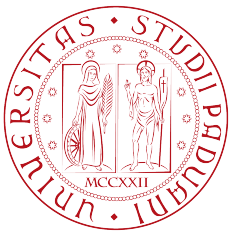
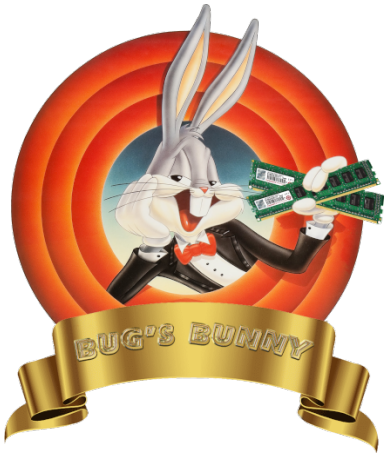


Università degli Studi di Padova
Ingegneria del Software
Anno Accademico: 2021/2022



Bug's Bunny

Specifica Architetturale

bugsbunnyteam@protonmail.com

Versione: 1.0.0
Redazione:
Marco Volpato,
Marco Bellò,
Matteo Tossuto
, Giulio Zanatta
Verifica: Angela Arena,
Tommaso Di Fant
Approvazione: Ruth Genevieve Bousapnamene
Uso: Esterno

Registro delle modifiche

Versione	Data	Nominativo	Ruolo	Descrizione
1.0.0	04-09-2022	Ruth Genevieve Bousapnamene	Progettista	Approvazione 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

Elenco delle figure

1 Introduzione

1.1 Scopo

Il documento seguente ha lo scopo di descrivere in modo esaustivo e coerente le specifiche e caratteristiche architetturali 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 architetturali:**
  `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-si
  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

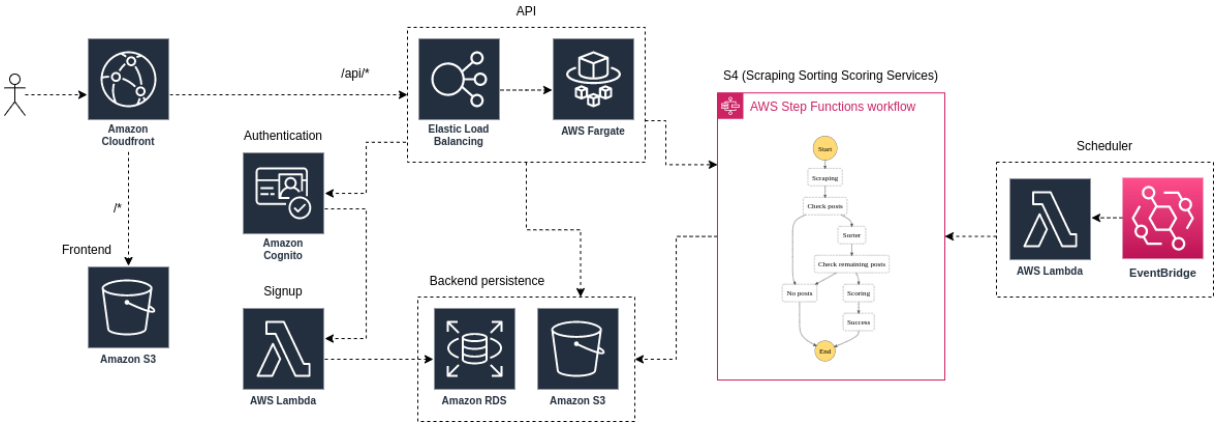


Figura 1: Visione d'insieme del sistema

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 lora 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** (**S**craping **S**orting **S**coring **S**ervices):
  - **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** (**S**craping **S**orting **S**coring **S**ervices). Essi vengono eseguiti in funzioni Lambda, le quali sono orchestrate attraverso *AWS Step Functions*.

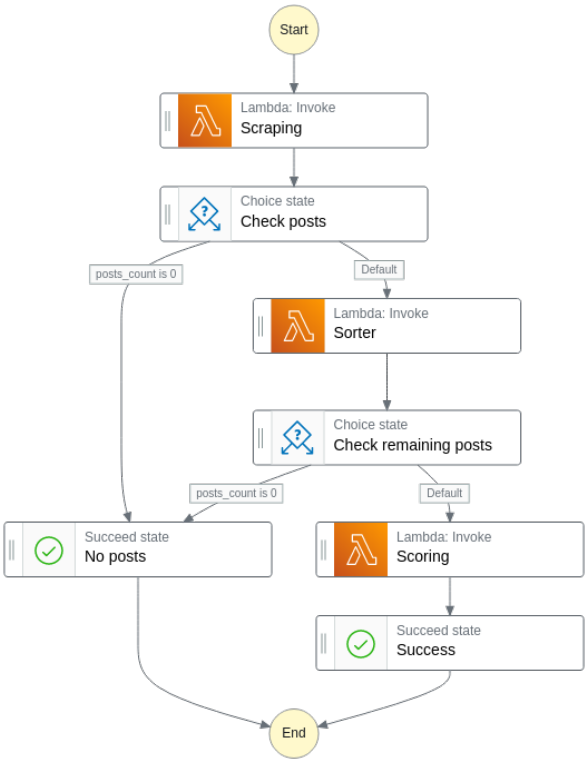


Figura 2: Visualizzazione macchina a stati

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*$_G$.

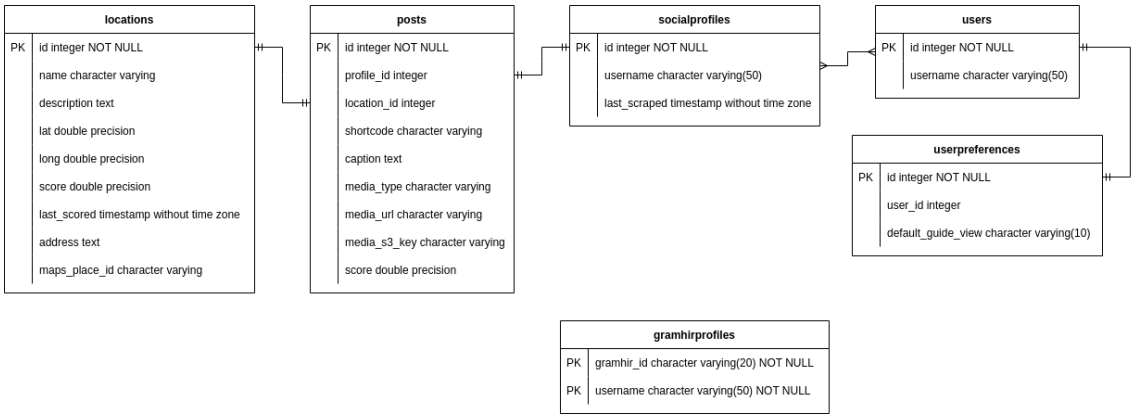


Figura 3: Schema ER database

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 schemo 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.

```
"""add location address and maps_place_id
Revision ID: 6ce8d444b850
Revises: a557f1403100
Create Date: 2022-09-01 10:43:05.074767
"""

from alembic import op
import sqlalchemy as sa

# revision identifiers, used by Alembic.
revision = '6ce8d444b850'
down_revision = 'a557f1403100'
branch_labels = None
depends_on = None

def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('locations', sa.Column('address', sa.Text(), nullable=
        True))
    op.add_column('locations', sa.Column('maps_place_id', sa.String(),
        nullable=True))
    # ### 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

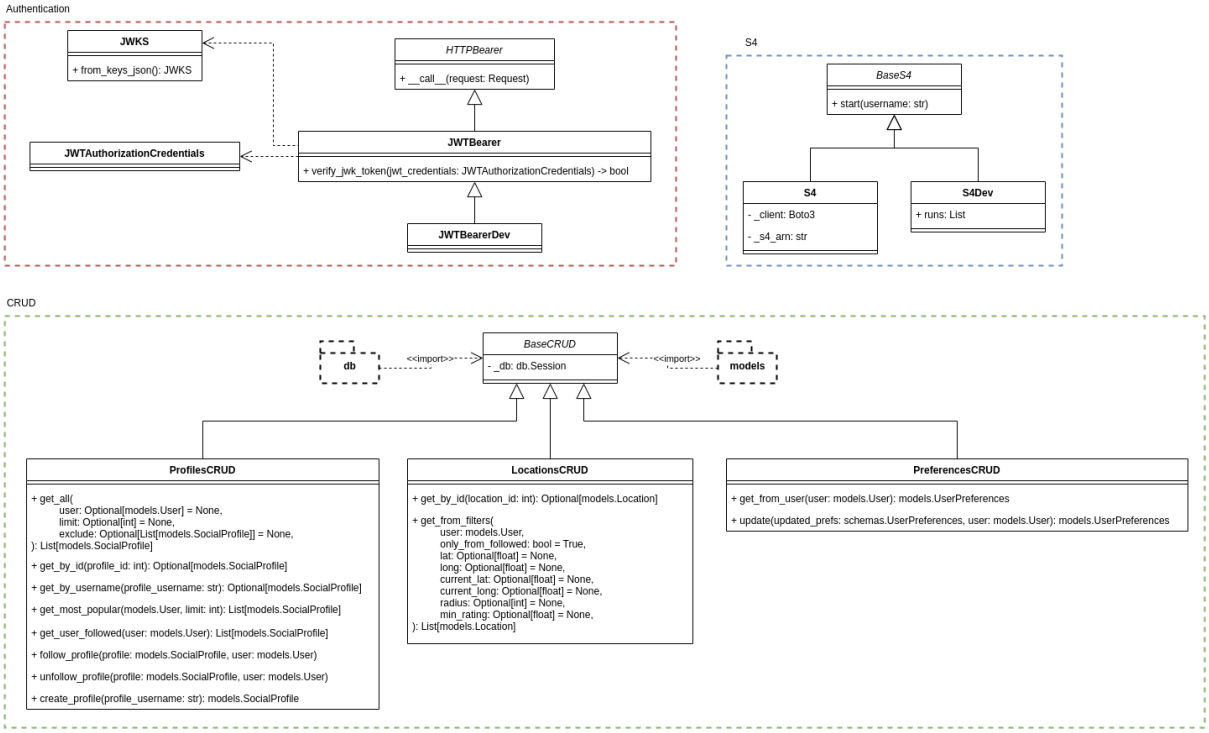


Figura 4: API Service - Diagramma delle classi

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 increntrato 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 decoratori 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,
```

```
    response_model_exclude_unset=True,
)
def get_location(
    location_id: int,
    locations: LocationsCRUD = Depends(get_locations_crud),
):
    location = locations.get_by_id(location_id)
    if not location:
        raise HTTPException(status_code=HTTP_404_NOT_FOUND, detail='
            Location not found')
    return location
```

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.

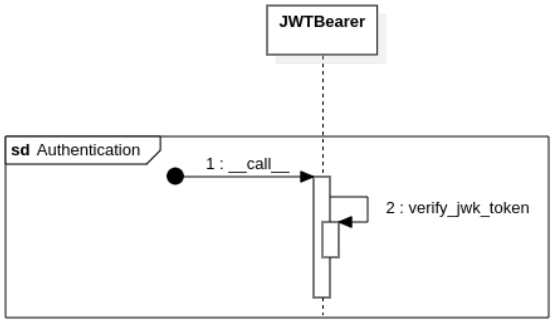


Figura 5: Autenticazione - Diagramma di sequenza

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

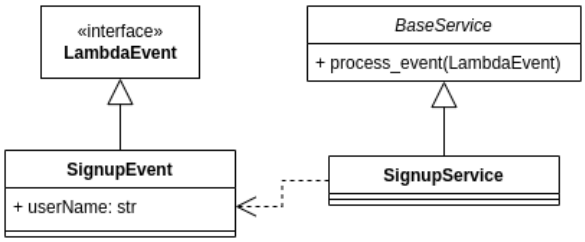


Figura 6: Signup Service - Diagramma delle classi

3.2.3 Schemi I/O

Input

Evento *pre signup* in format JSON_G, come definito nella documentazione di Cognito_G.

Output

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

3.3 Scraping Service

3.3.1 Descrizione generale

Questo servizio implementa le funzionalità di scraping_G ed in generale ottenimento dati da $\text{Instagram}_\text{G}$ e altre piattaforme ad esso correlate.

3.3.2 Diagramma delle classi

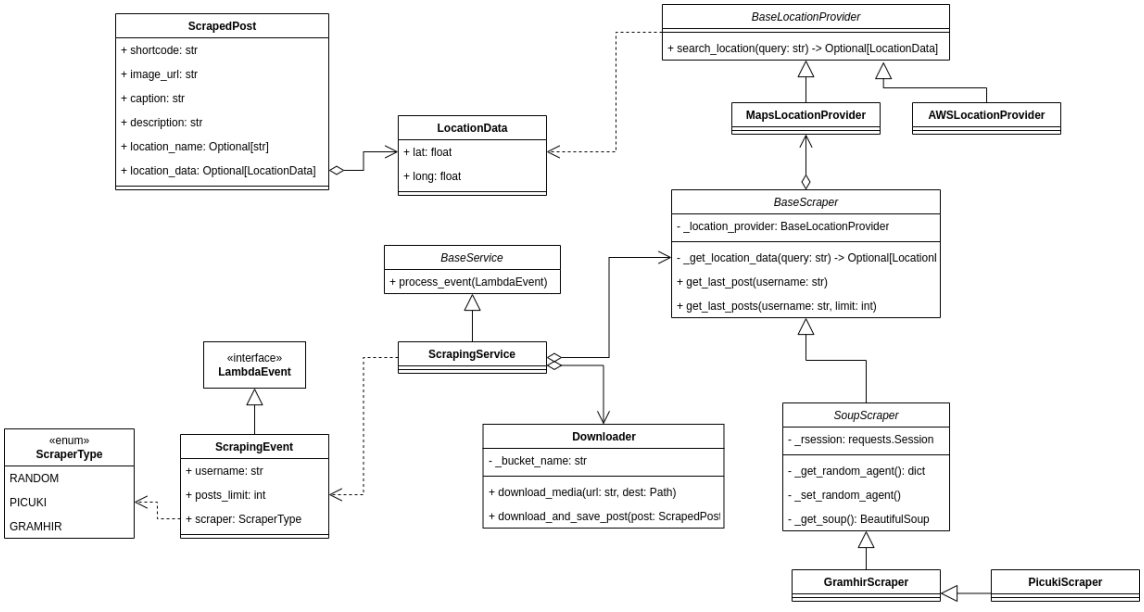


Figura 7: Scraping Service - Diagramma delle classi

3.3.3 Diagramma di sequenza

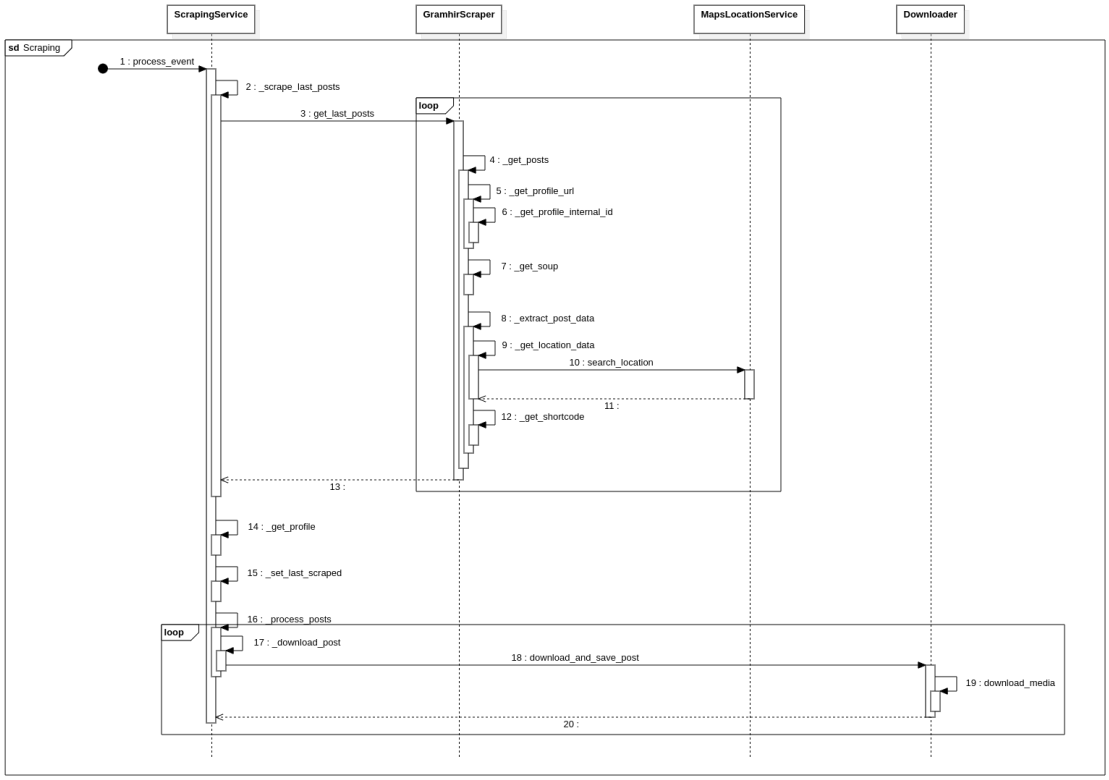


Figura 8: Scraping Service - Diagramma di sequenza

3.3.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

```
{
    "username": "antoniorazzi",
    "posts_limit": 10
}
```

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.

```
{
    "posts_count": 2,
    "posts": [
        {"id": 4},
        {"id": 5}
    ]
}
```

Descrizione:

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

3.3.5 Strategie di scraping

Le azioni di scraping$_G$ non avvengono direttamente su Instagram$_G$, questo perchè comporterebbe il rischio di incorrere in *rate limiting* o addirittura l'impossibilità totale di usufruire del servizio dall'infrastruttura cloud di AWS$_G$. Per aggirare queste problematiche, lo scraping$_G$ avviene da piattaforme chiamate *viewer*, che propongono in larga parte gli stessi contenuti di Instagram$_G$, 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 scraper$_G$ 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*.

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

Listing 3: Ottenimento URL profilo in *Gramhir*

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

Listing 4: Ottenimento URL profilo in *Picuki*

Lo scraping vero e proprio avviene tramite parsing$_G$ del contenuto HTML$_G$ delle pagine web fornite dai *viewer*.

```
def _extract_post_data(self, post_result: Tag) -> ScrapedPost:
    location_name = post_result.find(attrs={'class': 'photo-location'}).
        get_text(
         strip=True
    )
    location_name = location_name if location_name else None

    location_data = None
    if location_name:
        location_data = self._get_location_data(location_name)

    details_url = post_result.find('a').get('href')
    shortcode = self._get_shortcode(details_url)

    return ScrapedPost(
        image_url=post_result.find('img').get('src'),
        caption=post_result.find(attrs={'class': 'photo-description'}).
            get_text(
             strip=True
        ),
        description='',
        location_name=location_name,
```

```
        location_data=location_data,
        shortcode=shortcode,
    )
```

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.

```
def search_location(self, query: str) -> Optional[LocationData]:
    geocode_result = self._maps.geocode(query)

    if not geocode_result:
        return None

    geocode_result = geocode_result[0]
    coords = geocode_result['geometry']['location']
    maps_place_id = geocode_result.get('place_id')

    place = self._maps.place(place_id=maps_place_id)
    address = place['result']['formatted_address']
    name = place['result']['name']
    types = place['result']['types']

    return LocationData(
        lat=coords['lat'],
        long=coords['lng'],
        address=address,
        maps_name=name,
        maps_place_id=maps_place_id,
        types=types
    )
```

Listing 6: Ottenimento informazioni da *Google Maps API*

3.3.7 Salvataggio dei dati

I dati relativi ai post ottenuti e le relative posizioni, vengonono salvati nel database. Le immagini o altri contenuti multimediali vengonono 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:

```
def __init__(self):
    self._rekognition = boto3.client(
        service_name='rekognition',
        region_name=aws_region
    )
```

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

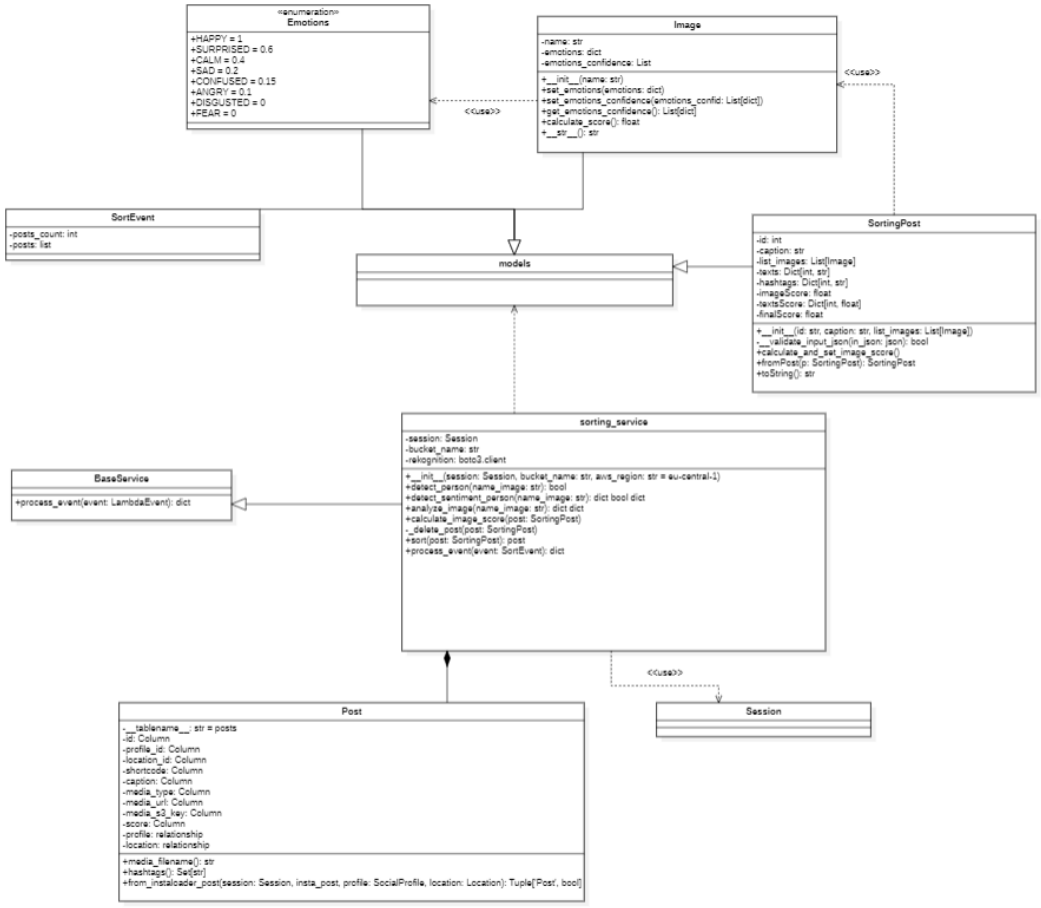


Figura 9: Sorting Service - Diagramma delle classi

3.4.3 Diagramma di sequenza

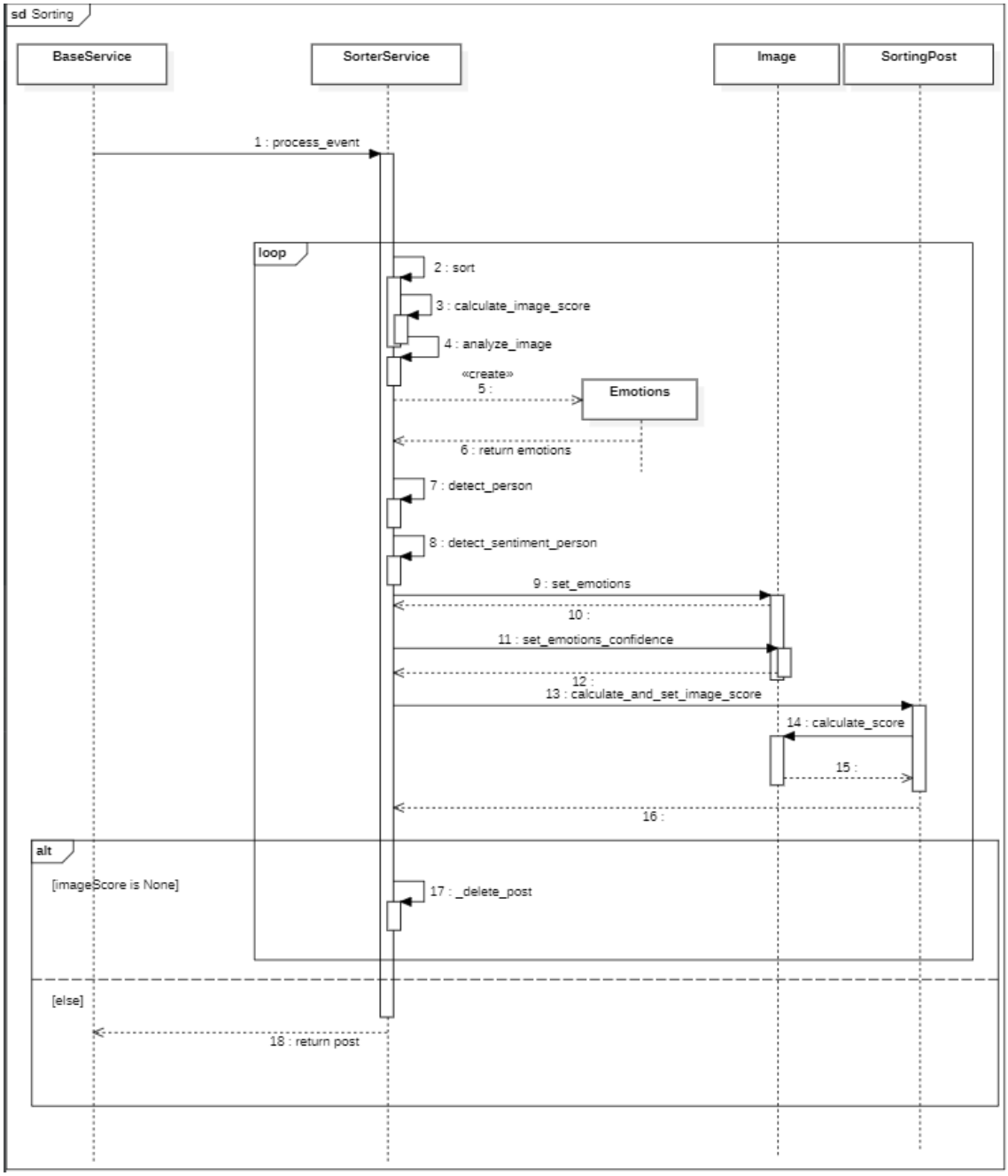


Figura 10: Sorting Service - Diagramma di sequenza

3.4.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

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

```
        {"id": 4}
    ]
}
```

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.

```
{
    "posts_count": 2,
    "posts": [
        {"id": 2},
        {"id": 4}
    ]
}
```

Descrizione:

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

3.4.5 Calcolo dell'imageScore

Segue una spiegazione più dettagliata del processo per il calcolo di imageScore 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`.

```
for p in event.posts:
    db_post = self._session.query(models.Post).filter_by(id=p['id']).
        first()
    sorting_post = SortingPost.fromPost(db_post)
    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 imageScore.

```
self.calculate_image_score(post)
```

```
image.set_emotions(emotions)
image.set_emotions_confidence(emotions_confidence)

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:

```
def set_emotions(self, emotions: dict):
    self.emotions = emotions

def set_emotions_confidence(self, emotions_confid: List[dict]):
    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`:

```
if self.list_images:
    count = 0
    for image in self.list_images:
        score = image.calculate_score()
        if score is not None:
            count += 1
            self.imageScore = (
                (self.imageScore + score)
                if self.imageScore is not None
                else score
            )
    if self.imageScore:
        self.imageScore = self.imageScore / count
```

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

```
if self.emotions:
    value_sum = 0.0
    sum_weights = 0.0
    for emotion, num in self.emotions.items():
        value_sum += Emotions[emotion].value * num
        sum_weights += num
    return 100 * value_sum / sum_weights
else:
    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`:

```
for img in post.list_images:
    s3_delete_file(self._bucket_name, img.name)
self._session.delete(
    self._session.query(models.Post).filter_by(id=post.id).first()
)
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

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

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$_G$ contente una sentiment analysis dei volti presenti nell'immagine. Segue un esempio ove presenti solo i dati necessari al nostro caso:

```
{
"FaceDetails": [
    {
      ....
      ....
        "Emotions": [
            {
                "Type": "ANGRY",
                "Confidence": 55.18563461303711
            },
            {
                "Type": "HAPPY",
                "Confidence": 37.01131820678711
            },{},{},{},{},{},{}
        ],
        ....
        ....
        ....
        ....
    }
}
```

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:

```
contain_emotion = False
emotions_dict = {}
emotions_confid = []

for faceDetail in response['FaceDetails']:
    emotions = faceDetail['Emotions']
    confid_single_face = {}
    for emotion in emotions:
        emotion_name = emotion['Type']
        emotion_confid_value = emotion['Confidence']
        if emotion_name != 'UNKNOWN':
            confid_single_face[emotion_name] = emotion_confid_value

            if emotion_confid_value >= 90:
```

```
                if emotion_name in emotions_dict:
                    emotions_dict[emotion_name] += 1
                else:
                    emotions_dict[emotion_name] = 1
                contain_emotion = True
    emotions_confid.append(confid_single_face)
return emotions_dict, contain_emotion, emotions_confid
```

3.4.6.2 Response from detect_labels

```
response = self._rekognition.detect_labels(
            Image={'S3Object': {'Bucket': self._bucket_name, 'Name':
               name_image}},
            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$_G$ contente 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:

```
{
  "Labels": [
    {
      "Name": "Person",
            "Confidence": 99.88428497314453,
            "Instances": [
              {
                  "BoundingBox": {
                    "Width": 0.27858132123947144,
                    "Height": 0.44550982117652893,
                    "Left": 0.2236727923154831,
                    "Top": 0.35303789377212524
                     },
                 "Confidence": 99.88428497314453
                },
        {
                    "BoundingBox": {
                    "Width": 0.20297367870807648,
                    "Height": 0.6778767108917236,
                    "Left": 0.5247262716293335,
                    "Top": 0.10922279208898544
                     },
                "Confidence": 99.81658172607422
        }
      ]
            ....
            ....
            ....
            ....
          ....
  }]
}
```

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

```
contain_person = False

for label in response['Labels']:
    if label['Confidence'] >= 90:
        if label['Name'] == 'Person':
            contain_person = True

return contain_person
```

In questo nostro esempio possiamo vedere, nel file $JSON_G$, 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:

```
def __init__(self):
    self._rekognition = boto3.client(service_name='rekognition')
    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 $JSON_G$ 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 contente 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

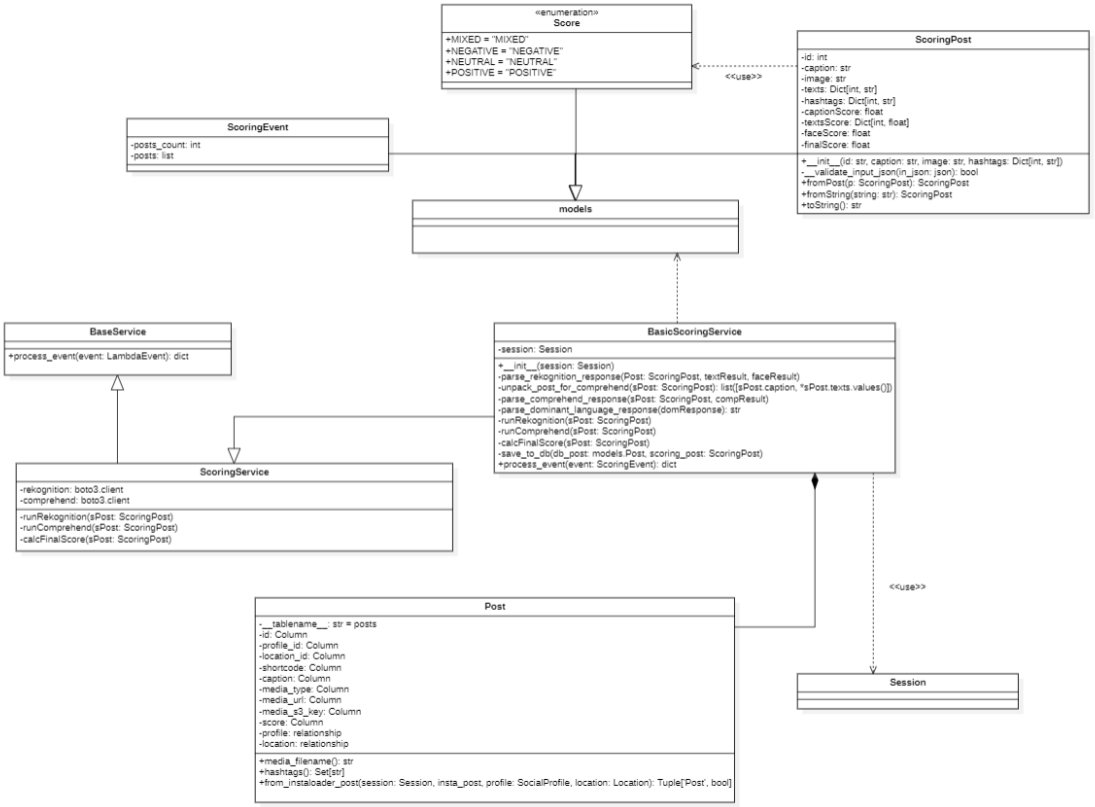


Figura 11: Scoring Service - Diagramma delle classi

3.5.3 Diagramma di sequenza

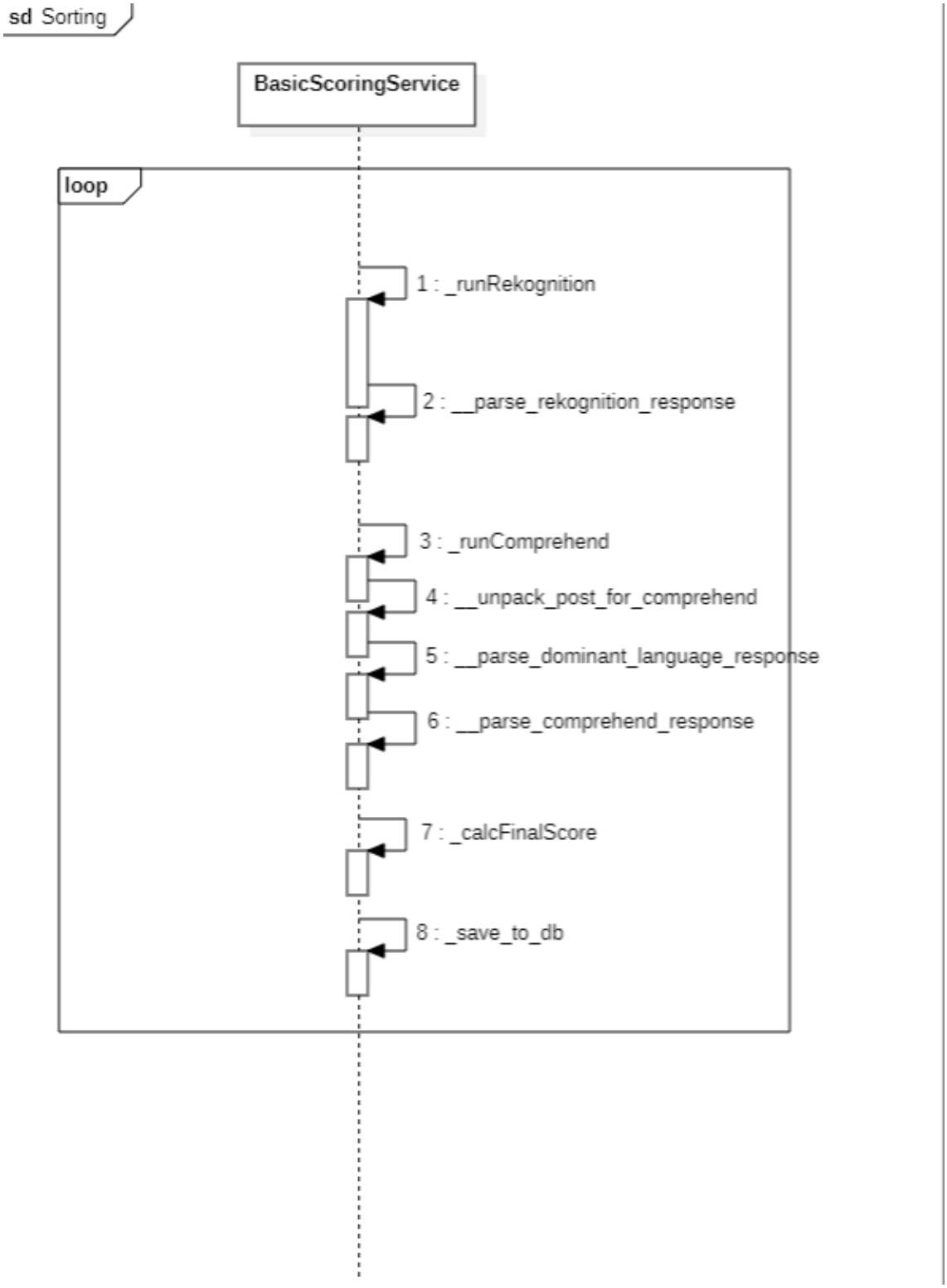


Figura 12: Scoring Service - Diagramma delle classi

3.5.4 Schemi I/O

Input esempio di evento in input, in formato JSON_{G}.

```
{
    "posts_count": 2,
    "posts": [
        {"id": 4},
        {"id": 5}
    ]
}
```

Descrizione:

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

Output esempio di risposta in output, in formato JSON_G.

```
{
    "scored_posts_count": 8,
    "scored_posts": [
        {
            "id": 4,
            "score": 4.830453701069928
        },
        {
            "id": 5,
            "score": 2.562992335879244
        },
    ]
}
```

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`:

```
Image = {
    'S3Object': {
        'Bucket': os.environ['ENV_BUCKET_NAME'],
        'Name': sPost.image,
    }
}
textResponse = self._rekognition.detect_text(Image=Image)
faceResponse = self._rekognition.detect_faces(Image=Image)
BasicScoringService.__parse_rekognition_response(
    sPost, textResponse, faceResponse
)
```

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$_G$ contente una sentiment analysis dei vari volti presenti.
Segue un esempio del file in questione (lasciando implicito ciò che non serve):

```
{
"FaceDetails": [
    {
        "BoundingBox": {},
        "AgeRange": {},
        "Smile": {},
        "Eyeglasses": {},
        "Sunglasses": {},
        "Gender": {},
        "Beard": {},
        "Mustache": {},
        "EyesOpen": {},
        "MouthOpen": {},
        "Emotions": [
            {
                "Type": "ANGRY",
                "Confidence": 55.18563461303711
            },
            {
                "Type": "HAPPY",
                "Confidence": 37.01131820678711
            },{},{},{},{},{},{}
        ],
        "Landmarks": [],
        "Pose": {
            "Roll": 3.602341890335083,
            "Yaw": -82.46586608886719,
            "Pitch": -18.774751663208008
        },
        "Quality": {
            "Brightness": 92.66178894042969,
            "Sharpness": 9.912903785705566
        },
        "Confidence": 99.85897064208984
    }
}
```

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:

```
loop for all volti{
    if(volto dritto){
        score_volto = confidence_HAPPY + (confidence_CALM * 0.5)
        if(confidence_DISGUSTED too high){
            score_volto = 0
        }
    }
}
score_immagine_finale = somma_score_volti_dritti / numero_volti_dritti
```

Segue ora l'algoritmo:

```
faceCount = 0
scoreSum = 0
if len(faceResult['FaceDetails']) > 0:
    for face in faceResult['FaceDetails']:
        pose = face['Pose']
        # SE VOLTO DRITTO
        if (abs(pose['Yaw']) <= 50) and (
            abs(pose['Pitch']) <= 50
        ):  # abs() perche' deve essere -50<pose<50
            # CALCOLARE SCORE EMOZIONI
            faceCount = faceCount + 1
            faceSum = 0
            disgusted = False
            for emotion in face['Emotions']:
                if emotion['Type'] == 'HAPPY':
                    faceSum = faceSum + emotion['Confidence']
                if emotion['Type'] == 'CALM':
                    faceSum = (
                        faceSum + emotion['Confidence'] * 0.5
                    )  # ha peso minore di happy
                if emotion['Type'] == 'DISGUSTED':
                    if (
                        emotion['Confidence'] >= 50
                    ):  # se disgust troppo elevato azzera il punteggio
                        #                                 della faccia
                        disgusted = True
            if not disgusted:
                scoreSum = (
                    scoreSum + faceSum
                )
        # UN VOLTO STORTO VIENE IGNORATO NEL CALCOLO
    faceScore = scoreSum / faceCount    # =[0,100]
```

```
    sPost.faceScore = faceScore / 100  # normalizzato a [0,1]
else:
    sPost.faceScore = None  # se num facce =0 si ignora nel calcolo
                            #                       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:

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

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 `DISGUSTED`≥ 50 riceve uno score pari a 0.
Senza questo controllo sarebbe teoricamente possibile che `HAPPY` fosse pari a 50 (contemporaneamente a `DISGUSTED`=50) e quindi che lo score del volto fosse 50 su 100.

```
    if emotion['Type'] == 'DISGUSTED':
        if (
            emotion['Confidence'] >= 50
        ):
            disgusted = True
# end of for
if not disgusted:
scoreSum = (
    scoreSum + faceSum
)  # 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:

```
#if len(faceResult['FaceDetails']) > 0:
# corpo dell' algoritmo
    faceScore = scoreSum / faceCount
    sPost.faceScore = faceScore / 100  # normalizzato a [0,1]
else:
    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`:

```
Image = {
    'S3Object': {
        'Bucket': os.environ['ENV_BUCKET_NAME'],
        'Name': sPost.image,
    }
}
textResponse = self._rekognition.detect_text(Image=Image)
faceResponse = self._rekognition.detect_faces(Image=Image)
BasicScoringService.__parse_rekognition_response(
    sPost, textResponse, faceResponse
)
```

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$_G$ contente il testo presente nell'immagine.
Segue un esempio del file in questione (lasciando implicito ciò che non serve):

```
{
"TextDetections": [
    {
        "DetectedText": "PIZZA LAB",
        "Type": "LINE",
        "Id": 0,
        "Confidence": 99.8796615600586,
        "Geometry": {}
    },
    {
        "DetectedText": "Buon Ferragasta",
        "Type": "LINE",
        "Id": 1,
        "Confidence": 67.9867935180664,
        "Geometry": {}
    },{},{},{}
    ],
    "TextModelVersion": "3.0"
}
```

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

```
for line in textResult['TextDetections']:
    if line['Type'] == 'LINE':
        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:

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

A questo punto è Comprehend$_G$ che deve occuparsi di dare degli effettivi sentiment scores a queste stringhe di testo.
Il punto di partenza si trova nella funzione `BasicScoringService._runComprehend`:

```
dominantLanguageResponse = self._comprehend.detect_dominant_language(
    Text=sPost.caption
)
response = self._comprehend.batch_detect_sentiment(
    TextList=BasicScoringService.__unpack_post_for_comprehend(sPost),
    LanguageCode=BasicScoringService.__parse_dominant_language_response(
        dominantLanguageResponse
    ),
)
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`:

```
def __unpack_post_for_comprehend(self, sPost: ScoringPost):
    if not sPost.caption:
        sPost.caption = ' '
    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 Comprehend$_G$, e il risultato della funzione (cioè quindi ciò su cui viene lanciata la funzione di sentiment detection di Comprehend$_G$) è il seguente (sempre prendendo ad esempio i file di cui sopra):

```
TextList = [
    'Caption',
    'PIZZA LAB',
    'Buon Ferragasta'
]
```

`TextList` viene quindi inoltrata a Comprehend$_G$tramite l'oggetto `_comprehend` e la sua funzione `batch_detect_sentiment`, che ritorna nella variabile `response` un file JSON$_G$ contenete la sentiment analysis per i vari pezzi di testo.
L'output di Comprehend$_G$è strutturato come segue:

```
{
    'ResultList': [
        {
            'Index': 0,
            'Sentiment': 'POSITIVE',
            'SentimentScore': {
                'Positive': 0.9763978719711304,
                'Negative': 9.167650568997487e-05,
                'Neutral': 0.02348191849887371,
                'Mixed': 2.8541926440084353e-05
                }
        }, {}, {}
```

```
    ],
    'ErrorList': [],
    'ResponseMetadata': {}
}
```

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

```
for item in compResult['ResultList']:
    idx = item['Index']
    score = Score(item["Sentiment"])
    float_score = (
        item["SentimentScore"]["Positive"]
        - item["SentimentScore"]["Negative"]
        if (score != Score.MIXED)
        else 0.0
    )
    if idx == 0:
        sPost.captionScore = float_score
    else:
        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:

```
float_score = (
    item["SentimentScore"]["Positive"]
    - item["SentimentScore"]["Negative"]
    if (score != Score.MIXED)
    else 0.0
)
if idx == 0:
    sPost.captionScore = float_score
else:
    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`:

```
dominantLanguageResponse = self._comprehend.detect_dominant_language(
    Text=sPost.caption
)
response = self._comprehend.batch_detect_sentiment(
    TextList=BasicScoringService.__unpack_post_for_comprehend(sPost),
```

```
    LanguageCode=BasicScoringService.__parse_dominant_language_response(
        dominantLanguageResponse
    ),
)
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`:

```
def __unpack_post_for_comprehend(self, sPost: ScoringPost):
    if not sPost.caption:
        sPost.caption = ' '
    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 Comprehend$_G$, e il risultato della funzione (cioè quindi ciò su cui viene lanciata la funzione di sentiment detection di Comprehend$_G$) è il seguente (sempre prendendo ad esempio i file di cui sopra):

```
TextList = [
    'Caption',
    'PIZZA LAB',
    'Buon Ferragasta'
]
```

A questo punto `TextList` viene inoltrata a Comprehend$_G$ tramite l'oggetto `_comprehend` e la sua funzione `batch_detect_sentiment`, che ritorna nella variabile `response` un file JSON$_G$ contenete la sentiment analysis per i vari pezzi di testo.
L'output di Comprehend$_G$è strutturato come segue:

```
{
    'ResultList': [
        {
            'Index': 0,
            'Sentiment': 'POSITIVE',
            'SentimentScore': {
                'Positive': 0.9763978719711304,
                'Negative': 9.167650568997487e-05,
                'Neutral': 0.02348191849887371,
                'Mixed': 2.8541926440084353e-05
                }
        }, {}, {}
    ],
    'ErrorList': [],
    'ResponseMetadata': {}
}
```

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

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

```
        if (score != Score.MIXED)
        else 0.0
    )
    if idx == 0:
        sPost.captionScore = float_score
    else:
        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:

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

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$_G$ 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:

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

```

```
# se presente la caption normalizzo la sua score
if sPost.caption and not sPost.caption.isspace():
    normalizedCaptionScore = (sPost.captionScore + 1) / 2  # =[0,1]

# 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:

```
# CALCOLO DI FINAL SCORE
# F,T,C = faceScore, textScore, captionScore
# vuote = post analizzato non contiene: volti(F), testo a schermo(T)
#                                                    ,caption(C)
# SE TUTTE VUOTE
if (
    not (sPost.caption and not sPost.caption.isspace())
    and len(sPost.textsScore) == 0
    and sPost.faceScore is None
):
    sPost.finalScore = None
# SE F,T VUOTE
elif sPost.faceScore is None and len(sPost.textsScore) == 0:
    sPost.finalScore = normalizedCaptionScore * 5
# SE C,T VUOTE
elif (
    not (sPost.caption and not sPost.caption.isspace())
    and len(sPost.textsScore) == 0
):
    sPost.finalScore = sPost.faceScore * 5
# SE F,C VUOTE
elif sPost.faceScore is None and not (
    sPost.caption and not sPost.caption.isspace()
):
    sPost.finalScore = normalizedTextScore * 5
# SE T VUOTA
elif len(sPost.textsScore) == 0:
    sPost.finalScore = sPost.faceScore * 2.5
                     + normalizedCaptionScore * 2.5
# SE F VUOTA
elif sPost.faceScore is None:
    sPost.finalScore = normalizedTextScore * 2
                     + normalizedCaptionScore * 3
# SE C VUOTA
elif not (sPost.caption and not sPost.caption.isspace()):
    sPost.finalScore = sPost.faceScore * 3
                     + normalizedTextScore * 2
# SE NESSUNA VUOTA
else:
    sPost.finalScore = (
        normalizedCaptionScore * 2
        + sPost.faceScore * 2
        + normalizedTextScore
```

```
44          )
```

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 accqusisce 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

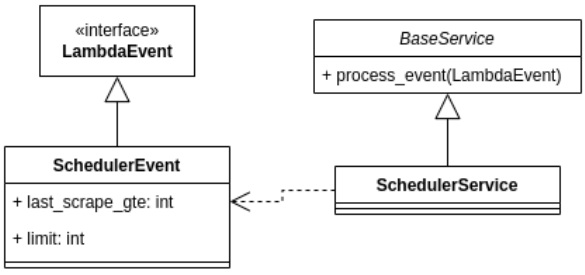


Figura 13: Scheduler Service - Diagramma delle classi

3.6.3 Diagramma di sequenza

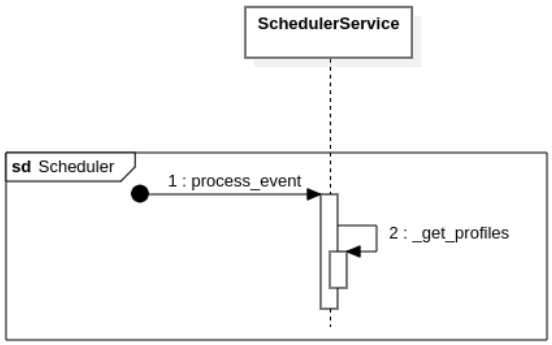


Figura 14: Scheduler Service - Diagramma di sequenza

3.6.4 Schemi I/O

Input esempio di evento in input, in formato JSON_G.

```
{
    "last_scrape_gte": 12,
    "limit": 10
}
```

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.

```
{
    "profiles_count": 2,
    "profiles": [
        {
            "id": 1,
            "username": "testuser1"
        },
        {
            "id": 2,
            "username": "testuser2"
        }
    ],
}
```

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.

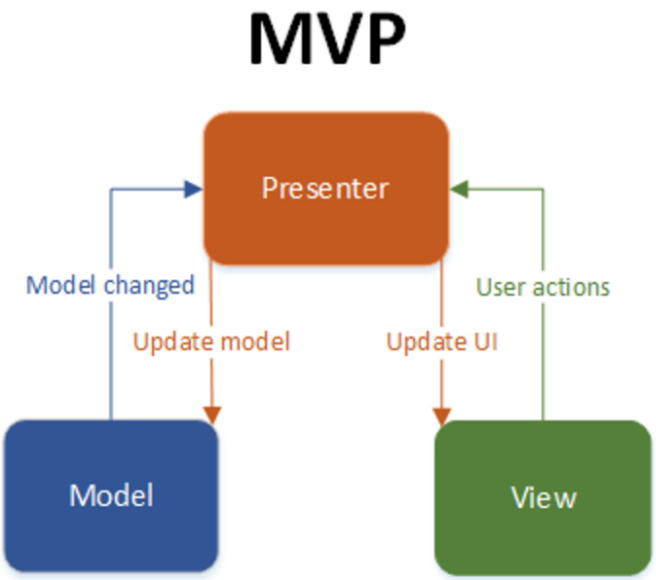


Figura 15: Schema pattern MVP

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

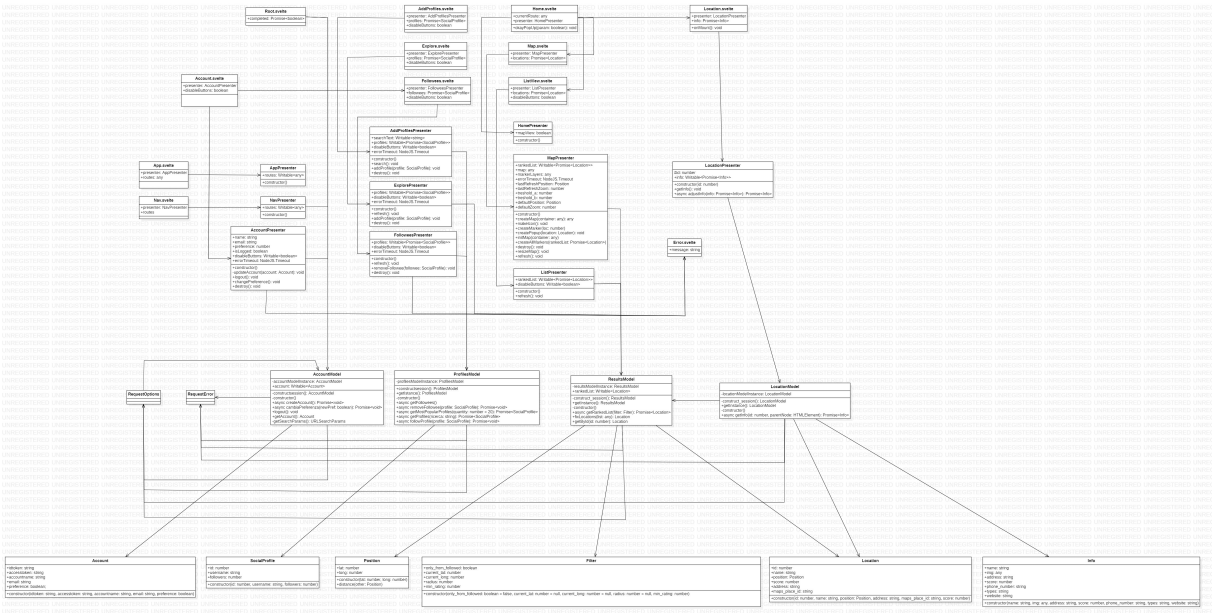


Figura 16: Frontend - Diagramma delle classi