**TP2 - Partie 2 (25%)**

(Notions des semaines 4 à 13)



**Consignes :**

* Ce projet est la suite du **TP2 partie 1**. Continuez de travailler sur le même projet. Si vous n’avez pas complété toutes les fonctionnalités de la **partie 1**, n’hésitez pas à vous présenter en classe pour obtenir une version de départ.
* Le **TP2** est à faire de manière individuelle. Attention au plagiat.
* Il faut encore utiliser Git. Si vous souhaitez changer de repository pour la partie 2, envoyez une nouvelle invitation à votre enseignant(e) avant le 10 mai. ([chantal.vallieres@cegepmontpetit.ca](mailto:chantal.vallieres@cegepmontpetit.ca) ou [maxime.pelletier@cegepmontpetit.ca](mailto:maxime.pelletier@cegepmontpetit.ca)) Nous utiliserons tout de même la remise sur Léa, mais nous inspecterons votre répertoire Git.

**Mise en situation :**

* Des fonctionnalités supplémentaires devrons être ajoutées dans l’application SussyKart.
* De nouvelles données de test seront ajoutées pour augmenter la charge de la base de données.
* Quelques tests seront créés.
* Encore une fois, vous remarquerez qu’il n’y a quasiment rien à faire dans les **Views** et les actions à coder sont déjà préparées. C’est parce que vous êtes l’expert BD du projet et un collègue en a déjà fait le plus possible côté Web pour vous orienter 🙄
* Précision : Oui, cet énoncé est beaucoup plus long que pour la **partie 1** et peut paraître intimidant, mais ce n’est pas parce que le travail est plus long, c’est parce que vous serez très guidés pour les modifications à faire, donc voyez-le du bon côté.

**1 - Migrations versionnées avec Evolve 📈**

Tous les changements que vous ferez à la BD devront faire partie de migrations standardisées exécutables avec Evolve. Vous n’êtes pas censés modifier les migrations existantes. (Vous pouvez seulement modifier vos migrations si elles contenaient des erreurs. Vous ne devez pas modifier vos migrations pour travailler sur de nouvelles fonctionnalités) Assurez-vous que si on ouvre votre projet avec une BD vide, toutes les migrations fonctionnent à l’aide de la commande Evolve.

**TP2 partie 1** : Les versions des migrations seront **2.0**, **2.1**, etc.

**TP2 partie 2** : Les versions des migrations seront **3.0**, **3.1**, etc.

**2 - Models et DbContext générés par scaffolding 🏗**

Tous vos **Models** et votre **DbContext** devront être générés grâce à la commande **dotnet ef scaffold** que nous avons abordée. Vous ne pouvez pas modifier les **Models** et le **DbContext** qui ont été générés par cette commande. Lors de la correction, le **DbContext** et les **Models** doivent déjà être présents dans le projet. (La seule chose à « ajouter » par le correcteur sera l’exécution des migrations)

**3 - Modifications simples** 🤏

Il y a quelques modifications à faire dans le projet pour préparer les prochaines étapes. C’est essentiellement du copié-collé, ça ira vite. Dans tous les fichiers avec l’extension .txt, il y a des instructions supplémentaires pour vous aider à bien intégrer le code au projet. Soyez très attentifs !

* Remplacez les migrations **1.1**, **1.2** et **1.3** par celles fournies avec cet énoncé. (Cela inclut toutes celles avec les versions **1\_3\_X**)
  + Elles contiennent simplement plus de données.(Sauf la 1.1, qui corrige les noms de courses à cause d’un bug dans la **partie 1**)
* Ajoutez les deux ViewModels fournis à votre projet. Attention ! Ce sont juste les **class**, sans le bloc **namespace**. Pour être sûrs que votre **namespace** sera bon, vous pouvez créer un ViewModel manuellement avec le nom correspondant avant de copier-coller.
* Ajoutez les vues Razor **Amis.cshtml** et **Avatar.cshtml** dans le dossier Views/Utilisateurs. (Avec Profil, Connexion et Inscription qui sont déjà là)
  + Elles ne fonctionneront pas bien si vous n’avez pas encore ajouté les ViewModels au-dessus.
* Il y a du CSS à ajouter dans **wwwroot/css/styles.css**
* Il y a des ajouts et modifications à faire dans la vue **Views/Utilisateurs/Profil.cshtml**
* Les squelettes de 6 nouvelles actions à ajouter dans **UtilisateursController** sont fournis et contiennent quelques directions gracieusement offertes par votre collègue qui est développeur Web.

**4 - Avatar** 🌪

Les utilisateurs doivent pouvoir *upload* une image pour en faire leur avatar (image de profil) sur l’application Web. Cette image sera affichée dans le profil de l’utilisateur.

Modifications dans la BD :

* Configuration d’un Filegroup à l’emplacement C:\EspaceLabo. (Rappel : ça ne peut pas se faire dans une migration et c’est à répéter à chaque fois que vous changez de poste de travail ou recréez la BD. De toute façon, vous devez glisser ce bloc dans votre InitialCreate.sql pour la correction)
* Ajout d’une nouvelle table qui permet de stocker les avatars des utilisateurs dans le système de fichiers. Vous devez utiliser le FILESTREAM. Un avatar par utilisateur maximum !

Modifications dans l’application Web :

* Dans l’action Utilisateurs/Profil, si l’utilisateur possède un avatar, l’envoyer à la vue via le ProfilVM.
* Complétez l’action Utilisateurs/Avatar.
* Complétez l’action Utilisateurs/Avatar. (POST)

**5 - Le pouvoir de l’amitié** 👥🤝

On doit ajouter une fonctionnalité « Amitié » dans l’application. Les amitiés sont à sens unique : on peut ajouter qui que ce soit dans notre liste d’amis, sans forcément que ce soit réciproque. Ajouter un autre utilisateur dans notre liste d’amis ... :

* Ne nous ajoute pas dans sa liste d’amis.
* Ne lui envoie pas de notification ou de « demande d’ami » quelconque.

Dans le profil, nous pourrons inspecter notre propre liste d’amis (si on est connecté) et voir certaines statistiques sur chaque ami.

Modifications dans la BD :

* Ajoutez une table dans la BD qui représente une amitié à sens unique.
* Votre table aura, au minimum, deux colonnes qui sont des FK associées à UtilisateurID. (Une colonne représente l’utilisateur qui a ajouté l’ami dans sa liste et l’autre colonne représente l’utilisateur qui a été ajouté par le premier)
* Comme ces deux FK font référence à la même PK et génèrent un cycle, les deux contraintes FK devront être créées avec délicatesse.
* Une contrainte doit empêcher d’ajouter le même ami deux fois dans sa liste d’amis.
* Créez un déclencheur qui est activé lorsqu’on tente de supprimer un (ou des) utilisateur(s). Ce déclencheur permet de ... :
  + Au lieu de vraiment supprimer les utilisateurs, faire un *soft delete* grâce à une *colonne stratégique* qui existe déjà dans la table Utilisateur.
  + Supprimer toutes amitiés qui concernent les utilisateurs « supprimés ».
  + Supprimer l’avatar de l’utilisateur s’il en avait un.
  + Gérer les suppressions simples et les suppressions multiples d’utilisateurs. (Au cas où une action permettrait de supprimer plusieurs comptes)
  + Notez que les participations de l'utilisateur sont bel et bien conservées.
  + Notez qu'on ne doit pas être capable d'ajouter un utilisateur supprimé dans sa liste d'amis.
  + Il se peut que vous ayez à supprimer un déclencheur existant...
* Finalement, la procédure stockée d’authentification doit être légèrement modifiée : un utilisateur « supprimé » ne doit pas pouvoir se connecter. (Il ne devrait pas y avoir de changements à faire dans l’application Web, on veut gérer cela avec la procédure)

Modifications dans le projet Web :

* Dans l’action **Utilisateurs/Profil**
  + Calculer le nombre d’amis et l’afficher dans la vue Profil.cshtml. (Que ce soit en passant l’information par le ProfilVM ou avec ViewData)
* Compléter l’action **Utilisateurs/Amis**
  + Notez bien qu’on doit entre autre afficher les avatars de nos amis dans la vue Razor Amis.cshtml.
* Compléter l’action **Utilisateurs/AjouterAmi**
  + Remarquez que nous ne récupérerons pas l’utilisateur via son cookie, exceptionnellement. Le paramètre **utilisateurID** correspond à l’utilisateur qui fait l’ajout et le paramètre **pseudoAmi** correspond au pseudo de l’ami à ajouter.
* Compléter l’action **Utilisateurs/SupprimerAmi**
  + Remarquez que nous ne récupérerons pas l’utilisateur via son cookie, exceptionnellement. Le paramètre **utilisateurID** correspond à l’utilisateur qui fait l’ajout et le paramètre **amiID** correspond au pseudo de l’ami à retirer.
* Compléter l’action **Utilisateurs/DesactiverCompte**
  + Encore une fois, on ne récupère pas l’utilisateur via son cookie, mais en utilisant l’ID reçu en paramètre.
  + Ne pas mettre la ligne de code await HttpContext.SignOutAsync() même si ce serait judicieux.
  + Supprimez tout simplement l’utilisateur du Context et laissez votre déclencheur faire le travail.

**6 - Optimisation**

Pour s’initier à l’optimisation, nous allons faire deux modifications dans notre projet.

Modification dans la BD :

* Ajoutez trois index non clustered de votre choix dans une migration. Pour chaque index, justifiez, avec un court commentaire (directement au sein de la migration), pourquoi vous avez choisi la / les colonne(s).

Modification dans le projet Web :

* Quand on fait **.ToListAsync()** sur un **DbSet**, Entity Framework doit charger toutes les données dans la mémoire de l’application Web. Avec \_context.ParticipationCourse.ToListAsync(), par exemple, il faut charger les 300 000 rangées de la table dans l’application. (Ce qui est très mauvais)
  + Seulement pour l’action **Stats/ToutesLesParticipations**, retardez le plus possible l’usage de ToListAsync(), après avoir filtré les données non pertinentes et utilisez le type IQueryable<T> puisque le filtrage se fait avec plusieurs instructions. (Voir diapo 30 du cours de la semaine 9 : Intro aux projets Web BD-First)
  + Grâce à ce mode opératoire, Entity Framework chargera seulement les quelques données vraiment nécessaires dans l’application et la mémoire jonglera moins avec des données impertinentes.

**7 - Tests**

Pour s’initier aux **tests d’intégration**, (Pas de tests unitaires car vous avez été évalués sur ça dans un autre cours) nous allons faire trois tests (qui touchent au total deux contrôleurs) dans un projet **xUnit**. (Dans la même solution que le projet principal) Pour simplifier les tests, vous remarquerez que :

* L’action **Utilisateurs/DesactiverCompte** n’est pas sécuritaire. (Car elle reçoit l’id de l’utilisateur à supprimer en paramètre au lieu de le récupérer dans le cookie ET le context HTTP n’est pas vidé immédiatement, ce qui fait que l’utilisateur reste capable d’utiliser son compte tant que son cookie est valide.)
* L’action **Utilisateurs/AjouterAmi** n’est pas sécuritaire. (Car elle reçoit l’id de l’utilisateur qui ajoute un ami en paramètre au lieu de le récupérer dans le cookie)
* Test 1 ⚙ : Tester l’action **Stats/ToutesParticipationsFiltre**
  + Lorsqu’on lui transmet un **FiltreParticipationVM** qui permet de ne garder que les participations de « **sussyprincessZ** » dans la course « **Donut** », est-ce que l’action retourne bel et bien une **Vue** avec un Model de type **FiltreParticipationVM** dont la liste de participations contient **3** éléments ?
* Test 2 ⚙ : Tester l’action **Utilisateurs/AjouterAmi**
  + Lorsqu’on appelle cette action avec des paramètres valides, (1 et "Chantal", donc l’utilisateur Maxime ajoute Chantal en amie) sommes-nous bien redirigés vers l’action « Amis » ?
  + De plus, dans le même test, appelez une deuxième fois l’action avec les mêmes paramètres. (1 et "Chantal") Est-ce que la table **Amitie** contient bel et bien une seule rangée après avoir tenté de créer la même amitié deux fois ?
* Test 3 ⚙ : Tester l’action **Utilisateurs/DesactiverCompte**
  + Lorsqu’on appelle cette action avec le paramètre 1 (Donc désactiver le compte de « Maxime »), est-ce que cet utilisateur est encore bel et bien dans la BD, mais avec la valeur *true* pour la colonne EstSuppr ?
    - Ne faites pas de vérifications en lien avec les tables Amitie et Avatar, car Entity Framework ne peut pas « tracker » les opérations que votre déclencheur fait en cachette et cela cause des problèmes lors du test.

**Grille de correction**

|  |  |  |
| --- | --- | --- |
| Section | Critères | Points |
| Migrations | - Toutes les migrations sont fonctionnelles avec la commande Evolve.  - Les migrations sont bien versionnées et segmentées. | 3  1 |
| Models et DbContext | - Les Models et le DbContext sont présents et sont compatibles avec la base de données. | 1 |
| Avatar | - Configuration d’un Filegroup  - Création d’une table pour les avatars.  - Afficher l’avatar de l’utilisateur dans son profil.  - Permettre à l’utilisateur d’ajouter ou modifier un avatar. | 1  4  2  3 |
| Amitié | - Création d’une table et de ses contraintes  - Création d’un déclencheur  - Modifier d’une procédure stockée  - Modifier l’action Profil pour afficher le nombre d’amis  - Afficher les infos nécessaires des amis  - Ajouter un ami s’il n’est pas déjà dans notre liste  - Supprimer un ami  - Désactivation du compte | 4  4  1  1  3  2  2  1 |
| Optimisation | - Trois index *non clustered* créés  - Justifications et pertinence des index  - Optimisation de la requête pour filtrer les participations | 3  3  1 |
| Tests | - Test pour Stats/ToutesParticipations  - Test pour Utilisateurs/AjouterAmi  - Test pour Utilisateurs/DesactiverCompte | 3  4  3 |
| Git | - Git n’a pas été utilisé. (-5)  - L’enseignant n’a pas été ajouté en tant que collaborateur à temps. (-1)  - La quantité de commits est trop petite ou n’est pas stratégique. (-1) | -5 à 0 |
| Total | | 50 |