SAE Semestre 4

Valette, Coulon, Moreira, Ferhani , Meliand

SOMMAIRE

Qualité de code et architecture

Performance :

Dans plusieurs fichiers php, des imports d'autres fichiers sont effectués au début, mais ne sont pas utilisés. Cela peut mener à des ralentissements de l'application si les fichiers à charger sont trop imposants. Pour régler ce problème, il suffit de supprimer ces imports.

Dans ControleurEvenement.php, il y a
use App\VeryBadSplit\Modèle\HTTP\Cookie

Dans ConnexionUtilisateur.php, il y a
use App\VeryBadSplit\Modele\DataObject\Utilisateur
use App\VeryBadSplit\Modele\Repository\UtilisateurRepository

Dans Evenement.php, il y a
use App\VeryBadSplit\Modele\Repository\DepenseRepository

Dans le projet, le développeur a eu recours plusieurs fois à la méthode "array_filter". Cette méthode a pour but de trier des tableaux en supprimant des éléments qui ne correspondent pas à certaines conditions. Le problème est que cette méthode est utilisée énormément de fois dans le filtrage des différents utilisateurs, ce qui peut charger la mémoire, car l'on récupère des données dont on n'a pas besoin, ce qui consomme plus de mémoire. Cela aussi peut provoquer un ralentissement important puisque le filtrage s'effectue après la récupération de donnée.

Exemples :

```
$filtredUtilisateurs = array_filter($utilisateurs, function ($u)
use ($evenement) {return
!$evenement->estMembre($u->getLogin());});
```

Par exemple, dans cette ligne, le développeur récupère tous les utilisateurs existants ($utilisateurs) et retourne tous les utilisateurs qui ne sont pas membres de l'événement ($evenement). Cela veut dire que pour chaque utilisateur existant, la méthode vérifie s'il fait partie de l'événement.

Solution : Faire une méthode qui effectue le filtrage, mais au niveau de la base de donnée en sql.

Qualitée :

Non-respect de DRY (Don’t repeat yourself)

Dans les classes “UtilisateurRepository”, EvenementRepository ainsi que DepenseRepository, l’instanciation des objets (depenses, utilisateurs et évenements) est répétée plusieurs fois, ce qui réduit la lisibilité dans le code, augmente le risque d’erreurs dans le code et rend difficile la maintenance de celui-ci.

Solution : Nous avons créé une classe AbstratDataObjet abstraite, qui contient la méthode creerObjet(), et qui est donc étendue par les différentes classes qui créent des objets (Dépense, Evenement et Utilisateur). Cela nous permettra d’améliorer la lisibilité et d’éviter la répétition, mais cela favorise aussi la flexibilité et le polymorphisme. Cela nous permet aussi de centraliser la gestion des propriétés des différents objets, forçant les classes à respecter une certaine structure.

- Dans les contrôleurs, une clause de garde qui permet de vérifier si l’utilisateur est connectée, suivi d’une redirection, se trouve dans presque chaque méthode dans cette classe.

Solution : Nous avons créé une méthode “exigerConnexion(controleur, action, vérification)” afin d’éviter la répétition de cette structure conditionnelle (if).

Problèmes de conceptions :
Il existe 3 fonctions dans notre application qui ne sont pas remplies.
Cela pose un problème de compréhension du code.

Dans ConnexionUtilisateur.php :
La fonction “tel” implique la possibilité de récupérer le numéro de téléphone d’un utilisateur, mais ce n’est actuellement pas prévu ni dans l’application, lors de l’inscription ou de la modification, ou alors dans la Base de Donnée.

```
public static function tel() : int
{
        //TO-DO
        return 5;
}
```

De plus elle n’est utilisée qu’une seule fois dans tout le projet, pour être stockée dans une variable inutile :

```
public static function mettreAJour(): void
{
        $x = ConnexionUtilisateur::tel();
```

```
        …
}
```

Dans MotDePasse.php :
La fonction “important” n’a pas de spécification, ce qui rend impossible la création d’une fonctionnalité peut être majeure. Cette méthode ne respecte pas les conventions de nommages, en ayant des paramètres “x” et “y”. Il est impossible de savoir pourquoi cette méthode a été créée.

```
public static function important($x,$y)
{
        //Je crois que ça ne marche pas hahahaha
        //Je vais simplement retirer le code pour le moment
}
```

Dans EvenementRepository.php
Même remarque pour la fonction ‘unlog’, il est compliqué de savoir ce que les développeurs ont voulu faire avec cette fonction, et il est laissé un “todo” sans aucune spécification. Il est donc compliqué de savoir vers où se diriger

```
public function unlog()
{
        if(ConnexionUtilisateur::estConnecte()) {
                echo “To-DO unlog”;
        }
}
```

Afin de régler ces problèmes de conceptions, nous devons soit supprimer entièrement ces fonctions, soit demander des précisions sur la volonté de notre client afin de savoir pourquoi ces fonctions ont été mises là en premier lieu.

Lors de la création d’une session, si celle ci échoue, une exception est jetée :

```
private function __construct()
{
        if (session_start() === false) {
                throw new Exception("La session n'a pas réussie à démarrer.");
        }
}
```

Lors de l’instanciation d’une session, la possibilité de recevoir une exception n’est pas prise en considération :

```
public static function getInstance(): Session
{
        if (is_null(static::$instance)) {
```

```
                static::$instance = new Session();
                // Durée d'expiration des sessions en secondes
                $dureeExpiration = ConfigurationSite::getDureeExpirationSession();
                static::$instance->verifierDerniereActivite($dureeExpiration);
        }
        return static::$instance;
}
```

Problèmes de formulaires :
Il n'y a pas de vérification dans le code php des données renvoyées par les différents formulaires, ce qui permet aux utilisateurs de ne pas respecter les restrictions de mot de passes s' ils modifient le contenu des différents formulaires à l'aide de leurs navigateurs.

```
<input id="mdp" name="mdp" class="input is-large"
type="password" placeholder="********" minlength="6" maxlength="50"
pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[!@#$%^&*_=+\-]).{6,50}"
title="6 à 50 caractères, au moins une minuscule, une majuscule et un caractère spécial"
required>
```

Puisque qu'il n'y a aucune vérification du format en dehors de ces formulaires, il suffit de retirer ci-dessus les patterns et les attributs "minlength" et "maxlength" pour pouvoir insérer des données non-prévues par l'application, comme un mot de passe d'un seul caractère dans ce cas là. De la même manière, il est possible de créer une dépense au coût négatif.

Failles de sécurité et bugs

- L'utilisateur est supprimé lorsqu'il n'est le propriétaire d'aucun évènement.
- L'évènement est supprimé lorsque aucune dépense n'existe.
- mot de passe stocké en clair dans la bd
- il est possible d'accéder aux mots de passe avec uniquement l'adresse email de quelqu'un
- Il est possible d'avoir 2 comptes différents avec la même adresse email.
- il n'y a aucune vérification de l'adresse email.
- Une injection SQL est possible dans l'application

  La plupart de ces failles et bugs seront réglés avec une modification complète de la base de données. En effet, ce sont les limitations de la BD qui créent ces bugs.

  Afin de régler les autres problèmes, il faut repenser l'architecture de l'application. En effet, une vérification inexistante de l'adresse email, par exemple, permet à 2 utilisateurs de s'inscrire avec le même mail.
  Cela pose aussi problème lors de la perte du mot de passe, car dans notre application, aucun mail n'est envoyé, mais au contraire une page s'affiche avec le mot de passe écrit en clair.

Base de données

La base de données actuelle ne consiste que d'une seule table (annexe).
La clé primaire (loginProprietaire, idEvenement, idDepense) n'est pas une clef primaire de qualité. En effet, cette clé primaire permet de la duplication d'information, des incohérences, des problèmes lors de l'insertion, de la modification et de la suppression de données.
Toute la base de données étant sur une seule table, il est impossible d'avoir un utilisateur qui n'a pas d'événement, ni un événement sans dépense. Cela cause beaucoup de problèmes lors de la gestion des événements.
Les mots de passe sont stockés en clair dans la base de données, ce qui, dans le cas d'un piratage où de fuite de données, met en difficulté toute notre application. C'est quelque chose a ne jamais faire dans une DB.
Exemple 1 :
A créé un évènement en plus de l'évènement d'exemple, il invite B.
B invite A sur son évènement d'exemple: il n'y a qu'une seule dépense, référençant A en tant que payeur.
A quitte l'événement de B.
Résultat:
L'unique dépense de l'événement de B est supprimée,
donc l'unique évènement dont B est propriétaire est supprimé,
donc le compte de B est supprimé.
Donc l'évènement de A fait référence à B comme étant un membreEvenement, or ce dernier n'est plus utilisateur,
et on ne peut pas le supprimer de l'évènement.

Exemple 2 :
Voici le type de problèmes qu'on peut rencontrer avec la conception actuelle de la base de données. Une dépense unique peut être attribuée à plusieurs évenements. Un évenement qui n'est censé n'avoir qu'un seul propriétaire peut avoir 2 propriétaires.

loginPropriétaire	idEvenement	idDepense
1	1	1
1	2	1
2	1	1

Il serait donc idéal de changer la base de donnée, afin de faire disparaitre ces problèmes de sécurité. Nous proposons le schéma suivant (annexe 2)

(loginPropriétaire, nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire, idEvenement, codeSecretEvenement, titreEvenement, dateEvenement, membreEvenement, idDepense, titreDepense, dateDepense, montantDepense, loginPayeur, nomPayeur, prenomPayeur, participantsDepense)

Cette base de donnée n'est pas en première forme normale car il n'y a pas de DF entre la clef primaire et membresEvenement et participantsDepense. En effet, ces attributs peuvent contenir plusieurs informations, et non une seule.

PREMIÈRE FORME NORMALE :

App_db(loginPropriétaire, nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire, idEvenement, codeSecretEvenement, titreEvenement, dateEvenement, idDepense, titreDepense, dateDepense, montantDepense, loginPayeur, nomPayeur, prenomPayeur)

ParticiperEvenement(idEvenement,membreEvenement)

ParticiperDepense(idDepense, participantsDepense)

Liste Dépendances Fonctionnelles :

Après une analyse de l'application, nous avons déterminé les DF suivantes :

loginPropriétaire,idDepense,idEvenement -> nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire, codeSecretEvenement, titreEvenement, dateEvenement, titreDepense, dateDepense, montantDepense, loginPayeur, nomPayeur, prenomPayeur

loginPropriétaire -> nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire

idEvenement -> codeSecretEvenement, titreEvenement, dateEvenement,loginPropriétaire

idDepense -> titreDepense, dateDepense, montantDepense, loginPayeur

loginPayeur->nomPayeur, prenomPayeur

La fermeture transitive :

loginPropriétaire -> nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire
idEvenement -> codeSecretEvenement, titreEvenement, dateEvenement,loginPropriétaire,nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire

idDepense -> titreDepense, dateDepense, montantDepense, loginPayeur,nomPayeur, prenomPayeur

loginPayeur->nomPayeur, prenomPayeur

Pour les deux tables que nous avons créés, voici les clefs primaires

ParticiperEvenement(idEvenement,membreEvenement)
ParticiperDepense(idDepense, participantsDepense)

Notre schéma est maintenant en 1ere forme normale.
Les tables ParticiperEvenement et ParticiperDepense sont en troisième forme normale.

Pour une meilleure lisibilitée, nous posons :
LoginPropriétaire = A
nomPropriétaire = B
prénomPropriétaire = C
emailPropriétaire = D
mdpHashePropriétaire = E
mdpPropriétaire = F
idEvenement = G
codeSecretEvenement = H
titreEvenement = I
dateEvenement = J
idDepense = K
titreDepense = L
dateDepense = M
montantDepense = N
loginPayeur = O
nomPayeur = P
prenomPayeur = Q
membreEvenement = R
participantDepense = S

Notre base de donnée est donc représentée comme ceci :

App(ABCDEFGHIJKLMNOPQ)
R2(GR)
R3(KS)
Les deux dernières tables sont en 3ème forme normale.

Les df de R sont :

App{AGK-> B,C,D,E,F,H,I,J,L,M,N,O,P,Q
A->B,C,D,E,F

G-> H,I,J,A
K->L,M,N,O
O->P,Q}

La fermeture transitive de R est :
App+{A-> B,C,D,E,F
G-> H,I,J,A,B,C,D,E,F
K-> L,M,N,O,P,Q
O-> P,Q}

La seule clef possible est GK, car ce sont les deux seuls elements qui ne sont pas a droite et ils sont clefs ensemble.
C'est une clef minimale.

Casey - Delobel :
App(ABCDEFGHIJKLMNOPQ)

K-> LMNOPQ

App1(KLMNOPQ) App2(ABCDEFGHIJK)

O->PQ G-> ABCDEFHIJ

App11(OPQ) **App12(KLMNO)** App21(ABCDEFGHIJ) **App22(GK)**

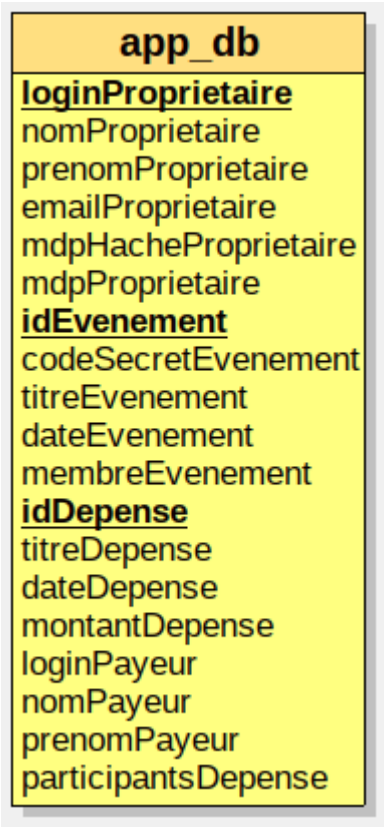
A->BCDEF

App211(ABCDEF) **App212(AGHIJ)**

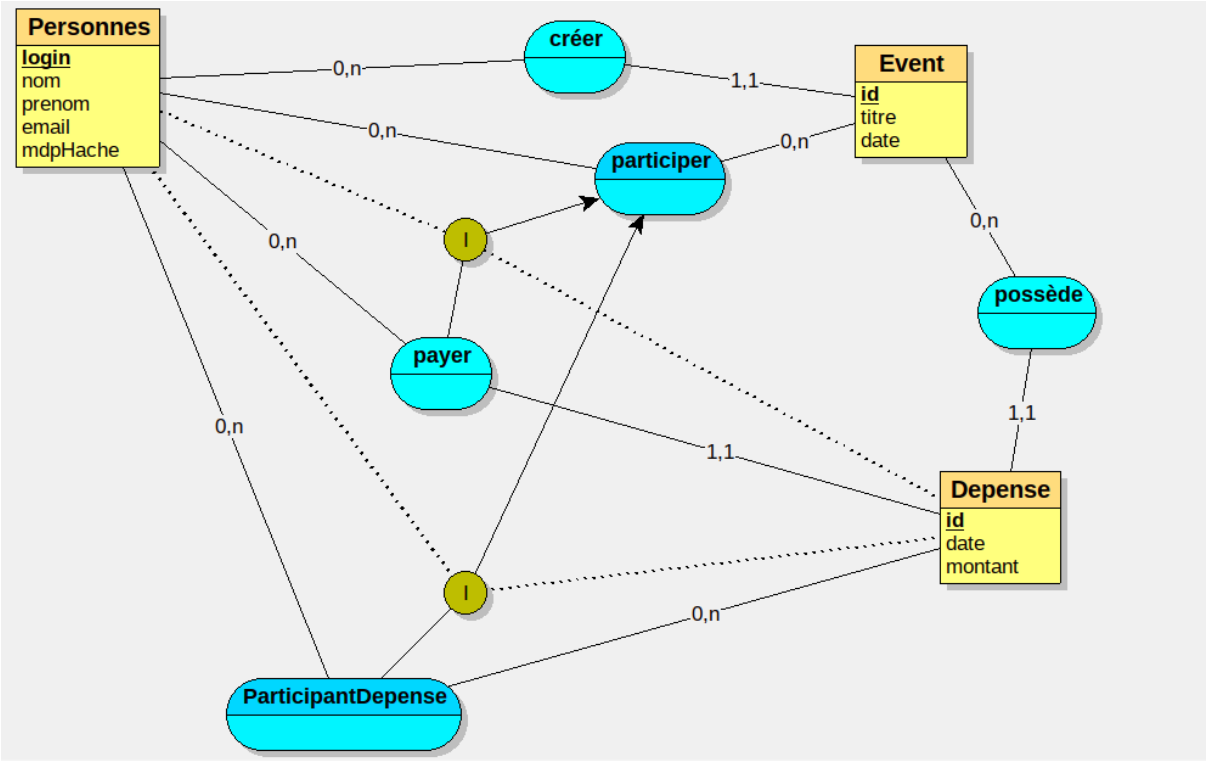
App11(loginPayeur,nomPayeur,PrénomPayeur)
App12(idDepense, titreDepense, dateDepense, montantDepense, loginPayeur)
App22(idEvenement, idDepense)
App211(loginPropriétaire, nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHashePropriétaire, mdpProprétaire)
App212(idEvenement, codeSecretEvenement, titreEvenement, dateEvenement, loginPropriétaire)
R2(idEvenement,membreEvenement)
R3(idDepense, participantDepense)

Annexes

Annexe 1 :



Annexe 2 :


Personnes
login
nom
prenom
email
mdpHache
créer
Event
id
titre
date
participer
possède
payer
Depense
id
date
montant
ParticipantDepense
0,n
1,1
0,n
0,n
0,n
1,1
1,1
0,n

Paulo fait des trucs (tkt)

La base de données actuelle ne consiste que d'une seule table (annexe).
La clé primaire (loginProprietaire, idEvenement, idDepense) n’est pas une clef primaire de qualité. En effet, cette clé primaire permet de la duplication d’information, des incohérences, des problèmes lors de l’insertion, de la modification et de la suppression de données.
Toute la base de données étant sur une seule table, il est impossible d’avoir un utilisateur qui n’a pas d'événement, ni un événement sans dépense. Cela cause beaucoup de problèmes lors de la gestion des événements.
Les mots de passe sont stockés en clair dans la base de données, ce qui, dans le cas d’un piratage où de fuite de données, met en difficulté toute notre application. C’est quelque chose à ne jamais faire dans une DB.
Exemple 1 :
A créé un évènement en plus de l’évènement d’exemple, il invite B.
B invite A sur son évènement d’exemple: il n’y a qu’une seule dépense, référençant A en tant que payeur.
A quitte l'événement de B.
Résultat:
L’unique dépense de l'événement de B est supprimée,
donc l’unique évènement dont B est propriétaire est supprimé,
donc le compte de B est supprimé.
Donc l’évènement de A fait référence à B comme étant un membreEvenement, or ce dernier n’est plus utilisateur,
et on ne peut pas le supprimer de l’évènement.

Exemple 2 :
Voici le type de problèmes qu’on peut rencontrer avec la conception actuelle de la base de données. Une dépense unique peut être attribuée à plusieurs évenements. Un évenement qui n’est censé n’avoir qu’un seul propriétaire peut avoir 2 propriétaires.

loginPropriétaire	idEvenement	idDepense
1	1	1
1	2	1
2	1	1

Il serait donc idéal de changer la base de données, afin de faire disparaître ces problèmes de sécurité. Nous proposons le schéma suivant (annexe 2)

(loginPropriétaire, nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire, idEvenement, codeSecretEvenement, titreEvenement, dateEvenement, membreEvenement, idDepense, titreDepense, dateDepense, montantDepense, loginPayeur, nomPayeur, prenomPayeur, participantsDepense)

Cette base de donnée n'est pas en première forme normale car il n'y a pas de DF entre la clef primaire et membresEvenement et participantsDepense. En effet, ces attributs peuvent contenir plusieurs informations, et non une seule.

PREMIÈRE FORME NORMALE :

App_db(loginPropriétaire, nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire, idEvenement, codeSecretEvenement, titreEvenement, dateEvenement, idDepense, titreDepense, dateDepense, montantDepense, loginPayeur, nomPayeur, prenomPayeur)

ParticiperEvenement(idEvenement,membreEvenement)

ParticiperDepense(idDepense, participantsDepense)

Liste Dépendances Fonctionnelles :

Après une analyse de l'application, nous avons déterminé les DF suivantes :

loginPropriétaire,idDepense,idEvenement -> nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire, codeSecretEvenement, titreEvenement, dateEvenement, titreDepense, dateDepense, montantDepense, loginPayeur, nomPayeur, prenomPayeur

loginPropriétaire -> nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire

idEvenement -> codeSecretEvenement, titreEvenement, dateEvenement,loginPropriétaire

idDepense -> titreDepense, dateDepense, montantDepense, loginPayeur

loginPayeur->nomPayeur, prenomPayeur

La fermeture transitive :

loginPropriétaire -> nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire
idEvenement -> codeSecretEvenement, titreEvenement, dateEvenement,loginPropriétaire,nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHachePropriétaire, mdpPropriétaire

idDepense -> titreDepense, dateDepense, montantDepense, loginPayeur,nomPayeur, prenomPayeur

loginPayeur->nomPayeur, prenomPayeur

Pour les deux tables que nous avons créés, voici les clefs primaires

ParticiperEvenement(idEvenement,membreEvenement)
ParticiperDepense(idDepense, participantsDepense)

Notre schéma est maintenant en 1ere forme normale.
Les tables ParticiperEvenement et ParticiperDepense sont en troisième forme normale.

Pour une meilleure lisibilitée, nous posons :
LoginPropriétaire = A
nomPropriétaire = B
prénomPropriétaire = C
emailPropriétaire = D
mdpHashePropriétaire = E
mdpPropriétaire = F
idEvenement = G
codeSecretEvenement = H
titreEvenement = I
dateEvenement = J
idDepense = K
titreDepense = L
dateDepense = M
montantDepense = N
loginPayeur = O
nomPayeur = P
prenomPayeur = Q
membreEvenement = R
participantDepense = S

Notre base de donnée est donc représentée comme ceci :

App(ABCDEFGHIJKLMNOPQ)
R2(GR)
R3(KS)
Les deux dernières tables sont en 3ème forme normale.

Les df de R sont :

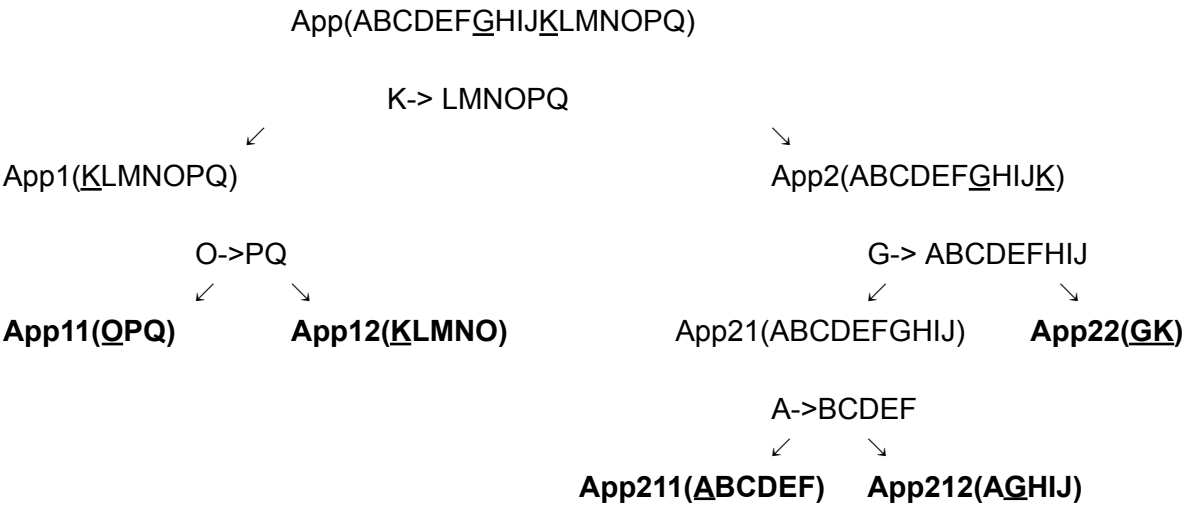
App{AGK-> B,C,D,E,F,H,I,J,L,M,N,O,P,Q
A->B,C,D,E,F

G-> H,I,J,A
K->L,M,N,O
O->P,Q}

La fermeture transitive de R est :
App+{A-> B,C,D,E,F
G-> H,I,J,A,B,C,D,E,F
K-> L,M,N,O,P,Q
O-> P,Q}

La seule clef possible est GK, car ce sont les deux seuls elements qui ne sont pas a droite et ils sont clefs ensemble.
C'est une clef minimale.

Casey - Delobel :



App11(loginPayeur,nomPayeur,PrénomPayeur) Table Payeur
App12(idDepense, titreDepense, dateDepense, montantDepense, loginPayeur) Table Depense
App22(idEvenement, idDepense)
App211(loginPropriétaire, nomPropriétaire, prénomPropriétaire, emailPropriétaire, mdpHashePropriétaire, mdpProprétaire) Table Propriétaire
App212(idEvenement, codeSecretEvenement, titreEvenement, dateEvenement, loginPropriétaire) Table Evenement
R2(idEvenement,membreEvenement) Table ParticiperEvenemtn
R3(idDepense, participantDepense) Table ParticiperDepense

Nous pouvons remarquer que la table Payeur et la table Propriétaire sont assez similaire, nous avons donc fait le choix de les rassembler en une seule table "membres"

Nous avons aussi supprimé la colonne “mdpPropriétaire” pour des raisons de sécuritées.