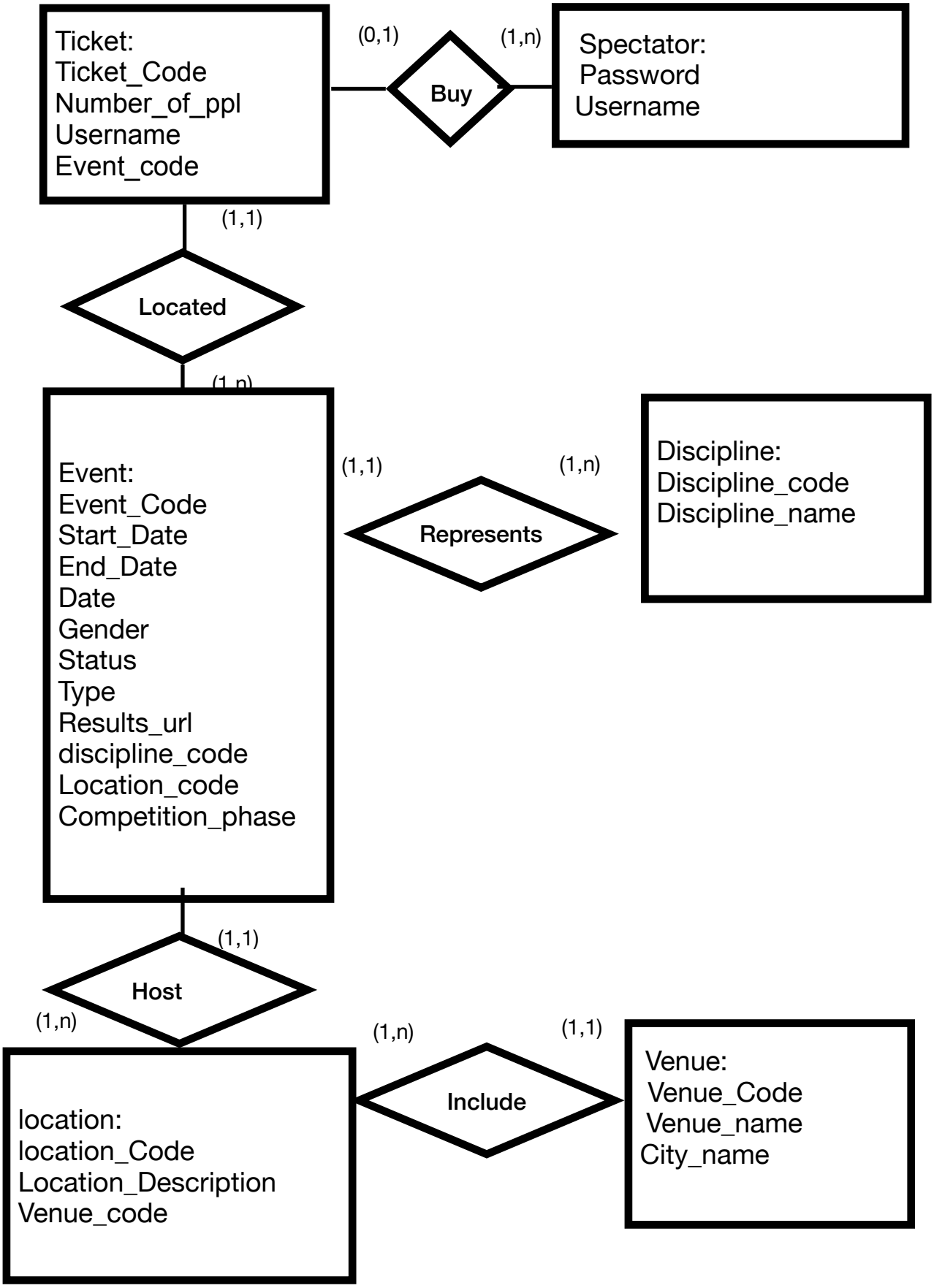
TP SIP

Armand COIFFE
Jennifer EIGHO

Q1.



Q2.

Relational Model (3NF)

1. **Spectator Table**
    - Username (PK)
    - Password
2. **Ticket Table**
    - Ticket_Code (PK)
    - Username (FK, references Spectator)
    - Number_of_People
3. **Venue Table**
    - Venue_Code (PK)
    - Venue_Name
    - City_Name
4. **Location Table**
    - Location_Code (PK)
    - Venue_Code (FK, references Venue)
    - Location_Description
5. **Event Table**
    - Event_Code (PK)
    - Start_Date
    - End_Date
    - Competition_Phase
    - Type
    - Gender
    - Status
    - Results_URL
    - Location_Code (FK, references Location)
    - Discipline_Code (FK, references Discipline)
6. **Discipline Table**
    - Discipline_Code (PK)
    - Discipline_Name

PK pour private key
FK pour foreign key

Q9.

On ne voit pas Hubert dans la liste car il manque la commande "conn.commit()" avant la fermeture de la database

Il faudrait mettre directement conn.commit() dans test_spectator.py.
Sinon, on peut le mettre dans spectator.py, sous condition d'ajouter à toutes les fonctions l'argument conn pour exécuter le conn.commit() dans la bonne classe.

Q10.

On obtient l'erreur suivante:
Inserting spectator Hubert...
An integrity error occurred while insert the spectator: UNIQUE constraint failed: Spectator.username
Impossible to add Hubert …

On obtient cette erreur car hubert est déjà présent dans la Database, il faut alors implémenter la fonction update_password()

Q12.

La fonction essaie de mettre à jour le mot de passe d'un spectateur qui n'est pas dans la base de données.
L'opération échoue car le spectateur n'existe pas.
Aucune donnée n'est modifiée, et le test renvoie que la mise à jour n'a pas réussi.

Q14.

L'application utilise Flask pour créer un serveur web. Elle enregistre des blueprints, qui définissent des routes spécifiques qui gèrent différentes parties de l'application (par exemple consulter les spectateurs). Enfin, elle initialise une base de données et s'exécute sur le port 5001.
Par exemple :
http://localhost:5001/spectators retourne une liste de spectateurs

Q15.

Le chemin de la route pour obtenir tous les spectateurs est: /spectators.
Le type de requête HTTP pour accéder à cette route est une requête GET

On peut donc accéder à la liste des spectateurs avec une requête GET sur l'URL:

http://localhost:5001/spectators.

Q16.

On utilise pour la premiere fois la librairie request, qui nous permet de mener de requête depuis notre fichier test.py pour toutes la suite du TP. Ce serait très utilise pour tester les méthodes patch et GET

Q17.
On complete le fichier et les fonctions dans ./routes/spectators.py: en utilisant si possible les fonctions de ./db/spectators.py:

Q18.

On initialise les blueprint puis on complète les requîtes HTTP de ./routes/data.py qui permettent d'avoir toutes les données de notre DB

Q19.
On fait pareil qu'à Q18

Q20.
Grâce à bcrypt on créer une fonction de HASHache pour protéger les mots de passes car une fois hacher on ne peut retourner en arrière

Q21.

On modifier insert_spectator() pour qu'on puisse hasher les mots de passes quand on ajoute un utilisateur à la base de donné. Ces deux questions ajoutent un niveau de sécurité supplémentaire à notre base de donnée

Q22.
La librairie PyJWT nous permet de généré des tockens d'utilisations pour certifier qu'on ai le droit de par exemple modifier son mot de passe.

Q23.
Il faut cependant être connecter pour avoir ce mot de passe, on développe Alors la fonction login, dont on lui associe un blueprint pour s'enregistrer. Pour créer ce token on se base sur la clé privé qui est stocké dans la configuration.

Q24. Finalement dans utils ou app_utils dans mon projet on a une fonction token required() qui est ce qu'on appelle un décorateur. Avant chaque blueprint qui nécessite une protection on met alors @token_required pour nécéssiter une connexion pour pouvoir modifier son mot de passe ou vendre sa place. Car on est pas censé modifier la place de quelqu'un d'autre ou vendre la place de quelqu'un d'autre sans son accord.