

Manipuler ses entités avec Doctrine

Table des matières

I. Création d'une entité et de son repository	3
A. Prérequis avant la création d'une entité	3
B. Création d'une entité User	4
C. Le Repository	4
II. Exercice : Quiz	6
III. Manipulation des entités	7
A. CRUD	7
B. Rôle du repository dans la manipulation des entités.....	9
1. EntityManager	9
2. QueryBuilder.....	10
3. Requête SQL.....	10
IV. Exercice : Quiz	11
V. Essentiel	12
VI. Auto-évaluation	13
A. Exercice	13
B. Test.....	13
Solutions des exercices	14

I. Création d'une entité et de son repository

Durée : 1 h

Prérequis : notions de PHP et notions de POO

Contexte

Le framework de Symfony s'appuie sur les entités et Doctrine pour manipuler la base de données. Une entité étant un objet représenté sous forme de classe va permettre à Doctrine, qui est un ORM (*Object-Relational Mapping*), utilisé par défaut avec Symfony, d'être transformé en table dans une base de données relationnelle de type MySQL, PostgreSQL, SQLite, etc.

Chaque entité a ses propriétés qui représentent les champs des tables de la base de données que l'on appelle aussi parfois « *propriété* ». Une entité a ses méthodes et ses propriétés ainsi qu'un Repository qui permettra d'effectuer des requêtes vers la base de données.

Un des gros avantages d'un ORM comme Doctrine est qu'il n'est pas nécessaire de connaître le SQL pour utiliser la base de données. Doctrine permet d'insérer, de récupérer, de modifier ou de supprimer les entités et, par extension, les tables et champs de la base de données. Il est possible aussi de créer, de récupérer, de modifier et supprimer des objets issus des entités, c'est-à-dire les données qui constituent notre BDD (Base De Données).

Alors des questions se posent : comment manipule-t-on les entités avec Doctrine ? Quels sont les outils mis à notre disposition ? Peut-on, malgré tout, faire des requêtes un peu plus poussées ?

A. Prérequis avant la création d'une entité

Avant tout, assurez-vous que votre application peut accéder à votre service local de base de données, comme MySQL par exemple. Pour cela, dans votre fichier .env qui se trouve à la racine du projet, décommentez la ligne qui contient la variable d'environnement DATABASE_URL avec le service de base de données que vous utilisez et paramétrez-le en suivant l'exemple ci-dessous :

Exemple

```
DATABASE_URL=« db_service://db_user:db_password@127.0.0.1:db_port/db_name?serverVersion=db_version& charset=utf8 »
```

- db_service : exemple postgresql ou mysql.
- db_user : est à remplacer par votre nom d'utilisateur de la base de données.
- db_password : est à remplacer par le mot de passe pour accéder à la base de données.
- db_name : est à remplacer par le nom que vous souhaitez utiliser pour la base de données.
- db_port : 5 432 est le port par défaut de base postgresql et 3 306 de mysql. Changez le port si le vôtre est différent.
- db_version : doit contenir la version du service BDD → mysql -version par exemple que votre ordinateur utilise.

Vérifiez bien que Doctrine et MakerBundle sont installés dans votre fichier composer.json, sinon, installez-les et créez votre base de données avec les lignes de commandes suivantes :

```
1 $ composer require symfony/orm-pack
2 $ composer require --dev symfony/maker-bundle
3 $ php bin/console doctrine:database:create
```

B. Création d'une entité User

Pour pleinement utiliser la puissance de Symfony et Doctrine, avant même de créer l'entité User, nous allons installer un pack de sécurité qui va, entre autres, configurer le fichier **security.yaml**. Ce dernier se trouvera dans le dossier packages, lui-même dans le dossier config à la racine du projet « **config/packages/security.yaml** ». Ce fichier va permettre à Doctrine de charger l'entité User et utiliser la bonne propriété qui servira d'identification, de connaître le hachage choisi pour le mot de passe et retenir ce qui nous intéresse.

Puis, commençons à créer l'entité User. Tapez les lignes de commande suivantes dans le terminal situé dans votre projet :

```
1 $ composer require symfony/security-bundle
2 $ php bin/console make:user
```

« The name of the security user class (e.g. User) [User]: »

Le terminal vous demande maintenant de choisir un nom pour la classe sécurité user et vous propose par défaut User (vous pouvez donc changer ce nom qui est une convention qui correspond à la plupart des situations). Validez !

« Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]: »

Puis, il vous demande si vous voulez stocker les données utilisateur dans la base de données par Doctrine. Validez !

« Enter a property name that will be the unique « display » name for the user (e.g. email, username, uuid) [email]: »

L'invite de commande propose maintenant d'ajouter une propriété qui sera unique, c'est-à-dire qui servira d'identifiant. L'email qui est proposé est excellent, alors validez !

« Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed by some other system (e.g. a single sign-on server). »

Le terminal vous pose la question si vous allez utiliser un mot de passe qu'il faudra crypter (hasher). Validez !

Félicitations ! Vous venez de créer votre première entité qui est très complexe mais avec une incroyable facilité. Notez que l'entité User est situé au « **src/Entité/User.php** » mais que repository, dont nous reparlerons plus tard, a aussi été crée au « **src/Repository/UserRepository.php** » et que le fichier **security.yaml** a été mis en jour.

Maintenant, pour que l'entité corresponde à notre base de données, il va falloir construire une migration, c'est-à-dire créer une classe qui décrit les changements nécessaires et mettre à jour le schéma de la base de données. Puis, on doit mettre à jour la base de données à partir de ce nouveau schéma. Tapez les lignes de commandes suivantes :

```
1 $ php bin/console make:migration
2 $ php bin/console doctrine:migrations:migrate
```

Validez lorsque l'invite vous demande si vous êtes sûr de faire ces changements.

C. Le Repository

Définition Qu'est-ce qu'un repository avec Doctrine ?

Un repository est une classe dont la responsabilité est de faire des requêtes vers la base de données. Ainsi, grâce à **EntityManager**, les entités pourront être manipulées et grâce à **QueryBuilder**, il sera possible de faire des requêtes personnalisées. Nous reviendrons sur ces deux classes dans la seconde partie de ce cours.

Revenons sur le fichier **UserRepository** qui a été créé, qui se trouve dans le dossier « **/src/Repository/UserRepository.php** ».

Voici le fichier créé :

```
1 <?php
2
3 namespace App\Repository;
4
5 use App\Entity\User;
6 use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
```

```

7 use Doctrine\Persistence\ManagerRegistry;
8 use Symfony\Component\Security\Core\Exception\UnsupportedUserException;
9 use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
10 use Symfony\Component\Security\Core\User\PasswordUpgraderInterface;
11
12 /**
13  * @extends ServiceEntityRepository<User>
14  *
15  * @method User|null find($id, $lockMode = null, $lockVersion = null)
16  * @method User|null findOneBy(array $criteria, array $orderBy = null)
17  * @method User[]    findAll()
18  * @method User[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset =
19  * null)
20 */
21 class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
22 {
23     public function __construct(ManagerRegistry $registry)
24     {
25         parent::__construct($registry, User::class);
26     }
27
28     public function save(User $entity, bool $flush = false): void
29     {
30         $this->getEntityManager()->persist($entity);
31
32         if ($flush) {
33             $this->getEntityManager()->flush();
34         }
35     }
36
37     public function remove(User $entity, bool $flush = false): void
38     {
39         $this->getEntityManager()->remove($entity);
40
41         if ($flush) {
42             $this->getEntityManager()->flush();
43         }
44     }
45
46     /**
47      * Used to upgrade (rehash) the user's password automatically over time.
48      */
49     public function upgradePassword(PasswordAuthenticatedUserInterface $user, string
50     $newHashedPassword): void
51     {
52         if (!$user instanceof User) {
53             throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.',
54             \get_class($user)));
55         }
56
57         $user->setPassword($newHashedPassword);
58
59         $this->save($user, true);
60     }
61 }

```

Dans le code ci-dessus, avant la classe UserRepository, il y a des annotations qui s'appellent **ServiceEntityRepository** qui utilisent quatre méthodes :

- find
- findOneBy
- findAll
- findBy

Grâce à elles, il sera possible d'effectuer les différentes recherches proposées.

Dans la classe nous retrouvons les méthodes :

- save() : qui permet de sauvegarder un objet de cette entité
- remove() : qui permet de supprimer un objet de cette entité
- upgradePassword() : qui permet de modifier le mot de passe d'un utilisateur

Exercice : Quiz

[solution n°1 p.15]

Question 1

Avec Symfony, il est nécessaire de toujours créer un repository associé à une entité.

- ☐ Vrai
- ☐ Faux

Question 2

L'Entité User est une entité un peu particulière pour Symfony.

- ☐ Vrai
- ☐ Faux

Question 3

Quelle ligne de commande permet de mettre à jour la base de données ?

- ☐ \$ php bin/console make:migration
- ☐ \$ php bin/console doctrine:database:create
- ☐ \$ php bin/console doctrine:migrations:migrate

Question 4

Un repository permet de récupérer le contenu de notre entité depuis la base de données.

- ☐ Vrai
- ☐ Faux

Question 5

Quel fichier faut-il configurer pour accéder à la base de données ?

- ☐ .env
- ☐ .services
- ☐ .repository

III. Manipulation des entités

A. CRUD

Pour continuer, il va falloir maintenant créer un contrôleur. En MVC (Modèle-Vue-Contrôleur) c'est le contrôleur, le moteur de l'application, qui doit être utilisé pour modifier les entités.

Nous pouvons créer un CRUD (*Create, Read, Update, Delete*) pour l'entité User simplement en tapant la commande suivante :

```
1 $ php bin/console make:crud User
```

« Choose a name for your controller class (e.g. UserController) [UserController]: »

Le terminal vous demande de choisir un nom pour le contrôleur et vous propose par défaut UserController. Validez !

« Do you want to generate tests for the controller?. [Experimental] (yes/no) [no]: »

Maintenant, le terminal vous demande si vous souhaitez générer des tests. La réponse par défaut est non, que vous validez.

Voici les fichiers que Symfony vient de créer :

« created: src/Controller/UserController.php

created: src/Form/ UserType.php

created: templates/base.html.twig

created: templates/user/_delete_form.html.twig

created: templates/user/_form.html.twig

created: templates/user/edit.html.twig

created: templates/user/index.html.twig

created: templates/user/new.html.twig

created: templates/user/show.html.twig »

Un fichier **form** qui contient un formulaire pour créer un nouvel utilisateur et six fichiers **twig** dont un **base.html.twig** qui sert de gabarit. Il est aussi inséré grâce à **extends** à tous les autres fichiers twig. Chaque fichier twig correspond à une vue que peut appeler le contrôleur. Et enfin, le fichier contrôleur qui nous intéresse particulièrement que nous allons détailler. Voici ce que vous devriez avoir :

```
1 <?php
2
3 namespace App\Controller;
4
5 use App\Entity\User;
6 use App\Form\UserType;
7 use App\Repository\UserRepository;
8 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
9 use Symfony\Component\HttpFoundation\Request;
10 use Symfony\Component\HttpFoundation\Response;
11 use Symfony\Component\Routing\Annotation\Route;
12
13 #[Route('/user')]
14 class UserController extends AbstractController
15 {
16     #[Route('/', name: 'app_user_index', methods: ['GET'])]
17     public function index(UserRepository $userRepository): Response
18     {
19         return $this->render('user/index.html.twig', [
20             'users' => $userRepository->findAll(),
21         ]);
```

```

22     }
23
24     #[Route('/new', name: 'app_user_new', methods: ['GET', 'POST'])]
25     public function new(Request $request, UserRepository $userRepository): Response
26     {
27         $user = new User();
28         $form = $this->createForm(UserType::class, $user);
29         $form->handleRequest($request);
30
31         if ($form->isSubmitted() && $form->isValid()) {
32             $userRepository->save($user, true);
33
34             return $this->redirectToRoute('app_user_index', [], Response::HTTP_SEE_OTHER);
35         }
36
37         return $this->renderForm('user/new.html.twig', [
38             'user' => $user,
39             'form' => $form,
40         ]);
41     }
42
43     #[Route('/{id}', name: 'app_user_show', methods: ['GET'])]
44     public function show(User $user): Response
45     {
46         return $this->render('user/show.html.twig', [
47             'user' => $user,
48         ]);
49     }
50
51     #[Route('/{id}/edit', name: 'app_user_edit', methods: ['GET', 'POST'])]
52     public function edit(Request $request, User $user, UserRepository $userRepository):
53     Response
54     {
55         $form = $this->createForm(UserType::class, $user);
56         $form->handleRequest($request);
57
58         if ($form->isSubmitted() && $form->isValid()) {
59             $userRepository->save($user, true);
60
61             return $this->redirectToRoute('app_user_index', [], Response::HTTP_SEE_OTHER);
62         }
63
64         return $this->renderForm('user/edit.html.twig', [
65             'user' => $user,
66             'form' => $form,
67         ]);
68     }
69
70     #[Route('/{id}', name: 'app_user_delete', methods: ['POST'])]
71     public function delete(Request $request, User $user, UserRepository $userRepository):
72     Response
73     {
74         if ($this->isCsrfTokenValid('delete.'.$user->getId(), $request->request->get('_token')))
75         {
76             $userRepository->remove($user, true);
77
78             return $this->redirectToRoute('app_user_index', [], Response::HTTP_SEE_OTHER);
79         }
80     }

```



```
78 }
```

Un CRUD permet d'avoir les méthodes essentielles pour manipuler nos entités. Ici, nous avons cinq méthodes :

- `index()` : qui retourne la liste des utilisateurs joint à la Vue sur **index.html.twig**.
- `new()` : qui retourne le formulaire User joint à la Vue **new.html.twig**. Après la validation du formulaire, il ajoutera le nouvel objet User en base et redirigera vers la liste des users **1app_user_index**.
- `show()` : qui retourne dans la Vue **show.html.twig** l'utilisateur demandé.
- `edit()` : qui retourne le formulaire pour modifier un utilisateur et le modifie après validation, puis le renvoie vers la liste des utilisateurs.
- `delete()` : qui, si le token est valide, supprime l'utilisateur demandé et renvoie vers la liste des utilisateurs.

B. Rôle du repository dans la manipulation des entités

1. EntityManager

Vous avez peut-être noté que **UserRepository** est appelé dans presque toutes les méthodes de **UserController**. C'est parce que, comme nous avons commencé à l'expliquer au début de ce cours, le repository a pour rôle de manipuler les entités.

À chaque fois que le contrôleur en a besoin, UserRepository est injecté dans la méthode. Puis, la méthode du contrôleur l'utilise en appelant les méthodes du repository dont il a besoin.

Par exemple :

- `index()` : retourne un tableau d'utilisateurs grâce à la méthode **findAll()** de UserRepository.
- `new()` : sauvegarde le nouvel utilisateur grâce à la méthode **save()** de UserRepository.
- `edit()` : sauvegarde les modifications apportées à l'utilisateur sélectionné à la méthode **save()** de UserRepository.
- `delete()` : supprime l'utilisateur sélectionné grâce à la méthode **remove()** de UserRepository.

Doctrine (l'ORM de Symfony) a besoin d'**EntityManager** pour récupérer un objet issu d'une entité et mettre à jour la base de données. Il est possible de manipuler EntityManager directement dans le contrôleur mais dans le cas présent, c'est UserRepository qui est appelé et qui, dans le fond, utilise EntityManager. C'est une bonne pratique car le contrôleur doit être le plus simplifié possible.

Reprenons la méthode `save()` dans UserRepository pour bien comprendre :

```
1 public function save(User $entity, bool $flush = false): void
2 {
3     $this->getEntityManager()->persist($entity);
4     if ($flush) {
5         $this->getEntityManager()->flush();
6     }
7 }
```

Vous constatez que EntityManager a été utilisé avec **persist()** et **flush()**.

La méthode `persist()` va permettre de lier à Doctrine l'objet sélectionné et `flush()` va demander à Doctrine de mettre à jour les changements sur la base de données.

2. QueryBuilder

QueryBuilder est une classe qu'utilise Doctrine de par un repository pour créer des requêtes en langage PHP vers la base de données sans passer par le SQL. L'avantage c'est qu'il n'y a pas besoin de connaître le SQL et surtout que Doctrine va convertir les requêtes aussi bien pour une base de données MySQL, PostgreSQL, SQLite, etc.

Dans UserRepository, il y avait ces lignes commentées :

```
1 // /**
2 //  * @return User[] Returns an array of User objects
3 //  */
4 // public function findByExampleField($value): array
5 // {
6 //     return $this->createQueryBuilder('u')
7 //         ->andWhere('u.exampleField = :val')
8 //         ->setParameter('val', $value)
9 //         ->orderBy('u.id', 'ASC')
10 //         ->setMaxResults(10)
11 //         ->getQuery()
12 //         ->getResult()
13 //     ;
14 // }
15
16 // public function findOneBySomeField($value): ?User
17 // {
18 //     return $this->createQueryBuilder('u')
19 //         ->andWhere('u.exampleField = :val')
20 //         ->setParameter('val', $value)
21 //         ->getQuery()
22 //         ->getOneOrNullResult()
23 //     ;
24 // }
```

Il s'agit de DQL (*Doctrine Query language*). Ces exemples sont presque prêts à l'emploi. Il suffit de remplacer quelques éléments par les valeurs voulues.

Il n'y a effectivement pas besoin de connaître le SQL, mais ce langage hybride s'en inspire largement et avoir quelques notions de SQL vous aidera à mieux comprendre.

Nous vous encourageons à aller sur la documentation officielle de ce langage pour effectuer les requêtes les plus adaptées à vos besoins :

Site officiel : Doctrine Query Language¹

3. Requête SQL

Notez qu'il est tout de même possible de faire des requêtes SQL dans un repository. Le site officiel de Symfony nous donne l'exemple suivant :

```
1 class ProductRepository extends ServiceEntityRepository
2 {
3     public function findAllGreaterThanOrPrice(int $price): array
4     {
5         $conn = $this->getEntityManager()->getConnection();
6
7         $sql = '
8             SELECT * FROM product p
9             WHERE p.price > :price
```

1 <https://www.doctrine-project.org/projects/doctrine-orm/en/2.13/reference/dql-doctrine-query-language.html#doctrine-query-language>

```
10         ORDER BY p.price ASC
11     ';
12     $stmt = $conn->prepare($sql);
13     $resultSet = $stmt->executeQuery(['price' => $price]);
14
15     // returns an array of arrays (i.e. a raw data set)
16     return $resultSet->fetchAllAssociative();
17 }
18 }
```

Libre à vous si vous êtes plus à l'aise avec le SQL de faire des requêtes spécifiques de cette façon.

Les différents types de propriétés proposés et gérés par Doctrine :

- Main types
 - string
 - text
 - boolean
 - integer(or smallint or bigint)
 - float
- Relationships/Associations
 - ManyToOne
 - OneToMany
 - ManyToMany
 - OneToOne
- Array/Object Types
 - array (or simple_array)
 - json
 - object
 - binary
 - blob
- Date/Time Types
 - datetime (or datetime_immutable)
 - datetimetz (or datetimetz_immutable)
 - date (or date_immutable)
 - time (or time_immutable)
 - dateinterval
- Other Types
 - ascii_string
 - decimal
 - guid

Exercice : Quiz

[solution n°2 p.16]

Question 1

EntityManager est un service de Doctrine qui permet de manipuler les Entités.

- ☐ Vrai
- ☐ Faux

Question 2

Qu'est-ce qu'un CRUD ?

- ☐ Create Release Upgrade Delete
- ☐ Create Read Update Delete
- ☐ Create Read Update Display

Question 3

Query Builder est un service intégré aux repository.

- ☐ Vrai
- ☐ Faux

Question 4

Doctrine ne gère pas le SQLite.

- ☐ Vrai
- ☐ Faux

Question 5

Quelle propriété n'est pas compatible avec Doctrine ?

- ☐ Integer
- ☐ Double
- ☐ Float
- ☐ Toutes sont compatibles

V. Essentiel

Les entités jouent un rôle essentiel avec le framework de Symfony. En s'appuyant sur l'ORM (Object-Relational Mapping) Doctrine, une entité représente en quelque sorte une table dans la base de données et c'est lui qui va gérer celle-ci. Il n'y a donc pas besoin de se soucier de la base de données sous-jacente, pas besoin d'écrire de SQL. Cela permet une réduction de code à créer et à maintenir, donne de l'homogénéité au code, accélère le temps de développement et permet de changer de services de base de données comme, par exemple, de passer de MySQL à PostgreSQL sans reprendre le code.

Le repository est l'élément clé pour manipuler la base de données. Le repository s'appuie sur deux classes :

- EntityManagerInterface qui va vous aider à manipuler les entités,
- QueryBuilder qui va vous permettre de créer des requêtes en PHP.

Le repository contient de nombreuses méthodes qui vous permettront, à partir d'un Contrôleur d'être amené à faire un CRUD (*Create read Update Delete*) sur la base de données, c'est-à-dire de créer un nouvel objet, de le lire, seul ou en liste, de le modifier ou de le supprimer.

Rappelez-vous qu'il est aussi possible de créer une requête en SQL si vous êtes plus à l'aise, même si vous pouvez perdre quelques avantages comme la compatibilité de services de base de données.

Il est vrai qu'utiliser Doctrine contient quelques inconvénients (son installation, la maîtrise de DQL), mais cela vous permet de bénéficier pleinement des avantages d'un framework comme Symfony.

VI. Auto-évaluation

A. Exercice

Vous êtes un développeur junior inexpérimenté dans une agence web. Votre mentor vous a formé sur les connaissances de base du Framework de Symfony.

Question 1

[solution n°3 p.17]

On vous demande de travailler sur un projet de blog en Symfony 5.4 à partir d'un diagramme de classes qui contient 3 entités, User, Article et Comment. L'objectif est de mettre en place les éléments voulus avec les outils les plus simples afin que les entités puissent être manipulées. Décrivez les actions que vous allez mener.

Question 2

[solution n°4 p.17]

Expliquez le rôle du repository dans la manipulation des entités.

B. Test

Exercice 1 : Quiz

[solution n°5 p.17]

Question 1

Pour faire des requêtes avec QueryBuilder, il faut écrire en :

- ☐ HQL
- ☐ PSQL
- ☐ DQL

Question 2

Quel est l'avantage du DQL dans symfony ?

- ☐ Il est plus facile à utiliser
- ☐ Il est plus performant
- ☐ Il permet à Doctrine de retranscrire la requête pour la plupart des services de base de données

Question 3

Quelle variable parmi les suivantes est une variable d'environnement ?

- ☐ DATABASE_URL
- ☐ EntityManager
- ☐ QueryBuilder

Question 4

Pour mettre à jour la base de données il faut utiliser la méthode flush() d'EntityManager.

- ☐ Vrai
- ☐ Faux

Question 5

Quelle méthode permettant de supprimer un objet de la base de données Doctrine fournit-il lors de création du repository ?

- ☐ dump()
- ☐ remove()
- ☐ log()


Solutions des exercices

Exercice p. 6 Solution n°1**Question 1**

Avec Symfony, il est nécessaire de toujours créer un repository associé à une entité.

☒ Vrai

☐ Faux


 Le repository est un outil de Doctrine qui permet de manipuler les entités.

Question 2

L'Entité User est une entité un peu particulière pour Symfony.

☒ Vrai

☐ Faux

 User est très souvent utilisé dans des applications web. Ainsi, en parallèle avec le package security, l'entité User sera configurée par défaut pour répondre au besoin de base de chaque projet.


Question 3

Quelle ligne de commande permet de mettre à jour la base de données ?

☐ \$ php bin/console make:migration

☐ \$ php bin/console doctrine:database:create

☒ \$ php bin/console doctrine:migrations:migrate


 La réponse 1 permet de créer une classe qui mettra à jour le schéma de la base de données, la réponse 2 permet de créer une base de données et la 3 de mettre à jour la base de données en se basant sur le schéma.

Question 4

Un repository permet de récupérer le contenu de notre entité depuis la base de données.

☒ Vrai

☐ Faux

 Le repository permet la récupération de données qui sont dans une ou plusieurs bases de données accessibles par les utilisateurs.


Question 5

Quel fichier faut-il configurer pour accéder à la base de données ?

☒ .env

☐ .services


☐ .repository

 Le fichier .env contient la variable d'environnement DATABASE_URL qui permet de configurer la connexion à la base de données.

Exercice p. 11 Solution n°2


Question 1

EntityManager est un service de Doctrine qui permet de manipuler les Entités.

- ☒ Vrai
- ☐ Faux
-  EntityManagerInterface possède de nombreuses fonctions qui permettent la manipulation des entités.


Question 2

Qu'est-ce qu'un CRUD ?

- ☐ Create Release Upgrade Delete
- ☒ Create Read Update Delete
- ☐ Create Read Update Display
-  En français, créer, lire, modifier, supprimer.


Question 3

Query Builder est un service intégré aux repository.

- ☒ Vrai
- ☐ Faux
-  QueryBuilder est une classe qui permet de créer de requête qu'interprètera Doctrine avant de les passer à la base de données.


Question 4

Doctrine ne gère pas le SQLite.

- ☐ Vrai
- ☒ Faux
-  Doctrine est compatible avec de nombreux services de base de données dont Sqlite.

Question 5

Quelle propriété n'est pas compatible avec Doctrine ?

- ☐ Integer
- ☒ Double
- ☐ Float
- ☐ Toutes sont compatibles
-  Double représente des nombres à virgule dans plusieurs langages de programmation, mais il n'est pas pris en charge par Doctrine.

p. 13 Solution n°3

Après avoir créé l'application Web en Symfony 5.4, vous allez connecter celle-ci avec la base de données locale en configurant la variable d'environnement `DATABASE_URL` qui se trouve dans le fichier `.env` à la racine du projet.

Puis, vous lancerez ces 3 lignes de commandes :

- `$ composer require symfony/orm-pack`
- `$ composer require --dev symfony/maker-bundle`
- `$ php bin/console doctrine:database:create`

Cela vous permet de construire la base de données et avoir les outils nécessaires pour cela. Pour chaque entité, vous suivez la procédure de la commande suivante « `$ php bin/console make:entity` », en tâchant d'attribuer les propriétés mentionnées dans le diagramme de classe avec types et relations voulus.

Vous mettez à jour la base de données avec les commandes suivantes :

- `$ php bin/console make:migration`
- `$ php bin/console doctrine:migrations:migrate`

Ensuite, vous mettez en place un CRUD avec « `$ php bin/console make:crud entity` » pour chaque entité et votre application devient en back-end fonctionnelle.


p. 13 Solution n°4

Lors de la création d'une entité, il est nécessaire de posséder un repository pour la manipulation de celle-ci. De base, avec une installation à partir de MakerBundle, le repository possédera des méthodes de recherches telles que `find()`, `findAll()`, `findBy()` et `findOneBy()` ainsi qu'une méthode `save()` et `remove()` pour sauvegarder ou supprimer une entité. Mais il est possible d'ajouter d'autres méthodes voir même d'utiliser QueryBuilder pour créer une méthode de recherche particulière.

Pour utiliser ces méthodes il faudra les appeler à partir du contrôleur, en veillant que le repository soit injecté dans la méthode et puis en l'utilisant par exemple avec une ligne de code comme celle-ci : `$userRepository->remove($user, true);`

Exercice p. 13 Solution n°5**Question 1**

Pour faire des requêtes avec QueryBuilder, il faut écrire en :

- ☐ HQL
- ☐ PSQL
- ☒ DQL
-  DQL signifie Doctrine Query Language.

Question 2

Quel est l'avantage du DQL dans symfony ?

- ☐ Il est plus facile à utiliser
- ☐ Il est plus performant
- ☒ Il permet à Doctrine de retranscrire la requête pour la plupart des services de base de données

Q Même si les services de base de données comme MySQL ou PostgreSQL sont basés sur le SQL, ils ont quelques différences de syntaxe, le DQL permet de changer de service sans changer de code.

Question 3

Quelle variable parmi les suivantes est une variable d'environnement ?

- ☒ DATABASE_URL
- ☐ EntityManager
- ☐ QueryBuilder

Q EntityManager et QueryBuilder sont des services de Doctrine.

Question 4

Pour mettre à jour la base de données il faut utiliser la méthode flush() d'EntityManager.

- ☒ Vrai
- ☐ Faux

Q Flush() va permettre à Doctrine de mettre à jour les changements effectués sur la base de données.

Question 5

Quelle méthode permettant de supprimer un objet de la base de données Doctrine fournit-il lors de création du repository ?

- ☐ dump()
- ☒ remove()
- ☐ log()

Q remove() est une méthode qui va permettre de supprimer un objet de la base de données.