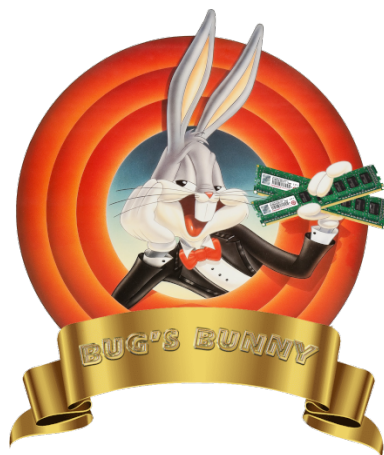




Università degli Studi di Padova
Ingegneria del Software
Anno Accademico: 2021/2022



Bug's Bunny

Specifica Architettuale

bugsbunnyteam@protonmail.com

Versione: 2.0.0

Redazione:

Tommaso Di Fant

Verifica: Angela Arena

Approvazione: Marco Bellò

Uso: Esterno



Registro delle modifiche

Versione	Data	Nominativo	Ruolo	Descrizione
2.0.0	27-09-2022	Marco Bellò	Responsabile	Approvazione del documento
1.0.1	25-09-2022	Tommaso Di Fant	Progettista	Stesura §4.2 (Verificatore: Angela Arena)
1.0.0	04-09-2022	Ruth Genevieve Bousapnamene	Progettista	Approvazione del documento
0.0.7	01-09-2022	Marco Volpato	Progettista	Stesura §3.3, §3.6 (verificatore: Angela Arena)
0.0.6	30-08-2022	Giulio Zanatta	Progettista	Stesura §4 (verificatore: Tommaso Di Fant)
0.0.5	30-08-2022	Matteo Tossuto	Progettista	Stesura §3.5.2, §3.5.3, §3.4 (verificatore: Angela Arena)
0.0.4	29-08-2022	Marco Bellò	Progettista	Stesura §3.5.1, §3.5.4, §3.5.5 (verificatore: Angela Arena)
0.0.3	24-08-2022	Matteo Tossuto	Progettista	Stesura §2, §3.2 (verificatore: Angela Arena)
0.0.2	15-08-2022	Marco Bellò	Progettista	Stesura §3.1 (verificatore: Angela Arena)
0.0.1	09-08-2022	Giulio Zanatta	Progettista	Creazione bozza documento, stesura Introduzione §1 (verificatore: Angela Arena)



Indice

1	Introduzione	5
1.1	Scopo	5
1.2	Scopo del prodotto	5
1.3	Glossario	5
1.4	Riferimenti	5
1.4.1	Riferimenti normativi	5
1.4.2	Riferimenti informativi	5
2	Architettura del sistema	7
2.1	Descrizione generale	7
2.2	Servizi S4	8
2.3	Database	9
2.3.1	Interfacciamento	9
2.3.2	Versionamento	9
3	Architettura dei componenti	11
3.1	API Service	11
3.1.1	Descrizione generale	11
3.1.2	Diagramma delle classi	11
3.1.3	Implementazione	11
3.1.3.1	Documentazione automatica	12
3.1.4	Autenticazione	12
3.2	Signup Service	12
3.2.1	Descrizione generale	12
3.2.2	Diagramma delle classi	13
3.2.3	Schemi I/O	13
3.3	Scraping Service	13
3.3.1	Descrizione generale	13
3.3.2	Diagramma delle classi	13
3.3.3	Diagramma di sequenza	14
3.3.4	Schemi I/O	14
3.3.5	Strategie di scraping	15
3.3.6	Ottenimento informazioni relative alle posizioni	16
3.3.7	Salvataggio dei dati	16
3.4	Sorting Service	16
3.4.1	Descrizione generale	16
3.4.2	Diagramma delle classi	18
3.4.3	Diagramma di sequenza	19
3.4.4	Schemi I/O	19
3.4.5	Calcolo dell'imageUrl	20
3.4.5.1	Inizio processo	20
3.4.5.2	Creazione e uso dei dizionari	21
3.4.5.3	Calcolo finale	21
3.4.5.4	Conclusione processo	21
3.4.6	Variabili	22
3.4.6.1	Response from detect_faces	22
3.4.6.2	Response from detect_labels	23



3.5	Scoring Service	24
3.5.1	Descrizione generale	24
3.5.2	Diagramma delle classi	26
3.5.3	Diagramma di sequenza	27
3.5.4	Schemi I/O	27
3.5.5	Calcolo degli Scores	28
3.5.5.1	faceScore	28
3.5.5.2	textScore	32
3.5.5.3	captionScore	34
3.5.5.4	finalScore	36
3.6	Scheduler Service	38
3.6.1	Descrizione generale	38
3.6.2	Diagramma delle classi	38
3.6.3	Diagramma di sequenza	38
3.6.4	Schemi I/O	39
3.6.5	Intervallo temporale	39
3.6.6	Regole di scheduling	39
4	Architettura Frontend	40
4.1	Diagramma delle classi	41
4.2	Descrizione delle classi	41
4.2.1	Models	41
4.2.2	Presenters	42
4.2.3	View	42



Elenco delle figure

1	Visione d'insieme del sistema	7
2	Visualizzazione macchina a stati	8
3	Schema ER database	9
4	API Service - Diagramma delle classi	11
5	Autenticazione - Diagramma di sequenza	12
6	Signup Service - Diagramma delle classi	13
7	Scraping Service - Diagramma delle classi	13
8	Scraping Service - Diagramma di sequenza	14
9	Sorting Service - Diagramma delle classi	18
10	Sorting Service - Diagramma di sequenza	19
11	Scoring Service - Diagramma delle classi	26
12	Scoring Service - Diagramma delle classi	27
13	Scheduler Service - Diagramma delle classi	38
14	Scheduler Service - Diagramma di sequenza	38
15	Schema pattern MVP	40
16	Frontend - Diagramma delle classi	41



1 Introduzione

1.1 Scopo

Il documento seguente ha lo scopo di descrivere in modo esaustivo e coerente le specifiche e caratteristiche architetture del prodotto software sviluppato da *Bug's Bunny*.

1.2 Scopo del prodotto

Lo scopo del *Bug's Bunny* e dell'azienda Zero12 è la creazione di un applicativo software (WebApp)_G in grado di analizzare e classificare molteplici contenuti digitali, creati e condivisi dagli utenti sulla piattaforma social Instagram, in base alle reazioni e alle impressioni ricavabili dal contenuto di essi. La WebApp_G deve poter fornire una guida per ogni luogo di ristorazione, basandosi sulla classificazione precedente. E' possibile, quindi, fare una classifica di questi luoghi grazie al contributo degli utenti.

1.3 Glossario

Per maggiore chiarezza del lessico usato, è stato creato un glossario, il quale contiene spiegazioni dei termini più importanti che sono stati usati.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- **Capitolato d'appalto C4:**
<https://www.math.unipd.it/~tullio/IS-1/2021/Progetto/C4.pdf>

1.4.2 Riferimenti informativi

- **Slide dell'insegnamento di Ingegneria del Software: (Slide 12, 19)**
<https://www.math.unipd.it/~tullio/IS-1/2021/Dispense/PD2.pdf>
- **Slide Progettazione e programmazione: diagrammi delle classi (UML):**
https://www.math.unipd.it/~rcardin/swea/2021/Diagrammi%20delle%20Classi_4x4.pdf
- **Slide Progettazione: i pattern architetture:**
<https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf>
- **Slide Progettazione: design pattern creazionali:**
<https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf>
- **Slide Progettazione: design pattern strutturali:**
<https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf>
- **Funzioni Lambda in Python**
<https://docs.aws.amazon.com/lambda/latest/dg/python-handler.html>



- **Overview AWS Step Functions**
<https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html#application>
- **Introduzione FastAPI**
<https://fastapi.tiangolo.com/tutorial/first-steps/>
- **Overview AWS Fargate**
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html
- **Evento *pre signup* Cognito:**
<https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-lambda-pre-signup.html>
- **Documentazione SQLAlchemy**
<https://docs.sqlalchemy.org/en/14/intro.html>
- **Migrazioni autogenerate Alembic**
<https://alembic.sqlalchemy.org/en/latest/autogenerate.html>
- **Analisi facciale Amazon Rekognition**
<https://docs.aws.amazon.com/rekognition/latest/dg/faces.html>
- **Sentiment analysis Amazon Comprehend**
<https://docs.aws.amazon.com/comprehend/latest/dg/how-sentiment.html>
- **Documentazione Svelte**
<https://svelte.dev/docs>

2 Architettura del sistema

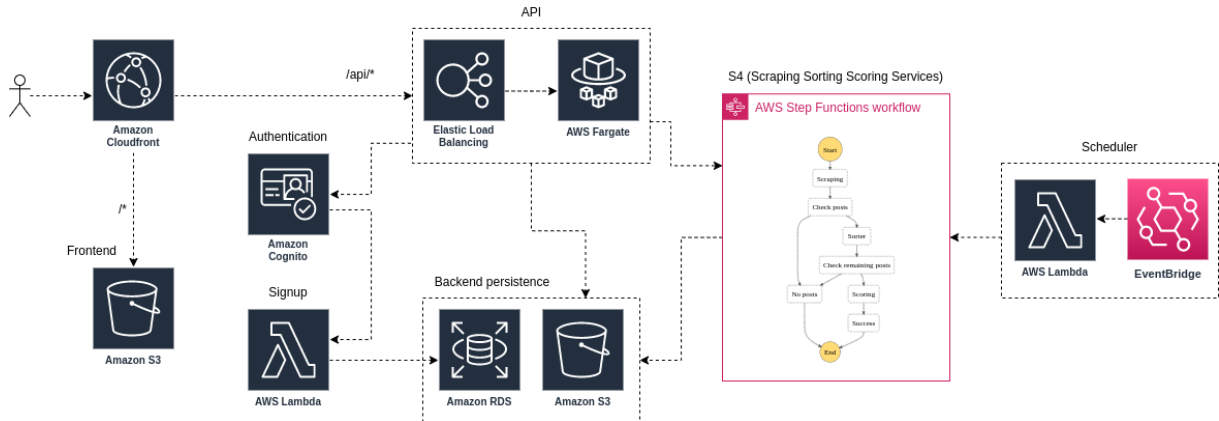


Figura 1: Visione d'insieme del sistema

2.1 Descrizione generale

Il Backend_G della piattaforma sfrutta un'architettura a microservizi, i quali comunicano fra di loro secondo regole e meccanismi appropriati. Le varie parti sono fra di loro indipendenti e i dati da esse prodotti persistono in Amazon RDS_G o bucket_G S3, a seconda della loro natura. Il lavoro di analisi e progettazione ha permesso di individuare i seguenti servizi:

- **API Service:** ha il compito di far interfacciare il Frontend_G con il resto del sistema, in particolare fornisce delle API_G RESTful che consentono l'ottenimento dei dati dal database e in generale di gestire tutte le funzionalità rese disponibili all'utente;
- **Signup Service:** risponde agli eventi *pre signup* di Amazon Cognito_G e permette di aggiungere automaticamente utenti al database della piattaforma, dopo la loro registrazione;
- **S4 (Scraping Sorting Scoring Services):**
 - **Scraping Service:** sfruttando apposite tecniche e librerie, effettua operazioni di scraping_G da Instagram_G al fine di ottenere dati che saranno successivamente processati dagli altri servizi;
 - **Sorting Service:** effettua un preprocessing sui dati ottenuti dallo scraping_G, scartando eventuali dati non conformi e che comporterebbero uno spreco di risorse nelle fasi successive di analisi;
 - **Scoring Service:** esegue analisi approfondite su dati testuali e multimediali, producendo una valutazione numerica;
- **Scheduler Service:** programma l'esecuzione di S4 secondo un intervallo di tempo predefinito.

2.2 Servizi S4

I servizi di scraping, sorting e scoring formano un raggruppamento logico chiamato **S4** (Scraping Sorting Scoring Services). Essi vengono eseguiti in funzioni Lambda, le quali sono orchestrate attraverso *AWS Step Functions*.

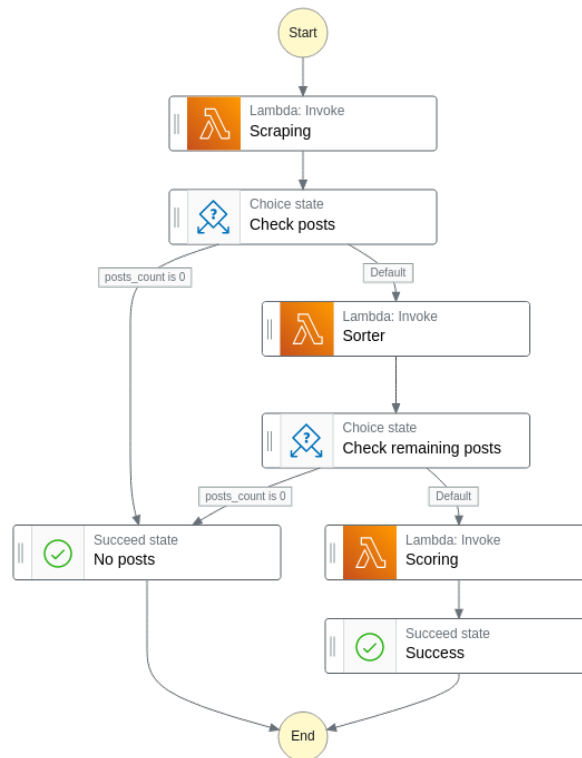


Figura 2: Visualizzazione macchina a stati

L'orchestratore permette di definire il flusso di esecuzione come una macchina a stati, dove l'output di uno stato diventa l'input di quello successivo.

Uno stato può anche controllare il flusso dell'esecuzione, per esempio se dopo l'operazione di sorting non restano post da analizzare e quindi da passare in input al servizio di scoring, si arriva subito ad uno stato terminale che quindi termina l'esecuzione senza sprecare risorse.

2.3 Database

La persistenza dei dati è affidata ad un database relazionale, su DBMS_G *PostgreSQL*. Il database è gestito dal servizio *Amazon RDS*_G.

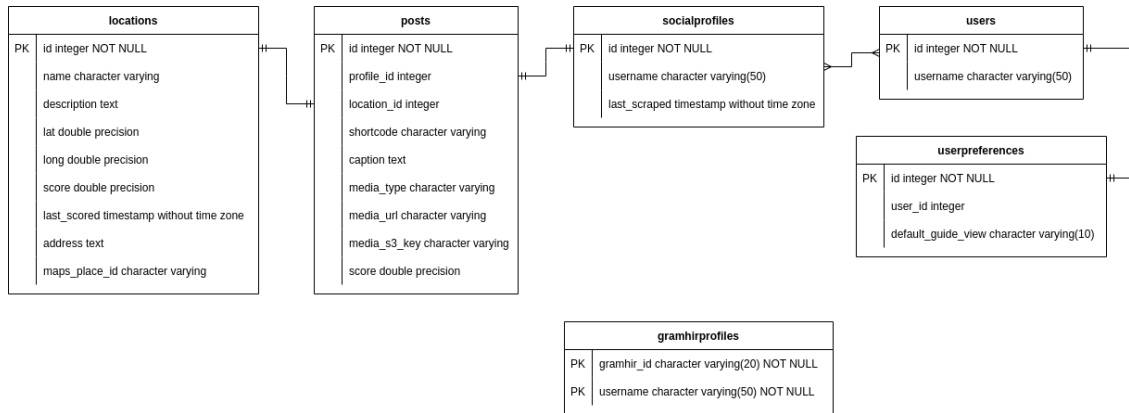


Figura 3: Schema ER database

2.3.1 Interfacciamento

L'interfacciamento con il database, da parte dei vari servizi, è affidato alla libreria Python_G *SQLAlchemy*. Essendo una libreria ORM_G, vengono definiti dei modelli che permettono l'interazione con le entità e le relazioni interne al database come dei normali oggetti.

2.3.2 Versionamento

Il versionamento della schema relazione del database è affidato allo strumento *alembic*, che in sintonia con la libreria sopracitata *SQLAlchemy*, gestisce in automatico i cambiamenti apportati allo schema. Questi cambiamenti vengono raggruppati in operazioni atomiche dette migrazioni.

```
1 """add location address and maps_place_id
2 Revision ID: 6ce8d444b850
3 Revises: a557f1403100
4 Create Date: 2022-09-01 10:43:05.074767
5 """
6
7 from alembic import op
8 import sqlalchemy as sa
9
10 # revision identifiers, used by Alembic.
11 revision = '6ce8d444b850'
12 down_revision = 'a557f1403100'
13 branch_labels = None
14 depends_on = None
15
16 def upgrade() -> None:
17     # ### commands auto generated by Alembic - please adjust! ###
18     op.add_column('locations', sa.Column('address', sa.Text(), nullable=True))
19     op.add_column('locations', sa.Column('maps_place_id', sa.String(), nullable=True))
20     # ### end Alembic commands ###
```



```
21
22 def downgrade() -> None:
23     # ### commands auto generated by Alembic - please adjust! ###
24     op.drop_column('locations', 'maps_place_id')
25     op.drop_column('locations', 'address')
26     # ### end Alembic commands ###
```

Listing 1: Esempio di migrazione autogenerata

3 Architettura dei componenti

3.1 API Service

3.1.1 Descrizione generale

Questo servizio svolge l'importante funzione di fornire delle API_G RESTful al Frontend_G della piattaforma. Queste permettono di svolgere le azioni richieste dall'utente, soprattutto l'ottenimento di dati dal database. Il servizio viene eseguito in ambiente container, gestito da AWS_G Fargate.

3.1.2 Diagramma delle classi

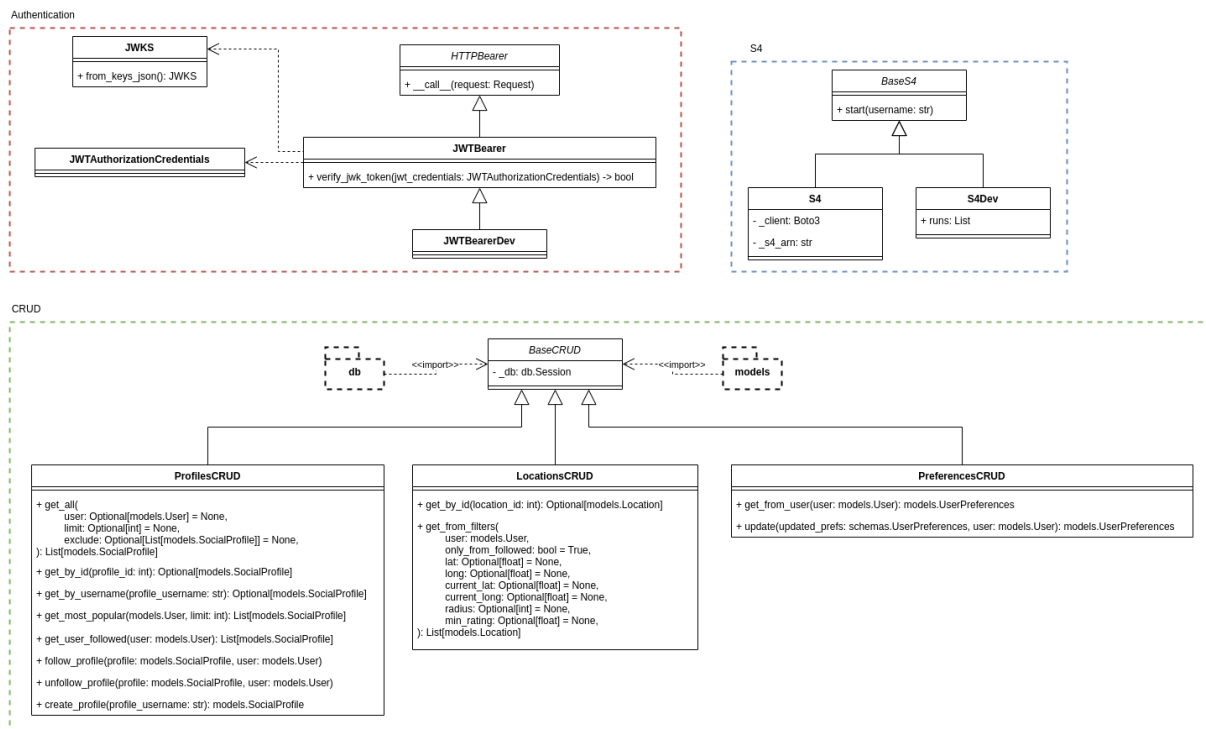


Figura 4: API Service - Diagramma delle classi

3.1.3 Implementazione

Per l'implementazione del servizio è stato scelto FastAPI, un framework_G Python_G per lo sviluppo di API_G.

Essendo Python_G un linguaggio multiparadigma, ed il framework_G non incentrato sul paradigma ad oggetti, non sono state definite gerarchie di classi particolarmente complesse. Degne di nota sono le classi derivanti da `BaseCRUD`, che si occupano di gestire operazioni di lettura e scrittura dal database, quindi buona parte della *business logic*.

La gestione degli *endpoint* delle API_G avviene tramite decorator a funzioni. I dati delle richieste, sia essi di percorso o nel *body* della richiesta stessa, vengono passati alla funzione come dei normali parametri grazie al funzionamento interno di FastAPI.

```
1 @router.get(  
2     '/locations/{location_id}',  
3     response_model=schemas.Location,
```

```
4     response_model_exclude_unset=True ,
5 )
6 def get_location(
7     location_id: int,
8     locations: LocationsCRUD = Depends(get_locations_crud),
9 ):
10     location = locations.get_by_id(location_id)
11     if not location:
12         raise HTTPException(status_code=HTTP_404_NOT_FOUND, detail='
13         Location not found')
```

Listing 2: Esempio di gestione di un *endpoint*

3.1.3.1 Documentazione automatica

Le API_G vengono documentate automaticamente dal framework_G, seguendo lo standard OpenAPI.

3.1.4 Autenticazione

L'autenticazione degli utenti avviene tramite JWT_G. Il token è memorizzato nell'header **HTTP Authentication**, e viene impostato dopo il login dall'interfaccia di Cognito_G. La verifica della validità avviene nella classe **JWTBearer**. Tale verifica avviene dopo ogni chiamata ad endpoint che necessitano di autenticazione.

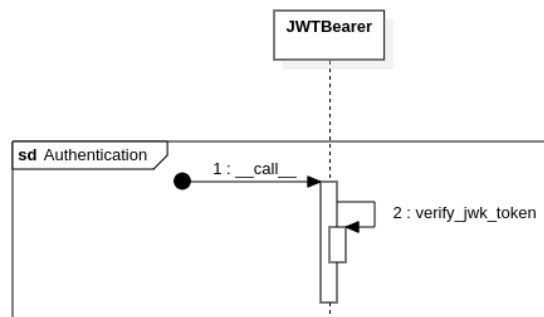


Figura 5: Autenticazione - Diagramma di sequenza

3.2 Signup Service

3.2.1 Descrizione generale

Questo servizio ha il compito di creare automaticamente, nel database della piattaforma, gli utenti che si registrano utilizzando l'interfaccia fornita da Cognito_G. Questo è necessario perchè gli utenti nella Cognito User Pool, normalmente, non hanno alcuna correlazione con quelli nel database. La funzione Lambda che esegue il codice il servizio, è invocata esternamente da un evento *pre signup*, inviato da Cognito_G quando un utente si registra.

3.2.2 Diagramma delle classi

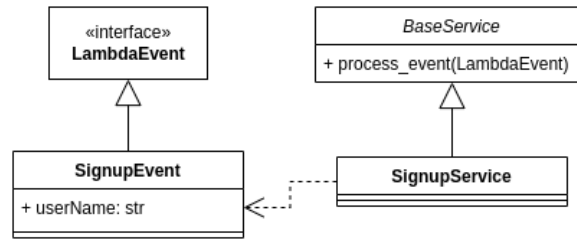


Figura 6: Signup Service - Diagramma delle classi

3.2.3 Schemi I/O

Input

Evento *pre signup* in format `JSONG`, come definito nella documentazione di `CognitoG`.

Output

Evento *pre signup* in format `JSONG` (lo stesso ottenuto in input).

3.3 Scraping Service

3.3.1 Descrizione generale

Questo servizio implementa le funzionalità di `scrapingG` ed in generale ottenimento dati da `InstagramG` e altre piattaforme ad esso correlate.

3.3.2 Diagramma delle classi

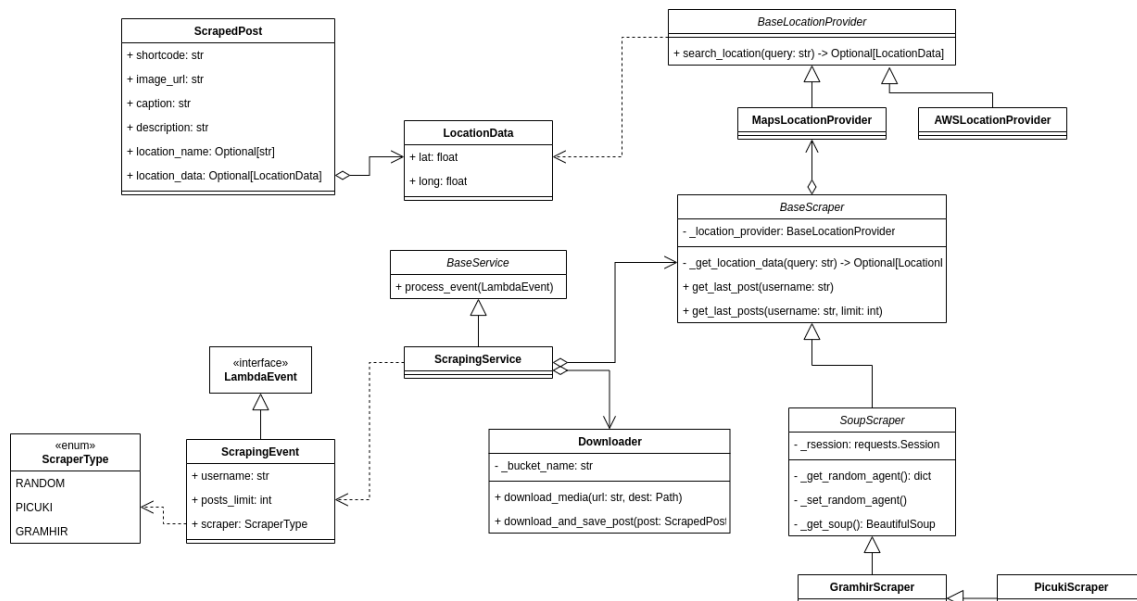


Figura 7: Scraping Service - Diagramma delle classi

3.3.3 Diagramma di sequenza

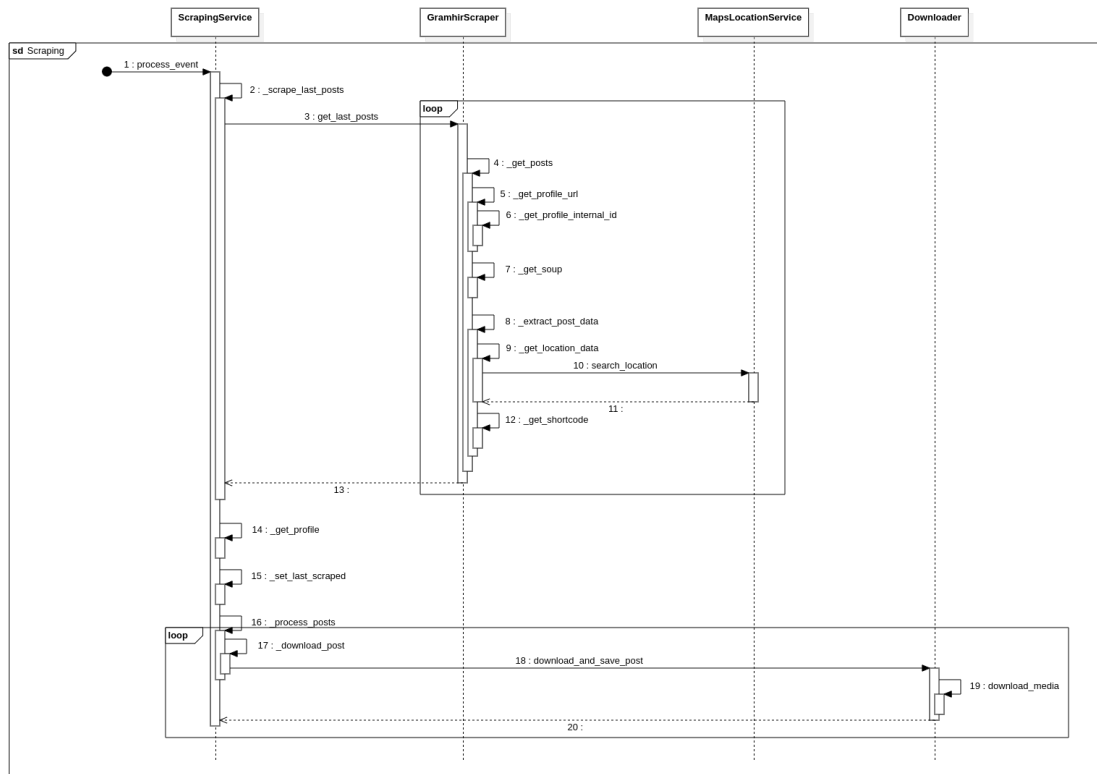


Figura 8: Scraping Service - Diagramma di sequenza

3.3.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

```
1 {
2   "username": "antoniorazzi",
3   "posts_limit": 10
4 }
```

Descrizione:

- **username**: username del profilo social da cui effettuare lo scraping_G;
- **posts_limit**: numero massimo di post di cui effettuare lo scraping_G.

Output esempio di risposta in output, in formato JSON_G.

```
1 {
2   "posts_count": 2,
3   "posts": [
4     { "id": 4 },
5     { "id": 5 }
6   ]
7 }
```

Descrizione:

- **posts_count**: numero di post ottenuti;
- **posts**: array di post.

3.3.5 Strategie di scraping

Le azioni di `scrapingG` non avvengono direttamente su `InstagramG`, questo perchè comporterebbe il rischio di incorrere in *rate limiting* o addirittura l'impossibilità totale di usufruire del servizio dall'infrastruttura cloud di `AWSG`. Per aggirare queste problematiche, lo `scrapingG` avviene da piattaforme chiamate *viewer*, che propongono in larga parte gli stessi contenuti di `InstagramG`, ma in un formato molto più semplice da trattare automaticamente.

Le classi `SoupScraper` e più in generale `BaseScraper`, si occupano di fornire interfacce comuni per gli `scraperG` che andranno implementati. Le classi `GramhirScraper` e `PicukiScraper` si occupano di effettuare lo scraping rispettivamente dai viewer *Gramhir* (<https://www.gramhir.com>) e *Picuki* (<https://www.picuki.com>). Gli algoritmi e le tecniche utilizzate dalle due classi, sono sostanzialmente identiche. Infatti, `PicukiScraper` eredita direttamente da `GramhirScraper`. Il funzionamento interno delle due piattaforme differisce solo nella necessità di ottenere un *ID* correlato al profilo social che si vuole visualizzare, necessario per *Gramhir*.

```
1 def _get_profile_url(self, username: str) -> str:
2     gramhir_id = self._get_profile_internal_id(username)
3     profile_url = self._PROFILE_URL.format(username=username, gramhir_id=gramhir_id)
4     return profile_url
```

Listing 3: Ottenimento URL profilo in *Gramhir*

```
1 def _get_profile_url(self, username: str) -> str:
2     return self._PROFILE_URL.format(username=username)
```

Listing 4: Ottenimento URL profilo in *Picuki*

Lo scraping vero e proprio avviene tramite `parsingG` del contenuto `HTMLG` delle pagine web fornite dai *viewer*.

```
1 def _extract_post_data(self, post_result: Tag) -> ScrapedPost:
2     location_name = post_result.find(attrs={'class': 'photo-location'}).get_text(
3         strip=True
4     )
5     location_name = location_name if location_name else None
6
7     location_data = None
8     if location_name:
9         location_data = self._get_location_data(location_name)
10
11     details_url = post_result.find('a').get('href')
12     shortcode = self._get_shortcode(details_url)
13
14     return ScrapedPost(
15         image_url=post_result.find('img').get('src'),
16         caption=post_result.find(attrs={'class': 'photo-description'}).get_text(
17             strip=True
18         ),
19         description='',
20         location_name=location_name,
```



```
21     location_data=location_data ,
22     shortcode=shortcode ,
23 )
```

Listing 5: Estrazioni dati tramite parsing_G

3.3.6 Ottenimento informazioni relative alle posizioni

Correlare i post ottenuto a precise posizioni reali è necessario ai fini del progetto. I dati relativi alla posizione, nelle piattaforme sopracitate, comprendono solo il nome del particolare punto di interesse, non le coordinate esatte in latitudine e longitudine. Per estrapolare questi due valori, viene utilizzato il servizio *Google Maps API_G*.

La classe `MapsLocationProvider`, che concretizza `BaseLocationProvider`, ha il compito di ottenere informazioni specifiche relative ad un punto di interesse, a partire dal nome.

```
1 def search_location(self, query: str) -> Optional[LocationData]:
2     geocode_result = self._maps.geocode(query)
3
4     if not geocode_result:
5         return None
6
7     geocode_result = geocode_result[0]
8     coords = geocode_result['geometry']['location']
9     maps_place_id = geocode_result.get('place_id')
10
11     place = self._maps.place(place_id=maps_place_id)
12     address = place['result']['formatted_address']
13     name = place['result']['name']
14     types = place['result']['types']
15
16     return LocationData(
17         lat=coords['lat'],
18         long=coords['lng'],
19         address=address,
20         maps_name=name,
21         maps_place_id=maps_place_id,
22         types=types
23 )
```

Listing 6: Ottenimento informazioni da *Google Maps API*

3.3.7 Salvataggio dei dati

I dati relativi ai post ottenuti e le relative posizioni, vengono salvati nel database. Le immagini o altri contenuti multimediali vengono salvati nell'apposito bucket_G in S3.

3.4 Sorting Service

3.4.1 Descrizione generale

La classe `SorterService` si occupa del servizio di sorting di uno o più post: innanzitutto fornisce l'entry point per sfruttare il servizio Amazon AWS Rekognition_G, dopo di che sono implementate tutte le funzioni contenenti l'effettiva algoritmica di sorting per il calcolo del punteggio del post dalle sue immagini.

Più nello specifico, `SorterService` implementa:



```
1 def __init__(self):
2     self._rekognition = boto3.client(
3         service_name='rekognition',
4         region_name=aws_region
5     )
```

Che si occupa di fornire un handle sotto forma di oggetto per accedere al servizio AWS Rekognition (`self._rekognition`). Poi definisce e implementa le seguenti funzioni:

- `def process_event(self, event: SortEvent) -> dict:`

Entry point per la funzione di sorting; si occupa di lanciare il servizio di sorting sui post salvati. Ritorna infine una lista con tutti i post validi;

- `def sort(self, post: SortingPost) -> Optional[SortingPost]:`

Chiama una funzione per calcolare lo score di un post dalle sue immagini. Se è possibile calcolare lo score di un post esso viene ritornato a `process_event`; Se invece non è possibile viene evocato la funzione `_delete_post` a cui viene passato il post da eliminare;

- `def _delete_post(self, post: SortingPost):`

Si occupa di eliminare il post inutilizzabile e di rimuovere le sua immagini dal bucket_G di S3 su cui erano state salvate;

- `def calculate_image_score(self, post: SortingPost):`

Si occupa inizialmente di calcolare lo score di ogni singola immagine di un post e dopo di che calcola lo score finale dell'insieme di immagini del post;

- `def analyze_image(self, name_image: str):`

Verifica che siano presenti persone nella foto (tramite `detect_person`) e ne estrapola le emozioni (servendosi di `detect_sentiment_person`). Ritorna, se presenti, un dizionario di emozioni e un dizionario dei rispettivi valori numerici;

- `def detect_sentiment_person(self, name_image: str):`

Utilizza il servizio AWS Rekognition_G per riconoscere le facce ed emozioni correlate in un immagine. Controllerà che le emozioni siano valide (devono avere un valore pari o superiore al 90% per essere considerate influenti sullo score finale) e verrà poi creato e ritornato a `analyze_image` un dizionario dei sentimenti, un dizionario dei rispettivi valori numerici e un booleano per segnalare di aver trovato o meno emozioni;

- `def detect_person(self, name_image: str):`

Utilizza il servizio di AWS Rekognition_G per analizzare l'immagine e riconoscere delle labels; queste labels verranno controllate per verificare la presenza di persone. Verrà ritornato un booleano per segnalare la presenza o meno di persone.



3.4.2 Diagramma delle classi

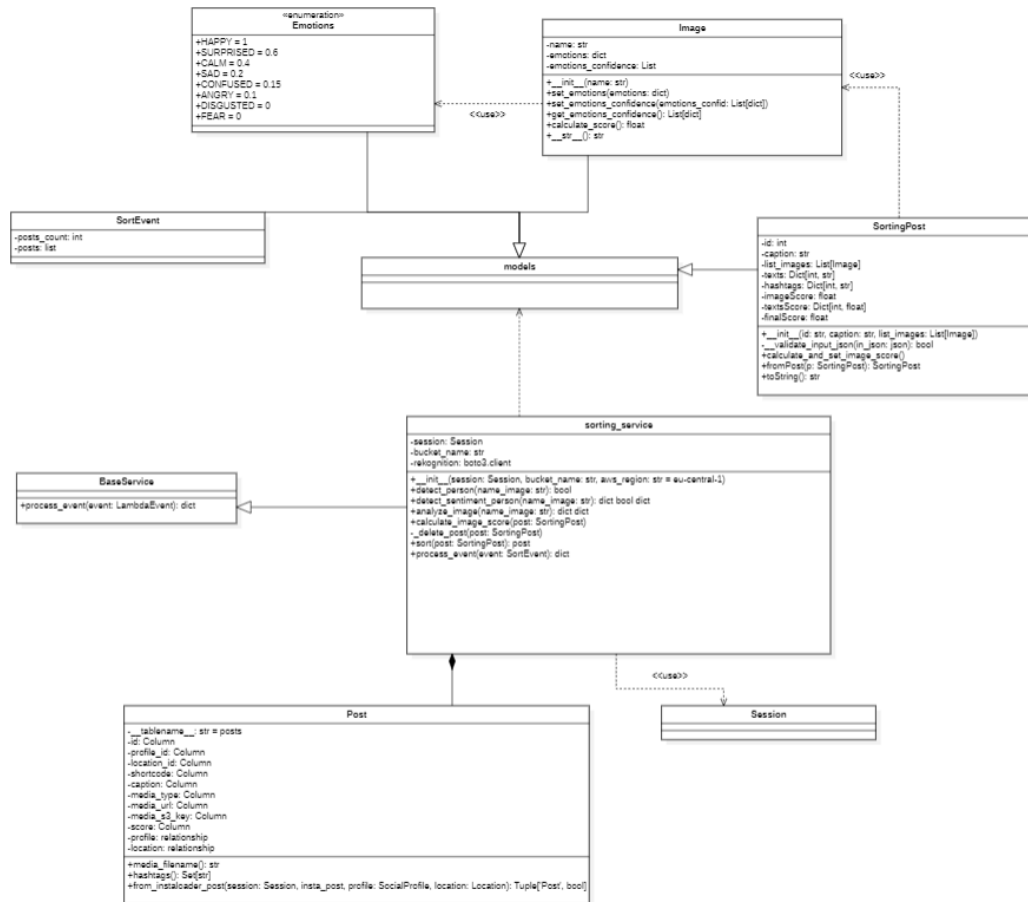


Figura 9: Sorting Service - Diagramma delle classi

3.4.3 Diagramma di sequenza

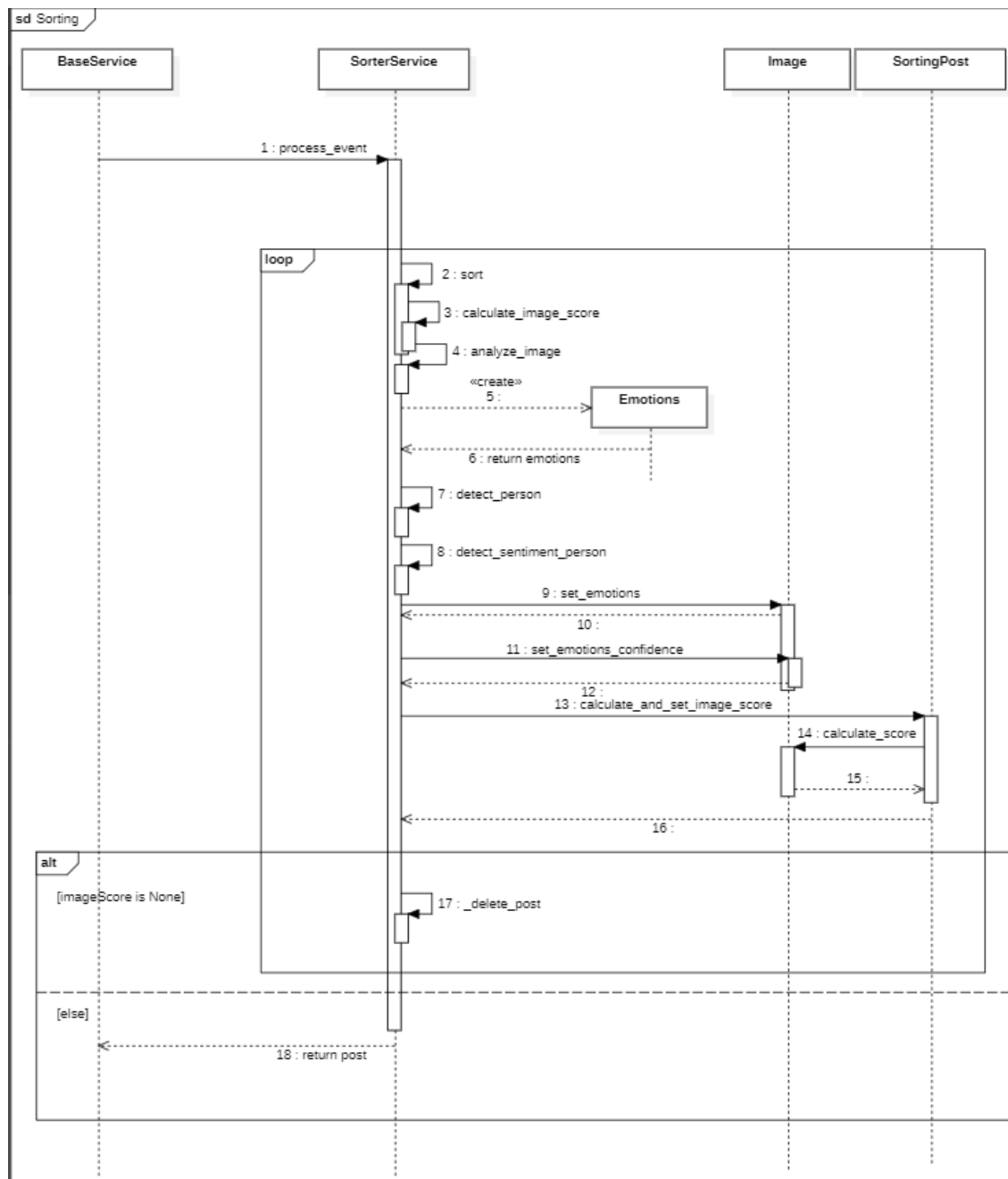


Figura 10: Sorting Service - Diagramma di sequenza

3.4.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

```
1 {
2   "posts_count": 4,
3   "posts": [
4     {"id": 1},
5     {"id": 2},
6     {"id": 3},
```



```
7     {"id": 4}
8 ]
9 }
```

Descrizione:

- `posts_count`: numero di post di cui effettuare il sorting;
- `posts`: array di post.

Output esempio di risposta in output, in formato JSON_G.

```
1 {
2   "posts_count": 2,
3   "posts": [
4     {"id": 2},
5     {"id": 4}
6   ]
7 }
```

Descrizione:

- `posts_count`: numero di post tenuto dopo il sorting;
- `posts`: array di post.

3.4.5 Calcolo dell'imageUrl

Segue una spiegazione più dettagliata del processo per il calcolo di imageUrl necessario al sort dei post.

3.4.5.1 Inizio processo

Il processo inizia con `process_event(self, event: SortEvent) -> dict` che si occupa di prendere dal database i post sui quali chiamerà la funzione `sort`.

```
1 for p in event.posts:
2     db_post = self._session.query(models.Post).filter_by(id=p['id']).
3         first()
4     sorting_post = SortingPost.fromPost(db_post)
5     sorting_post = self.sort(sorting_post)
```

Essa chiamerà a sua volta `calculate_image_score` che con un dizionario di emozioni e un dizionario sulla confidenza di tali emozioni permette il calcolo di imageUrl.

```
1 self.calculate_image_score(post)

1 image.set_emotions(emotions)
2 image.set_emotions_confidence(emotions_confidence)
3
4 post.calculate_and_set_image_score()
```



3.4.5.2 Creazione e uso dei dizionari

Per creare tale dizionari si utilizza la funzione `detect_sentiment_person` che utilizzerà a sua volta il servizio AWS Rekognition per il riconoscimento delle facce ed eseguirà poi un analisi sulla risposta ritornata dal servizio. Attraverso tale analisi si salveranno solo le emozioni valide ossia con una confidenza pari o superiore al 90%. Verranno ritornati i dizionari a `calculate_image_score` dove verranno usati per salvare le emozioni valide nell'oggetto "Image" nel seguente modo:

```
1 def set_emotions(self, emotions: dict):
2     self.emotions = emotions
3
4 def set_emotions_confidence(self, emotions_confid: List[dict]):
5     self.emotions_confidence = emotions_confid
```

3.4.5.3 Calcolo finale

Il calcolo finale di `imageScore` avviene con nell'algoritmo di seguito riportato, identificato come `calculate_and_set_image_score`:

```
1 if self.list_images:
2     count = 0
3     for image in self.list_images:
4         score = image.calculate_score()
5         if score is not None:
6             count += 1
7             self.imageScore = (
8                 (self.imageScore + score)
9                 if self.imageScore is not None
10                else score
11            )
12     if self.imageScore:
13         self.imageScore = self.imageScore / count
```

Dove lo score delle singole immagini viene calcolato da `calculate_score`:

```
1 if self.emotions:
2     value_sum = 0.0
3     sum_weights = 0.0
4     for emotion, num in self.emotions.items():
5         value_sum += Emotions[emotion].value * num
6         sum_weights += num
7     return 100 * value_sum / sum_weights
8 else:
9     return None
```

3.4.5.4 Conclusione processo

Se l'`imageScore` sarà stato calcolato il post è valido e verrà tenuto; se invece non è stato calcolato significa che il post non è valido e sarà eliminato insieme alle sua immagini salvate su un bucket_G S3 tramite la funzione `_delete_post`:

```
1 for img in post.list_images:
2     s3_delete_file(self._bucket_name, img.name)
3 self._session.delete(
4     self._session.query(models.Post).filter_by(id=post.id).first()
5 )
6 self._session.commit()
```

3.4.6 Variabili

Segue una spiegazione più dettagliata di alcune variabili usate dagli algoritmi spiegati precedentemente:

3.4.6.1 Response from detect_faces

```
1 response = self._rekognition.detect_faces(  
2     Image={'S3Object': {'Bucket': self._bucket_name, 'Name':  
3         name_image}},  
4     Attributes=['ALL'],  
5 )
```

La variabile `response` riceve il risultato della funzione `detect_faces` del servizio AWS Rekognition_G; tale funzione prende in input un'immagine presente in un bucket_G di S3 e ritorna come risultato un file JSON_G contenente una sentiment analysis dei volti presenti nell'immagine. Segue un esempio ove presenti solo i dati necessari al nostro caso:

```
1 {  
2     "FaceDetails": [  
3         {  
4             ....  
5             ....  
6             "Emotions": [  
7                 {  
8                     "Type": "ANGRY",  
9                     "Confidence": 55.18563461303711  
10                },  
11                {  
12                    "Type": "HAPPY",  
13                    "Confidence": 37.01131820678711  
14                }, {}, {}, {}, {}, {}, {}  
15            ],  
16            ....  
17            ....  
18            ....  
19            ....  
20        }  
21    ]  
22 }
```

Tale risultato sarà usato dalla funzione `detect_sentiment_person` per creare un dizionario di emozioni e un dizionario sulla confidenza di tali emozioni nel seguente modo:

```
1 contain_emotion = False  
2 emotions_dict = {}  
3 emotions_confid = []  
4  
5 for faceDetail in response['FaceDetails']:  
6     emotions = faceDetail['Emotions']  
7     confid_single_face = {}  
8     for emotion in emotions:  
9         emotion_name = emotion['Type']  
10        emotion_confid_value = emotion['Confidence']  
11        if emotion_name != 'UNKNOWN':  
12            confid_single_face[emotion_name] = emotion_confid_value  
13  
14        if emotion_confid_value >= 90:
```



```
15         if emotion_name in emotions_dict:
16             emotions_dict[emotion_name] += 1
17         else:
18             emotions_dict[emotion_name] = 1
19             contain_emotion = True
20     emotions_confid.append(confid_single_face)
21 return emotions_dict, contain_emotion, emotions_confid
```

3.4.6.2 Response from detect_labels

```
1 response = self._rekognition.detect_labels(
2     Image={'S3Object': {'Bucket': self._bucket_name, 'Name':
3         name_image}},
4     MaxLabels=10,
```

La variabile `response` riceve il risultato della funzione `detect_labels` del servizio AWS Rekognition_G; tale funzione prende in input un'immagine presente in un bucket_G di S3 e ritorna come risultato un file JSON_G contenente una labels analysis delle istanze di entità del mondo reale presenti nell'immagine. Segue un esempio ove presenti solo i dati necessari al nostro caso:

```
1 {
2     "Labels": [
3         {
4             "Name": "Person",
5             "Confidence": 99.88428497314453,
6             "Instances": [
7                 {
8                     "BoundingBox": {
9                         "Width": 0.27858132123947144,
10                        "Height": 0.44550982117652893,
11                        "Left": 0.2236727923154831,
12                        "Top": 0.35303789377212524
13                    },
14                    "Confidence": 99.88428497314453
15                },
16                {
17                    "BoundingBox": {
18                        "Width": 0.20297367870807648,
19                        "Height": 0.6778767108917236,
20                        "Left": 0.5247262716293335,
21                        "Top": 0.10922279208898544
22                    },
23                    "Confidence": 99.81658172607422
24                }
25            ]
26        },
27        ....
28        ....
29        ....
30        ....
31    ]
32 }
```

Tale risultato sarà usato dalla funzione `detect_person` per verificare la presenza di persone nel seguente modo:


```
1 contain_person = False
2
3 for label in response['Labels']:
4     if label['Confidence'] >= 90:
5         if label['Name'] == 'Person':
6             contain_person = True
7
8 return contain_person
```

In questo nostro esempio possiamo vedere, nel file `JSONG`, che sono presenti due persone indicate dal parametro "BoundingBox". La funzione sopra vedrà la presenza di queste due persone e tornerà un risultato positivo alla funzione `analyze_image` che quindi saprà di poter chiamare la funzione `detect_sentiment_person` il cui scopo è descritto sopra.

3.5 Scoring Service

3.5.1 Descrizione generale

Le classi `ScoringService` e `BasicScoringService` (che estende `ScoringService`) si occupano del servizio di scoring di un post: `ScoringService` fornisce gli entry point per sfruttare i servizi Amazon AWS Comprehend_G e Rekognition_G, mentre è in `BasicScoringService` che sono implementate tutte le funzioni contenenti l'effettiva algoritmica di scoring e i servizi I/O.

Più nello specifico, `ScoringService` implementa:

```
1 def __init__(self):
2     self._rekognition = boto3.client(service_name='rekognition')
3     self._comprehend = boto3.client(service_name='comprehend')
```

Che si occupa di fornire degli handles sotto forma di oggetti per accedere ai servizi AWS Rekognition_G e Comprehend_G (`self._rekognition` e `self._comprehend`).

`BasicScoringService` invece definisce e implementa le seguenti funzioni:

- `def process_event(self, event: ScoringEvent) -> dict:`

Entry point per l'intera funzione di scoring, si occupa di lanciare tutti i servizi necessari a effettuare i vari scores e salva poi il risultato nel database. Ritorna infine una lista con tutti i post analizzati e i loro scores;

- `def _save_to_db(self, db_post: models.Post, scoring_post: ScoringPost):`

Salva nel database lo score assegnandolo ai relativi post;

- `def _runRekognition(self, sPost: ScoringPost):`

Lancia, tramite l'oggetto `_rekognition`, le funzioni `detect_text` e `detect_faces` che si occupano di ritornare dei file `JSONG` contenenti rispettivamente il testo a schermo (se presente) e una sentiment analysis dei volti (se presenti) dell'immagine di cui si vuole ottenere uno scoring. Dopodiché lancia `__parse_rekognition_response`;

- `def _runComprehend(self, sPost: ScoringPost):`

Lancia, sfruttando l'oggetto `_comprehend` le funzioni `detect_dominant_language` e `batch_detect_sentiment`, che si occupano rispettivamente di ritornare la lingua dominante di un documento e fornire una sentiment analysis di una lista di testi. Dopodiché lancia `__parse_comprehend_response`;



- `def _calcFinalScore(self, sPost: ScoringPost):`

A seconda di quali scores siano presenti (caption, volti e testo a schermo), calcola lo score del post e lo salva in `sPost.finalScore`;

- `def __parse_rekognition_response(self, sPost: ScoringPost, textResult, faceResult):`

Prende in input due file JSON_G (`textResult` e `faceResult`), fa lo scoring dei volti ed estrapola da `textResult` (che contiene il testo rilevato dall'immagine) solo gli elementi di tipo "LINE", cioè quelli che contengono effettivamente il risultato voluto. Salva poi i risultati in `sPost.texts` e `sPost.faceScore`;

- `def __parse_comprehend_response(self, sPost: ScoringPost, compResult):`

Prende in input un file JSON_G contenente i risultati delle sentiment analysis effettuate sulla caption e sul testo a schermo e ne effettua uno scoring che viene poi salvato in `sPost.captionScore` e `sPost.textsScore` (quest'ultima è in realtà una lista di scores dei vari frammenti di testo rilevati);

- `def __parse_dominant_language_response(self, domResponse):`

Ritorna il codice della lingua dominante, oppure 'en' di default (cioè inglese);

- `def __unpack_post_for_comprehend(self, sPost: ScoringPost)`

Prepara il testo da fornire a Comprehend in modo che sia correttamente analizzato.

In linea generale lo scoring funziona come segue: `process_event` chiama le funzioni `_runRekognition`, `_runComprehend`, `_calcFinalScore` e `_save_to_db`.

`_runRekognition` riconosce il testo a schermo ed effettua uno scoring dei volti;

`_runComprehend` effettua uno scoring del testo a schermo e della caption;

`_calcFinalScore` somma, secondo un certo criterio, i vari scores per ottenere lo score finale del post, che viene quindi salvato nel corrispettivo post nel database da `_save_to_db`.



3.5.2 Diagramma delle classi

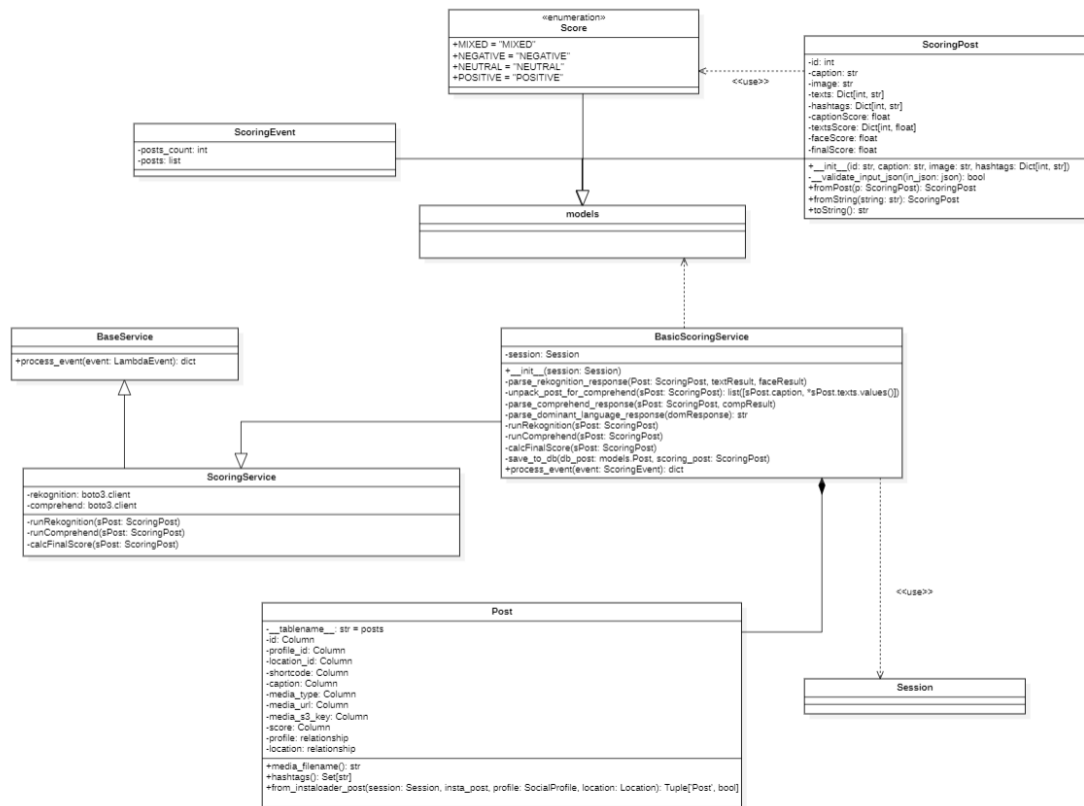


Figura 11: Scoring Service - Diagramma delle classi

3.5.3 Diagramma di sequenza

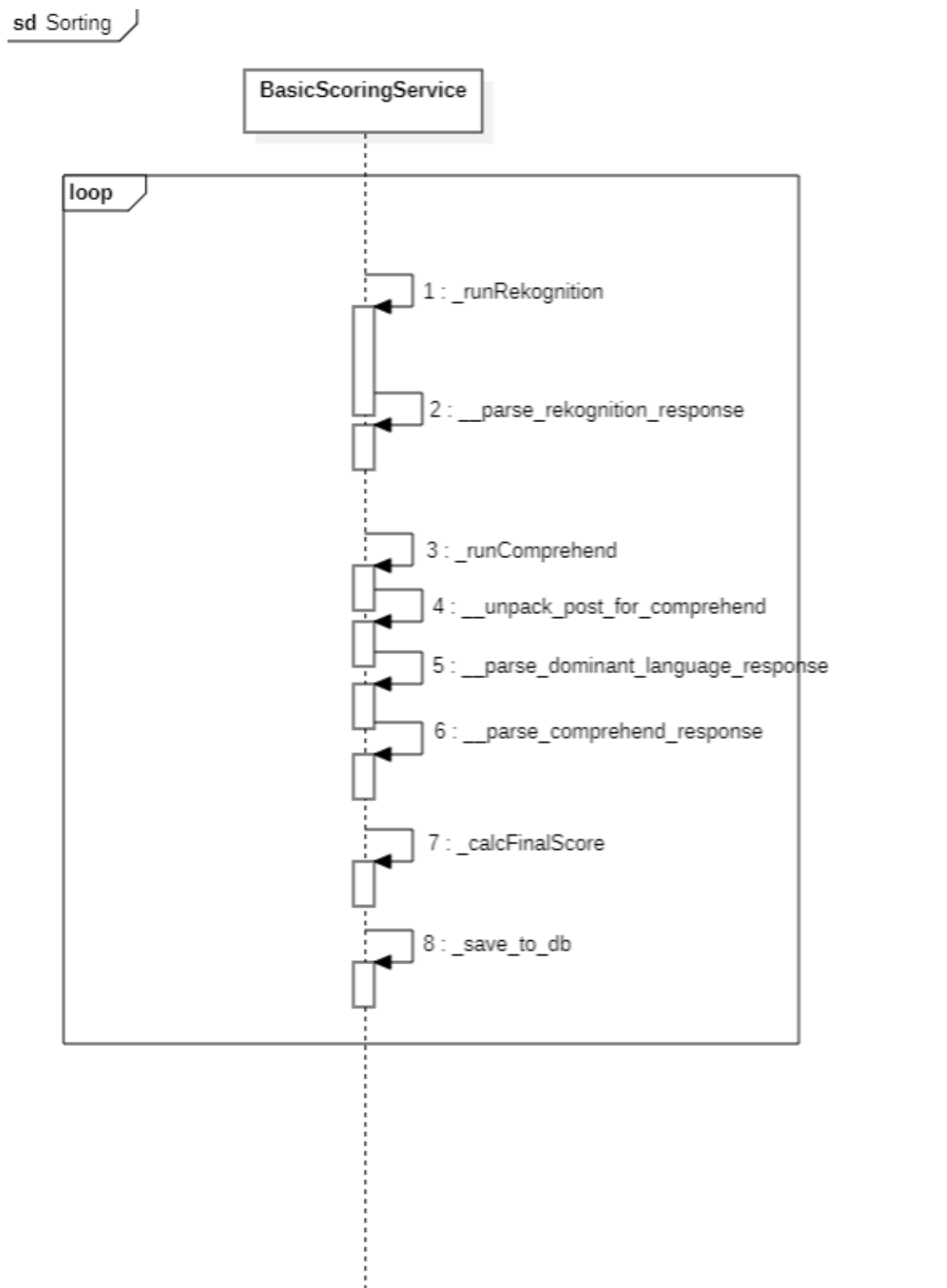


Figura 12: Scoring Service - Diagramma delle classi

3.5.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

```
1 {
2   "posts_count": 2,
3   "posts": [
4     {"id": 4},
5     {"id": 5}
6   ]
7 }
```

Descrizione:

- `posts_count`: numero di post di cui effettuare lo scoring;
- `posts`: array di post.

Output esempio di risposta in output, in formato JSON_G.

```
1 {
2   "scored_posts_count": 8,
3   "scored_posts": [
4     {
5       "id": 4,
6       "score": 4.830453701069928
7     },
8     {
9       "id": 5,
10      "score": 2.562992335879244
11    },
12  ]
13 }
```

Descrizione:

- `scored_posts_count`: numero di post di cui è stato effettuato lo score;
- `scored_posts`: array di post con relativo score ottenuto.

3.5.5 Calcolo degli Scores

Segue una spiegazione più dettagliata degli algoritmi e delle funzioni che si occupano di calcolare i quattro scores necessari (`faceScore`, `textScore`, `captionScore`, `finalScore`).

3.5.5.1 faceScore

Il punto di partenza si trova nella funzione `BasicScoringService._runRekognition`:

```
1 Image = {
2   'S3Object': {
3     'Bucket': os.environ['ENV_BUCKET_NAME'],
4     'Name': sPost.image,
5   }
6 }
7 textResponse = self._rekognition.detect_text(Image=Image)
8 faceResponse = self._rekognition.detect_faces(Image=Image)
9 BasicScoringService._parse_rekognition_response(
10   sPost, textResponse, faceResponse
11 )
```

La variabile `faceResponse` contiene il risultato della funzionalità di Rekognition_G chiamata `detect_faces` che prende in input un'immagine contenuta in un bucket_G S3 e ritorna un file JSON_G contenente una sentiment analysis dei vari volti presenti.

Segue un esempio del file in questione (lasciando implicito ciò che non serve):

```
1 {
2   "FaceDetails": [
3     {
4       "BoundingBox": {},
5       "AgeRange": {},
6       "Smile": {},
7       "Eyeglasses": {},
8       "Sunglasses": {},
9       "Gender": {},
10      "Beard": {},
11      "Mustache": {},
12      "EyesOpen": {},
13      "MouthOpen": {},
14      "Emotions": [
15        {
16          "Type": "ANGRY",
17          "Confidence": 55.18563461303711
18        },
19        {
20          "Type": "HAPPY",
21          "Confidence": 37.01131820678711
22        }, {}, {}, {}, {}, {}
23      ],
24      "Landmarks": [],
25      "Pose": {
26        "Roll": 3.602341890335083,
27        "Yaw": -82.46586608886719,
28        "Pitch": -18.774751663208008
29      },
30      "Quality": {
31        "Brightness": 92.66178894042969,
32        "Sharpness": 9.912903785705566
33      },
34      "Confidence": 99.85897064208984
35    }
36  }
```

Il risultato (contenuto in `faceResponse`), viene quindi passato come parametro alla funzione `BasicScoringService._parse_rekognition_response` che si occupa di effettuare il vero e proprio scoring.

I valori (contenuti in `FaceDetails`) che ci interessano sono:

- Emotions:
 - HAPPY
 - CALM
 - DISGUSTED
- Pose:
 - Yaw



– Pitch

Quality non è necessario poiché, tramite testing, abbiamo notato che immagini con valori di Sharpness anche molto bassi vengono analizzate senza alcun problema. L'algoritmo perde precisione solo nel caso in cui i volti siano ripresi da angolazioni troppo elevate (ad esempio da di lato).

Per quanto riguarda il campo Confidence delle Emotions può essere letto come una "quantità di emozione" compresa tra 0 e 100 (la somma di tutti i valori di Confidence è uguale a 100): in pratica una persona completamente felice avrà una Confidence dell'emozione HAPPY pari a 100 (e tutte le altre pari a 0), mentre una persona completamente arrabbiata avrà una Confidence dell'emozione ANGRY pari a 100 (e tutte le altre pari a 0) e così via.

Segue la formula dello score dei volti in linguaggio semi-naturale:

```
1 loop for all volti{
2     if(volto dritto){
3         score_volto = confidence_HAPPY + (confidence_CALM * 0.5)
4         if(confidence_DISGUSTED too high){
5             score_volto = 0
6         }
7     }
8 }
9 score_immagine_finale = somma_score_volto_dritti / numero_volto_dritti
```

Segue ora l'algoritmo:

```
1 faceCount = 0
2 scoreSum = 0
3 if len(faceResult['FaceDetails']) > 0:
4     for face in faceResult['FaceDetails']:
5         pose = face['Pose']
6         # SE VOLTO DRITTO
7         if (abs(pose['Yaw']) <= 50) and (
8             abs(pose['Pitch']) <= 50
9         ): # abs() perche' deve essere -50<pose<50
10            # CALCOLARE SCORE EMOZIONI
11            faceCount = faceCount + 1
12            faceSum = 0
13            disgusted = False
14            for emotion in face['Emotions']:
15                if emotion['Type'] == 'HAPPY':
16                    faceSum = faceSum + emotion['Confidence']
17                if emotion['Type'] == 'CALM':
18                    faceSum = (
19                        faceSum + emotion['Confidence'] * 0.5
20                    ) # ha peso minore di happy
21                if emotion['Type'] == 'DISGUSTED':
22                    if (
23                        emotion['Confidence'] >= 50
24                    ): # se disgust troppo elevato azzera il punteggio
25                        # della faccia
26                        disgusted = True
27                if not disgusted:
28                    scoreSum = (
29                        scoreSum + faceSum
30                    )
31            # UN VOLTO STORTO VIENE IGNORATO NEL CALCOLO
32            faceScore = scoreSum / faceCount # =[0,100]
```



```
33 sPost.faceScore = faceScore / 100 # normalizzato a [0,1]
34 else:
35     sPost.faceScore = None # se num facce =0 si ignora nel calcolo
36                             # di final Score
```

Il motivo per il quale CALM viene moltiplicato *0.5 è per dargli, nel calcolo del punteggio, un peso pari alla metà di HAPPY.

Infatti certamente non è un sentiment negativo, e va quindi valutato positivamente, però è anche vero che deve valere meno di HAPPY.

Il controllo riguardante l'angolazione del volto (cioè se il volto sia dritto o meno) viene effettuato dal codice:

```
1 if (abs(pose['Yaw']) <= 50) and (
2     abs(pose['Pitch']) <= 50
3 ):
```

Che controlla i valori d'imbardata e beccheggio del volto (in parole più semplici: Yaw è la rotazione della testa sull'asse perpendicolare al terreno, Pitch invece dell'asse parallelo al terreno e passante per le orecchie).

Si è scelto arbitrariamente il valore massimo di 50 e minimo di -50 per entrambi i valori al fine di evitare possibili errori: abbiamo riscontrato analisi del sentiment corrette anche con valori leggermente superiori, tuttavia è stata data maggiore priorità all'affidabilità dell'analisi.

La parte riguardante l'emozione DISGUSTED è stata inserita perché ci è sembrato limitante usare solo le emozioni HAPPY e CALM per dare una valutazione, soprattutto considerando che il tema generale del progetto riguarda ristoranti e locali. Abbiamo quindi ritenuto fondamentale dare il giusto peso a un'emozione così negativa.

Infatti un volto che abbia una valutazione di $\text{DISGUSTED} \geq 50$ riceve uno score pari a 0.

Senza questo controllo sarebbe teoricamente possibile che HAPPY fosse pari a 50 (contemporaneamente a $\text{DISGUSTED}=50$) e quindi che lo score del volto fosse 50 su 100.

```
1     if emotion['Type'] == 'DISGUSTED':
2         if (
3             emotion['Confidence'] >= 50
4         ):
5             disgusted = True
6 # end of for
7 if not disgusted:
8 scoreSum = (
9     scoreSum + faceSum
10 ) # se volto disgusted value >=50 allora face value = 0
```

Come si vede dal codice soprastante, durante il ciclo che "scorre" i vari volti presenti viene controllato il valore di DISGUSTED e, in caso sia superiore a 50, viene attivato un semaforo che impedisce di aggiungere lo score del volto alla somma generale (di fatto rendendo il valore di suddetto volto pari a zero).

Vediamo quindi la parte terminale dell'algoritmo, che si occupa di salvare il faceScore vero e proprio:

```
1 #if len(faceResult['FaceDetails']) > 0:
2 # corpo dell' algoritmo
3     faceScore = scoreSum / faceCount
4     sPost.faceScore = faceScore / 100 # normalizzato a [0,1]
5 else:
6     sPost.faceScore = None
```




Notiamo che `faceScore` viene calcolato come la media degli scores dei singoli volti, e viene poi normalizzato.

Infatti gli scores sono compresi tra 0 e 100, ma è comodo avere lo score finale compreso tra 0 e 1 (tornerà utile nel calcolo dello score finale).

Viene poi salvato in `sPost.faceScore`.

Se non sono presenti volti viene assegnato allo score il valore di controllo `None`.

3.5.5.2 textScore

Il punto di partenza si trova nella funzione `BasicScoringService._runRekognition`:

```
1 Image = {
2     'S3Object': {
3         'Bucket': os.environ['ENV_BUCKET_NAME'],
4         'Name': sPost.image,
5     }
6 }
7 textResponse = self._rekognition.detect_text(Image=Image)
8 faceResponse = self._rekognition.detect_faces(Image=Image)
9 BasicScoringService._parse_rekognition_response(
10     sPost, textResponse, faceResponse
11 )
```

La variabile `textResponse` contiene il risultato della funzionalità di Rekognition_G chiamata `detect_text` che prende in input un'immagine contenuta in un bucket_G S3 e ritorna un file JSON_G contenente il testo presente nell'immagine.

Segue un esempio del file in questione (lasciando implicito ciò che non serve):

```
1 {
2     "TextDetections": [
3         {
4             "DetectedText": "PIZZA LAB",
5             "Type": "LINE",
6             "Id": 0,
7             "Confidence": 99.8796615600586,
8             "Geometry": {}
9         },
10        {
11            "DetectedText": "Buon Ferragasta",
12            "Type": "LINE",
13            "Id": 1,
14            "Confidence": 67.9867935180664,
15            "Geometry": {}
16        }, {}, {}, {}
17    ],
18    "TextModelVersion": "3.0"
19 }
```

Ciò che andremo a considerare in questo caso sarà il contenuto di `DetectedText` per ogni elemento di tipo `LINE`.

Vediamo il codice:

```
1 for line in textResult['TextDetections']:
2     if line['Type'] == 'LINE':
3         sPost.texts[line['Id']] = line['DetectedText']
```

Ogni "pezzo" di testo rilevato nell'immagine viene salvato in un dizionario e viene identificato dal suo `Id`. Il dizionario in questione è `sPost.texts`.

Rispetto al JSON_G di esempio di cui sopra, il risultato di `sPost.texts` sarebbe il seguente:

```
1 {
2     0: 'PIZZA LAB',
3     1: 'Buon Ferragasta '
4 }
```

A questo punto è `ComprehendG` che deve occuparsi di dare degli effettivi sentiment scores a queste stringhe di testo.

Il punto di partenza si trova nella funzione `BasicScoringService._runComprehend`:

```
1 dominantLanguageResponse = self._comprehend.detect_dominant_language(
2     Text=sPost.caption
3 )
4 response = self._comprehend.batch_detect_sentiment(
5     TextList=BasicScoringService._unpack_post_for_comprehend(sPost),
6     LanguageCode=BasicScoringService._parse_dominant_language_response(
7         dominantLanguageResponse
8     ),
9 )
10 BasicScoringService._parse_comprehend_response(sPost, response)
```

Ciò che interessa a noi è la variabile `TextList`, ritornata dalla funzione `_unpack` (la variabile `LanguageCode` infatti è semplicemente una stringa che indica la lingua dominante della caption).

`TextList` infatti contiene sia la caption che la lista di testi rilevati a schermo, uniti in un'unica variabile dalla funzione `_unpack`:

```
1 def _unpack_post_for_comprehend(self, sPost: ScoringPost):
2     if not sPost.caption:
3         sPost.caption = ' '
4     return list([sPost.caption, *sPost.texts.values()])
```

Se la caption è un file vuoto la sostituisce con uno spazio, così da renderla accettabile in input da `ComprehendG`, e il risultato della funzione (cioè quindi ciò su cui viene lanciata la funzione di sentiment detection di `ComprehendG`) è il seguente (sempre prendendo ad esempio i file di cui sopra):

```
1 TextList = [
2     'Caption',
3     'PIZZA LAB',
4     'Buon Ferragasta '
5 ]
```

`TextList` viene quindi inoltrata a `ComprehendG` tramite l'oggetto `_comprehend` e la sua funzione `batch_detect_sentiment`, che ritorna nella variabile `response` un file JSON_G contenente la sentiment analysis per i vari pezzi di testo.

L'output di `ComprehendG` è strutturato come segue:

```
1 {
2     'ResultList': [
3         {
4             'Index': 0,
5             'Sentiment': 'POSITIVE',
6             'SentimentScore': {
7                 'Positive': 0.9763978719711304,
8                 'Negative': 9.167650568997487e-05,
9                 'Neutral': 0.02348191849887371,
10                'Mixed': 2.8541926440084353e-05
11            }
12        }, {}, {}
13     ]
14 }
```

```
13 ],
14 'ErrorList': [],
15 'ResponseMetadata': {}
16 }
```

Anche in questo caso lo scoring vero e proprio avviene nella funzione di parsing, vediamo quindi l'algoritmo di scoring:

```
1 for item in compResult['ResultList']:
2     idx = item['Index']
3     score = Score(item["Sentiment"])
4     float_score = (
5         item["SentimentScore"]["Positive"]
6         - item["SentimentScore"]["Negative"]
7         if (score != Score.MIXED)
8         else 0.0
9     )
10    if idx == 0:
11        sPost.captionScore = float_score
12    else:
13        sPost.textsScore[idx - 1] = float_score
```

La parte che si occupa di effettuare lo score vero e proprio è `float_score`, che si limita a dare come score la differenza tra i sentiments "Positive" e "Negative", oppure ignora la score se il valore di sentiment "Mixed" è troppo elevato (questo serve a filtrare i dati, ignorando quelli non rilevanti).

L'algoritmo riconosce la differenza tra caption e text-on-screen grazie agli id dei vari risultati contenuti in `ResultList`: infatti ci premuriamo di fare in modo che all'id pari a zero ci sia sempre la sentiment analysis della caption, e per gli altri valori di id ci siano i vari pezzi che compongono il testo rilevato a schermo.

Il risultato di questo scoring è salvato come una lista di valori in `sPost.textsScore`.

3.5.5.3 captionScore

Segue gli stessi passi di `textsScore`. In `__parse_comprehend_response` abbiamo:

```
1 float_score = (
2     item["SentimentScore"]["Positive"]
3     - item["SentimentScore"]["Negative"]
4     if (score != Score.MIXED)
5     else 0.0
6 )
7 if idx == 0:
8     sPost.captionScore = float_score
9 else:
10    sPost.textsScore[idx - 1] = float_score
```

E lo score che viene calcolato per `idx==0` è lo score della caption, che viene salvato in `sPost.captionScore`.

Tuttavia, per comodità di lettura, riportiamo a seguito comunque la spiegazione.

Il punto di partenza si trova nella funzione `BasicScoringService._runComprehend`:

```
1 dominantLanguageResponse = self._comprehend.detect_dominant_language(
2     Text=sPost.caption
3 )
4 response = self._comprehend.batch_detect_sentiment(
5     TextList=BasicScoringService._unpack_post_for_comprehend(sPost),
```

```
6     LanguageCode=BasicScoringService.__parse_dominant_language_response(  
7         dominantLanguageResponse  
8     ),  
9 )  
10 BasicScoringService.__parse_comprehend_response(sPost, response)
```

Ciò che interessa a noi è la variabile `TextList`, ritornata dalla funzione `__unpack` (la variabile `LanguageCode` infatti è semplicemente una stringa che indica la lingua dominante della caption).

`TextList` contiene sia la caption che la lista di testi rilevati a schermo, uniti in un'unica variabile dalla funzione `__unpack`:

```
1 def __unpack_post_for_comprehend(self, sPost: ScoringPost):  
2     if not sPost.caption:  
3         sPost.caption = ' '  
4     return list([sPost.caption, *sPost.texts.values()])
```

Se la caption è un file vuoto la sostituisce con uno spazio, così da renderla accettabile in input da `ComprehendG`, e il risultato della funzione (cioè quindi ciò su cui viene lanciata la funzione di sentiment detection di `ComprehendG`) è il seguente (sempre prendendo ad esempio i file di cui sopra):

```
1 TextList = [  
2     'Caption',  
3     'PIZZA LAB',  
4     'Buon Ferragasta '  
5 ]
```

A questo punto `TextList` viene inoltrata a `ComprehendG` tramite l'oggetto `_comprehend` e la sua funzione `batch_detect_sentiment`, che ritorna nella variabile `response` un file `JSONG` contenente la sentiment analysis per i vari pezzi di testo.

L'output di `ComprehendG` è strutturato come segue:

```
1 {  
2     'ResultList': [  
3         {  
4             'Index': 0,  
5             'Sentiment': 'POSITIVE',  
6             'SentimentScore': {  
7                 'Positive': 0.9763978719711304,  
8                 'Negative': 9.167650568997487e-05,  
9                 'Neutral': 0.02348191849887371,  
10                'Mixed': 2.8541926440084353e-05  
11            }  
12        }, {}, {}  
13    ],  
14    'ErrorList': [],  
15    'ResponseMetadata': {}  
16 }
```

Anche in questo caso lo scoring vero e proprio avviene nella funzione di parsing, vediamo quindi l'algoritmo di scoring:

```
1 for item in compResult['ResultList']:  
2     idx = item['Index']  
3     score = Score(item["Sentiment"])  
4     float_score = (  
5         item["SentimentScore"]["Positive"]  
6         - item["SentimentScore"]["Negative"]
```



```
7         if (score != Score.MIXED)
8             else 0.0
9     )
10    if idx == 0:
11        sPost.captionScore = float_score
12    else:
13        sPost.textsScore[idx - 1] = float_score
```

La parte che si occupa di effettuare lo score vero e proprio è `float_score`, che si limita a dare come score la differenza tra i sentiment "Positive" e "Negative", oppure ignora la score se il valore di sentiment "Mixed" è troppo elevato (questo serve a filtrare i dati, ignorando quelli non rilevanti).

L'algoritmo discrimina tra caption e text-on-screen grazie agli id della `ResultList`: infatti ci premuriamo di fare in modo che all'id pari a zero ci sia sempre la sentiment analysis di Caption che viene quindi salvata in `sPost.captionScore`.

3.5.5.4 finalScore

È questa la funzione che si occupa di calcolare lo score del post vero e proprio, e lo fa mettendo assieme tutte e tre le scores precedenti (ovvero `captionScore`, `faceScore` e `textsScore`) per poi salvare il risultato in `sPost.finalScore`.

La formula generale è la seguente:

```
1 sPost.finalScore = (
2     normalizedCaptionScore * 2    #=[0,2]
3     + sPost.faceScore * 2         #=[0,2]
4     + normalizedTextScore         #=[0,1]
5 )
```

Si noti che gli scores per `caption` e `texts` non vengono prese direttamente da `sPost`, a differenza di `faceScore`.

Il motivo è il seguente: `finalScore` è stato scelto sia sempre compreso tra 0 e 5, così da agevolarne la visualizzazione nel frontend.

Se gli scores che lo compongono sono tutti compresi tra 0 e 1 diventa semplice dar loro dei pesi nel calcolo del punteggio: si è deciso infatti di dare peso uguale a caption e analisi facciale, mentre il testo a schermo pesa molto meno.

Il motivo è che nei post di Instagram_C il testo a schermo viene usato di rado, e quasi sempre per presentare locandine di eventi o locali (o ancora per lanciare messaggi come "Ferragosto Aperti"). È altamente improbabile che un utente normale usi del testo a immagine per veicolare informazioni, soprattutto quando esiste un'alternativa che richiede molto meno impegno, ovvero la caption.

Si è deciso di dare all'analisi facciale peso pari a quello della caption, poiché in quest'ultima spesso non si trovano informazioni particolarmente rilevanti riguardo al locale (es: serata al ristorante, la caption potrebbe essere "Cena con gli zii"), e quindi la sentiment analysis dei volti delle persone è una fonte d'informazioni non trascurabile.

Riportiamo ora l'algoritmo di normalizzazione:

```
1 # se presente del text on screen normalizzo la sua (loro) score
2 if len(sPost.textsScore) != 0:
3     textScore = sum(sPost.textsScore.values()) / len(
4         sPost.textsScore
5     ) # =[-1,1]
6     normalizedTextScore = (textScore + 1) / 2 # =[0,1]
7
```



```
8 # se presente la caption normalizzo la sua score
9 if sPost.caption and not sPost.caption.isspace():
10     normalizedCaptionScore = (sPost.captionScore + 1) / 2 # =[0,1]
11
12 # face score e' gia' =[0,1]
```

Salviamo in `textScore` la media di tutti gli scores dei pezzi di testo analizzati, e otteniamo un valore compreso tra -1 e 1.

A questo punto non ci resta che normalizzarlo per renderlo compreso tra 0 e 1.

La `captionScore` invece, anch'essa compresa tra -1 e 1, può essere normalizzata direttamente portandola quindi ad avere un valore compreso tra 0 e 1.

Come riportato anche nel codice sotto forma di commenti, gli scores vengono normalizzati solo se presenti.

Riportiamo ora l'algoritmo di scoring:

```
1 # CALCOLO DI FINAL SCORE
2 # F,T,C = faceScore, textScore, captionScore
3 # vuote = post analizzato non contiene: volti(F), testo a schermo(T)
4 #                                     ,caption(C)
5 # SE TUTTE VUOTE
6 if (
7     not (sPost.caption and not sPost.caption.isspace())
8     and len(sPost.textsScore) == 0
9     and sPost.faceScore is None
10 ):
11     sPost.finalScore = None
12 # SE F,T VUOTE
13 elif sPost.faceScore is None and len(sPost.textsScore) == 0:
14     sPost.finalScore = normalizedCaptionScore * 5
15 # SE C,T VUOTE
16 elif (
17     not (sPost.caption and not sPost.caption.isspace())
18     and len(sPost.textsScore) == 0
19 ):
20     sPost.finalScore = sPost.faceScore * 5
21 # SE F,C VUOTE
22 elif sPost.faceScore is None and not (
23     sPost.caption and not sPost.caption.isspace()
24 ):
25     sPost.finalScore = normalizedTextScore * 5
26 # SE T VUOTA
27 elif len(sPost.textsScore) == 0:
28     sPost.finalScore = sPost.faceScore * 2.5
29                     + normalizedCaptionScore * 2.5
30 # SE F VUOTA
31 elif sPost.faceScore is None:
32     sPost.finalScore = normalizedTextScore * 2
33                     + normalizedCaptionScore * 3
34 # SE C VUOTA
35 elif not (sPost.caption and not sPost.caption.isspace()):
36     sPost.finalScore = sPost.faceScore * 3
37                     + normalizedTextScore * 2
38 # SE NESSUNA VUOTA
39 else:
40     sPost.finalScore = (
41         normalizedCaptionScore * 2
42         + sPost.faceScore * 2
43         + normalizedTextScore
```

L'algoritmo considera tutti i possibili casi, dando pesi corretti di volta in volta, e calcola sempre un `finalScore` compreso tra 0 e 5:

- Se c'è solo uno dei tre score, il suo punteggio acquisisce peso pari a 5 diventando di fatto il `finalScore`;
- Se manca uno dei tre scores, gli altri due vengono pesati a modo per ottenere il risultato finale, sempre tenendo peso uguale per volti e caption e minore per testo a schermo;
- Se nessuno degli scores è presente viene assegnato a `sPost.finalScore` il valore di controllo `None`.

3.6 Scheduler Service

3.6.1 Descrizione generale

Questo servizio ha l'importante compito di far partire automaticamente, ogni intervallo di tempo predefinito, l'esecuzione del gruppo di servizi **S4**. La funzione di timer, che andrà a notificare la funzione Lambda responsabile vera e propria dello scheduling, è affidata ad una regola di Amazon EventBridge.

3.6.2 Diagramma delle classi

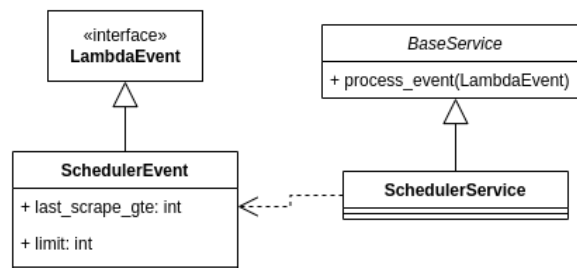


Figura 13: Scheduler Service - Diagramma delle classi

3.6.3 Diagramma di sequenza

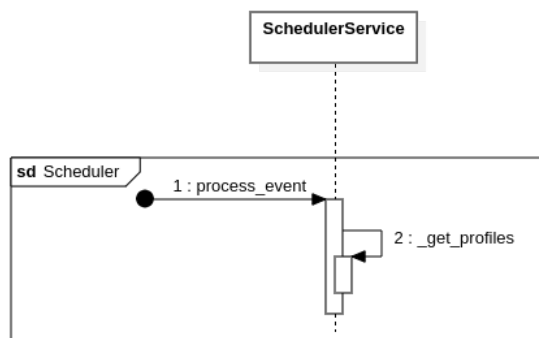


Figura 14: Scheduler Service - Diagramma di sequenza

3.6.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

```
1 {  
2   "last_scrape_gte": 12,  
3   "limit": 10  
4 }
```

Descrizione:

- **last_scrape_gte**: minimo intervallo di tempo passato dall'ultima azione di scraping per considerare un profilo social (misurato in ore);
- **limit**: numero massimo di profili social da trattare.

Output esempio di risposta in output, in formato JSON_G.

```
1 {  
2   "profiles_count": 2,  
3   "profiles": [  
4     {  
5       "id": 1,  
6       "username": "testuser1"  
7     },  
8     {  
9       "id": 2,  
10      "username": "testuser2"  
11    }  
12  ],  
13 }
```

Descrizione:

- **profiles_count**: numero di profili social trattati;
- **profiles**: array di profili social.

3.6.5 Intervallo temporale

L'intervallo temporale scelto, sulla base del quale poi eseguire ad intervalli regolari lo scheduling, è stato impostato a 30 minuti.

3.6.6 Regole di scheduling

La scelta di quali profili social sottoporre ad una esecuzione di **S4**, avviene sulla base del campo `SocialProfile.last_scraped`, memorizzato nel database. Verranno quindi scelti per l'esecuzione i profili social dove la differenza fra data e ora attuali e `last_scraped` è di 12 ore.

4 Architettura Frontend

Nel Frontend_G è stato utilizzato il pattern architetturale Model-View-Presenter (MVP), design diffuso nelle WebApp, con lo scopo di separare le componenti di visualizzazione dalla loro implementazione. Il pattern si suddivide in tre elementi:

- Model: elemento dove sono definiti i dati
- View: elemento passivo per la visualizzazione dei dati
- Presenter: elemento che fa da mediatore tra la View e il Model tramite data-binding ed eventi, prelevando o modificando dati del Model

Quando l'utente interagisce con l'applicazione, la parte di View è incaricata di visualizzare i dati e di notificare le azioni dell'utente, la parte del Presenter fa da tramite per le interazioni tra View e Model, prelevando i dati da quest'ultimo e visualizzandoli, oltre che a rispondere in modo corretto agli eventi sollevati dalla View, modificando i dati del Model di conseguenza. Il Model si occupa della persistenza dei dati oltre che a essere in grado di ottenerli tramite delle chiamate API_G (tramite metodo POST o GET) che si occuperà di interfacciarsi con il Backend_G, il quale restituirà i dati o gli errori risultati dalla chiamata.

A causa della scelta del framework_G Svelte_G la View non è una classe perchè è un Componente Svelte, che si compone da un blocco di script, un blocco di HTML_G ed un blocco di stile. Lo script viene utilizzato soltanto per inizializzare il Presenter, per il resto la View è passiva come previsto dal pattern MVP.

La reattività ai cambiamenti dei dati del Model (e anche nel Presenter) non è implementata con una pattern Observer standard, bensì a livello del singolo campo all'interno dei Model con un tipo speciale 'Writable' che può essere osservato tramite il metodo subscribe.

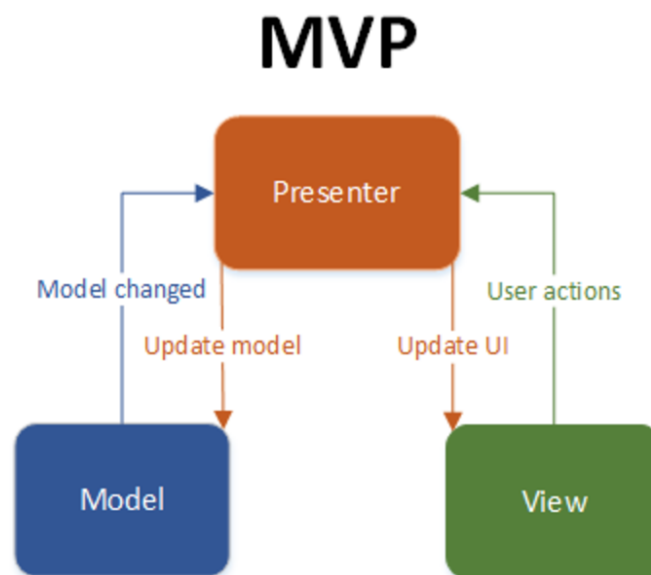


Figura 15: Schema pattern MVP



I punti di forza individuati in questo pattern sono che ogni classe Presenter possa gestire una classe View alla volta, quindi l'esistenza di una relazione uno a uno e questo permette di avere maggiore controllo sulle varie componenti, e la netta separazione presente tra Model e View. Quest'ultima risulta un punto chiave perchè permette di facilitare il testing relativo ai Presenter.

4.1 Diagramma delle classi

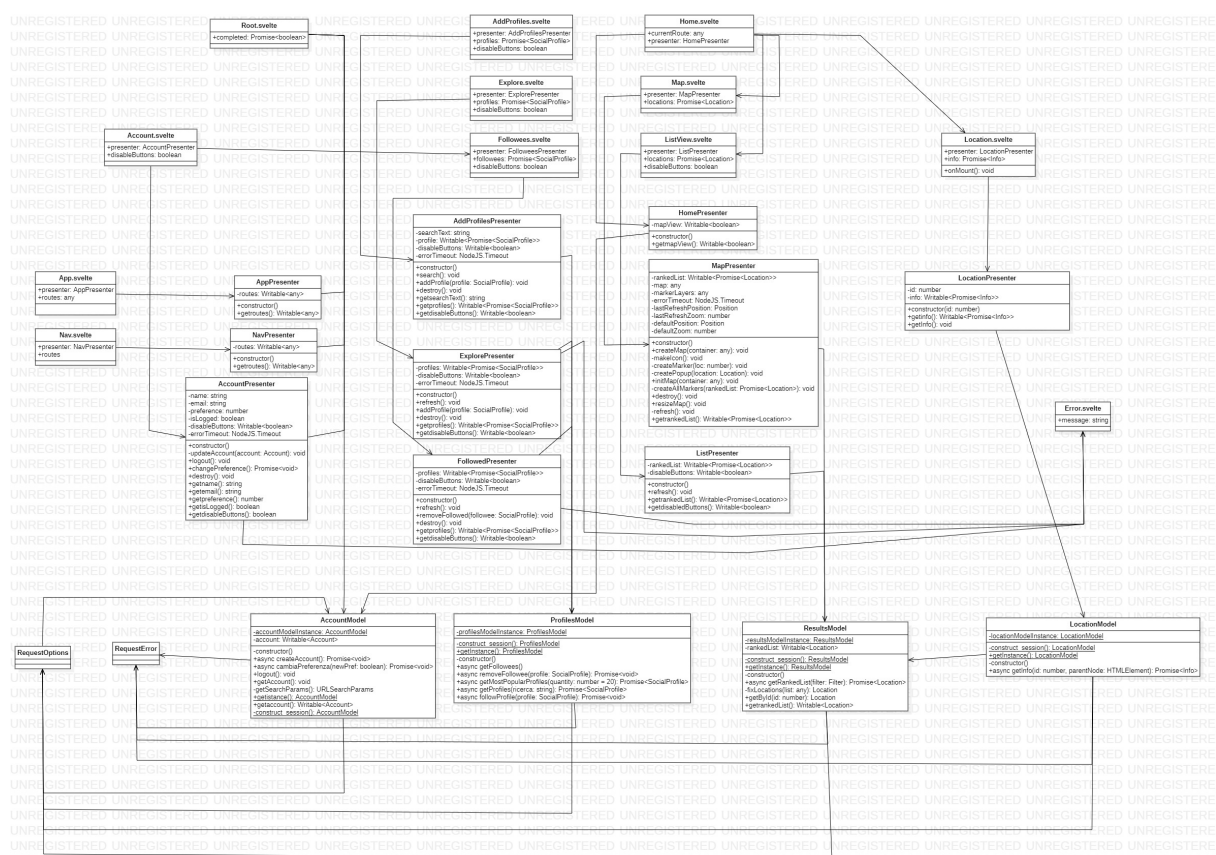


Figura 16: Frontend - Diagramma delle classi

4.2 Descrizione delle classi

4.2.1 Models

I Models si occupano della persistenza dei dati e dell'interfacciamento con le varie API usate. Utilizzano il pattern Singleton per la creazione di un'istanza, e assieme alla creazione viene anche controllato il SessionStorage per dati utili all'inizializzazione di un certo Model, sintomo di dati fondamentali che vanno mantenuti in memoria anche a seguito del cambio di pagina. Inoltre, mettono a disposizione alcuni membri Writable a cui ci si può iscrivere per ricevere i cambiamenti. Tutti i Model contengono codice asincrono che viene gestito tramite il sistema nativo delle Promise Javascript (inclusa la sintassi async/await).

- AccountModel: Si occupa delle funzionalità di login, come decifrare i token ritornati dal login gestito da AWS Cognito per ottenere le informazioni sull'account. Si



occupa delle funzionalità di gestione dell'account oltre a renderlo disponibile alle altre parti dell'applicazione.

- **ResultsModel:** Si occupa della gestione delle liste dei luoghi da visualizzare all'interno dell'applicazione.
- **ProfilesModel:** Si occupa delle funzionalità legate ai profili social, per esempio seguire o rimuovere dai seguiti un profilo, oppure ottenere le liste dei profili seguiti o più popolari.
- **LocationModel:** Si occupa della fornitura delle informazioni dettagliate legate ad un singolo luogo, interfacciandosi con la Maps API di Google per ottenere tali informazioni.
- **RequestOptions:** Si occupa della formattazione corretta delle opzioni di richiesta all'API, in modo da aderire alla specifica API.

4.2.2 Presenters

I Presenters in generale si occupano di collezionare dati dai vari Model di bisogno e contenerli nella formattazione corretta per la visualizzazione diretta all'interno delle View, essi presentano molti membri Writable che consentono un'integrazione molto naturale con le View in quanto un membro di questo tipo visualizzato si aggiorna automaticamente in seguito ai cambiamenti. I Presenter sono attenti a sottoscrivere ai cambiamenti dei Model quando necessario per mostrare dati sempre aggiornati e rilevanti.

4.2.3 View

Le View sono dei componenti bene o male passivi, in quanto il codice scritto si limita alla delegazione degli eventi verso il Presenter, ed alla visualizzazione di dati statici o dinamici (tipo Writable). Questi file vengono compilati automaticamente dalla libreria SvelteG per diventare delle View funzionali e responsive ai cambiamenti. Il codice asincrono viene gestito anch'esso nativamente, infatti quando la View visualizza un membro che ha dei dati non ancora pronti (per esempio in fase di download o di loading di una libreria) mostra invece una barra di caricamento e, a seconda del successo o meno del caricamento dei dati, visualizza i dati stessi oppure un errore con maggiori informazioni (se utili).