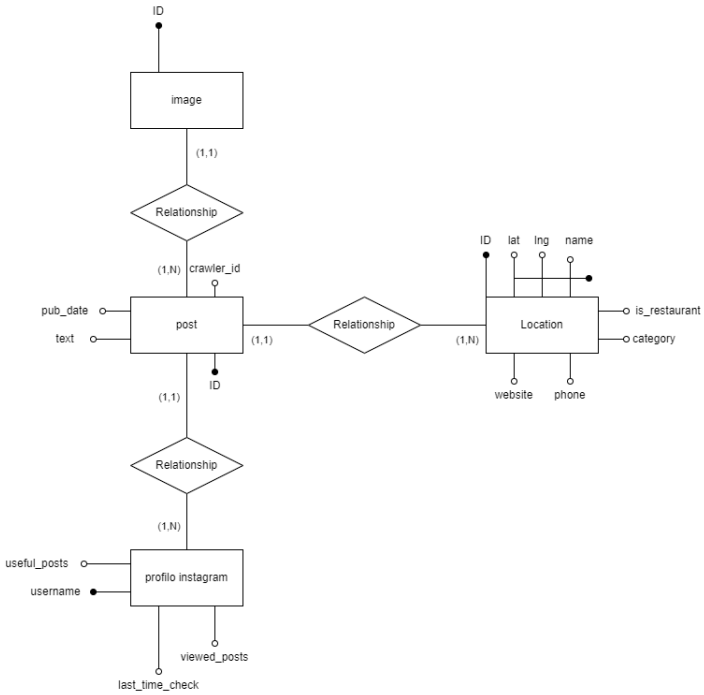


Figura 7: Crawling Service - Schema ER del database

## 2.3 Architettura del Ranking Service

### 2.3.1 Descrizione

Il microservizio denominato *Ranking Service* svolge le seguenti funzionalità:

- Gestione della classifica: ogni volta che un nuovo messaggio () viene aggiunto alla coda SQS, viene chiamati il metodo `refresh_ranking` che si occupa di recuperare e fare la parsificazione e deserializzazione dei messaggi nella coda (lavorando a batch di massimo 10 messaggi). Dopo di che i messaggi vengono analizzati tramite i servizi Comprehend[G]e Rekognition[G]di AWS e vengono generati i punteggi per le foto, per il testo e per le emoji. Viene quindi aggiornato il database, aggiungendo il ristorante e il media (se non presenti) e aggiornati i punteggi. Vengono inoltre esposti due API endpoint:
  - `getRanking`: restituisce una parte della classifica generale;
  - `getLabelAndPost`: restituisce i media (e le relative label[G]) di un particolare ristorante.
- Funzionalità di ricerca: viene esposto un API endpoint `searchByName` che restituisce tutte le informazioni presenti nel database relative ad un ristorante con un nome specifico (media compresi);
- Gestione dei preferiti: viene esposto un API endpoint `favorites` che abilità alla gestione della lista di ristoranti preferiti per ogni utente, fornendo le funzionalità di aggiunta, rimozione e visualizzazione per la lista dei preferiti.

### 2.3.2 Diagrammi delle classi

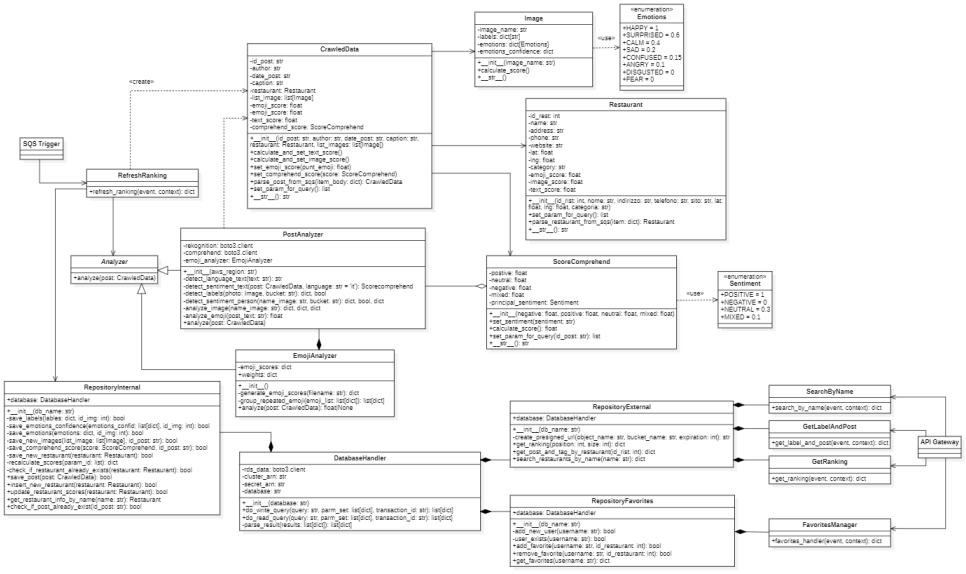


Figura 8: Ranking Service - Diagramma delle classi

### 2.3.3 Diagrammi di sequenza

In questa sezione vengono presentati i diagrammi di sequenza che modellano le operazioni principali del Ranking Service:

- Il processo di analisi di un media, assumendo che il media e il ristorante non siano già presenti nel database e che siano presenti persone delle immagini;
- Il processo di salvataggio del media analizzato e del ristorante, sempre assumendo che il media e il ristorante non siano già presenti nel database, e l'aggiornamento dei punteggi.

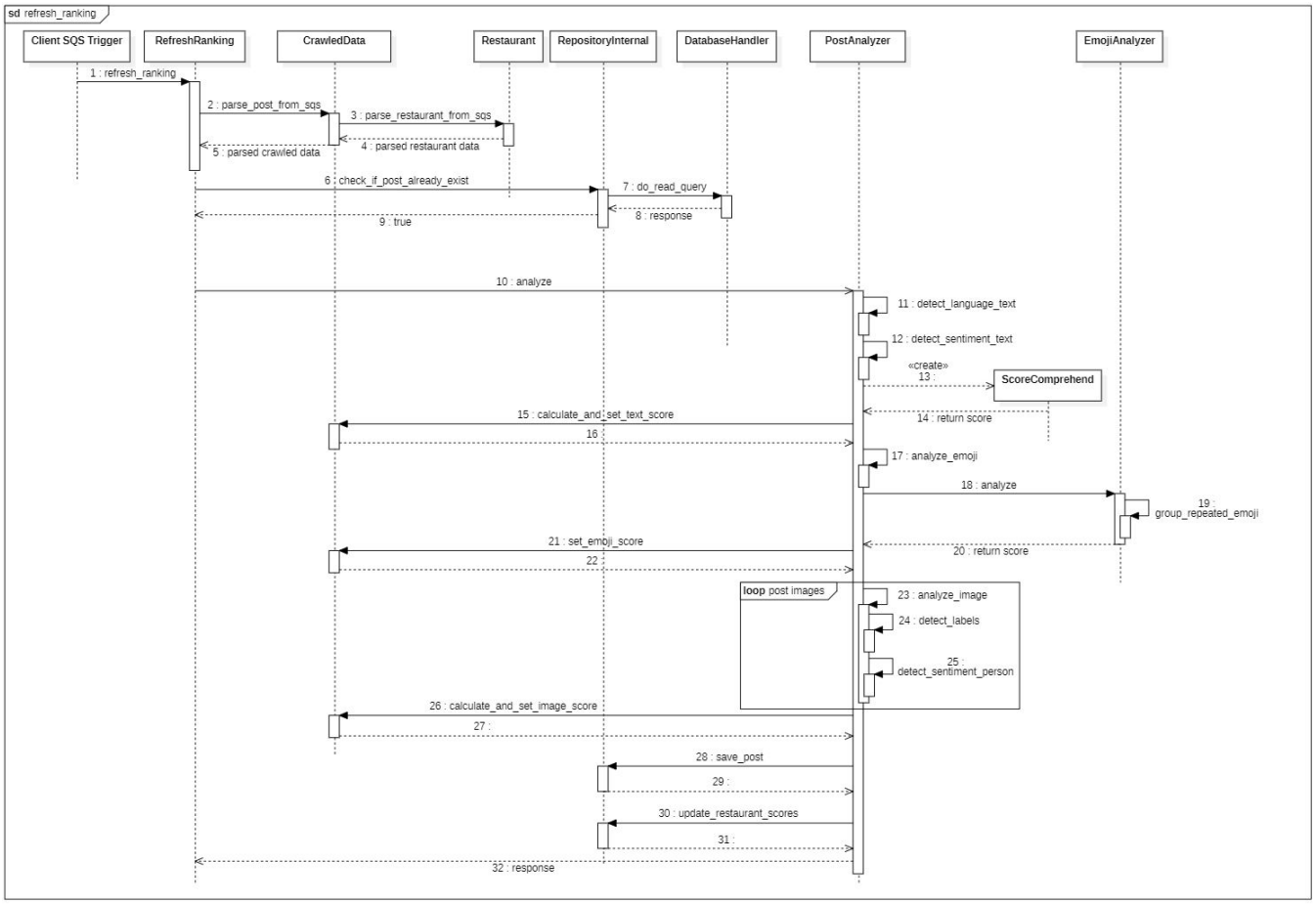


Figura 9: Ranking Service - Diagramma di sequenza - 1

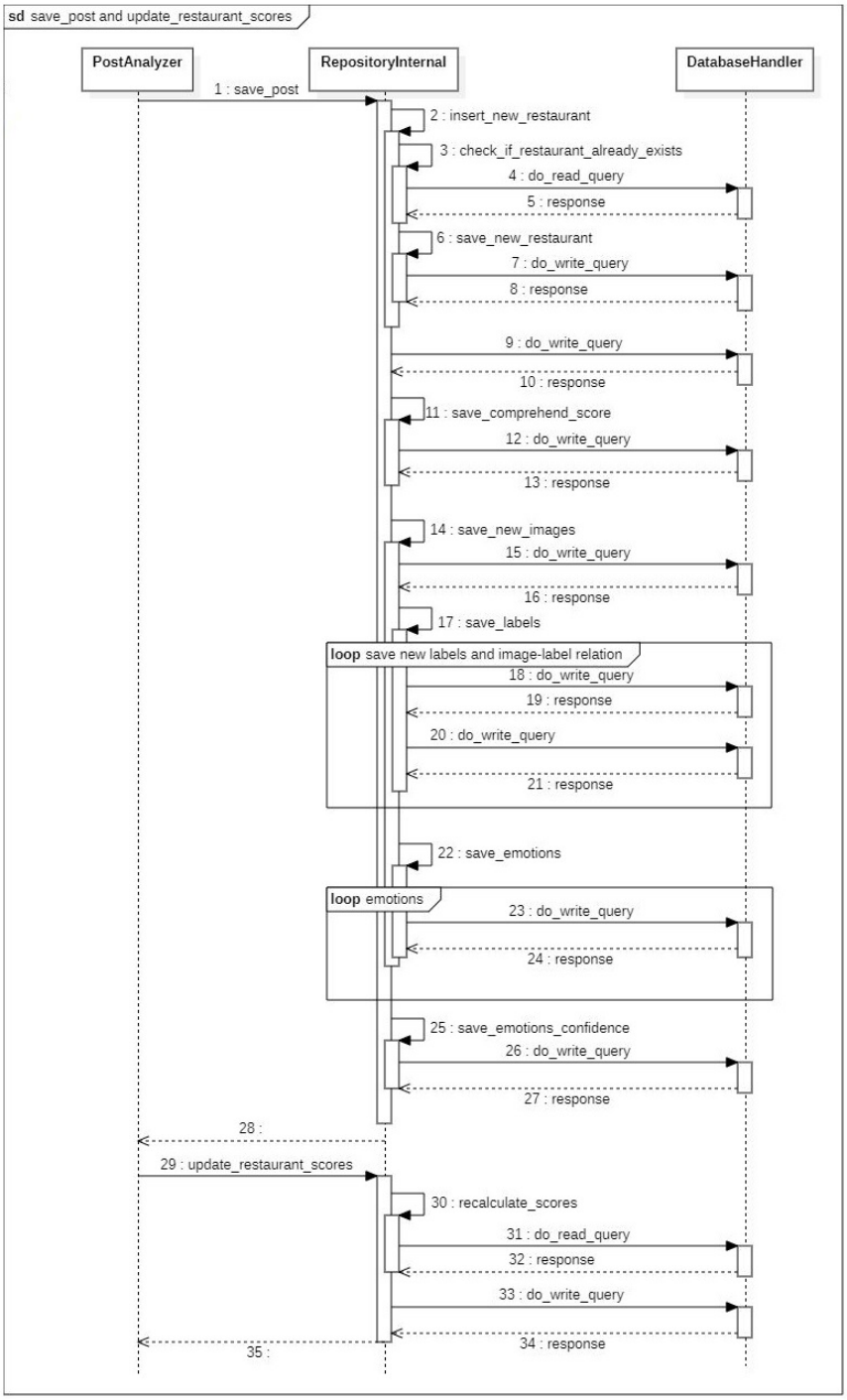


Figura 10: Ranking Service - Diagramma di sequenza - 2

### 2.3.4 Note sul processo di analisi

Dopo alcune prove ed osservazioni sono state fatte alcune decisioni relative al processo di analisi:

- Dopo l'analisi della immagini, verranno salvate nel database solo le label relative al cibo (quindi con padre "Food") e con confidenza di almeno il 90%;
- Solo se nelle immagini viene trovata la label "Person" verrà effettuata l'analisi dei sentimenti nell'immagine tramite Rekognition, anche in questo caso verranno salvati solo i sentimenti predominanti con una confidenza di almeno il 90%.

### 2.3.5 Design pattern notevoli utilizzati

Per La realizzazione del Ranking Service sono stati utilizzati i seguenti design pattern:

- **Strategy:** Utlizzato per la realizzazione di PostAnalyzer ed EmojiAnalyzer, è stato scelto principalmente per la sua versatilità.
  Questo pattern infatti ci permette, nel caso in cui in futuro ci sia bisogno di un algoritmo più avanzato per l'analisi dei post, con il minimo sforzo e modifica del codice, di implementare la nuova classe che erediterà anch'essa dalla classe base Analyzer.

### 2.3.6 Schema del database

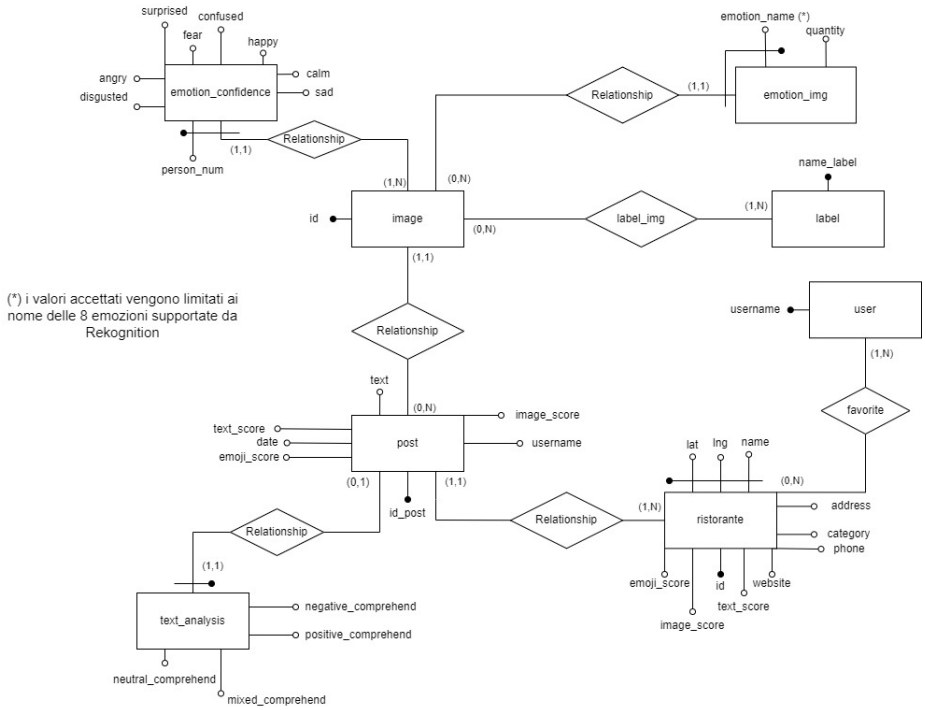


Figura 11: Ranking Service - Schema ER del database

## 2.4 Architettura del FrontEnd

Come già specificato in precedenza, nel Frontend è stato utilizzato il pattern architetturale *Model-View View-Model* (MVVM), ossia: quando l'utente esegue un'operazione nella WebApp (per esempio effettua una ricerca di un locale tramite il suo nome), il View-Model chiede al Model di scaricare i dati e/o effettuare le operazioni e rimane in attesa della risposta (nel caso di chiamate sincrone), oppure aspetta degli aggiornamenti e "osserva" (in caso di chiamate asincrone). Il Model invoca una API (che può essere una GET o una POST), la quale si interfaccerà con il Backend e, in caso di esito positivo, ritornerà la risposta o salverà i dati (in base all'operazione effettuata).
Una volta che il Model avrà terminato l'operazione, ritornerà direttamente la risposta al View-Model, nei casi in cui è necessaria, oppure lo notificherà se i dati "osservati" sono cambiati. In quest'ultimo caso, il View-Model chiederà al Model di restituirgli i dati aggiornati. Successivamente, quando anche i dati del View-Model saranno stati aggiornati, quest'ultimo invierà una notifica alla View, la quale gli dirà di re-renderizzarsi con i nuovi dati resi disponibili tramite il data-binding.

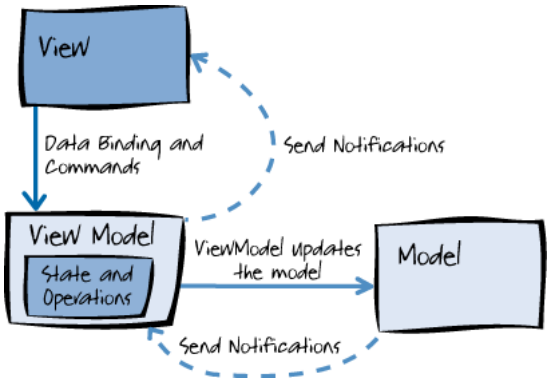


Figura 12: Schema MVVM

In questo modo si ha una netta separazione tra chi salva ed elabora i dati (il Model) e chi li mostra (la View), mentre il View-Model funge da tramite tra le due parti. In accordo con il proponente, per realizzare la parte Frontend della WebApp, abbiamo scelto di utilizzare la libreria JavaScript React. Per l'aggiornamento dei dati nel Model e la notifica dei cambiamenti al View-Model è stata utilizzata la libreria *MobX*. Questa ci ha permesso di implementare il meccanismo degli Observer su React. Invece per la notifica alla View da parte del View-Model è stata utilizzata la libreria *MobX-React*: questa permette di notificare la View e gli chiede di re-renderizzarsi, mostrando così i dati aggiornati.

### 2.4.1 Diagramma delle classi

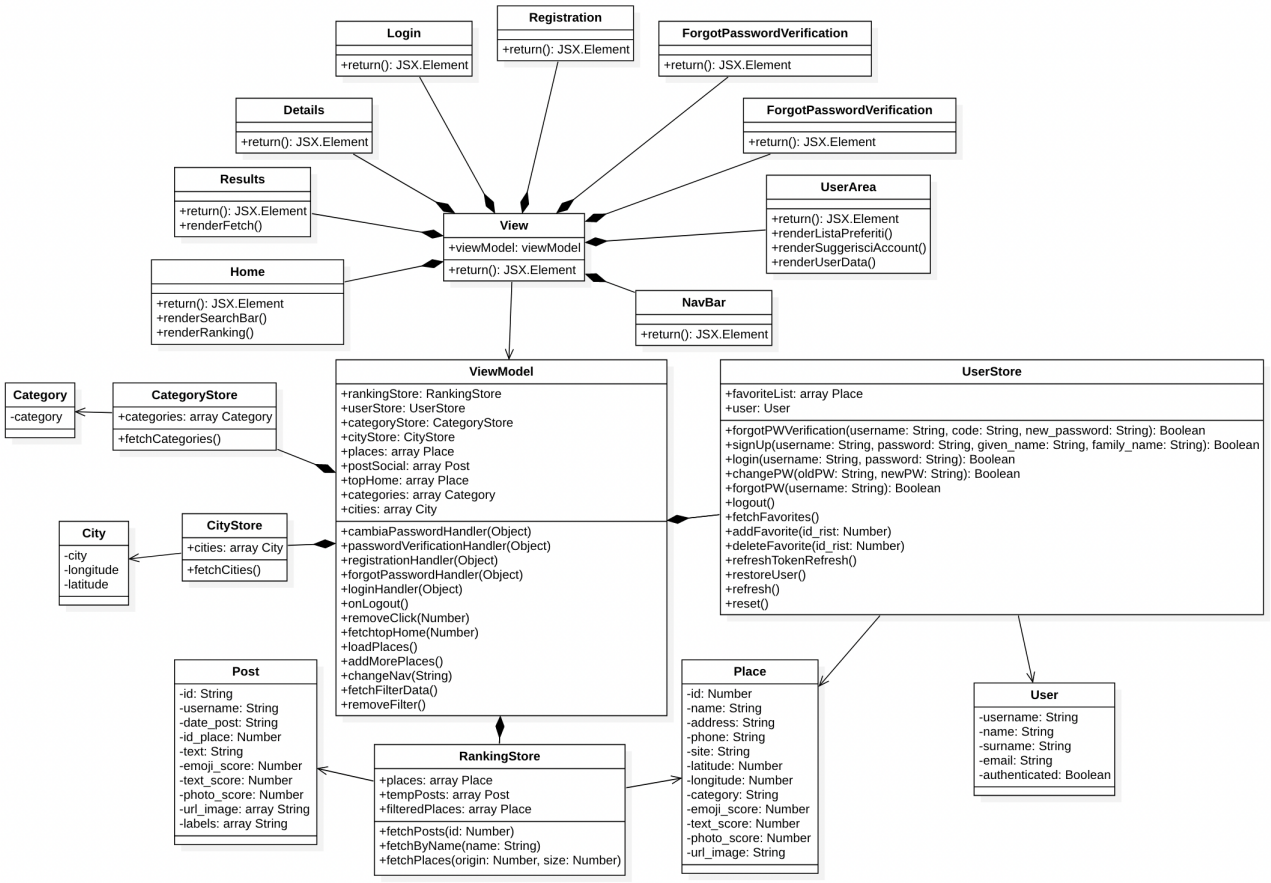


Figura 13: Architettura Frontend - Diagramma delle classi

### 2.4.2 Diagramma di sequenza

Abbiamo ritenuto non opportuno includere un diagramma di sequenza per la parte Frontend, perché le operazioni svolte risultano essere troppo semplici.

Infatti, le varie operazioni svolte lato Frontend sono delle semplici chiamate API (GET e POST) verso il Backend per ottenere i dati richiesti dall'utente (ad esempio i risultati che vengono mostrati dopo una ricerca oppure i migliori locali mostrati nella Home). Una volta ricevuto il responso dal Backend viene semplicemente aggiornata la vista della WebApp, come spiegato precedentemente.