Doctrine

< Javascript >>

1. Introduction à CodeIgniter
2. Helpers
3. Bibliothèques
4. Sessions CodeIgniter
5. Validation des formulaires
6. ORM IgnitedRecord
7. Doctrine
8. Javascript et codeIgniter

Doctrine est également un ORM qui peut être associé à CodeIgniter, il est beaucoup plus puissant, et plus complet.

- Site de référence Doctrine
- Documentation
- Téléchargement DoctrineORM-2.2.1-full

Installation

Doctrine est installé en tant que bibliothèque dans codeIgniter.
Créer un nouveau projet PHP, installer à nouveau CodeIgniter.
Avec l'archive Doctrine :

- Dézipper l'archive.
- Copier le dossier **Doctrine** de l'archive dans le dossier **application/libraries**.

Créer une classe **Doctrine.php** dans le dossier libraries :

|h application/libraries/Doctrine.php

```
<?php
class Doctrine
{
    // the Doctrine entity manager
    public $em = null;

    public function __construct()
    {
        // include our CodeIgniter application's database configuration
        require APPPATH.'config/database.php';
        // include Doctrine's fancy ClassLoader class
        require_once APPPATH.'libraries/Doctrine/Common/ClassLoader.php';

        // load the Doctrine classes
        $doctrineClassLoader = new \Doctrine\Common\ClassLoader('Doctrine',
APPPATH.'libraries');
        $doctrineClassLoader->register();

        // load Symfony2 helpers
```

```
        // Don't be alarmed, this is necessary for YAML mapping files
        $symfonyClassLoader = new \Doctrine\Common\ClassLoader('Symfony',
APPPATH.'libraries/Doctrine');
        $symfonyClassLoader->register();

        // load the entities
        $entityClassLoader = new \Doctrine\Common\ClassLoader('Entities',
APPPATH.'models');
        $entityClassLoader->register();

        // load the proxy entities
        $proxyClassLoader = new \Doctrine\Common\ClassLoader('Proxies',
APPPATH.'models');
        $proxyClassLoader->register();

        // set up the configuration
        $config = new \Doctrine\ORM\Configuration;

        if(ENVIRONMENT == 'development')
            // set up simple array caching for development mode
            $cache = new \Doctrine\Common\Cache\ArrayCache;
        else
            // set up caching with APC for production mode
            $cache = new \Doctrine\Common\Cache\ApcCache;
        $config->setMetadataCacheImpl($cache);
        $config->setQueryCacheImpl($cache);

        // set up proxy configuration
        $config->setProxyDir(APPPATH.'models/Proxies');
        $config->setProxyNamespace('Proxies');

        // auto-generate proxy classes if we are in development mode
        $config->setAutoGenerateProxyClasses(ENVIRONMENT == 'development');

        // set up annotation driver
        //$yamlDriver = new
\Doctrine\ORM\Mapping\Driver\YamlDriver(APPPATH.'models/Mappings');
        $driverImpl = $config->newDefaultAnnotationDriver(APPPATH.'models');
        $config->setMetadataDriverImpl($driverImpl);

        // Database connection information
        $connectionOptions = array(
            'driver' => 'pdo_mysql',
            'user' => $db['default']['username'],
            'password' => $db['default']['password'],
            'host' => $db['default']['hostname'],
            'dbname' => $db['default']['database']
        );

        // create the EntityManager
        $em = \Doctrine\ORM\EntityManager::create($connectionOptions,
$config);

        // store it as a member, for use in our CodeIgniter controllers.
        $this->em = $em;
    }
```

```
}
?>
```

Doctrine doit être ensuite chargé automatiquement avec autoload.php :

```
$autoload['libraries'] = array('database','doctrine');
```

Le chargement de la bibliothèque **database** est indispensable

Logiquement, Doctrine est prêt à fonctionner.
Vérifier que votre page d'accueil ne produit pas d'erreurs : http://localhost/siteURL/

Création des classes métier

Une classe métier correspond à la notion d'**entity** dans Doctrine.

Considérons la base de données suivante :

La base de données sera composée de 2 entities: utilisateur et categorie. La relation de type CIF entre utilisateurs et categories peut s'exprimer de la façon suivante :

- Chaque utilisateur appartient à 1 catégorie (manyToOne)
- Dans chaque categorie, on peut compter de 0 à n utilisateurs (oneToMany)

Le model utilisateur

Dans le dossier **application/models** :

- créer le fichier **utilisateur.php**
- générer ensuite les accesseurs sur les membres
- créer un constructeur sans paramètres

|h application/models/utilisateur.php

```
<?php
/**
 * @Entity
 * @Table(name="utilisateurs")
 */
class Utilisateur {
    /**
     * @Id @Column(type="integer")
     * @GeneratedValue
     */
    private $id;
```

```
    /**
     * @Column(type="string")
     * @var string
     */
    private $nom;
    /**
     * @Column(type="string")
     * @var string
     */
    private $prenom;
    /**
     * @Column(type="integer")
     * @var integer
     */
    private $age;
    /**
     * @Column(type="boolean")
     * @var string
     */
    private $adulte;
    /**
     * @ManyToOne(targetEntity="Categorie")
     * @JoinColumn(name="categorie_id", referencedColumnName="id")
     */
    private $categorie;
}
?>
```

- Un **model** est un fichier contenant une classe dont le nom commence par une majuscule.
- Le nom du fichier doit être le même que celui de la classe, mais en minuscule.
- Le fichier doit être enregistré dans le dossier **application/models/**
- Doctrine peut utiliser plusieurs systèmes pour définir les modèles :
    - avec annotations dans le code php comme dans l'exemple
    - avec fichiers xml
    - avec fichiers yaml

Le model categorie

Dans le dossier **application/models** :

- créer le fichier **categorie.php**
- générer ensuite les accesseurs sur les membres
- créer un constructeur sans paramètres

|h application/models/categorie.php

```
<?php
/**
 * @Entity
 * @Table(name="categories")
 */
class Categorie{
```

```
    /**
    * @Id @Column(type="integer")
    * @GeneratedValue
    */
    private $id;
    /**
     * @Column(type="string")
     * @var string
     */
    private $nom;

    /**
     * @OneToMany(targetEntity="Utilisateur",mappedBy="categorie")
     */
    private $utilisateurs;
}
?>
```

Chargement des models

Le chargement peut être automatique, par le biais de application/config/autoload.php

```
$autoload['model'] = array('categorie','utilisateur');
```

ou bien se faire dans un contrôleur :

```
$this->load->model('utilisateur');
```

Il est nécessaire de créer à la main le dossier **Proxies** dans **application/models** pour permettre la génération à la volée des classes de mapping

Gestion des utilisateurs

Contrôleur utilisateurs

Ajouter un contrôleur utilisateurs dans controllers :

- la méthode **all** charge tous les utilisateurs, et leur catégorie correspondante.
- Elle appelle ensuite la vue **v_utilisateurs** et lui passe les utilisateurs chargés (**users**)

|h application/controllers/utilisateurs.php

```
<?php
class Utilisateurs extends CI_Controller{
    public function all(){
        $query = $this->doctrine->em->createQuery("SELECT u FROM Utilisateur u
join u.categorie c");
        $users = $query->getResult();
        $this->load->view('v_utilisateurs',array('utilisateurs'=>$users));
    }
}
```

```
?>
```

Vues

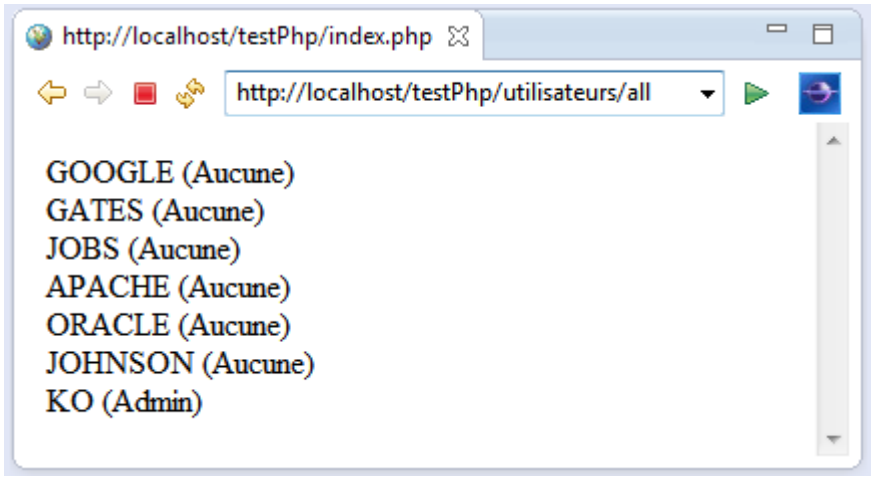
Liste des utilisateurs

Créer la vue v_utilisateurs pour afficher la liste des utilisateurs : La variable $utilisateurs est récupérée par la méthode all du contrôleur utilisateurs

|h application/views/v_utilisateurs.php

```
<?php
foreach ($utilisateurs as $user){
    echo($user->getNom()." (".$user->getCategorie()->getNom().")<br>");
}
?>
```

Tester en allant à l'adresse /utilisateurs/all/



Ajout d'utilisateur

Modification du contrôleur

Modifier le contrôleur utilisateurs :

- La méthode **add** permet d'afficher un formulaire **v_utilisateur_add** permettant d'ajouter un utilisateur en saisissant son nom.
- La méthode **submit_add** effectue la validation du formulaire en cas de succès de la validation puis appelle la vue **v_success_add**

|h application/controllers/utilisateurs.php

```
<?php
class Utilisateurs extends CI_Controller{
    public function add(){
```

```
        $this->load->helper(array('form', 'url'));

        $this->load->library('form_validation');

        $this->form_validation->set_rules('username', 'Username',
'trim|required|min_length[5]|max_length[12]|xss_clean');
        if ($this->form_validation->run() == FALSE)
        {
            $this->load->view('v_utilisateur_add');
        }
        else
        {
            $this->submit_add($_POST["username"]);
        }
    }

    public function submit_add($name){
        $user = new Utilisateur();
        $user->setNom($name);
        $this->doctrine->em->persist($user);
        $this->doctrine->em->flush();
    }

    public function all(){
        $query = $this->doctrine->em->createQuery("SELECT u FROM Utilisateur u
join u.categorie c");
        $users = $query->getResult();
        $this->load->view('v_utilisateurs',array('utilisateurs'=>$users));
    }
}
?>
```

Ajout des vues

La vue **v_utilisateur_add** sera appelée par l'intermédiaire du contrôleur **utilisateurs/add**

|h application/views/v_utilisateur_add.php

```
<html>
<head>
<title>Ajout utilisateur</title>
</head>
<body>

<?php echo validation_errors(); ?>

<?php echo form_open('utilisateurs/add/'); ?>

<h5>Nom d'utilisateur</h5>
<input type="text" name="username" value="<?php echo set_value('username');
?>" size="50" />

<div><input type="submit" value="Ajouter utilisateur" /></div>
```

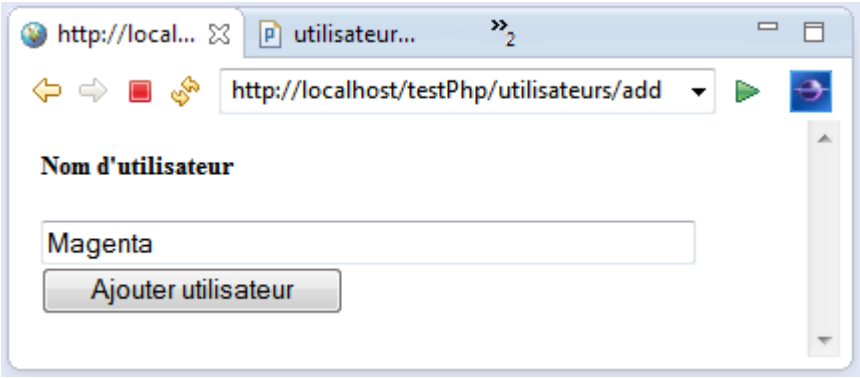
```
</form>

</body>
</html>
```

La vue **v_success_add** sera appelée après soumission du formulaire par le contrôleur **utilisateurs/submit_add**

|h application/views/v_success_add.php

```
<?php
echo($user->nom." ajouté");
?>
```

Tester en allant à l'adresse : http://localhost/testPhp/utilisateurs/add/



Vérifier l'insertion dans la base de données du nouvel utilisateur.

Sur le même principe que pour les utilisateurs, en respectant MVC :

- Créer un contrôleur categories
- Afficher la liste des catégories, et les utilisateurs correspondants
- Créer la fonctionnalité d'ajout de catégorie
- Créer la fonctionnalité de modification d'une catégorie existante
- Créer la fonctionnalité de suppression d'une catégorie

Génération des classes métier

L'accès à la base de données doit-être correctement configuré dans le fichier **config/database.php**.
Doctrine est installé en tant que Librairie.

Configuration de l'outil en ligne de commande

Créer le fichier **doctrine-cli.php** dans le dossier **application**, et modifier la variable **APPPATH** : partie siteURL

```
<?php
```

```
// trailing slash is important!
define('APPPATH', 'c:/xampp/htdocs/siteURL/application/');
define('BASEPATH', APPPATH);
define('ENVIRONMENT', 'production');

require APPPATH.'libraries/Doctrine.php';

$doctrine = new Doctrine();

$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new
\Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($doctrine->em->getConnection()
),
    'em' => new
\Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($doctrine->em)
));

$cli = new \Symfony\Component\Console\Application('Doctrine Command Line Interface
(CodeIgniter integration by Joel Verhagen)', Doctrine\ORM\Version::VERSION);
$cli->setCatchExceptions(true);
$cli->setHelperSet($helperSet);
$cli->addCommands(array(
    // DBAL Commands
    new \Doctrine\DBAL\Tools\Console\Command\RunSqlCommand(),
    new \Doctrine\DBAL\Tools\Console\Command\ImportCommand(),

    // ORM Commands
    new \Doctrine\ORM\Tools\Console\Command\ClearCache\MetadataCommand(),
    new \Doctrine\ORM\Tools\Console\Command\ClearCache\ResultCommand(),
    new \Doctrine\ORM\Tools\Console\Command\ClearCache\QueryCommand(),
    new \Doctrine\ORM\Tools\Console\Command\SchemaTool\CreateCommand(),
    new \Doctrine\ORM\Tools\Console\Command\SchemaTool\UpdateCommand(),
    new \Doctrine\ORM\Tools\Console\Command\SchemaTool\DropCommand(),
    new \Doctrine\ORM\Tools\Console\Command\EnsureProductionSettingsCommand(),
    new \Doctrine\ORM\Tools\Console\Command\ConvertDoctrine1SchemaCommand(),
    new \Doctrine\ORM\Tools\Console\Command\GenerateRepositoriesCommand(),
    new \Doctrine\ORM\Tools\Console\Command\GenerateEntitiesCommand(),
    new \Doctrine\ORM\Tools\Console\Command\GenerateProxiesCommand(),
    new \Doctrine\ORM\Tools\Console\Command\ConvertMappingCommand(),
    new \Doctrine\ORM\Tools\Console\Command\RunDqlCommand(),
    new \Doctrine\ORM\Tools\Console\Command\ValidateSchemaCommand(),

));
$cli->run();
?>
```

Exécution en mode console

Lancer la console, et redéfinir la variable PATH pour permettre l'accès au programme php en ligne de commande :

```
SET PATH=%PATH%;c:\xampp\php
echo %PATH%
```

Aller dans le dossier application du site, et exécuter :

```
php doctrine-cli.php
```

La sortie écran devrait produire le résultat suivant :

```
Doctrine Command Line Interface (CodeIgniter integration by Joel Verhagen) version
2.0.5

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v Increase verbosity of messages.
  --version        -V Display this program version.
  --ansi           -a Force ANSI output.
  --no-interaction -n Do not ask any interactive question.

Available commands:
  help                          Displays help for a command (?)
  list                          Lists commands
dbal
  :import                       Import SQL file(s) directly to Database.
  :run-sql                      Executes arbitrary SQL directly from the command
line.
orm
  :convert-d1-schema            Converts Doctrine 1.X schema into a Doctrine 2.X
schema.
  :convert-mapping              Convert mapping information between supported
formats.
  :ensure-production-settings   Verify that Doctrine is properly configured for a
production environment.
  :generate-entities            Generate entity classes and method stubs from your
mapping information.
  :generate-proxies             Generates proxy classes for entity classes.
  :generate-repositories        Generate repository classes from your mapping
information.
  :run-dql                      Executes arbitrary DQL directly from the command
line.
  :validate-schema              Validate that the mapping files.
orm:clear-cache
  :metadata                     Clear all metadata cache of the various cache
drivers.
  :query                        Clear all query cache of the various cache drivers.
  :result                       Clear result cache of the various cache drivers.
orm:schema-tool
  :create                       Processes the schema and either create it directly
on EntityManager Storage Connection or generate the SQL output.
  :drop                         Drop the complete database schema of EntityManager
Storage Connection or generate the corresponding SQL output.
  :update                       Processes the schema and either update the database
schema of EntityManager Storage Connection or generate the SQL output.
```

Génération des metadonnées de mapping

Nous allons générer les metadonnées de mapping à partir de la base de données existante, au format YAML :

- Modifier le driver Doctrine dans le fichier Doctrine.php de application/libraries :

```
<?php
class Doctrine
{
    // the Doctrine entity manager
    public $em = null;

    public function __construct()
    {
        // include our CodeIgniter application's database configuration
        require APPPATH.'config/database.php';
        // include Doctrine's fancy ClassLoader class
        require_once APPPATH.'libraries/Doctrine/Common/ClassLoader.php';

        // load the Doctrine classes
        $doctrineClassLoader = new \Doctrine\Common\ClassLoader('Doctrine',
APPPATH.'libraries');
        $doctrineClassLoader->register();
        // load Symfony2 helpers
        // Don't be alarmed, this is necessary for YAML mapping files
        $symfonyClassLoader = new \Doctrine\Common\ClassLoader('Symfony',
APPPATH.'libraries/Doctrine');
        $symfonyClassLoader->register();

        // load the entities
        $entityClassLoader = new \Doctrine\Common\ClassLoader('Entities',
APPPATH.'models');
        $entityClassLoader->register();

        // load the proxy entities
        $proxyClassLoader = new \Doctrine\Common\ClassLoader('Proxies',
APPPATH.'models');
        $proxyClassLoader->register();

        // set up the configuration
        $config = new \Doctrine\ORM\Configuration;
        if(ENVIRONMENT == 'development')
            // set up simple array caching for development mode
            $cache = new \Doctrine\Common\Cache\ArrayCache;
        else
            // set up caching with APC for production mode
            $cache = new \Doctrine\Common\Cache\ApcCache;
        $config->setMetadataCacheImpl($cache);
        $config->setQueryCacheImpl($cache);

        // set up proxy configuration
        $config->setProxyDir(APPPATH.'models/Proxies');
```

```
        $config->setProxyNamespace('Proxies');
        // auto-generate proxy classes if we are in development mode
        $config->setAutoGenerateProxyClasses(ENVIRONMENT == 'development');

        // set up annotation driver
        $yamlDriver = new
\Doctrine\ORM\Mapping\Driver\YamlDriver(APPPATH.'models/Mappings');
        $config->setMetadataDriverImpl($yamlDriver);
        //$driverImpl = $config->newDefaultAnnotationDriver(APPPATH.'models');
        //$config->setMetadataDriverImpl($driverImpl);

        // Database connection information
        $connectionOptions = array(
            'driver' => 'pdo_mysql',
            'user' => $db['default']['username'],
            'password' => $db['default']['password'],
            'host' => $db['default']['hostname'],
            'dbname' => $db['default']['database']
        );
        // create the EntityManager
        $em = \Doctrine\ORM\EntityManager::create($connectionOptions, $config);
        // store it as a member, for use in our CodeIgniter controllers.
        $this->em = $em;
    }
}
?>
```

- créer le dossier **Mappings** dans le dossier **application/models**
- Exécuter en mode console, dans le dossier application, la commande de génération des métadonnées en YAML :

```
php doctrine-cli.php orm:convert-mapping --from-database yml models/Mappings
```

- L'application doit retourner quelque chose de ce genre :

```
Processing entity "Categories"
Processing entity "Droit"
Processing entity "Projet"
Processing entity "Utilisateurs"

Exporting "yml" mapping information to "C:\xampp\htdocs\doctrine_CI\application\
models\Mappings"
```

Les fichier yml associés à chaque table de la BDD doivent être présents dans le dossier application/models/Mappings :

```
Utilisateurs:
  type: entity
  table: utilisateurs
  fields:
```

```
    id:
      id: true
      type: integer
      unsigned: false
      nullable: false
      generator:
        strategy: IDENTITY
    prenom:
      type: string
      length: 50
      fixed: false
      nullable: true
    dateinscription:
      type: date
      nullable: true
      column: dateInscription
    age:
      type: integer
      unsigned: false
      nullable: true
    nom:
      type: string
      length: 50
      fixed: false
      nullable: true
    adulte:
      type: boolean
      nullable: false
  manyToMany:
    iddroit:
      targetEntity: Droit
      cascade: {  }
      mappedBy: null
      inversedBy: idutilisateur
      joinTable:
        name: utilisateur_droit
        joinColumns:
          -
            name: idUtilisateur
            referencedColumnName: id
        inverseJoinColumns:
          -
            name: idDroit
            referencedColumnName: id
      orderBy: null
  oneToOne:
    categorie:
      targetEntity: Categories
      cascade: {  }
      mappedBy: null
      inversedBy: null
      joinColumns:
        categorie_id:
          referencedColumnName: id
      orphanRemoval: false
  lifecycleCallbacks: {  }
```

Génération des classes

Nous allons maintenant générer les classes à partir des données de mapping : Avant la génération, vérifier les données, et procédez à d'éventuelles corrections.

- en mode console, exécuter l'instruction suivante :

```
php doctrine-cli.php orm:generate-entities --generate-annotations=true models
```

L'exécution devrait retourner un résultat semblable à celui-ci :

```
Processing entity "Categories"
Processing entity "Droit"
Processing entity "Projet"
Processing entity "Utilisateurs"

Entity classes generated to "C:\xampp\htdocs\doctrine_CI\application\models"
```

Les classes sont générées dans application/models :

```
<?php



use Doctrine\ORM\Mapping as ORM;

/**
 * Utilisateurs
 */
class Utilisateurs
{
    /**
     * @var integer $id
     */
    private $id;

    /**
     * @var string $prenom
     */
    private $prenom;

    /**
     * @var date $dateinscription
     */
    private $dateinscription;

    /**
     * @var integer $age
```

```
     */
    private $age;

    /**
     * @var string $nom
     */
    private $nom;

    /**
     * @var boolean $adulte
     */
    private $adulte;

    /**
     * @var Categories
     */
    private $categorie;

    /**
     * @var \Doctrine\Common\Collections\ArrayCollection
     */
    private $iddroit;

    public function __construct()
    {
        $this->iddroit = new \Doctrine\Common\Collections\ArrayCollection();
    }
    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set prenom
     *
     * @param string $prenom
     * @return Utilisateurs
     */
    public function setPrenom($prenom)
    {
        $this->prenom = $prenom;
        return $this;
    }

    /**
     * Get prenom
     *
     * @return string
     */
    public function getPrenom()
    {
```

```
        return $this->prenom;
    }

    /**
     * Set dateinscription
     *
     * @param date $dateinscription
     * @return Utilisateurs
     */
    public function setDateinscription($dateinscription)
    {
        $this->dateinscription = $dateinscription;
        return $this;
    }

    /**
     * Get dateinscription
     *
     * @return date
     */
    public function getDateinscription()
    {
        return $this->dateinscription;
    }

    /**
     * Set age
     *
     * @param integer $age
     * @return Utilisateurs
     */
    public function setAge($age)
    {
        $this->age = $age;
        return $this;
    }

    /**
     * Get age
     *
     * @return integer
     */
    public function getAge()
    {
        return $this->age;
    }

    /**
     * Set nom
     *
     * @param string $nom
     * @return Utilisateurs
     */
    public function setNom($nom)
    {
        $this->nom = $nom;
```

```
        return $this;
    }

    /**
     * Get nom
     *
     * @return string
     */
    public function getNom()
    {
        return $this->nom;
    }

    /**
     * Set adulte
     *
     * @param boolean $adulte
     * @return Utilisateurs
     */
    public function setAdulte($adulte)
    {
        $this->adulte = $adulte;
        return $this;
    }

    /**
     * Get adulte
     *
     * @return boolean
     */
    public function getAdulte()
    {
        return $this->adulte;
    }

    /**
     * Set categorie
     *
     * @param Categories $categorie
     * @return Utilisateurs
     */
    public function setCategorie(\Categories $categorie = null)
    {
        $this->categorie = $categorie;
        return $this;
    }

    /**
     * Get categorie
     *
     * @return Categories
     */
    public function getCategorie()
    {
        return $this->categorie;
    }
```

```
    /**
     * Add iddroit
     *
     * @param Droit $iddroit
     * @return Utilisateurs
     */
    public function addDroit(\Droit $iddroit)
    {
        $this->iddroit[] = $iddroit;
        return $this;
    }

    /**
     * Get iddroit
     *
     * @return Doctrine\Common\Collections\Collection
     */
    public function getIddroit()
    {
        return $this->iddroit;
    }
}
```

- Après génération, remettez le driver sur la position Annotations dans le fichier Doctrine.php :

```
$driverImpl = $config->newDefaultAnnotationDriver(APPPATH.'models');
$config->setMetadataDriverImpl($driverImpl);
```

Sérialisation, session php

Pour permettre la sérialisation d'instances de models en session, il est nécessaire de respecter certaines étapes :

Au niveau des models

Sur-définir éventuellement la méthode magique **_sleep** des classes, pour définir les membres à sérialiser :

```
    public function __sleep(){
        return
array('id','login','password','nom','prenom','mail','monde','groupe');
    }
```

S'il faut également prévoir la sérialisation des objets dépendants, préciser CASCADE="ALL" et FETCH="EAGER" sur les annotations concernées :

Exemple : sérialisation du groupe de l'utilisateur :

```
    /**
     * @var \Groupe
     *
     * @ManyToOne(targetEntity="Groupe",cascade={"all"}, fetch="EAGER")
     * @JoinColumns({
     *   @JoinColumn(name="groupe_id", referencedColumnName="id")
     * })
     */
    private $groupe;
```

Sauvegarde en session

Utilisation de **detach** :

```
    public function save(){
        $user=DAO\getOne("Utilisateur", 1);
        $this->doctrine->em->detach($user);
        $this->session->set_userdata("user",$user);
        echo $user->getNom()." enregistré";
    }
```

restauration depuis la session

Utilisation de **merge** :

```
    public function load(){
        var_dump($this->session->all_userdata());
        $user=$this->session->userdata("user");
        $user=$this->doctrine->em->merge($user);
        var_dump($user);
    }
```