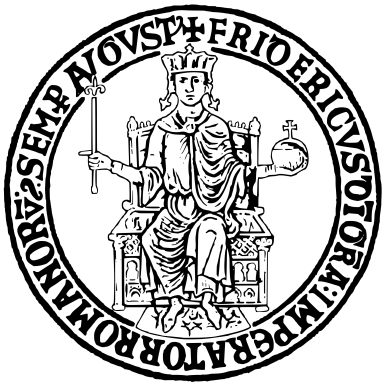


# UninaSocialGroup

Esame di Basi di Dati I

Anno Accademico 2023/2024

*Gruppo 30*
Sabrina CASSONE N86004481
Matteo CHIAPPETTA N86004782

.

# Indice

# 1 Presentazione del progetto

## 1.1 Cos'è UninaSocialGroup

UninaSocialGroup è un social network pensato per gli studenti della Federico II di Napoli, ovvero un punto d'incontro virtuale che permette agli utenti di condividere pensieri, domande e opinioni sottoforma di contenuti multimediali. Gli utenti possono interagire tra di loro lasciando un commento o un mi piace ai post condivisi da altri.
Su UninaSocialGroup e possibile iscriversi a gruppi, così da vedere solo i contenuti a cui si è interessati. Infatti, ogni gruppo è caratterizzato da un tema: se il tema a cui si è interessati ancora non esiste, allora basterà creare un nuovo gruppo!
Se si vuole conoscere meglio un altro utente, è sufficiente seguire quell'utente.

## 1.2 Primo approccio

Per potersi iscrivere ed effettuare l'accesso, ogni utente deve avere un'e-mail e una password valide e, per essere riconosciuto più facilmente dagli altri utenti, dispone anche di un username. Un utente può scegliere di personalizzare il suo Profilo aggiungendo un'immagine e una descrizione.
Un utente può seguire altri utenti.

Un utente può iscriversi a uno o più gruppi, oppure crearne dei propri. Ogni gruppo ha un nome specifico, ma anche uno o più temi generici che permettono agli utenti di riconoscere il tipo di contenuto pubblicato all'interno del gruppo stesso. Gli iscritti ad un gruppo possono visualizzare e pubblicare post all'interno di esso, e interagire ai post degli altri utenti mettendo un mi piace, oppure lasciando dei commenti.
L'utente viene notificato ogni volta che viene inserito un nuovo post in un gruppo di cui fa parte.
Ogni post è caratterizzato da informazioni testuali, che possono essere eventualmente accompagnate da immagini.

Se l'utente è creatore di un gruppo, allora può scegliere se accettare o meno le richieste di ingresso nel suo gruppo da parte di altri utenti.

# 2 Progettazione Concettuale

## 2.1 Diagramma ER

## 2.2 Diagramma delle classi UML

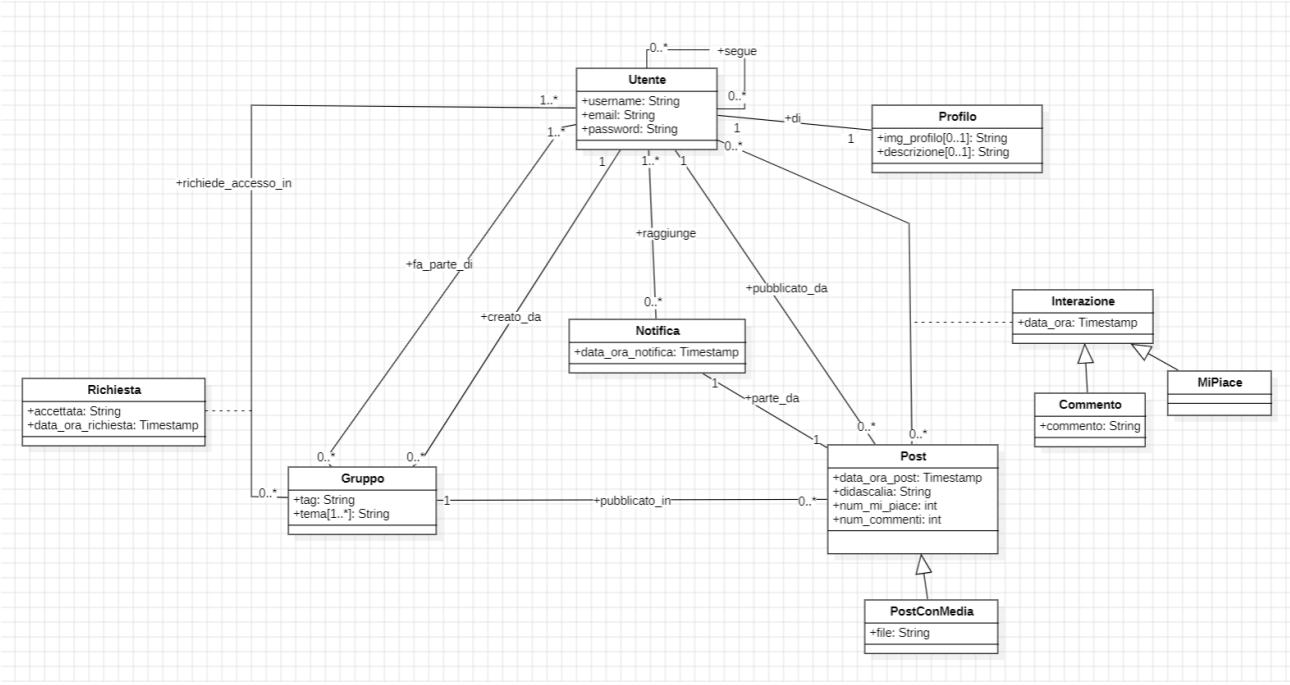


Figure 1: Diagramma UML

### 2.2.1 Precisazioni sul Class Diagram

"Richiesta" è una classe di associazione: un Utente può richiedere l'accesso ad un Gruppo e, nel caso la richiesta venisse rifiutata, allora potrebbe voler riprovare ad accedere tramite una nuova richiesta per lo stesso gruppo.

"Interazione" è una classe di associazione: un Utente può interagire con uno stesso post più volte (lasciando sia un mi piace che un commento), ma soprattutto può farlo in momenti diversi (ad esempio aggiungendo più commenti in date diverse).

Una Notifica riguarda un solo Post, ma raggiunge tutti gli Utenti iscritti ad un Gruppo (quello in cui è stato pubblicato il Post).

La classe "PostConMedia" si riferisce ai post che oltre alla didascalia contengono anche immagini.

## 2.3 Ristrutturazione del Class Diagram

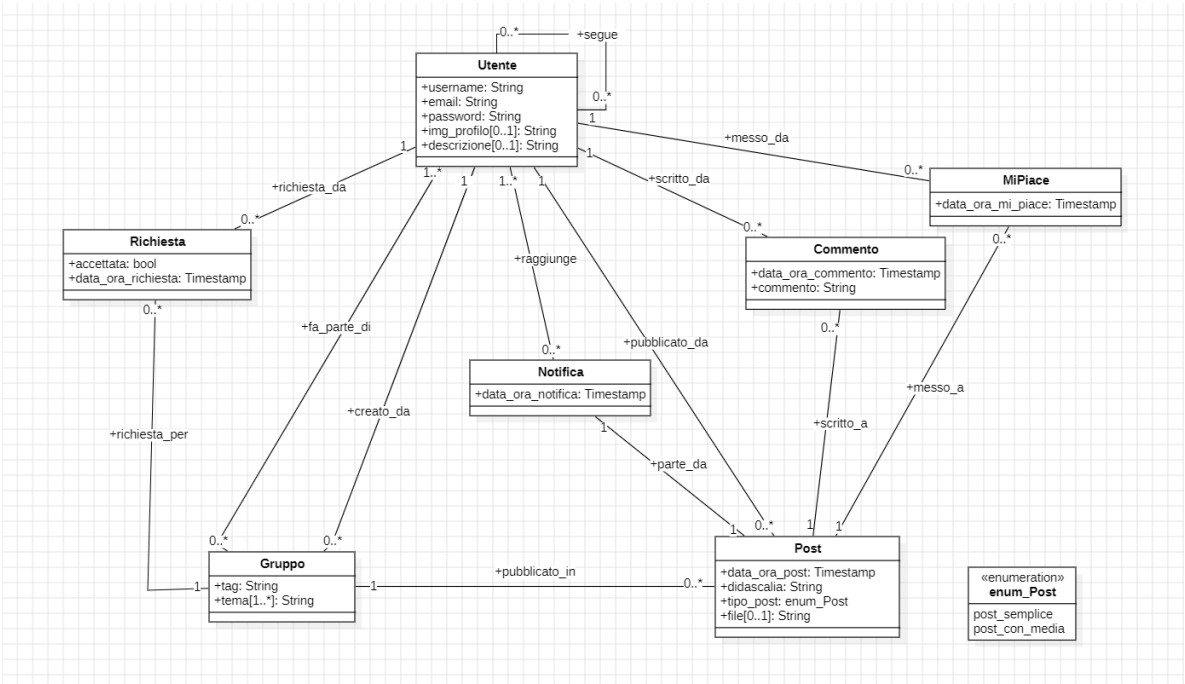


Figure 2: Diagramma UML ristrutturato

### 2.3.1 Accorpamento di entità ed associazioni

Le classi di associazione "Richiesta" e "Interazione" sono state trasformate in classi a sé, ognuna con le proprie associazioni alle altre classi.

La classe Profilo è stata accorpata alla classe Utente, poiché l'associazione tra le due è 1:1, nonostante l'immagine di profilo e la descrizione potrebbero essere attributi nulli.

### 2.3.2 Analisi e eliminazione delle ridondanze

Notiamo che nella classe Post abbiamo gli attributi *"num_mi_piace"* e *"num_commenti"* che, rispettivamente, indicano il numero di mi piace e di commenti aggiunti ad un post. Ma questi attributi sono davvero necessari? I due attributi sono degli interi, facilmente calcolabili se si è a conoscenza di quante interazioni (ovvero quanti Commenti e Mi Piace) sono stati inseriti per un determinato Post.
Per questo nel diagramma ristrutturato i due attributi sono stati eliminati.

### 2.3.3 Eliminazione delle gerarchie

- La gerarchia *"Interazione - Commento / MiPiace"* è totale/disgiunta: un'interazione è sicuramente o un commento o un mi piace, e non può essere entrambi.
  La classe Interazione è stata portata nelle sue sottoclassi Commento e MiPiace, aggiungendo ad esse l'attributo *"data_ora"* (diventato rispettivamente *"data_ora_commento"* e *"data_ora_mi_piace"*).

- La gerarchia *"Post - PostConMedia"* è parziale: un post può essere un post contenente un'immagine, ma può anche non averla.
  La sottoclasse PostConMedia è stata accorpata nella sua classe generale Post: se un post non contiene immagini, allora l'attributo *"file"* sarà nullo. Per distinguere un post con è senza immagini è stato introdotto l'attributo *"tipo_post"* che può assumere due valori (da *enum_post*, *"post_semplice"* e *"post_con_media"*).

# 3 Dizionari

## 3.1 Dizionario delle Classi

| Classe | Descrizione | Attributi |
|---|---|---|
| Utente | Descrittore di ciascun utente iscritto a UninaSocialGroup. | **username** (String): è il nome dell'utente.<br>**email** (String): è un indirizzo mail unico e valido che permette ad ogni utente di iscriversi e accedere alla piattaforma.<br>**password** (String): una password permette ad ogni utente di proteggere il suo account e le sue informazioni.<br>**img_profilo** (String): indica il file path dell'immagine di profilo.<br>**descrizione** (String): una o più frasi che l'utente può utilizzare per descrivere sé stesso brevemente. |
| Gruppo | Descrittore di ciascun gruppo creato dagli utenti. | **tag** (String): è un nome unico che identifica un gruppo.<br>**tema** (String): contiene una o più parole chiave che permettono agli utenti di riconoscere i contenuti di un gruppo. |
| Richiesta | Descrittore di ciascuna richiesta elaborata da un utente per l'accesso ad un gruppo. | **accettata** (bool): è "true" se la richiesta è stata accettata, "false" se è stata rifiutata ed è nulla se non è stata ancora considerata.<br>**data_ora_richiesta** (Timestamp): è la data e l'ora in cui è stata effettuata la richiesta. |

Tabella 1.1: Dizionario delle Classi - cont.

| Classe | Descrizione | Attributi |
|---|---|---|
| Post | Descrittore di ogni contenuto pubblicato da un utente per un certo gruppo. | **data_ora_post** (Timestamp): indica data e ora di pubblicazione del post.<br>**didascalia** (String): sono le informazioni testuali inserite dall'utente che ha pubblicato il post.<br>**tipo_post** (enum_Post): indica se il post ha un'immagine ("post_con_media") oppure no ("post_semplice").<br>**percorso_file** (String): è il file path dell'immagine (se il post ne ha una). |
| Notifica | Descrittore del messaggio di avviso inviato ad ogni utente alla creazione di un post pubblicato in un gruppo di cui fa parte. | **data_ora_notifica** (Timestamp): indica data e ora di invio di una notifica. |
| Commento | Descrittore del contenuto testuale che un utente lascia per commentare un post. | **data_ora_commento** (Timestamp): indica data e ora di pubblicazione di un commento.<br>**commento** (String): è l'effettivo contenuto testuale del commento. |
| MiPiace | Descrittore del "mi piace" lasciato da un utente ad un post. | **data_ora_mi_piace** (Timestamp): indica data e ora in cui un utente mette mi piace al post. |

Tabella 1.2: Dizionario delle Classi - fine.

## 3.2 Dizionario delle Associazioni

| Associazione | Descrizione | Molteplicità delle classi coinvolte |
|---|---|---|
| segue | Un utente può seguire molti utenti, o nessun utente. | Utente [0..*] – Utente[0..*] |
| richiesta_da | Una richiesta è fatta da un solo utente, ma un utente può fare molte richieste. | Richiesta [1] – Utente [0..*] |
| richiesta_per | Una richiesta è fatta per l'accesso a un solo gruppo, ma per un gruppo possono essere fatte molte richieste d'ingresso. | Richiesta [1] – Gruppo [0..*] |
| creato_da | Un gruppo è creato da un solo utente, e un utente può creare molti gruppi (o nessuno). | Gruppo [1] – Utente [0..*] |
| fa_parte_di | Un utente può fare parte di nessuno o molti gruppi; di un gruppo possono far parte molte persone, e sicuramente vi parteciperà il creatore. | Utente [0..*] – Gruppo [1..*] |
| pubblicato_in | Un post è pubblicato in un solo gruppo, e in un gruppo possono essere pubblicati molti post (o nessuno). | Post [1] – Gruppo [0..*] |
| pubblicato_da | Un post è pubblicato da un solo utente, e un utente può pubblicare molti post (o nessuno). | Post [1] – Utente [0..*] |
| parte_da | Una notifica parte da un solo post, e alla creazione di un post è associata una sola notifica. | Notifica [1] – Post [1] |

Tabella 2.1: Dizionario delle Associazioni - cont.

| Associazione | Descrizione | Molteplicità delle classi coinvolte |
|---|---|---|
| raggiunge | Una notifica può raggiungere molti utenti (sicuramente raggiunge il creatore del gruppo in cui è stato pubblicato il post); un utente può ricevere molte notifiche (o nessuna). | Notifica [1..*] – Utente [0..*] |
| scritto_da | Un commento è scritto da un solo utente, e un utente può scrivere molti commenti (o nessuno). | Commento [1] – Utente [0..*] |
| scritto_a | Un commento è scritto sotto ad un unico post, mentre un post può avere molti commenti (o nessuno). | Commento [1] – Post [0..*] |
| messo_da | Un mi piace è messo da un solo utente, e un utente può mettere molti mi piace (o nessuno). | MiPiace [1] – Utente [0..*] |
| messo_a | Un mi piace è messo ad un unico post, mentre un post può avere molti mi piace (o nessuno). | MiPiace [1] – Post [0..*] |

Tabella 2.2: Dizionario delle Associazioni - fine.

## 3.3 Dizionario dei Vincoli

| Vincolo | Tipo | Descrizione |
|---|---|---|
| notNullUserName | di dominio | L'attributo "username" della classe Utente non può essere NULL. |
| checkEmailValida | di dominio | L'attributo "email" della classe Utente deve contenere una combinazione non nulla di lettere, numeri e simboli, seguiti da una chiocciola ("@"), altre lettere/simboli, un punto ("."), e finire con almeno due lettere. |
| checkPasswordValida | di dominio | Il valore dell'attributo "password" della classe Utente deve essere lungo almeno 8 caratteri, e avere almeno una lettera maiuscola, una minuscola, un numero e un carattere speciale. |
| notNullTema | di dominio | L'attributo "tema" della classe Gruppo non può essere NULL. |
| notNullDidascalia | di dominio | L'attributo "didascalia" della classe Post non può essere NULL. |
| typePost | di n-pla | Se l'attributo "file" è NULL, allora l'attributo "tipo_post" assume il valore "post_semplice", altrimenti sarà "post_con_media". |
| validFile | di dominio | Il valore dell'attributo "file" della classe Post deve terminare per ".jpg" oppure ".png". |
| notNullCommento | di dominio | L'attributo "commento" della classe Commento non può essere NULL. |
| validTimeCommento | interrelazionale | Data e ora di un commento non possono essere antecedenti alla data di pubblicazione del post a cui si riferisce il commento. |
| validTimeMiPiace | interrelazionale | Data e ora di un mi piace non possono essere antecedenti alla data di pubblicazione del post a cui si riferisce il mi piace. |

Tabella 3.1: Dizionario dei Vincoli - cont.

| Vincolo | Tipo | Descrizione |
|---|---|---|
| uniqueMiPiace | intrarelazionale | Un utente può mettere mi piace ad un post solo una volta. |
| validTimeNotifica | interrelazionale | Data e ora di una notifica non possono essere antecedenti alla data di pubblicazione del post a cui si riferisce la notifica. |
| uniqueNotifica | intrarelazionale | Per ogni post può esistere una sola notifica, e una notifica deve raggiungere un utente una sola volta. |
| uniqueSeguito | intrarelazionale | Un utente può seguire un altro utente solo una volta. |
| uniquePartecipazione | intrarelazionale | Un utente può far parte di un gruppo una sola volta. |
| validRichiesta | intrarelazionale | Se esiste già una richiesta (di un dato utente per un dato gruppo) ancora da considerare, allora non si può aggiungere una nuova richiesta. |
| validRichiestaPartecipazione | interrelazionale | Se un utente fa già parte di un gruppo, allora non può fare una nuova richiesta per quel gruppo. |
| immutableDataOra | di dominio | Gli attributi che indicano una data e un'ora (vedi *Post, Notifica, Commento e Mi_piace* in 3.1) non possono essere modificati. |

Tabella 3.2: Dizionario dei Vincoli - fine.

# 4 Progettazione Logica

## 4.1 Tabelle

Leggenda:

- le parole in **grassetto** sono i nomi delle tabelle;
- le parole sottolineate sono le chiavi primarie;
- le parole sottolineate due volte sono le chiavi esterne;
- se più attributi insieme formano la chiave primaria, allora sono chiusi tra [parentesi quadre].

| | |
|---|---|
| **Utente** | (email, username, password, img_profilo, descrizione) |
| **Seguiti** | ([email_utente_1, email_utente_2]) |
| **Gruppo** | (tag, tema, email_creatore) |
| **Richiesta** | ([email_utente, tag_gruppo, data_ora_richiesta], accettato) |
| **Partecipanti_al_gruppo** | (email_membro, tag_gruppo) |
| **Post** | (id_post, data_ora_post, didascalia, tipo_file, percorso_file, email_creatore_post, tag_gruppo) |
| **Notifica** | (id_notifica, data_ora_notifica, id_post_pubblicato) |
| **Raggiunge** | (email_utente, id_notifica) |
| **Commento** | (id_commento, id_post, email_utente, data_ora_commento, commento) |
| **Mi_Piace** | (id_mi_piace, id_post, email_utente, data_ora_mi_piace) |

Tabella 4: Schema logico

## 4.2 Traduzione delle associazioni in tabelle

- **segue** è un'associazione *N:M*, per questo è stata tradotta come una tabella separata, "Seguiti", con i suoi attributi *"email_utente_1"* (l'utente che segue) e *"email_utente_2"* (l'utente seguito);
- **richiesta_da** e **richiesta_per** sono state accorpate in due nuovi attributi di Richiesta, rispettivamente *"email_utente"* e *"tag_gruppo"*;
- **creato_da** è stata tradotta come attributo di Gruppo, *"email_creatore"*;
- **fa_parte_di** è un'altra associazione *N:M*, quindi è stata tradotta nella tabella "Partecipanti_al_gruppo" che ha due attributi: *"email_membro"* e *"tag_gruppo"*;
- **pubblicato_da** e **pubblicato_in** sono state tradotte come due nuovi attributi di Post, *"email_creatore"* e *"tag_gruppo"*;
- **parte_da** è stata tradotta come attributo di Notifica, *"id_post_pubblicato"*;
- **raggiunge** è un'associazione *N:M*, quindi è stata tradotta nella tabella "Raggiunge", con gli attributi *"email_utente"* e *"id_notifica"*;
- **scritto_da** e **scritto_a** sono stati rispettivamente tradotti negli attributi *"email_utente"* e *"id_post"* di Commento.
- **messo_da** e **messo_a** sono stati rispettivamente tradotti negli attributi *"email_utente"* e *"id_post"* di MiPiace.

## 4.3 Scelta delle Primary Key

Per la tabella Post un possibile candidato ad essere PK è l'insieme di attributi *"tag_gruppo, email_creatore_post, data_ora_post"*. Per comodità è stato introdotto un ID, *"id_post"*. Lo stesso ragionamento è stato fatto per le tabelle Commento, Mi_Piace e Notifica.

# 5 Progettazione Fisica

Per la progettazione fisica è stato utilizzata *pgAdmin*, ovvero un'interfaccia grafica che consente di amministrare database di PostgreSQL.

## 5.1 Creazione tabelle

### 5.1.1 Utente

```
create table Utente(
    username varchar(20) not NULL,
    email varchar(60) Primary key,
    password varchar(60) Not NULL,
    img_profilo varchar(100),
    descrizione varchar(300)
);
```

Figure 3: Creazione Utente - cont.

```
ALTER TABLE Utente
ADD CONSTRAINT check_email_valida
        CHECK (email ~'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$');
```

Figure 4: creazione Utente - cont.

```
ALTER TABLE Utente
ADD CONSTRAINT check_password_valida
CHECK (
    password ~ '[0-9]' AND
    password ~ '[A-Z]' AND
    password ~ '[a-z]' AND
    password ~ '[!?@#%&$]' AND
    LENGTH(password) >= 8
);
```

Figure 5: Creazione Utente - fine.

### 5.1.2 Seguiti

```
create table seguiti(
    email_utente_1 varchar(60),
    email_utente_2 varchar(60),
    constraint fk_utente_1 Foreign key (email_utente_1)references Utente(email) ON DELETE CASCADE,
    constraint fk_utente_2 Foreign key (email_utente_2)references Utente(email) ON DELETE CASCADE,
    constraint segui_unico UNIQUE (email_utente_1, email_utente_2)
);

```

Figure 6: Creazione Seguiti.

### 5.1.3 Gruppo

```
create table Gruppo(
    tag varchar(20) Primary key,
    tema varchar(20) not null,
    email_creatore varchar(60)not null,
    constraint fk_creatore foreign key(email_creatore) references Utente (email)
);
```

Figure 7: Creazione Gruppo.

### 5.1.4 Richiesta

```
create table richiesta(
    accettato BOOLEAN,
    data_ora_richiesta timestamp default current_timestamp,
    email_utente varchar(60),
    tag_gruppo varchar(20),
    constraint fk_tag foreign key(tag_gruppo) references Gruppo (tag),
    constraint fk_utente foreign key(email_utente) references Utente (email)
)
```

Figure 8: Creazione Richiesta.

### 5.1.5 Partecipanti_al_gruppo

```
create table Partecipanti_al_gruppo(
    email_membro varchar(60),
    tag_gruppo varchar(20),
    constraint fk_email foreign key (email_membro) references Utente(email) on delete cascade,
    constraint fk_tag foreign key (tag_gruppo) references gruppo(tag) on delete cascade
);
```

Figure 9: Creazione Partecipanti_al_gruppo - cont.

```
ALTER TABLE Partecipanti_al_gruppo
    ADD CONSTRAINT partecipa_una_volta UNIQUE(email_membro, tag_gruppo);
```

Figure 10: Creazione Partecipanti_al_gruppo - fine.

### 5.1.6 Post

```
CREATE TYPE tipo_post AS ENUM ('post_semplice','post_con_media');
```

Figure 11: Creazione Enum *tipo_post*.

```
create table Post(
    id_post SERIAL PRIMARY KEY,
    data_ora_post timestamp default current_timestamp,
    didascalia varchar(300) NOT NULL ,
    percorso_file varchar(100),
    tipo_file tipo_post,
    email_creatore_post varchar(60),
    tag_gruppo varchar(20),
    constraint fk_tag foreign key (tag_gruppo) references gruppo(tag),
    constraint fk_email_creatore_post foreign key (email_creatore_post) references utente(email)
);
```

Figure 12: Creazione Post.

### 5.1.7 Notifica

```sql
create table Notifica(
    id_notifica SERIAL primary key ,
    data_ora_notifica timestamp default current_timestamp,
    id_post_pubblicato integer,
    constraint fk_post foreign key (id_post_pubblicato) references post(id_post)
);
```

Figure 13: Creazione Notifica - cont.

```sql
ALTER TABLE Notifica
ADD CONSTRAINT  notifica_per_post UNIQUE (id_notifica, id_post_pubblicato);
```

Figure 14: Creazione Notifica - fine.

### 5.1.8 Raggiunge

```sql
create table Raggiunge(
    email_utente varchar(60),
    id_notifica INTEGER,
    constraint fk_email foreign key (email_utente) references utente(email),
    constraint fk_id_notifica foreign key (id_notifica) references notifica(id_notifica),
    Constraint notifica_unica unique (email_utente,id_notifica)
);
```

Figure 15: Creazione Raggiunge.

### 5.1.9 Commento

```sql
create table Commento(
    id_commento SERIAL primary key,
    data_ora_commento timestamp default current_timestamp ,
    commento varchar(300)not null,
    email_utente varchar(60),
    id_post integer,
    constraint fk_email foreign key (email_utente) references Utente(email),
    constraint fk_post foreign key (id_post) references post(id_post)
);
```

Figure 16: Creazione Commento.

### 5.1.10 Mi_piace

```sql
create table Mi_piace(
    id_mi_piace SERIAL primary key,
    data_ora_mi_piace timestamp default current_timestamp,
    email_utente varchar(60),
    id_post integer,
    constraint fk_email foreign key (email_utente) references Utente(email),
    constraint fk_post foreign key (id_post) references post(id_post),
    Constraint mi_piace_unica unique (email_utente, id_post)
);
```

Figure 17: Creazione Mi_piace.

## 5.2 Funzioni e Triggers

### 5.2.1 filePresenteInPost

Alla creazione di un post, il trigger seguente assegna il tipo al post (vedi *"typePost"* in 3.3 ).

```
create or replace function filePresenteInPost()
returns trigger
language plpgsql
as $filePresenteInPost$
BEGIN
    IF NEW.percorso_file IS NULL THEN
        NEW.tipo_file = 'post_semplice';
    ELSE
        NEW.tipo_file = 'post_con_media';
    END IF;
    return new;
END;
$filePresenteInPost$;
```

Figure 18: Funzione FilePresenteInPost

```
CREATE TRIGGER filePresenteInPost
BEFORE INSERT ON Post
FOR EACH ROW EXECUTE FUNCTION filePresenteInPost();
```

Figure 19: Trigger FilePresenteInPost

### 5.2.2 primoPartecipante

Alla creazione di un nuovo gruppo, tramite il seguente trigger verrà aggiunto il creatore del gruppo alla lista dei partecipanti (vedi *"Partecipanti_al_gruppo"* in 4.1 ).

```
CREATE OR REPLACE FUNCTION primoPartecipante()
RETURNS TRIGGER
language plpgsql
AS $primoPartecipante$
BEGIN

    INSERT INTO Partecipanti_al_gruppo
        VALUES(NEW.email_creatore, NEW.tag);
return new;

END;
$primoPartecipante$;
```

Figure 20: Funzione primoPartecipante

```
CREATE TRIGGER primoPartecipante
AFTER INSERT ON Gruppo
FOR EACH ROW EXECUTE FUNCTION primoPartecipante();
```

Figure 21: Trigger primoPartecipante

### 5.2.3 accettaNuovoMembro

Quando una richiesta viene accettata, allora l'utente che ha fatto la richiesta viene inserito tra i partecipanti al gruppo indicato nella richiesta (vedi *"Richiesta"* in 3.1 ).

```
1  CREATE OR REPLACE FUNCTION accettaNuovoMembro()
2  RETURNS TRIGGER
3  LANGUAGE plpgsql
4  AS $accettaNuovoMembro$
5  BEGIN
6      IF OLD.accettato IS NULL AND NEW.accettato = true THEN
7          INSERT INTO Partecipanti_al_gruppo (email_membro, tag_gruppo)
8          VALUES (NEW.email_utente, NEW.tag_gruppo);
9      END IF;
10
11     RETURN NEW;
12 END;
13 $accettaNuovoMembro$;
```

Figure 22: Funzione AccettaNuovoMembro

```
1  CREATE TRIGGER accettaNuovoMembro
2  AFTER UPDATE ON Richiesta
3  FOR EACH ROW
4  EXECUTE FUNCTION accettaNuovoMembro();
```

Figure 23: Trigger AccettaNuovoMembro

### 5.2.4 fileValido

Prima dell'inserimento di un nuovo post, il seguente trigger permette di verificare che l'attributo *percorso_file* contenga un file di tipo jpg o png. (vedi *"Post"* in 3.1 ).

```
CREATE OR REPLACE FUNCTION fileValido()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $fileValido$
DECLARE
    valid INTEGER;
    lunghezza INTEGER;
    finale TEXT;
BEGIN
    IF (NEW.percorso_file IS NULL) THEN
        RETURN NEW;
    END IF;
    valid := 0;
    lunghezza := LENGTH(NEW.percorso_file);
    finale := SUBSTR(NEW.percorso_file, lunghezza-3, lunghezza);

    IF (finale = '.png' OR finale = '.jpg') THEN
        valid := 1;
    END IF;

    IF (valid <> 1) THEN
        RAISE EXCEPTION 'il file deve essere png o jpg';
    END IF;

    RETURN NEW;
END;
$fileValido$;
```

Figure 24: Funzione FileValido

```
CREATE TRIGGER fileValido
BEFORE INSERT ON Post
FOR EACH ROW EXECUTE FUNCTION fileValido();
```

Figure 25: Trigger FileValido

### 5.2.5 imgProfiloValido

Analoga alla funzione precedente, questa volta il controllo viene fatto sulla validità di *"img_profilo"* all'inserimento di un nuovo utente. (vedi *"Utente"* in 3.1 ).

```
1  CREATE OR REPLACE FUNCTION ImgProfiloValido()
2  RETURNS TRIGGER
3  LANGUAGE plpgsql
4  AS $ImgProfiloValido$
5  DECLARE
6      valid INTEGER;
7      lunghezza INTEGER;
8      finale TEXT;
9  BEGIN
10     IF (NEW.img_profilo IS NULL) THEN
11         RETURN NEW;
12     END IF;
13
14     valid := 0;
15     lunghezza := LENGTH(NEW.img_profilo);
16     finale := SUBSTR(NEW.img_profilo, lunghezza-3, lunghezza);
17
18     IF (finale = '.png' OR finale = '.jpg') THEN
19         valid := 1;
20     END IF;
21
22     IF (valid <> 1) THEN
23         RAISE EXCEPTION 'il file per l immagine del profilo deve essere png o jpg';
24     END IF;
25
26     RETURN NEW;
27 END;
28 $ImgProfiloValido$;
```

Figure 26: Funzione ImgProfiloValido

```
1  CREATE TRIGGER ImgProfiloValido
2  BEFORE INSERT ON utente
3  FOR EACH ROW
4  EXECUTE FUNCTION ImgProfiloValido();
```

Figure 27: Trigger ImgProfiloValido

*Nota:* Se l'utente ha la possibilità di modificare la sua immagine di profilo, allora la funzione ImgProfilo valido dovrà essere eseguita non solo prima di un'INSERT (vedi riga 3 del Trigger 27), ma anche prima di un UPDATE.

### 5.2.6 nuovaNotifica

Successivamente alla creazione di un nuovo post, viene creata anche la notifica corrispondente. (vedi *"Notifica"* in 3.1 ).

```
CREATE OR REPLACE FUNCTION NuovaNotifica()
RETURNS TRIGGER
language plpgsql
AS $NuovaNotifica$
DECLARE
BEGIN
    insert into notifica(id_post_pubblicato) values (NEW.id_post);
RETURN NEW;
END;
$NuovaNotifica$;
```

Figure 28: Funzione nuovaNotifica

```
CREATE TRIGGER notificaRaggiungeUtenti
AFTER INSERT ON Notifica
FOR EACH ROW EXECUTE FUNCTION notificaRaggiungeUtenti();
```

Figure 29: Trigger nuovaNotifica

### 5.2.7 notificaRaggiungeUtenti

Successivamente alla creazione di una nuova notifica, questa viene "inviata" a tutti i partecipanti del gruppo in cui il post è stato pubblicato (vedi *"Raggiunge"* in 4.1 ).

```
CREATE OR REPLACE FUNCTION notificaRaggiungeUtenti()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $notificaRaggiungeUtenti$
DECLARE
    recupera_utenti CURSOR FOR
        SELECT PG.email_membro
        FROM Post P
        JOIN Gruppo G ON P.tag_gruppo = G.tag
        JOIN Partecipanti_al_gruppo PG ON G.tag = PG.tag_gruppo
        WHERE P.id_post = NEW.id_post_pubblicato;

    email Utente.email%TYPE;
BEGIN
    OPEN recupera_utenti;

    LOOP
        FETCH recupera_utenti INTO email;
        EXIT WHEN NOT FOUND; -- Corretto NOT FOUND
        INSERT INTO Raggiunge VALUES (email, NEW.id_notifica);
    END LOOP;

    CLOSE recupera_utenti;
RETURN NEW;
END;
$notificaRaggiungeUtenti$;
```

Figure 30: Funzione notificaRaggiungeUtenti

```
CREATE TRIGGER notificaRaggiungeUtenti
AFTER INSERT ON Notifica
FOR EACH ROW EXECUTE FUNCTION notificaRaggiungeUtenti();
```

Figure 31: Trigger notificaRaggiungeUtenti

### 5.2.8 postGruppoValido

La funzione seguente controlla se l'utente che pubblica il post appartiene al gruppo in cui lo si sta pubblicando, altrimenti segnala un errore. (vedi *"Partecipanti_al_gruppo"* in 4.1 ).

```
CREATE OR REPLACE FUNCTION postGruppoValido()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $postGruppoValido$
DECLARE
    valid INTEGER := 0;
BEGIN
    SELECT COUNT(*) INTO valid
    FROM Partecipanti_al_gruppo as PG
    WHERE PG.tag_gruppo = NEW.tag_gruppo AND PG.email_membro = NEW.email_creatore_post;

    IF (valid <> 1) THEN
        RAISE EXCEPTION 'L''utente non può pubblicare in questo gruppo';
    END IF;

    RETURN NEW;
END;
$postGruppoValido$;
```

Figure 32: Funzione postGruppoValido

```
CREATE TRIGGER postGruppoValido
BEFORE INSERT ON Post
FOR EACH ROW
EXECUTE FUNCTION postGruppoValido();
```

Figure 33: Trigger postGruppoValido

### 5.2.9 richiestaGruppoValida

La funzione seguente controlla che non ci sia già una richiesta di un determinato utente per un determinato gruppo, altrimenti segnala un errore (vedi *"validRichiesta"* in 3.3 ).

```
CREATE OR REPLACE FUNCTION richiestaGruppoValida()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $richiestaGruppoValida$
DECLARE
    richieste_in_corso INT;
BEGIN
    SELECT COUNT(*) INTO richieste_in_corso
    FROM Richiesta AS R
    WHERE R.email_utente = NEW.email_utente
        AND R.tag_gruppo = NEW.tag_gruppo
        AND R.accettato IS NULL;

    IF richieste_in_corso > 0 THEN
        RAISE EXCEPTION 'Richiesta inserita non valida: esiste già una richiesta ancora da considerare.
    END IF;

    RETURN NEW;
END;
$richiestaGruppoValida$;
```

Figure 34: Funzione richiestaGruppoValida

```
CREATE TRIGGER richiestaGruppoValida
BEFORE INSERT ON Richiesta
FOR EACH ROW
EXECUTE FUNCTION richiestaGruppoValida();
```

Figure 35: Trigger richiestaGruppoValida

### 5.2.10 utenteInGruppoValida

La funzione seguente controlla che l'utente che sta effettuando la richiesta per entrare in un gruppo non appartenga già a quel gruppo, altrimenti segnala un errore. (vedi *"validRichiestaPartecipazione"* in 3.3 ).

```
1  CREATE OR REPLACE FUNCTION utenteInGruppoValida()
2  RETURNS TRIGGER
3  language plpgsql
4  AS $utenteInGruppoValida$
5  DECLARE
6      utente_in_gruppo INT;
7  BEGIN
8
9      SELECT COUNT(*) INTO utente_in_gruppo
10         FROM Partecipanti_al_gruppo AS P
11         WHERE P.email_membro = NEW.email_utente
12                 AND P.tag_gruppo = NEW.tag_gruppo;
13
14     IF ( utente_in_gruppo > 0 ) THEN
15         RAISE EXCEPTION 'Richiesta inserita non valida: l utente fa gia parte del gruppo';
16     END IF;
17     RETURN NEW;
18 END;
19 $utenteInGruppoValida$;
20
```

Figure 36: Funzione utenteInGruppoValida

```
1  CREATE TRIGGER utenteInGruppoValida
2  BEFORE INSERT ON Richiesta
3  FOR EACH ROW
4  EXECUTE FUNCTION utenteInGruppoValida();
```

Figure 37: Trigger utenteInGruppoValida

### 5.2.11 miPiaceGruppoValido

Prima che un utente inserisca un mi piace, la funzione seguente controlla che l'utente appartenga al gruppo in cui il post è pubblicato. (vedi *"Partecipanti_al_gruppo"* in 4.1 ).

```
CREATE OR REPLACE FUNCTION MiPiaceGruppoValido()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $MiPiaceGruppoValido$
DECLARE
    gruppo post.tag_gruppo%TYPE;
    partecipazione INTEGER := 0;


BEGIN

    SELECT Post.tag_gruppo INTO gruppo
    FROM POST as P
    WHERE P.id_post = NEW.id_post;


    SELECT COUNT(*) INTO partecipazione
    FROM Partecipanti_al_gruppo as PG
    WHERE PG.tag_gruppo = gruppo AND PG.email_membro = NEW.email_utente;

    IF (partecipazione <> 1) THEN
        RAISE EXCEPTION 'L''utente non può mettere mi piace a questo Post';
    END IF;

    RETURN NEW;
END;
$MiPiaceGruppoValido$;
```

Figure 38: Funzione miPiaceGruppoValido

```
CREATE TRIGGER MiPiaceGruppoValido
BEFORE INSERT ON Mi_Piace
FOR EACH ROW
EXECUTE FUNCTION MiPiaceGruppoValido();
```

Figure 39: Trigger miPiaceGruppoValido

### 5.2.12 commentoGruppoValido

Analoga alla funzione precedente, ma il controllo viene fatto sulla tabella *Commento* (vedi *"Partecipanti_al_gruppo"* in 4.1 ).

```
CREATE OR REPLACE FUNCTION CommentoGruppoValido()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $CommentoGruppoValido$
DECLARE
    gruppo post.tag_gruppo%TYPE;
    partecipazione INTEGER := 0;


BEGIN

    SELECT P.tag_gruppo INTO gruppo
    FROM POST as P
    WHERE P.id_post = NEW.id_post;


    SELECT COUNT(*) INTO partecipazione
    FROM Partecipanti_al_gruppo as PG
    WHERE PG.tag_gruppo = gruppo AND PG.email_membro = NEW.email_utente;

    IF (partecipazione <> 1) THEN
        RAISE EXCEPTION 'L''utente non può commentare sotto a questo Post';
    END IF;

    RETURN NEW;
END;
$CommentoGruppoValido$;
```

Figure 40: Funzione commentoGruppoValido

```
CREATE TRIGGER CommentoGruppoValido
BEFORE INSERT ON Commento
FOR EACH ROW
EXECUTE FUNCTION CommentoGruppoValido();
```

Figure 41: Trigger commentoGruppoValido