Nunes Evan
Rostaing Damien
Haye Marc
Bodiguel Yann
Benhalima Mattéo

Q2



# SAE Very Bad Trip

# Rapport d'Analyse

# Qualité de Développement

Le fait de ne pas respecter la qualité de développement dans le code importe des problèmes notamment dans le fait que lorsque que l'on veut écrire du nouveau code, il va être nécessaire de réécrire du code déjà existant et donc de casser des conventions comme SOLID, de plus le fait que le code n'est pas d'interface aggrave fortement le manque du principe DRY.

- La méthode unlog() est incomplète dans EvenementRepository

- On ne peut pas supprimer un événement, ce qui ne respecte pas le principe CRUD

Violations du DRY :
Duplication de code dans les repositories pour la construction des objets
Répétition des requêtes SQL similaires

La fonction tel de ConnexionUtilisateur est appelée par la fonction MettreAJour du controleurUtilisateur mais ne l'utilise absolument pas après son appel.

Requêtes SQL Non Optimisées
Dans EvenementRepository.php :
"SELECT * FROM app_db WHERE $critere = '$valeur'"

Utilisation de SELECT * au lieu de sélectionner uniquement les colonnes nécessaires

- **Single Responsibility :**
  - La classe DepenseRepository gère à la fois les dépenses, les événements et les utilisateurs
  - La table app_db stocke toutes les données, mélangeant les responsabilités
  - La fonction recupererPar de EvenementRepository gère trop de choses.
- **Open/Closed :**
  - Impossible d'étendre les fonctionnalités sans modifier le code existant
- **Liskov Substitution :**
  - Pas d'utilisation d'interfaces, donc pas de substitution possible
- **Interface Segregation :**
  - Absence d'interfaces, donc pas de ségrégation
- **Dependency Inversion :**
  - Dépendance directe à la base de données
  - Pas d'injection de dépendances

Changements proposés :

- ajout d'interfaces et de classe abstraite
- des fonctions plus répartit notamment recupererPar qui fait trop de chose
- architecture MVC à revoir

# Failles de Sécurité

Il est possible d'exécuter du JavaScript, car un certain nombre d'informations saisies par l'utilisateur ne sont pas contrôlées. Il est possible ainsi, en nommant une dépense "<script>alert('boom')</script>" de lancer le morceau de JavaScript associé au nom sur un certain nombre de pages comme : le formulaire de modifications de dépense, la page d'accueil de l'événement associée.

Il est également possible de réaliser la même chose avec un événement sur les pages suivantes : la page d'accueil d'un événement, sur la page contenant tous les événements liés à un utilisateur, la page de modification de l'événement.

Le nom d'un utilisateur peut aussi être utilisé de cette manière sur la page d'ajout d'un utilisateur à un événement si l'utilisateur n'en fait pas partie. Pour résoudre cette manière, il faut penser à échapper les attributs sur ces pages de manière à ce que les noms et titres ne soient plus considérés comme du code, mais bien comme une chaîne de caractère.

Dans la classe ControleurEvenement, l'attribut login est non défini, ce qui pourrait causer une erreur critique et cesser le fonctionnement de l'application.

La méthode afficherEvenement() de ControleurEvenement effectue une division sur le nombre de participants. Dans le scénario où il n'y a aucun participant, cette méthode tente d'effectuer une division par zéro. Cela causera également une erreur critique.

Le code secret est visible dans le query string de l'URL. Cela ne devrait pas être visible en clair. Une correction possible serait d'utiliser le verbe HTTP POST.

<u>Pour la partie intermédiaire avec la base de données :</u>

L'attribut email dans recupererParEmail() de UtilisateurRepository n'est pas préparé. Un utilisateur malveillant peut donc saisir du code interprété au moment de saisir l'email et réaliser une injection SQL, par exemple "DELETE DATABASE ;//", ce qui supprimerait la base de données.

Cette attaque pourrait par exemple mettre en danger la base de données par suppression ou vol de données sensibles, comme par exemple les mots de passe. C'est également le cas dans la méthode ajouter() et recupererDepensesPayeesOuParticipeUtilisateur() de DepenseRepository pour l'attribut loginPayer.

Plusieurs paramètres de la classe EvenementRepository ne sont pas typés, notamment des attributs saisis par l'utilisateur.

Dans la méthode recupererPar() de EvenementRepository, il n'y a pas de vérification que $critère est un attribut valide. La méthode recupererEvenementsUtilisateur() de EvenementRepository appelle des colonnes qu'il utilise pas (nom prenom).

Également, des données sensibles de l'utilisateur sont stockées dans les cookies de l'utilisateur, cela pose des problèmes de sécurité car cela veut dire que coupler aux failles d'exécution de script déjà présente, n'importe qui peut récupérer les identifiants de l'utilisateur . Un exemple de d'utilisation de cette faille: Un utilisateur malveillant s'inscrit avec un prénom qui est en réalité un script qui récupère les cookies de l'utilisateur et les envoies sur un serveur lui appartenant, à partir de ce moment là, tout utilisateur qui charge la page d'ajout de membre à un événement donnera ses identifiant à cet utilisateur malveillant. Une solution consisterait à sauvegarder ses informations du côté serveur avec des sessions, de cette manière les données sont bien plus sécurisées et surtout non modifiables par l'utilisateur.

# Base de données

Dans la version originale, toute la base de données était contenue dans une seule table, contenant 20 attributs. Cela posait problèmes pour différentes raisons :

Les clés primaires ne remplissent pas réellement leur rôle, qui est d'assurer que chaque objet ait un identifiant unique. En effet, ce schéma admet par exemple deux utilisateurs différents ayant le même login, tant que l'identifiant d'évènement ou l'identifiant de dépense stocké avec est différent. Cela ne serait pas un problème si par exemple, un utilisateur était défini par un id d'événement, un login, et un id de dépense, mais ce n'est pas le cas ici. Additionnellement, la structure fait que (et c'est d'ailleurs ce qui est fait) si l'on souhaite par exemple ajouter un nouvel utilisateur, il faudra en plus créer un Evenement et une Depense qui ne sont pas utilisés, rien que pour satisfaire les contraintes de clé primaire.

L'application devient très complexe pour pallier à la simplicité excessive de la base de données. Cependant cet échange n'est pas équivalent, car une légère complexification de la base de donnée peut éviter beaucoup plus de complexité côté PHP qu'elle n'en crée.

Avec une seule table, l'utilisation de références (référer à un objet par sa clé primaire au lieu de stocker une copie de cet objet) devient assez difficile, et il n'est pas rare dans l'application, que des informations qui devraient être trouvables uniquement à partir d'un identifiant, soient stockées. Cela ne respecte donc pas le principe DRY.

L'utilisation de JSON pour stocker les associations multiples (par exemple entre les participants et une dépense) aurait été une solution assez créative s'il était techniquement impossible de créer plusieurs tables, mais ce n'est pas le cas ici. Cela contribue à la complexification de l'application côté PHP, beaucoup plus de travail doit être fait car il faut parser et traiter les données à chaque ajout ou modification. Du côté des performances, c'est aussi une mauvaise option, en raison de l'impossibilité d'indexation des valeurs contenues dans le JSON, et du fait qu'il faille récupérer, traiter, et renvoyer toute la liste JSON, à chaque ajout ou suppression (par exemple d'un membre d'un événement).

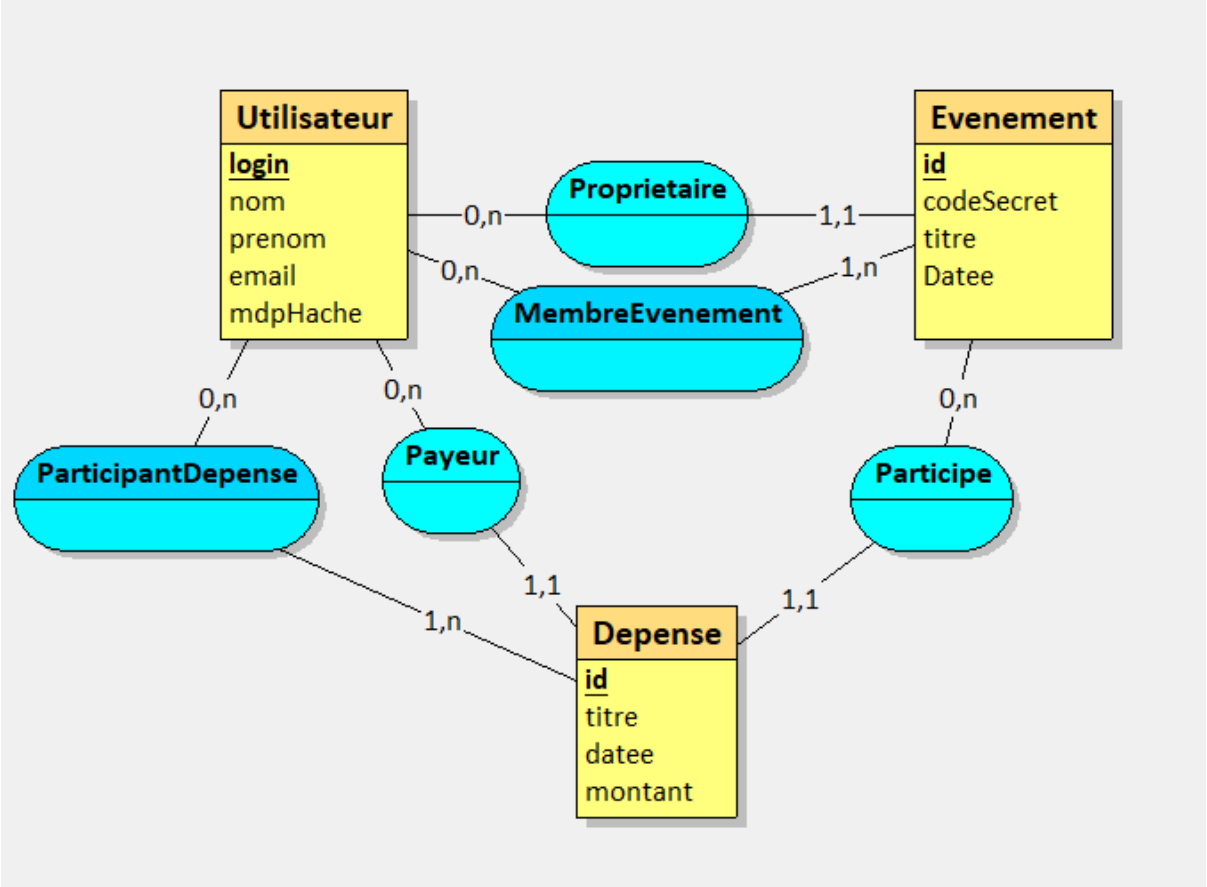
L'absence de Triggers constitue également une charge supplémentaire sur le code PHP, puisque beaucoup de méthodes interagissant avec la base de donnée doivent effectuer elles mêmes certaines choses qui pourraient être automatisés (Les triggers étant plus efficace en terme de quantité de code à écrire, en plus que ce qu'ils font est plus clair). Par exemple : la suppression d'un utilisateur engendrant la suppression des dépenses et des Evenements dont il est propriétaire, pourrait être un trigger, au lieu d'être dans la méthode de suppression d'un utilisateur sous la forme de code PHP.

D'un point de vue général, n'avoir qu'une seule table diminue la visibilité et la clarté de l'application et de la base de données. Comprendre les relations entre les différents éléments de la base de données devient impossible sans une connaissance approfondie du code PHP (par exemple, en observant uniquement la base de données, il est impossible de déterminer la relation entre une Depense, un Evenement, et un Utilisateur). Cela rend aussi le code PHP moins lisible, car il est responsable de beaucoup plus que ce qu'il devrait, rendant la compréhension plus difficile.

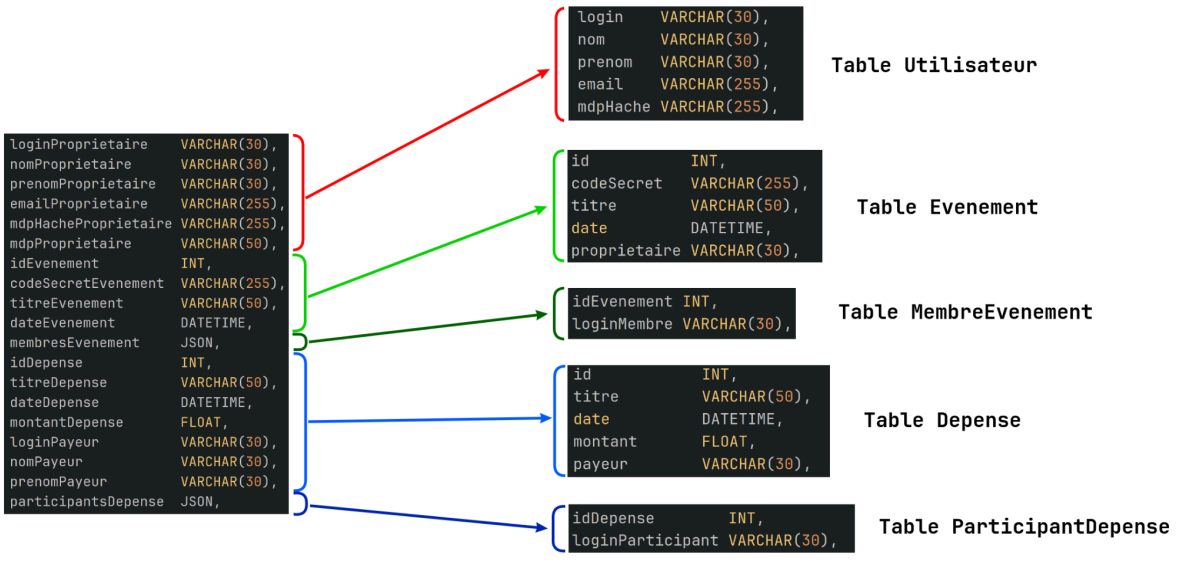
Nous avons noté quelques attributs dont le stockage n'est pas nécessaire. Premièrement, le mot de passe en clair d'un utilisateur, ce qui va à l'encontre du rgpd, qui stipule que les données sensibles telles que les mots de passe doivent être chiffrées. Deuxièmement, les attributs "nomPayeur" et "prenomPayeur", qui sont stockés en plus de "loginPayeur", alors que ces informations peuvent être retrouvées à partir du login uniquement.

Apporter des changements quelconques à la structure de la base de données devient aussi un challenge, car en l'état, elle n'est simplement pas faite pour être flexible, et d'importants efforts de modification côté PHP risquent d'être nécessaires pour des changements minimes.

Puisqu'il n'était donc pas envisageable de garder l'ancienne base de données, nous en avons modélisé, en essayant au mieux de corriger les problèmes évoqués ci-dessus.




La séparation des tables s'est faite assez facilement, car toutes les données étaient présentes, et n'avaient réellement besoin que d'une nouvelle organisation :




La séparation des attributs dans différentes table a été réalisé dans l'optique d'éviter au maximum la redondance (par exemple, stocker l'id et le nom d'un utilisateur, au lieu de seulement l'id), tout en améliorant la lisibilité :

- Séparation des Utilisateurs, des Événements, et des Dépenses dans une table différente pour chacun.
- Suppression de redondance :
    - mdpProprietaire supprimé (le mdp haché est déjà stocké),
    - nomPayeur et prenomPayeur supprimés (accessibles grâce au login utilisateur)
- Associations multiples mises sous la forme de table (précédemment sous Json), afin d'améliorer la lisibilité, et de centraliser la responsabilité sur la base de donnée, qui est plus optimisée pour ce genre de tâche :
    - suppression de l'attribut membreEvenements (de type Json), et création de la Table MembreEvenement,
    - suppression de l'attribut participantDepense (de type Json), et création de la Table Participant Dépense