INFO0009: Bases de données

Seconde partie du projet

Année académique 2019-2020

Boris Courtoy - s175068

Maxence Raquet - s153466

Louis Kennis - s162264

Alexandre Baré - s172388

Question 1: Architecture web

Le site web est lancé à partir de la page *index.php* qui vous demande vos identifiants. Les bons identifiants se trouvent dans la table *'users'* de *'group40'* et dans le fichier *credentials.php*

Ceux-ci sont sauvegardés via la classe *Connection* qui un wrapper pour l'objet de classe *PDO* qui se trouve dans le fichier *Connection.php*. Ainsi, on peut effectuer l'opération *serialize* sur l'instance de *Connection*, ce qui n'est pas possible autrement sur une instance de *PDO*, pour sauvegarder celle-ci dans la session actuelle. Et à chaque *deserialize*, la connection à la base de donnée via l'objet de classe *PDO* est réeffectuée automatiquement.

Si les identifiants sont bons, on arrive à la page *main_menu.php* qui se charge de présenter les différentes options du menu et un bouton de déconnection ramenant à *index.php.*

Chaque option mène à une page "*.php"* qui se charge des opérations à effectuer et qui obtient l'accès à la base de données en effectuant le *deserialize* de l'objet de type Connection sauvegardé dans la session actuelle. Ces pages sont respectivement pour les question 2. a), b), c), d), e): *search.php, country.php, add_episode.php, black_mirror.php, episodes_popularity.php*. Un bouton permet à chaque fois de revenir à *menu_principal.php*.

On note un fichiers *dynamic.js* contenant une fonction pour la création de tables HTML dynamiques qui sont alors inclus dans les fichiers de la question 2.

Pour éviter d'avoir accès à des pages du site web sans avoir encodé ses identifiants, chaque page renvoie à la page *index.php* si aucune connection n'a été enregistrée dans la session actuelle. Ainsi, même en notant l'adresse url d'une page interne sans être connecté, il n'est pas possible d'y accéder. Notons qu'ici, avec la classe PDO, la base de données n'est pas non plus accessible sans les identifiants.

Question 2: Initialisation de la base de données

Pour initialiser la base de données, il faut effectuer les opérations du fichier *DBInitialisation.sql* instruction par instruction, dans l'ordre donnée, dans phpMyAdmin. Ces opérations consistent à:

- créer la table users et y entrer les identifiants du groupe. Cette table contiendra ainsi les seuls identifiants qui pourront être utilisés pour se connecter sur le site web.
- créer les autres tables de la base de données.
- ajouter les bonnes contraintes de clés étrangères (*FOREIGN KEY*) aux tables.

Ensuite, il reste à remplir une à une les tables sur phpMyAdmin, en important les fichiers *".csv"* correspondants. Dans l'onglet *Import* et pour chaque fichier, on mentionne alors qu'il faut passer la première ligne car elle contient les headers de la table et que les colonnes sont séparées par le caractère ';'. Notons que la terminaison des lignes est reconnu automatiquement par phpMyAdmin. Finalement, on a veillé à ce que l'ordre des colonnes des fichiers *".csv"* correspond à l'ordre des attributs de la table.

Question 3: Requêtes SQL

Voici les requêtes utilisées:

pour la 2.a):
```
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE'
AND TABLE_SCHEMA = 'group40'
AND TABLE_NAME != 'users'
```
permet de récupérer le nom des tables de la base de donnée *'group40'* excepté celui de '*users*'. Ces informations sont utilisées pour proposer à l'utilisateur un choix de tables à visualiser.

```
"SELECT * FROM " . $_POST['select_table']
```

permet de récupérer les tuples de la table sélectionnée par l'utilisateur pour qu'ils puissent les voir.

```
"SELECT *
 FROM " . $_SESSION['select_table'] . "
 LIMIT 1"
```

permet de récupérer le premier tuple de la table sélectionnée par l'utilisateur pour après récupérer les titres de chaque colonne pour l'affichage de la table.

```
"SELECT *
 FROM " . $_SESSION['select_table']
WHERE … AND …
```

avec un nombre variable de contraintes d'égalité (pour les nombres et dates) et/ou de contenance (pour les chaînes de caractères) pour sélectionner uniquement certains tuples de la table sélectionnée par l'utilisateur. Les contraintes d'égalités sont du type: *… = '…'* et celles de contenance du type: *… LIKE '%...%'*

pour la 2.b): *SELECT nom*
FROM plateforme_streaming

permet de récupérer les noms des différentes plateformes de streaming pour que l'utilisateur en sélectionne un.

```
SELECT pays
FROM pays
```

permet de récupérer tous les pays des différentes plateformes lorsque l'utilisateur n'a pas encore fait de sélection de plateforme sur la page.

```
"SELECT pays
FROM plateforme_streaming NATURAL JOIN pays
WHERE nom = '" . $_POST['select_platform'] . "'"
```

permet de récupérer les pays de la plateforme de streaming sélectionnée par l'utilisateur.

pour la 2.c): *SELECT nom FROM serie*

permet de récupérer le nom de toutes les séries afin que l'utilisateur puisse choisir d'ajouter son épisode à une d'entre elles.

```
SELECT prenom, nom, numero
FROM acteur NATURAL JOIN personne
```

permet de récupérer le prénom, nom et numéro de tous les acteurs pour que l'utilisateur puisse en sélectionner un ou plusieurs qui jouent dans l'épisodes à rajouter.

```
"SELECT n_saison, n_episode
FROM episodes
WHERE nom = '" . $_POST['select_series'] . "'
ORDER BY n_saison DESC, n_episode DESC
LIMIT 1"
```

permet de récupérer les numéros du dernier épisode et de la dernière saison de la série spécifiées par l'utilisateur pour après pouvoir ajouter un épisode au bout de la saison en cours ou ajouter un épisode au début d'une nouvelle saison selon les inputs de l'utilisateur.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

permet de modifier le niveau d'isolation de la transaction à *SERIALIZABLE*, ce qui permet de s'assurer de la cohérence des données même dans le cas d'insertions de nouvelles valeurs dans la table. Cette garantie se fait néanmoins au prix d'un risque de deadlock lorsque plusieurs transactions simultanées ont lieu. Si le problème se présente, il faudra alors recommencer les transactions bloquées. Notons que dans les autres questions, nous avons décidé de garder les niveaux de transactions par défaut et nous n'avons pas vérouillé des tables ou des tuples car nos requêtes n'impliquent que des lectures non sensibles.

```
INSERT INTO episodes (n_saison, n_episode, duree, synopsis, nom)
VALUES (?, ?, ?, ?, ?)
```

permet d'insérer un épisode en fonction des inputs de l'utilisateur. Les 5 points d'interrogations sont remplacés par *$episode['n_saison'], $episode['n_episode'], $_POST['duration'], $_POST['synopsis'], $_POST['select_series'].* 3 sont des inputs de l'utilisateur (*$_POST[...]*) et le numéro d'épisode est calculé pour qu'il s'ajoute au bout de la saison en cours si la checkbox relative à *$_POST['is_new_season']* n'a pas été cochée. Sinon, le numéro d'épisode est 1 et le numéro de saison est calculée pour qu'il s'ajoute au bout des numéros de saisons déjà existantes.

```
INSERT INTO joue_dans (numero, n_saison, n_episode, nom)
VALUES (?, ?, ?, ?)
```

permet d'ajouter l'information qu'un acteur, donnée par le numéro, joue dans l'épisode ajouté. Ceci se fait en insérant un tuple dans la table *joue_dans* en fonction des inputs de l'utilisateur. Les 4 points d'interrogations sont remplacés par *$selectedActor, $episode['n_saison'], $episode['n_episode'], $_POST['select_series']*. On a 1 input de l'utilisateur (*$_POST[...]*). Le numéro d'épisode est calculé pour qu'il s'ajoute au bout de la saison en cours si la checkbox

relative à *$_POST['is_new_season']* n'a pas été cochée. Sinon, le numéro d'épisode est 1 et le numéro de saison est calculée pour qu'il s'ajoute au bout des numéros de saisons déjà existantes. Et l'insertion est répétée pour chaque *$selectedActor* dans *$_POST['select_actors']* comme plusieurs acteurs peuvent être sélectionnés par l'utilisateur.

Notons qu'on ne validera les insertions avec *commit()*, uniquement si elles ont toutes étés effectuées sans soucis. Autrement, un *rollBack()* aura lieu.

pour la 2.d):

```
SELECT numero, nom_serie, n_saison, n_episode
FROM regarde
WHERE nom_serie = 'Black Mirror'
```

en premier lieu, nous allons définir 'Black mirror' comme étant le nom de la série. Ceci va nous permettre de sélectionner le numéro, nom de la série (Black Mirror), le numéro de la saison et le numéro de l'épisode de chaque tuple dont le nom de la série correspond.

```
SELECT nom, COUNT(DISTINCT(n_saison)) AS nb_saison,
COUNT(n_episode) AS nb_epis
FROM episodes
WHERE nom = 'Black Mirror'
```

lors de cette étape, nous comptons le nombre de saisons et le nombre d'épisodes que contient la série 'Black Mirror'. Ces deux informations sont stockées sous de nouvelles variable : nb_saison et nb_epis.

```
SELECT numero, COUNT(DISTINCT(n_saison)) AS saison_watched,
COUNT(n_episode) AS episode_watched, nb_saison, nb_epis
FROM(
   SELECT *
   FROM(
      SELECT numero, nom_serie, n_saison, n_episode
      FROM regarde
      WHERE nom_serie = 'Black Mirror'
   ) AS t1
   NATURAL JOIN(
      SELECT nom, COUNT(DISTINCT(n_saison)) AS nb_saison,
COUNT(n_episode) AS nb_epis
      FROM episodes
      WHERE nom = 'Black Mirror'
   ) AS t2
```

nous effectuons ici une jointure naturelle entre deux sélections précédentes. Cette jointure s'effectue à la condition qu'il y ai des colonnes du même nom et de même type dans les 2 tables. Le résultat d'une jointure naturelle est la création d'un tableau avec autant de lignes qu'il y a de paires correspondant à l'association des colonnes de même nom.
De cette jointure, nous conservons le numéro, le nombre de saisons (différentes) regardées, le nombre d'épisodes regardés, le nombre de saisons et d'épisodes de la série.

```
SELECT numero
FROM (
   SELECT numero, COUNT(DISTINCT(n_saison)) AS saison_watched,
COUNT(n_episode) AS episode_watched, nb_saison, nb_epis
   FROM(
      SELECT *
      FROM(
         SELECT numero, nom_serie, n_saison, n_episode
         FROM regarde
         WHERE nom_serie = 'Black Mirror'
      ) AS t1
      NATURAL JOIN(
         SELECT nom, COUNT(DISTINCT(n_saison)) AS nb_saison,
COUNT(n_episode) AS nb_epis
         FROM episodes
         WHERE nom = 'Black Mirror'
      ) AS t2
   ) AS t3
GROUP BY numero
HAVING saison_watched = nb_saison AND episode_watched = nb_epis
```

ensuite, les résultats obtenus vont être groupés par numéros, ce qui nous permet d'avoir les informations relative à une personne dans un même tuple et de ce fait, d'identifier si cette personne a regardé tous les épisodes de 'Black Mirror'. Seuls le numéro des personnes ayant regardé l'intégralité de la série sont conservés.

```
SELECT nom, prenom
FROM personne
WHERE numero IN(
        SELECT numero
   FROM (
      SELECT numero, COUNT(DISTINCT(n_saison)) AS saison_watched,
COUNT(n_episode) AS episode_watched, nb_saison, nb_epis
      FROM(
```

```
            SELECT *
            FROM(
               SELECT numero, nom_serie, n_saison, n_episode
               FROM regarde
               WHERE nom_serie = 'Black Mirror'
            ) AS t1
            NATURAL JOIN(
               SELECT nom, COUNT(DISTINCT(n_saison)) AS nb_saison,
        COUNT(n_episode) AS nb_epis
               FROM episodes
               WHERE nom = 'Black Mirror'
            ) AS t2
         ) AS t3
      GROUP BY numero
      HAVING saison_watched = nb_saison AND episode_watched = nb_epis
      ) AS t4
   )
```

permet de récupérer les noms et prénoms dans la table “personne”. Seuls les noms et prénoms des tuples dont le numéro sera présent dans les résultats de la requête précédente seront affichés.

pour la 2.e):

```
SELECT nom_serie, n_saison, n_episode,
                  COUNT(DISTINCT(numero)) as nb_watchers
FROM regarde
GROUP BY nom_serie, n_saison, n_episode
ORDER BY nb_watchers DESC, nom_serie ASC, n_saison ASC,
                  n_episode ASC
```

Cette requête SQL groupe la table ‘regarde’ grâce aux attributs nom_serie, n_saison et n_episode, en d’autres termes groupe la table par épisode distinct d’une saison d’une série. Ceci crée par la même occasion une colonne nb_watchers qui correspond au nombre d’utilisateurs différents qui ont regardé cet épisode. La fonction ORDER BY sert à trier les résultats selon 4 critères différents. Le premier est le nombre de viewers (nb_watchers) avec un tri décroissant. Le second est le nom de la série (nom_serie) avec un tri croissant. Le suivant est le numéro de la saison (n_saison) avec un tri croissant. Et enfin, le dernier est le numéro de l’épisode (n_episode) avec un tri croissant.
Pour finir, nous ne sélectionnons dans le résultat que les colonnes nom_serie, n_saison, n_episode et nb_watchers car ceux-ci sont les seuls attributs nécessaires à notre requête.

Nous pouvons voir avec cette requête que, dans notre base de donnée, l'épisode le plus vu est l'épisode 2 de la saison 2 de Black Mirror.
Cette requête a été trouvée en écrivant l'énoncé en algèbre relationnelle comme ci-dessous :

$$\pi_{\text{nom\_serie, n\_saison, n\_episode, nb\_watchers}} (\tau_{\text{nv\_watchers, nom\_serie, n\_saison, n\_episode}} (\gamma_{\text{nom\_serie, n\_saison, n\_episode}}), \text{COUNT(numero)} \rightarrow \text{nb\_watchers(regarde)})$$

Question 4: URL du site Web

http://www.student.montefiore.ulg.ac.be/~s172388/