Note-Bridge Forum Design Report

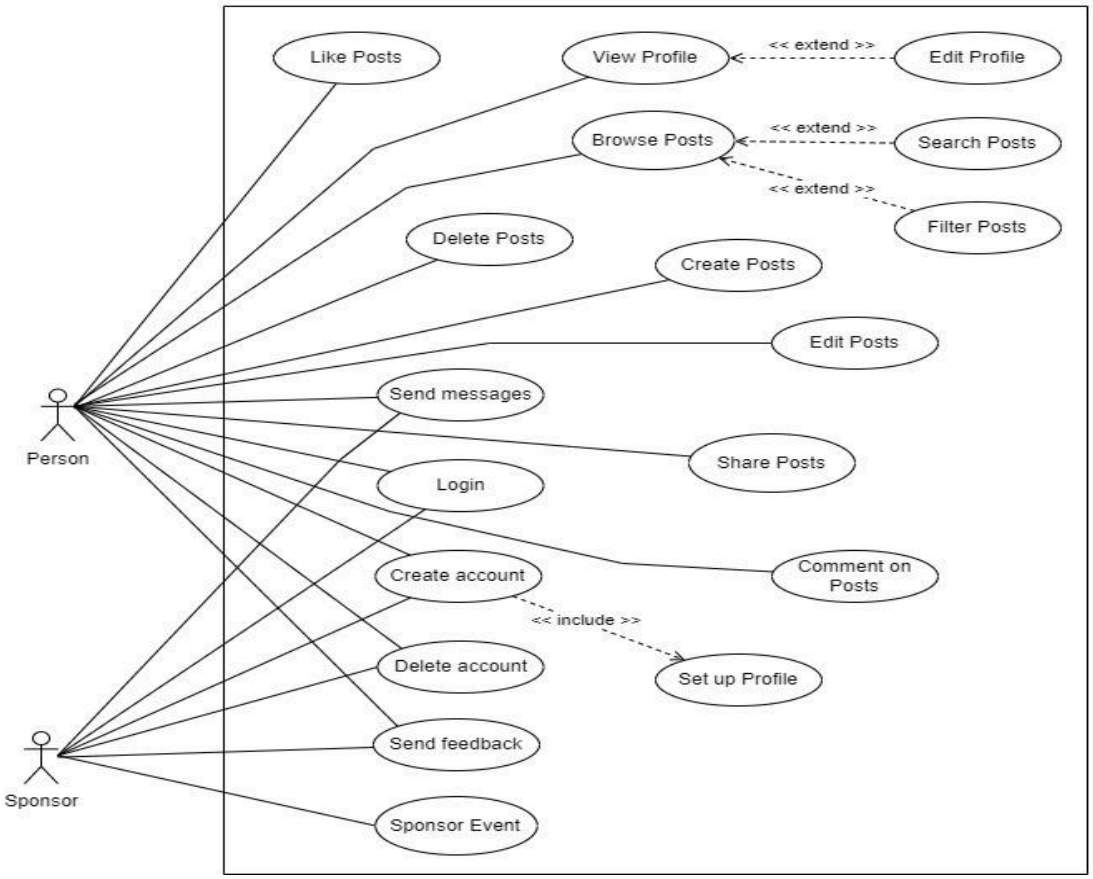
Group 1

Rebah Ozkoc 3363775
Anil Sen 3367606
Ilgin Simay Ozcan 3362779
Enescu Alexandru-Cristian - 3055779
Simeon Nikolov - 3163989
Adham Ehab Magdy Selim - 2948486

07/06/2024

Content Table

UML Use Case Diagram



The use case diagram illustrates a top-level description of the functionality of our web application. It shows how users can make use of the system. There are two main actors which interact with the system which are **Person** and **Sponsor**. There are some common and some specific activities for each user type.

The common activities are **logging in, creating an account, deleting an account, and sending messages.**

The use cases of the **Person** can be explained like this:

Person Interactions

Create Account: A user can create a new account. This is the foundational step for a user to access additional functionalities. To register a new account, the user completes and submits a form.
Login: After creating an account, a user must log in to access their personalized experience and interact with the platform. The user enters his unique credentials.
Create Posts: Allows the user to create new content on the platform. The user completes a form in which he enters the details of the post.
Edit Posts: Users can modify their previously created posts.
Delete Posts: Users have the ability to remove their posts.
Browse Posts: Users can view a collection of posts from other users.
Search Posts: Enhanced browsing capability allowing users to search for posts based on keywords or tags.
Filter Posts: Users can apply filters to refine the list of posts they are browsing.
Like Posts: Users can express their appreciation for a post by "liking" it.
Comment on Posts: Users can add comments to a post.
Add Interest on Posts: Users can indicate that they are interested in a post.
Delete Comment on Posts: Users have the ability to remove comments on posts.
Share Posts: This function allows users to share posts with others, either within the same platform or externally.
Send Messages: Users can send private messages to other users.
View Profile: Allows users to view their own or other users' profiles.
Edit Profile: Users can modify their profile information.
Delete Account: Users can permanently remove their account from the platform.
Send Feedback: Persons can provide feedback to the platform, possibly about the overall experience or other features.

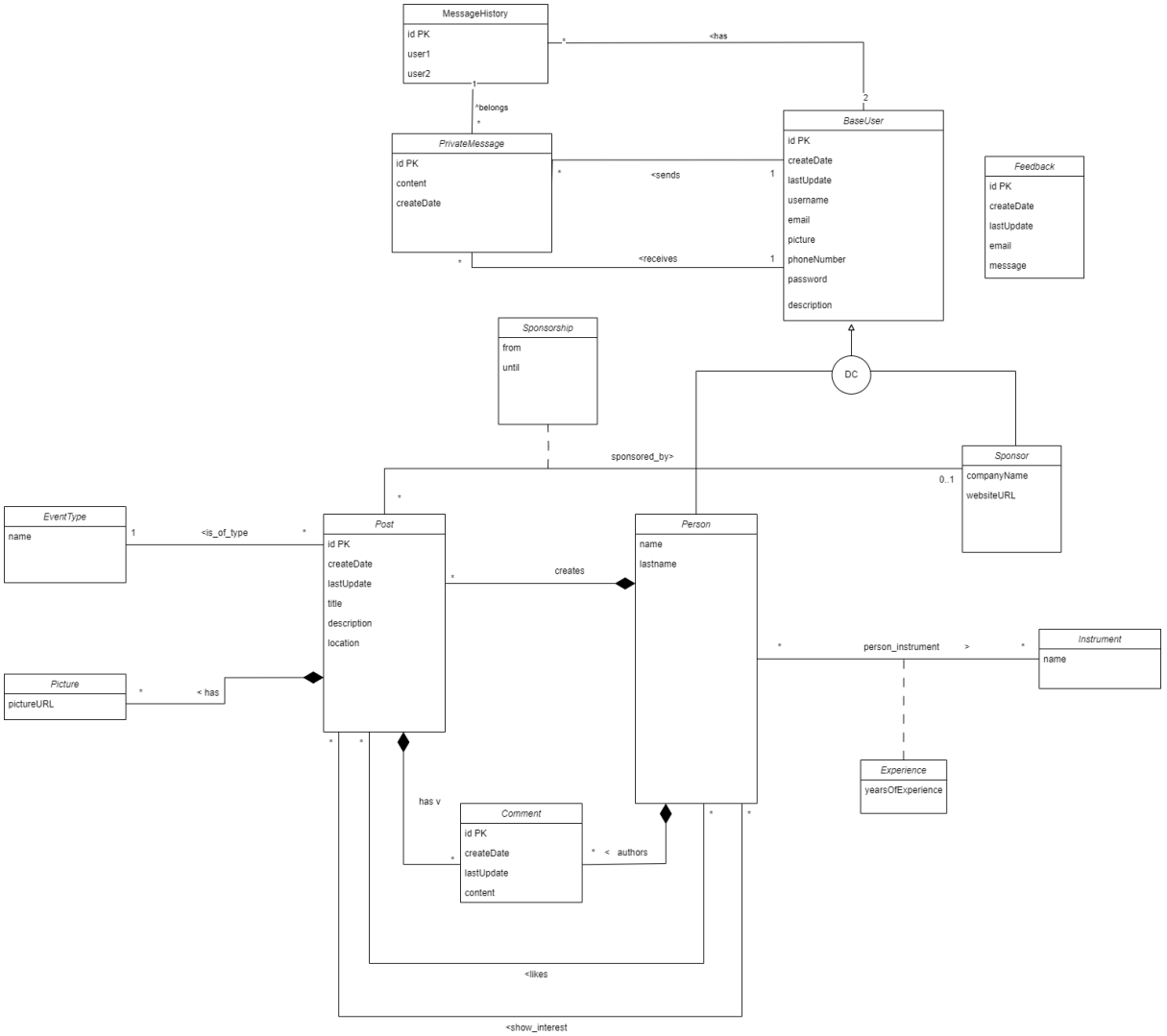
The use cases of the **Sponsor** can be explained like this.

Sponsor Interactions

Create Account: Similar to regular users, sponsors can create an account to engage with the platform.
Login: Sponsors need to log in to manage their sponsored content and events.
Sponsor Event: Sponsors can sponsor the existing events, which could be advertised or featured on the platform.

Send Feedback: Sponsors can provide feedback to the platform, possibly about the sponsorship experience or other features.

UML Class Diagram



In this diagram, there are 13 classes and each of them will be explained. Firstly, we have a **BaseUser** class which is the superclass of two other classes namely **Person** and **Sponsor**. The relation between the superclass and the subclasses is disjoint and complete. This provides that each user must be either a sponsor or a normal user and not both. Common fields like **username** and

password are stored in the BaseUser class and specific fields like **companyName** are stored in the subclasses.

The **BaseUser** class is connected to the **PrivateMessage** table with two many-to-one relationships. These relationships are **sends** and **receives** relationships. This table will help us to store the content and the other details of the private messages between the users. **PrivateMessage** is connected to the **BaseUser** class instead of the subclass and this allows messaging between the different user types.

The **Post** class has a one-to-many composition relationship with the **Person** class. Each post has to belong to one person and the post is deleted if the person is deleted. The **Post** class has attributes like **title, description,** and **createDate**. The **Post** class has a many-to-one composition relationship with the **Picture** class. Each picture has to belong to one post and the picture is deleted if the post is deleted. Each post has an event type and this information is stored in the **EventType** class. The **Post** class has a many-to-many relationship with the **Person** class which shows the **likes** information.

The **Comment** class is connected to the **Post** class with a one-to-many composition relationship. Each comment has to belong to one post and the comment is deleted if the post is deleted. The **Comment** class is connected to the **Person** class with a one-to-many composition relationship. So each user can have multiple comments but a comment must belong to one and only one person and the comment is deleted if the user's account is deleted.

The **Sponsor** class has a **sponsoredBy** relationship with the **Post** class. A sponsor can be a sponsor to many posts and a post can be sponsored by one or zero sponsor. The association class **Sponsorship** stores the information about this relation such as the duration of the sponsorship.

The **Instrument** class has a many-to-many relationship with the **Person** class. The **Experience** association class for this relationship stores the information on how long the user is experienced with this instrument.

The **Feedback** class is a separate class that is not connected to another class, since it doesn't require having an account or login.

SQL Schema

```
CREATE TABLE BaseUser (
    id SERIAL PRIMARY KEY,
    createDate TIMESTAMP NOT NULL,
    lastUpdate TIMESTAMP,
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    picture VARCHAR(255),
    phoneNumber VARCHAR(15),
    password VARCHAR(255) NOT NULL
   description TEXT
);
```

Because **BaseUser** is a class in our class diagram from which **Sponsor** and **Person** generalize, we put the fields applying for both **Sponsor** and **Person**. Moreover, a unique ID is assigned to that table in order to uniquely define each **BaseUser**.

```
CREATE TABLE Person(
    id SERIAL PRIMARY KEY,
    name VARCHAR(255),
    lastname VARCHAR(255),
    FOREIGN KEY(id) REFERENCES BaseUser(id) ,
);
```

Since **Person** is generalizing from **BaseUser** and there is a disjoint and covering relationship, we made use of a generic solution, hence creating two more tables for our subclasses, which are **Person** and **Sponsor**. In this table, we refer to the superclass ID as a result of this generalization.

```
CREATE TABLE Sponsor (
    id SERIAL PRIMARY KEY,
    companyName VARCHAR(255),
    websiteURL VARCHAR(255),
    FOREIGN KEY(id) REFERENCES BaseUser(id)
);
```

The **Sponsor** class is also generalizing from the **BaseUser**; therefore, we adopted the same approach as the one adopted in the **Person** table. We again referred to the superclass **BaseUser** to give the meaning for the generalization relationship as a result of our generic solution.

```
CREATE TABLE Instrument (
    name VARCHAR(255) PRIMARY KEY

);
```

The **Instrument** table is also created to later keep the records of the persons interested in instruments. In this table, we typically put one column which is the name of the instrument, and make it uniquely define a row.

```
CREATE TABLE PersonInstrument (
    personId INT REFERENCES Person(id) ON DELETE CASCADE,
    instrumentName VARCHAR(255) REFERENCES Instrument(name)  ON DELETE
CASCADE,
    yearsOfExperience FLOAT,
    PRIMARY KEY (personId, instrumentName)
);
```

Having defined **Person** and **Instrument** tables, we aimed to have the many-to-many relationship between our **Person** and **Instrument** class and their relationship with each other, which is “**Person** is interested in an **Instrument**”. As a result of using the association class, our relationship table references two keys from the **Person** and **Instrument** table and has the combination of these as its primary key.

```
CREATE TABLE Post (
    id SERIAL PRIMARY KEY,
    createDate TIMESTAMP NOT NULL,
    lastUpdate TIMESTAMP,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    location VARCHAR(255),
    sponsoredBy INT REFERENCES Sponsor(id),
    sponsoredFrom TIMESTAMP,
    sponsoredUntil TIMESTAMP,
    eventType VARCHAR(255) REFERENCES EventType(name),
    personId INT NOT NULL,
    FOREIGN KEY(personId) REFERENCES Person(id) ON DELETE CASCADE
);
```

The **Post** table is another crucial table just like the **BaseUser** table, due to its many relationships among other classes in the class diagram. Since **Post** is created by a **Person** and once the **Person** is deleted, their **Posts** are also deleted. We referenced Person ID and put “ON DELETE CASCADE” to satisfy the requirements of a composition relationship in the class diagram. Moreover, the **Post** is sponsored by a **Sponsor**, and similarly, we also referenced its ID as “sponsoredBy”. We also have “sponsoredFrom” and “sponsoredUntil” fields as a result of having an association class in “many-to-one” relationship with **Sponsor**. Lastly, we referenced **eventType** inside our table since a **Post** is guaranteed to have exactly one **eventType**.

```
CREATE TABLE EventType (
    name VARCHAR(255) PRIMARY KEY
);
```

Because we are also planning to put event types on **Posts**, just like our **Instrument** table, we again used only names and treated it as the primary key.

```
CREATE TABLE Picture (
    pictureURL VARCHAR(255) PRIMARY KEY,
    postId INT NOT NULL,
    FOREIGN KEY(postId) REFERENCES Post(id) ON DELETE CASCADE
);
```

Since we are aiming to also support a feature in which a **Person** is able to upload pictures for their **Post** while creation, a separate **Picture** table is needed because a **Post** can have multiple **Pictures**. Furthermore, once a **Post** is deleted, there is no need for storing its **Pictures**; therefore, while referencing a **Post** inside the table, we also added “ON DELETE CASCADE” to fully comply with our class diagram.

```
CREATE TABLE Comment (
    id SERIAL PRIMARY KEY,
    createDate TIMESTAMP NOT NULL,
    lastUpdate TIMESTAMP,
    content TEXT,
    postId INT NOT NULL,
    personId INT NOT NULL,
    FOREIGN KEY(postId) REFERENCES Post(id) ON DELETE CASCADE,
    FOREIGN KEY (personId) REFERENCES Person(id) ON DELETE SET NULL
);
```

Another feature we are aiming for regarding a **Post** is that a **Person** should be able to make comments on a **Post**. Since a **Comment** can belong to exactly one **Person** and one **Post**, we referenced these tables inside our **Comment** table. Moreover, since there is a composition relationship between **Post**-**Comment**, we put an additional statement “ON DELETE CASCADE " to delete the **Comments** in case the **Post** they belong to is deleted.

```
CREATE TABLE PersonLikesPost (
    personId INT REFERENCES Person(id)  ON DELETE CASCADE,
    postId INT REFERENCES Post(id)  ON DELETE CASCADE,
    PRIMARY KEY (personId, postId)
```

);

Similar to how a **Person** interacts with a **Post** through commenting, they can also hit the thumbs up if they like it. To keep the records for that, we created a separate table which takes two id's of a **Person** and a **Post** and uses the combined two id's as a primary key. Thanks to that, we ensure that a **Person** can like a **Post** exactly once.

```
CREATE TABLE PrivateMessage (
    id SERIAL PRIMARY KEY,
    content TEXT,
    createDate TIMESTAMP NOT NULL,
    senderId INT NOT NULL,
    receiverId INT NOT NULL,
    FOREIGN KEY(senderId) REFERENCES BaseUser(id),
    FOREIGN KEY (receiverId) REFERENCES BaseUser(id)
);
```

To keep the records for private messages between **Persons**, we created a separate table for **PrivateMessage** referencing **BaseUser** twice, one as a sender and another one as a receiver. By putting the separate IDs for this table, we ensure that there can be multiple messages among the same tuple (Sender, Receiver).

```
CREATE TABLE Feedback (
    id SERIAL PRIMARY KEY,
    createDate TIMESTAMP NOT NULL,
    lastUpdate TIMESTAMP,
    email VARCHAR(255) NOT NULL,
    message TEXT NOT NULL
);
```

To collect user feedback about the application, a separate table called **Feedback** is designed. This table allows users to submit their feedback without requiring an account or login, ensuring that anyone can provide their input easily. By including

essential fields such as the user's email and message content, it is ensured that the feedback can be followed up if necessary.

Deployed Working Prototype

The deployed working prototype can be reached from the following link.

https://notebridge1.paas.hosted-by-previder.com/notebridge/

In this prototype, users can navigate through some essential pages. The users can see the posts on the main page. It allows users to create a new post from the “Create Post” page. The newly created post can be seen at the end of the main page.