Nouveaux Paradigmes de Bases de Données

TD 1 : Programmer une implantation générique de l'ActiveRecord - fabrication d'un micro-ORM

Objectif : Comprendre l'anatomie d'un ORM et l'implantation générique d' ActiveRecord.

On souhaite implanter une classe abstraite réalisant toutes les fonctionnalités de l'ActiveRecord. Cette classe abstraite est destinée à être sous classée pour une utilisation concrète. Elle utilise une classe offrant des services pour construire des requêtes SQL.

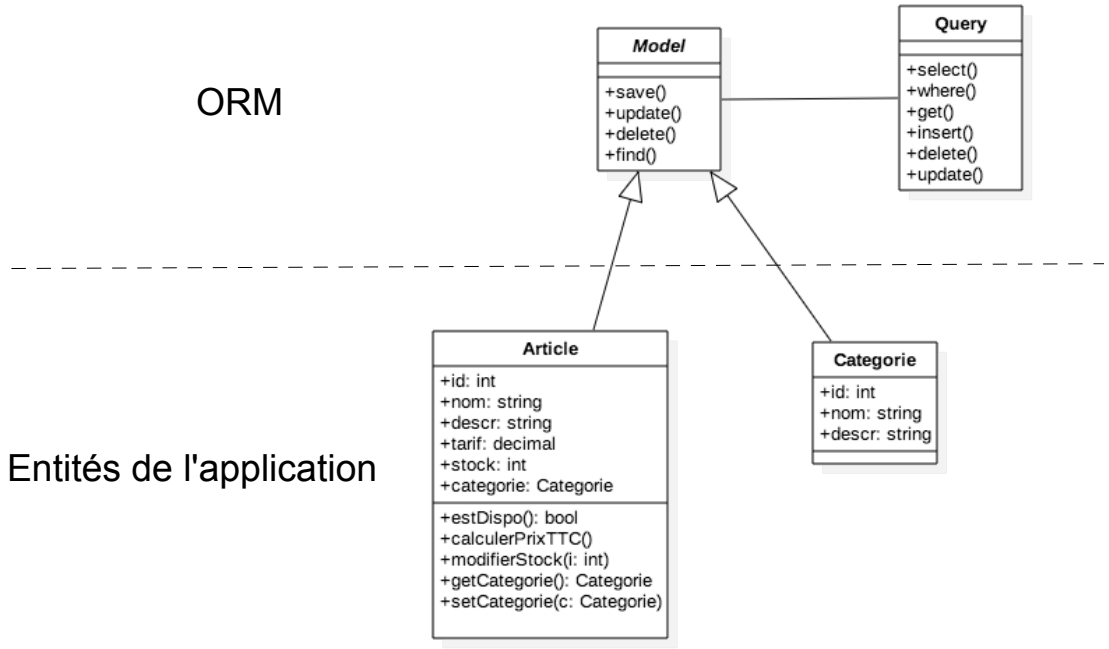
A l'utilisation, cette classe abstraite sera sous-classée pour obtenir un ActiveRecord associé à une table, comme dans l'exemple ci-dessous :

```
class Article extends \iutnc\hellokant\model\Model { … }
```

On pourra ensuite utiliser la classe Article pour interagir avec la base de données :

```
$a = new \iutnc\hellokant\model\Article() ;
$a->nom= 'A12609' ;
$a->descr= 'beau velo de course rouge' ;
$a->tarif= 59.95 ;
$a->insert() ;
```

Dans cet exemple, les classes utilisées correspondent à ce schéma :



1. la classe Query

On souhaite programmer la classe `Query` qui réalise une interface de construction-exécution de requêtes SQL.

Le nom complet de la classe doit être `\iutnc\hellokant\query\Query` et elle doit être placée dans le répertoire `src/query/` . Il est demandé d'utiliser un autoloader de classes PSR4 généré avec l'outil composer.

Important : toutes les requêtes construites et exécutées par la classe Query doivent être des requêtes PDO préparées.

Dans un premier temps, on programme la classe de façon à ce l'exécution d'une requête se contente d'afficher le code de la requête SQL et ses paramètres, sans l'exécuter. Il n'y a donc pas lieu de connecter l'exécution à une base de données. Lorsque le résultat est satisfaisant, on cherchera à établir la connexion avec la base et exécuter réellement les requêtes (étape 2 dans la suite).

1. Créer la classe `Query` et déclarer les attributs permettant de stocker les différentes parties de la requête (colonnes, table, conditions where),
2. créer la méthode statique `table( … )` qui créé une instance de `Query` et stocke le nom de la table concernée ; elle retourne le nouvel objet `Query`,
3. créer la méthode `where( … )` qui reçoit 3 paramètres correspondant aux composant d'une condition et stocke cette condition dans la requête ; retourne l'objet `Query` courant.
4. créer la méthode `get()` qui construit le texte complet de la requête et son tableau d'arguments, **et affiche cette requête**.
5. créer la méthode `select()` qui permet de choisir les colonnes dans le résultat, adapter la méthode `get()` pour en tenir compte ; retourne l'objet `Query` courant.
6. créer la méthode delete() qui permet de supprimer une ligne dans une table.
7. créer la méthode `insert()` qui reçoit en paramètre un tableau et insère ce tableau comme une nouvelle ligne dans la table.

2. Gestion de la connexion à la base

On gère la connexion à la base dans une classe dédiée en utilisant PDO.

Construire la classe "`ConnectionFactory`" qui contient 2 méthodes statiques :

- `makeConnection(array $conf)` : Cette méthode reçoit un tableau contenant les paramètres de connexion, établit cette connexion en créant un objet PDO, stocke cet objet dans une variable statique et la retourne en résultat. Elle est utilisée 1 seule fois au démarrage d'une application pour configurer la connexion à la base.
- `getConnection()` : permet d'obtenir une connexion à condition qu'elle ait été créée auparavant. Cette méthode retourne le contenu de la variable statique, et s'utilise chaque fois que cela est nécessaire d'exécuter une requête sur la base de données – par exemple la la classe `Query`.

Ces méthodes s'utiliseront ainsi :

```
// une seule fois au lancement de l'application
```

```
$conf = parse_ini_file('conf/db.conf.ini') ;
ConnectionFactory::makeConnection($conf);

// chaque fois qu'il est nécessaire d'obrenir une connexion
$myPdo = ConnectionFactory::getConnection() ;
```

Par défaut, on établit une connexion persistante (`PDO::ATTR_PERSISTENT=>true`), avec les erreurs en mode exception (`PDO::ATTR_ERRMODE = PDO::ERRMODE_EXCEPTION`). Par précaution, on positionne `PDO::ATTR_EMULATE_PREPARES=> false` et `PDO::ATTR_STRINGIFY_FETCHES => false`

3. Finaliser la classe Query

Finaliser la classe Query de façon à ce que les différentes méthodes aient un effet sur la base de données et exécutent les requêtes préparées SQL avec PDO. Pour cela, chaque méthode devant déclencher une exécution effective doit être compléter avec :

- obtention d'un objet PDO en utilisant la classe `ConnectionFactory` de la question 2,
- préparation de la requête,
- exécution de la requête avec transmission des arguments,
- renvoi d'un résultat.

Dans le cas de la méthode `insert()`, faire en sorte que le résultat retourné soit la valeur de la clé primaire attribuée à la nouvelle ligne lorsque cette clé primaire est une colonne dont la valeur est auto-incrémentée.

Indication : utiliser la méthode `PDO::lastInsertId()`

4. la classe Model

Créer la classe abstraite `\iutnc\hellokant\model\Model`, et la classe `Article` qui hérite de Model (vides toutes les 2 pour l'instant).

Dans la classe `Model`, on choisit de représenter les attributs d'un modèle sous la forme d'un tableau qui associe nom d'attribut et valeur : `[ <nom_att> => <val_att> ]`.

1. déclarer un attribut de la classe `Model` pour stocker ce tableau et l'initialiser avec un tableau vide.
2. construire les accesseurs pour accéder et modifier les valeurs des attributs du modèle. Pour cela, on utilise les méthodes "magic" `__get()` et `__set()` (voir la doc php pour des détails).
3. ajouter un constructeur pour la classe `Model`. Ce constructeur reçoit un paramètre optionnel qui est un tableau d'attributs.
4. tester en utilisant la classe `Article` : créer un programme qui importe les 2 classes, puis crée un article et le remplit, puis accède aux attributs pour les afficher.
5. La méthode `delete()` : la méthode permet de supprimer les données dans la base

correspondant à l'objet. La méthode ne reçoit aucun paramètre et supprime la ligne correspondant à l'objet dans la ***table*** concernée. Ecrire cette méthode dans la classe `Model`. **Attention :** elle doit vérifier que l'objet possède une valeur pour sa ***clé primaire***, puis utilise la classe `Query` pour exécuter la requête.

Pour réaliser cela, la méthode `delete()` doit connaître le nom de la table associée au modèle ainsi que le nom de la colonne clé primaire. Où va-t-elle trouver ces informations ?

6. tester en utilisant la classe Article : créer un objet Article, le remplir puis appeler la méthode delete() et vérifier que la requête SQL est correcte.
7. Ecrire maintenant la méthode `insert()` qui transforme l'objet courant en une ligne de table. Elle utilise la classe `Query` pour exécuter la requête. Elle crée systématiquement une nouvelle ligne, et stocke dans l'objet la valeur de son identifiant qui a été créé par auto-incrément par la base de données.

5. les "Finders"

On veut construire maintenant des méthodes permettant de retrouver des objets modèles stockés dans la base. Ces méthodes sont des **méthodes statiques**, qui retournent des **tableaux de modèles**.

1. programmer la méthode all() qui retourne l'ensemble des lignes de la table sous la forme d'un tableau de modèles :

   ```
   $liste = Article::all() ;
   foreach ($liste as $article) print $article->nom ;
   ```

   Pour cela, elle exécute la requête correspondante à l'aide de la classe `Query` puis transforme chaque ligne en 1 objet.

2. On veut maintenant construire la méthode `find()` qui permet de sélectionner des lignes dans la base et **retourne toujours un tableau d'objets modèles**.

   a) construire la méthode `find` pour le cas où elle reçoit un paramètre entier, correspondant à la valeur de l'identifiant de l'objet recherché :

   ```
   $l = Article::find(27) ;
   $article = $l[0] ;                          // retrouve l'article d'id 27
   ```

   b) compléter la méthode `find()` pour quelle accepte un critère de recherche en paramètre, à la place d'un identifiant de d'objet. Ce critère de recherche est un tableau indiquant le nom de la colonne, l'opérateur de comparaison, la valeur. Par exemple :

   ```
   Article::find( ['tarif', '<=', 100 ], ['nom', 'tarif'] ) ;
   ```

6. Gestion des associations

Créer le modèle pour les catégories. Bien sur il doit hériter de `Model`.

On veut maintenant programmer des méthodes génériques définies dans la classe `Model` pour gérer les associations entre objets : pour un article, obtenir l'objet correspondant à sa catégorie ; pour une catégorie, obtenir l'ensemble d'objets articles appartenant à cette catégorie.

1. programmer la méthode `belongs_to()` qui permet de suivre une association de multiplicité 1 ; elle est appelée sur un modèle correspondant à une table détenant une clé étrangère vers une table liée. Elle retourne un objet instance du modèle correspondant à la table liée. La méthode reçoit en paramètre : le nom du modèle lié, le nom de la clé étrangère. Ainsi, pour un article on obtient sa marque:

```
$a=Article::first(56);
$categorie = $a->belongs_to('Categorie', 'id_categ') ;
```

2. programmer la méthode `has_many()` qui permet de suivre une association de multiplicité n ; elle est appelée sur un modèle dont la clé primaire est associée à une clé étrangère dans la table liée. Elle retourne un tableau d'objets instances du modèle correspondant à la table liée. La méthode reçoit en paramètre : le nom du modèle lié, le nom de la clé étrangère. Ainci, pour une catégorie, on obtient les articles de cette catégorie :

```
$m = Categorie::first(1) ;
$list_article = $m->has_many('Article', 'id_marque') ;
```

3. Utiliser ces 2 méthodes pour créer :
    1. la méthode `categorie()` de la classe `Article`, qui retourne la categorie d'un article :

    ```
    $categorie = Article::first(56)->categorie() ;
    ```

    2. la méthode `articles()` de la classe `Categorie`, qui retourne tous les articles d'une catégorie :

```
$list = Categorie::first(1)->articles() ;
```