TP 3 : Le modèle d'un projet d'application Web

L'objectif de ce TP est de développer le modèle d'un projet d'application PHP, suivant l'architecture MVC présentée dans le TP1. Un modèle définit un ensemble d'objets en lien direct avec l'application visée.

Dès lors qu'un objet fera référence à une entité stockée dans la base de données, nous ferons appel à l'ORM (Object Relational Mapping) Doctrine 2 pour définir cet objet et le manipuler.

Dans ce TP, vous devez comprendre et implémenter ce modèle dans son ensemble.

Doctrine 2:
Vous trouverez ci-dessous un ensemble d'informations vous permettant d'appréhender Doctrine 2. Pour plus d'informations, nous vous renvoyons évidemment aux différentes documentations dont certains liens sont donnés ci-dessous.

Définition des entités par annotation manuelle (http://doctrine-orm.readthedocs.org/projects/doctrine-orm/en/latest/reference/annotations-reference.html)
Comme mentionné plus haut, certains objets du modèle feront référence à des entités, chacune liée à une table de la base de données ou à plusieurs (cas de jointure entre tables)[1]. Doctrine 2 fournit différents formats pour définir une entité : langage YAML, XML ou sous la forme d'une annotation textuelle. Nous choisissons ici cette dernière solution.
Les annotations textuelles sont :
- des lignes de codes spécifiques
- encadrées par les symboles /** et */, très similaires à des commentaires
- des entrées permettant de définir la ou les tables de la bases de données à laquelle l'entité est associée, les liens entre colonnes d'une table et propriétés de l'entité, d'éventuelles jointures entre tables.

Chaque classe, définissant un objet se rapportant à une entité, devra être précédée d'une annotation de type *@Entity.* Si seule l'annotation *Entity* est fournie, le nom de la classe fait référence au nom de la table en base de données. Sinon, l'annotation *@Table(name='' nom_de_la_table'') sera utilisée.*

```
/**
 * @Entity
 * @Table(name="fredouil.utilisateur")
 */
class utilisateur{
...
}
```

La définition d'un objet se rapportant à une entité s'effectue au travers de propriétés de la classe, chacune faisant référence à une colonne de la table de la base de données. L'annotation textuelle *@column* permet d'associer une propriété à une colonne. Elle est localisée avant la définition de la propriété. Une annotation *@Column* peut contenir différents paramètres, la plupart facultatif, permettant de caractériser la colonne : *type* (type doctrine ayant une coorspondance avec les types SQL), *name* (nom de la colonne), *length* (longueur d'une *string*), *unique, nullable, precision* (nombre de chiffres pour un type *decimal*), *scale (*nombre de chiffres après la virgule pour un type *decimal), options*. L'annotation *@Id* permet de déterminer les clés primaires dans les tables et les propriétés associés.

```
/** @Id @Column(type="integer")
 *  @GeneratedValue
 */
public $id;

/** @Column(type="string", length=45) */
public $identifiant;
```

1 Doctrine 2 fournit des mécanismes permettant, à partir de la définition d'entités, de créer directement dans la base de données les tables correspondantes. Ici, les tables existent déjà.

Il est possible dans une classe de définir des propriétés supplémentaires faisant référence à des propriétés d'autres classes/entités et dont les valeurs seront obtenues par une relation entre entités.
Les relations entre entités et attribuées à une propriété sont définies à partir des annotations suivantes :
- *@OneToMany* prenant comme paramètres : *targetEntity* (nom de la classe ciblée), *mappedBy* (nom de la propriété de l'entité cible avec laquelle la jointure est réalisée)
- *@ManyToOne* prenant comme paramètres : *targetEntity* (nom de l'entité ciblée), *inversedBy* (nom de la propriété de l'entité cible permettant la relation inverse)
- *@OneToOne* prenant comme paramètres : *targetEntity* (nom de l'entité ciblée), *inversedBy* (nom de la propriété de l'entité cible permettant la relation inverse)
- *@ManyToMany* prenant comme paramètres : *targetEntity* (nom de l'entité ciblée), , *mappedBy* (nom de la propriété de l'entité cible avec laquelle la jointure est réalisée), *inversedBy* (nom de la propriété de l'entité cible permettant la relation inverse)

Egalement :
- *@JoinColumn* prenant comme paramètres : *name* (nom de la colonne de l'entité courante identifiée comme clé étrangère), *referencedColumnName* (nom de la clé primaire utilisée dans l'entité cible). *@JoinColumn* peut être utilisée pour compléter les relations *@ManyToOne* et *@OneToOne.*

Ces relations vous permettront notamment de récupérer, par exemple, l'ensemble des messages postés par un utilisateur donné et ce, par une simple instruction, de type : $user->messages

Classes et méthodes utiles (liste non exhaustive!!!)

→ *EntityRepository* : Une fois définie, chaque entité est associée à un *repository* qui va nous permettre de récupérer et de manipuler les données associées.

Méthodes	**Définition**
find, findAll, findBy, findOneBy	Méthodes de récupération des données
CreateQueryBuilder	Méthodes de création de requêtes

(http://www.doctrine-project.org/api/orm/2.4/class-Doctrine.ORM.EntityRepository.html)

→ *EntityManager* : cette classe va permettre la connexion à la base de données, fournir une instance de connexion donnant accès aux repositories et fournir un ensemble de méthodes utiles pour la gestion des données.

Méthodes	**Définition**
create	Création d'une instance de connexion à la base dont la config est passée en paramètre
getRepository	Renvoi du repository associé à une entité associée à l'instance de connexion
persist	Persistance de l'objet (associé à une entité) passé en paramètre en base de données
remove	Suppression de l'objet (associé à une entité) passé en paramètre en base de données
CreateQueryBuilder	Méthodes de création de requêtes

(http://www.doctrine-project.org/api/orm/2.4/class-Doctrine.ORM.EntityManager.html)

→ *QueryBuilder :* classe permettant de créer des requêtes personnalisées.
(http://www.doctrine-project.org/api/orm/2.4/class-Doctrine.ORM.QueryBuilder.html)

Application :

Pour rappel, l'arborescence du projet est présentée dans la figure 1.

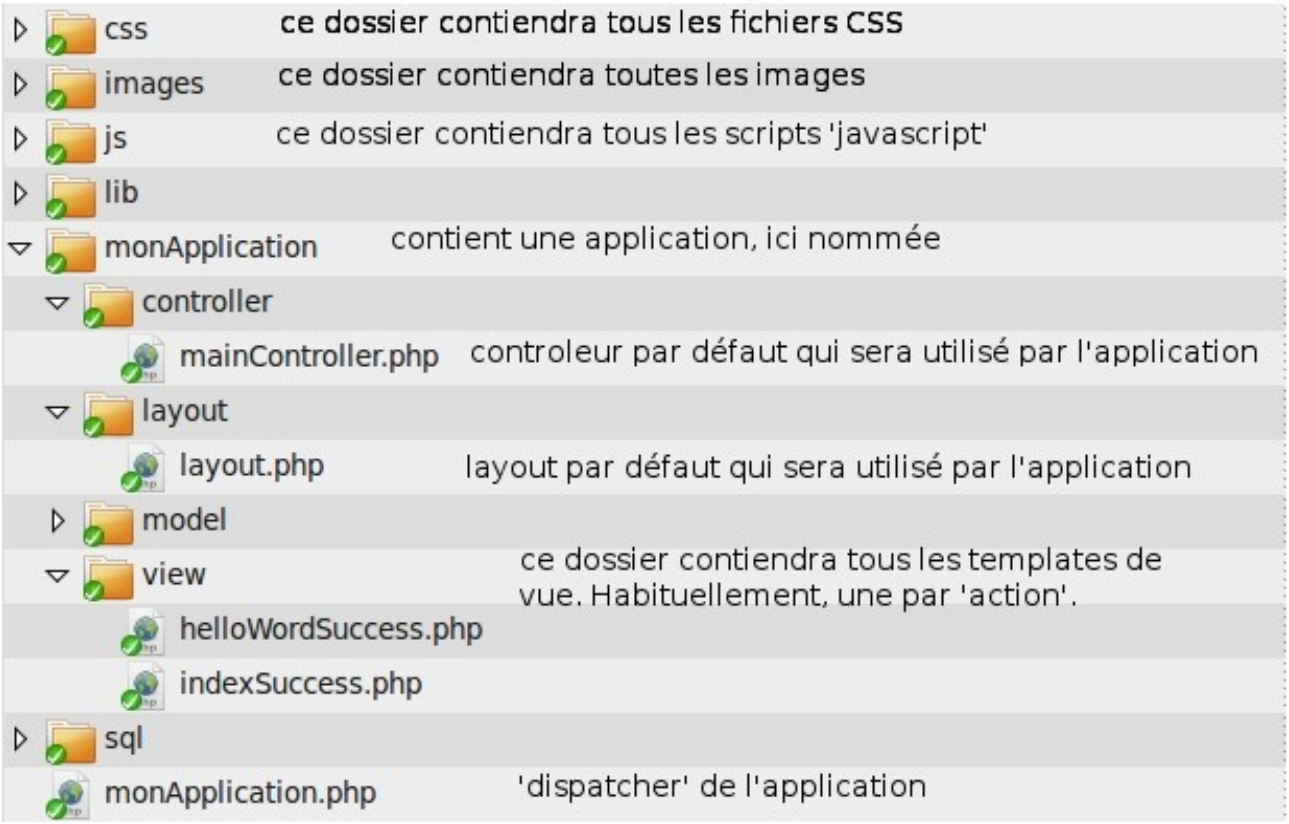

Figure 1: Arborescence de l'architecture MVC

Le dossier « model » contient les classes propres à l'application. Pour l'instant, seules les classes « utilisateur » et « utilisateurTable » sont présentes. La classe « utilisateur » propose un exemple d'annotation brute pour définir l'entité correspondante. Seules les propriétés sont définies. Aucune relation n'est spécifiée. La classe « utilisateurTable » contient la méthode permettant à l'utilisateur de se connecter à son profil par son identifiant et son mot de passe. Nous implémenterons dans ce TP les classes d'entités « utilisateur », « message », « post » et « chat », puis les classes fonctionnelles si nécessaire, soit « utilisateurTable », « postTable », « messageTable », « chatTable ». Les méthodes définies dans ces classes seront a priori statiques (à moins que votre implémentation demande le contraire) et appelées directement sans instancier d'objet de classes. Concrètement, ces classes font la correspondance entre le modèle de données et notre application. Elles renverront des instances de message construites à partir des lignes récupérées en base.

* Un exemple : la classe *message* représente une instance (= un enregistrement dans la table) d'un message. La table *messageTable* est la classe qui permet de récupérer à travers la base de données des instances de message (= représente l'accès à la table).
Ces deux classes fournissent une couche d'abstraction à la base de données et apportent une structure métier qui colle au modèle de données.

Implémentation :

Ce TP est à réaliser en binôme. Une première partie sera commune pour les deux partenaires puis chacun réalisera une partie qui lui sera propre. Ces parties seront ensuite regroupées et harmonisées en vue d'un seul rendu. Pour chaque morceau de code commun ou séparé, nous devrons être en mesure de connaître l'auteur. Nous vous recommandons fortement d'ajouter avant chaque méthode ou morceau de code quelques lignes (si possible normalisées suivant des conventions que vous mettrez en place) expliquant l'objectif du morceau de code, son

auteur ; s'il s'agit d'une méthode, vous pouvez rajouter les paramètres en entrée et en sortie, ...

Partie commune

- Vous devez implémenter l'ensemble des classes *entités* manquantes.
- Vous devez implémenter les méthodes dans le contrôleur vous permettant de tester votre modèle.

Premier partenaire

Vous devez implémenter la classe *utilisateurTable.* Cette classe devra contenir au minimum :

- L'une d'elle est déjà implémentée et est utilisée pour la connexion d'un utilisateur à son profil, c'est la méthode « getUserByLoginAndPass ».

- Une seconde méthode, *getUserById($id),* est destinée à récupérer les informations d'un utilisateur selon un identifiant (*id*). Nous pourrons ultérieurement afficher le profil de quelqu'un d'autre.
- Enfin, une méthode *getUsers()* est nécessaire afin de collecter l'ensemble des utilisateurs utilisant notre plate-forme de réseau social.

Deuxième partenaire

Pour la classe *chatTable*, vous devez implémenter :
- une méthode *getChats()* permettant de collecter l'ensemble des chats, via une requête récupérant les données de la table *chat*. Cette méthode retournera une collection contenant des objets de type *chat*.
- une méthode *getLastChat()* permettant de récupérer le dernier message posté sur le chat.

Pour finir

De manière individuelle, vous devez répondre à cette question dans un fichier readme. Expliquer **clairement** les mécanismes php permettant d'appeler et exécuter les méthodes - *findOneByidentifiant() ou findByidentifiant()* **-** sur un repository associé à l'entité *utilisateur* sachant que *identifiant* est une de ces propriétés.

Vous pourrez évidemment enrichir le modèle si vous le désirez par toutes autres méthodes jugées nécessaires pour la mise en place de votre application (en commençant par la classe *messageTable* par exemple).