

# Débuter avec Doctrine\*

Version 3.1.2

28 avril 2024

Ce tutoriel couvre la prise en main de l'ORM Doctrine. Après l'avoir parcouru, vous devez savoir :

- Comment installer et configurer Doctrine en le connectant à une base de données ;
- Mapper des objets PHP aux tables de la base de données ;
- Générer un schéma de base de données à partir d'objets PHP ;
- Utiliser l'**EntityManager** pour insérer, mettre à jour, supprimer et rechercher des objets dans la base de données.

## 1 Pré-requis

Ce tutoriel est conçu pour des débutants qui n'ont jamais travaillé avec Doctrine ORM auparavant. Certains pré-requis pour doivent être installés :

- PHP (dernière version stable)
- Composer Package Manager ([Install Composer](#))

Le code (une version quelque peu différente tout de même) de ce tutoriel est [disponible sur Github](#).

## 2 Qu'est-ce-que Doctrine ?

Doctrine ORM est un [mapping objet-relationnel](#) (en anglais object-relational mapping ou ORM) pour PHP qui fournit une persistance transparente pour les objets PHP. Il utilise un modèle Data Mapper, qui vise une séparation complète entre votre logique de domaine/métier et sa persistance dans un SGBDR (Système de Gestion de Base de Données Relationnelle).

L'avantage de Doctrine pour le programmeur est la capacité de se concentrer sur la logique métier orientée objet et de se soucier de la persistance uniquement comme problème secondaire. Cela ne signifie pas que la persistance est minimisée par Doctrine 2, mais nous pensons qu'il y a des avantages considérables pour la programmation orientée objet si la persistance et les entités restent séparées.

---

\*<https://www.doctrine-project.org/projects/doctrine-orm/en/current/tutorials/getting-started.html>

## 2.1 Les entités, c'est quoi ?

Les entités sont des objets PHP qui peuvent être identifiés sur de nombreuses requêtes par un identifiant unique ou une clé primaire. Ces classes n'ont pas besoin d'étendre une classe de base abstraite ou une interface.

Une entité contient des propriétés persistantes. Une propriété persistante est une variable d'instance de l'entité qui est enregistrée et récupérée dans la base de données par les capacités de mappage de données de Doctrine.

Une classe d'entité ne doit être ni finale, ni en lecture seule, bien qu'elle puisse contenir des méthodes finales ou des propriétés en lecture seule.

## 3 Un exemple de modèle : traqueur de bugs

Pour ce tutoriel, nous implémenterons le modèle de domaine « traqueur de bugs » de la documentation [Zend\\_Db\\_Table](#). En lisant leur documentation, nous pouvons extraire les exigences :

- Un Bug a une description, une date de création, un statut, un rapporteur et un ingénieur.
- Un Bug peut survenir sur différents Produits (plateformes).
- Un produit a un nom.
- Les rapporteurs de bogues et les ingénieurs sont tous deux des utilisateurs du système.
- Un utilisateur peut créer de nouveaux bugs.
- L'ingénieur désigné peut fermer un bug.
- Un utilisateur peut voir tous ses bugs signalés ou attribués.
- Les bugs peuvent être paginés dans une vue de type liste.

## 4 configuration du projet

### 4.1 Travail à faire



1. Créez un nouveau dossier vide pour ce projet, nommez le `doctrine2-tutorial`. Dans ce répertoire, créez un nouveau fichier intitulé `composer.json` avec le contenu suivant :

```
{
    "require": {
        "doctrine/orm": "^3",
        "doctrine/dbal": "^4",
        "symfony/cache": "^7"
    },
    "autoload": {
        "psr-0": {"": "src/"}
    }
}
```

2. Ensuite, placez vous dans votre dossier de travail (`doctrine2-tutorial`) et ouvrez une fenêtre PowerShell. Installez Doctrine à l'aide de l'outil de gestion des dépendances *Composer*, en exécutant la commande suivante :

```
composer install
```

Cela installe les packages Doctrine Common, Doctrine DBAL, Doctrine ORM, dans le répertoire du fournisseur (`vendor`).

3. Enfin, ajoutez les répertoires suivants :

```
doctrine2-tutorial
|-- config
|   |-- xml
|-- src
```



Dans le répertoire de votre projet (`doctrine2-tutorial`), créez un nouveau référentiel git (`git init`). Ajoutez tout (`git add --all`) puis validez avec le message « configuration du projet » (`git commit -m "configuration du projet"`).

Ensuite, par exemple sur le site [github](https://github.com), créez un dépôt distant nommé « doctrine2-tutorial » et poussez-y le référentiel local que vous venez de créer. Pour ce faire, vous pouvez suivre les consignes données par votre dépôt distant.

Enfin, n'oubliez pas de m'inviter.

## 5 Obtention d'EntityManager

L'interface publique de Doctrine se fait via **EntityManager**. Cette classe fournit des points d'accès à la gestion complète du cycle de vie de vos entités et transforme les entités depuis et vers la persistance. Vous devez le configurer et le créer pour utiliser vos entités avec Doctrine ORM. Je vais montrer les étapes de configuration, puis les discuter étape par étape.

### 5.1 Travail à faire



Dans le répertoire `doctrine2-tutorial`, créez un fichier `bootstrap.php` avec le contenu suivant :

```
<?php
// bootstrap.php
use Doctrine\DBAL\DriverManager;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\ORMSetup;

require_once "vendor/autoload.php";

// Create a simple "default" Doctrine ORM configuration for Attributes
$config = ORMSetup::createAttributeMetadataConfiguration(
    paths: array(__DIR__."/src"),
    isDevMode: true,
);
// or if you prefer XML
// $config = ORMSetup::createXMLMetadataConfiguration(
//     paths: array(__DIR__."/config/xml"),
//     isDevMode: true,
// );

// configuring the database connection
$connection = DriverManager::getConnection([
    'driver' => 'pdo_sqlite',
    'path' => __DIR__ . '/db.sqlite',
], $config);

// obtaining the entity manager
$entityManager = new EntityManager($connection, $config);
```

### Explications :

- L'instruction `require_once` configure le chargement automatique de classe pour Doctrine et ses dépendances à l'aide du chargeur automatique de Composer.
- Le deuxième bloc consiste en l'instanciation de l'objet ORM **Configuration** à l'aide de l'assistant `ORMSetup`. Cela suppose un certain nombre de valeurs par défaut dont vous n'avez pas à vous soucier pour l'instant. Notez que vous pouvez lire les détails de la configuration dans [le chapitre de référence sur la configuration](#).
- Le troisième bloc montre les options de configuration requises pour se connecter à une base de données. Dans ce tutoriel, nous utiliserons une base de données SQLite (donc basée sur des fichiers). Cependant, toutes les options de configuration pour tous les pilotes livrés sont indiquées dans [la section Configuration DBAL du manuel](#).

- Le dernier bloc montre comment `EntityManager` est obtenu à partir d'une méthode de fabrique (factory method).



Ajoutez tout, validez avec le message « Obtention d'EntityManager » et poussez.

## 6 Génération du schéma de la base de données

Doctrine dispose d'une interface de ligne de commande qui vous permet d'accéder au *SchemaTool*, un composant capable de générer un schéma de base de données relationnelle entièrement basé sur les classes d'entités définies et leurs métadonnées. Pour que cet outil fonctionne, vous devez créer un script de console exécutable comme décrit dans [le chapitre outils](#).

### 6.1 Travail à faire



1. Dans le répertoire `doctrine2-tutorial`, créez un nouveau dossier nommé `bin`. Dans ce dernier créez un nouveau fichier `doctrine` (sans extension) avec le contenu suivant :

```
#!/usr/bin/env php
<?php
// bin/doctrine

use Doctrine\ORM\Tools\Console\ConsoleRunner;
use Doctrine\ORM\Tools\Console\EntityManagerProvider\SingleManagerProvider;

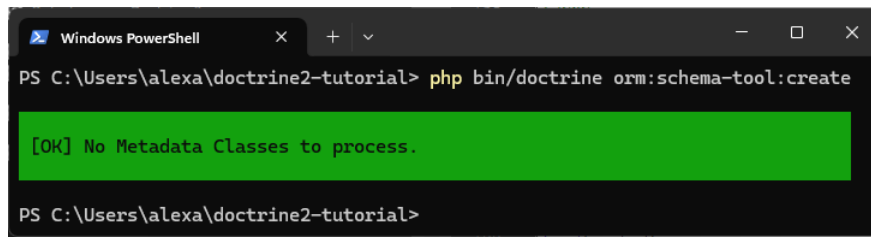
// Adjust this path to your actual bootstrap.php
require __DIR__ . '/../bootstrap.php';

ConsoleRunner::run(
    new SingleManagerProvider($entityManager)
);
```

2. Exécutez la commande suivante :

```
php bin/doctrine orm:schema-tool:create
```

Puisque nous n'avons pas encore ajouté de métadonnées d'entité dans le répertoire `src`, vous verrez un message indiquant qu'il n'y a aucune classe de métadonnées à traiter :



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php bin/doctrine orm:schema-tool:create

[OK] No Metadata Classes to process.

PS C:\Users\alexa\doctrine2-tutorial>
```

Dans la section suivante, nous allons créer une entité **Product** avec les métadonnées correspondantes et exécuter à nouveau cette commande.



Notez que lorsque vous modifiez les métadonnées de vos entités pendant le processus de développement, vous devrez mettre à jour le schéma de votre base de données pour rester synchronisé avec les métadonnées. Vous pouvez facilement recréer la base de données à l'aide des commandes suivantes :

```
php bin/doctrine orm:schema-tool:drop --force
php bin/doctrine orm:schema-tool:create
```

Ou vous pouvez utiliser la fonctionnalité de mise à jour :

```
php bin/doctrine orm:schema-tool:update --force
```

La mise à jour des bases de données utilise un algorithme de comparaison pour un schéma de base de données donné. Il s'agit d'une pierre angulaire du package **Doctrine\DBAL**, qui peut même être utilisé sans le package **Doctrine ORM**.



Ajoutez tout, validez avec le message « Génération du schéma de la base de données » et poussez.

## 7 Commençons par l'entité Product

Nous commençons par l'entité la plus simple, le produit.

### 7.1 Travail à faire



Créez un fichier `src/Product.php` pour contenir la définition de l'entité **Product** :

```

<?php
// src/Product.php

class Product
{
    private int|null $id = null;
    private string $name;

    // coder ici le getter de l'attribut $id
    // et les getter/setter de l'attribut $name
}

```



Lors de la création de classes d'entités, tous les champs doivent être **private**. Utilisez **protected** lorsque cela est strictement nécessaire et très rarement, voire jamais, **public**.

## 7.2 Ajoutons un comportement aux entités

*Cette section (théorique) peut être ommise en première lecture.*

Il existe deux options pour définir des méthodes dans les entités :

- les getters/setters pour les entités anémiques (anemic entity);
- les mutateurs et les DTO (Data Transfer Object) pour les entités riches (rich entity).

### 7.2.1 Entités anémiques : getters et setters

La méthode la plus populaire consiste à créer deux types de méthodes pour lire (getter) et mettre à jour (setter) les propriétés de l'objet.

Le champ id n'a pas de paramètre puisque, d'une manière générale, votre code ne doit pas définir cette valeur puisqu'il représente une valeur d'identifiant de base de données. (Notez que Doctrine lui-même peut toujours définir la valeur à l'aide de l'API Reflection au lieu d'une fonction de définition définie.)



Doctrine ORM n'utilise aucune des méthodes que vous avez définies : il utilise la réflexion (API Reflection) pour lire et écrire des valeurs sur vos objets, et n'appellera jamais de méthodes, pas même `__construct`.

Cette approche est principalement utilisée lorsque vous souhaitez vous concentrer sur des entités sans comportement et lorsque vous souhaitez avoir toute votre logique métier dans vos services plutôt que dans les objets eux-mêmes.

Les getters et setters sont une convention commune qui permet d'exposer chaque domaine de votre entité au monde extérieur, tout en vous permettant de maintenir une certaine sécurité de type en place.

Une telle approche est un bon choix pour le RAD (développement rapide d'applications), mais peut entraîner des problèmes plus tard, car fournir un moyen aussi simple de modifier n'importe quel champ dans votre entité signifie que l'entité elle-même ne peut pas garantir la validité de son état interne. Avoir un objet dans un état invalide est dangereux :

- Un état invalide peut entraîner des bugs dans votre logique métier.
- L'état peut être implicitement enregistré dans la base de données : tout vidage (flush) oublié peut conserver l'état cassé.
- Si elles persistent, les données corrompues seront récupérées plus tard dans votre application lors d'un nouveau chargement depuis la base de données, entraînant ainsi des bugs dans votre logique métier.
- Lorsque des bugs surviennent après la persistance de données corrompues, le dépannage deviendra beaucoup plus difficile et vous pourriez prendre conscience du bug trop tard pour le corriger de manière appropriée.



Cette méthode, bien que très courante, est inappropriée pour le Domain Driven Design (DDD) où les méthodes doivent représenter des opérations métier réelles et non de simples changements de valeur de propriété, et les invariants métier doivent être conservés à la fois dans l'état de l'application (entités dans ce cas) et dans la base de données.

Voici un exemple d'entité anémique simple :

```
<?php
class User
{
    private $username;
    private $passwordHash;
    private $bans;

    public function getUsername(): string
    {
        return $this->username;
    }

    public function setUsername(string $username): void
    {
        $this->username = $username;
    }
}
```



```

public function getPasswordHash(): string
{
    return $this->passwordHash;
}

public function setPasswordHash(string $passwordHash): void
{
    $this->passwordHash = $passwordHash;
}

public function getBans(): array
{
    return $this->bans;
}

public function addBan(Ban $ban): void
{
    $this->bans[] = $ban;
}
}

```

Dans l'exemple ci-dessus, nous évitons toute logique possible dans l'entité et nous soucions uniquement d'y insérer et de récupérer des données sans validation (sauf celle fournie par les types) ni prise en compte de l'état de l'objet.

Comme Doctrine ORM est un outil de persistance pour votre domaine, l'état d'un objet est vraiment important. C'est pourquoi nous recommandons fortement d'utiliser des entités riches.

### 7.2.2 Entités riches : mutateurs et DTO (Data Transfert Object)

Nous recommandons d'utiliser une conception d'entité riche et de nous appuyer sur des mutateurs plus complexes, et si nécessaire sur des DTO. Dans cette conception, vous ne devez pas utiliser de getters ni de setters, mais plutôt implémenter des méthodes qui représentent le comportement de votre domaine.

Par exemple, lorsqu'on a une entité Utilisateur, on pourrait prévoir le type d'optimisation suivant.

Exemple d'entité riche avec des accesseurs et des mutateurs appropriés :

```

<?php
class User
{
    private $banned;
    private $username;
    private $passwordHash;
    private $bans;
}

```

```

public function toNickname(): string
{
    return $this->username;
}

public function authenticate(string $password, callable $checkHash): bool
{
    return $checkHash($password, $this->passwordHash) && ! $this->hasActiveBans();
}

public function changePassword(string $password, callable $hash): void
{
    $this->passwordHash = $hash($password);
}

public function ban(\DateInterval $duration): void
{
    assert($duration->invert !== 1);

    $this->bans[] = new Ban($this);
}
}

```

Cet exemple n'est qu'un « bout » de code. En allant plus loin dans le tutoriel, nous modifierons cet objet avec plus de comportement et mettrons peut-être à jour certaines méthodes.

Les entités ne doivent muter leur état qu'après avoir vérifié que tous les invariants de la logique métier sont respectés. De plus, nos entités ne devraient jamais voir leur état changer sans validation. Par exemple, créer un nouvel objet `Product()` sans aucune donnée en fait un objet non valide. Les entités riches doivent représenter un comportement, pas des données, elles doivent donc être valides même après un appel `__construct()`.

Pour aider à créer de tels objets, nous pouvons nous appuyer sur des DTO, et/ou rendre nos entités toujours à jour. Cela peut être effectué avec des constructeurs statiques ou des mutateurs riches qui acceptent les DTO comme paramètres.



Le rôle du DTO (Data Transfert Object) est de maintenir l'état de l'entité et de nous aider à nous appuyer sur des objets qui représentent correctement les données utilisées pour muter l'entité. Un DTO est un objet qui transporte uniquement des données sans aucune logique. Son seul but est d'être transféré d'un service à un autre. Un DTO représente souvent des données envoyées par un client et qui doivent être validées, mais peut également être utilisé comme simple support de données dans d'autres cas.

En utilisant des DTO, si nous reprenons notre exemple utilisateur précédent, nous pourrions créer un DTO `ProfileEditingForm` qui sera un modèle simple, totalement étranger à notre base de données, qui sera renseigné via un formulaire et validé. Ensuite, nous pouvons ajouter un nouveau mutateur à notre utilisateur :

```

<?php
class User
{
    public function updateFromProfile(ProfileEditingDTO $profileFormDTO): void
    {
        // ...
    }

    public static function createFromRegistration(
        UserRegistrationDTO $registrationDTO): self
    {
        // ...
    }
}

```

Il y a plusieurs avantages à utiliser un tel modèle :

- L'état de l'entité est toujours valide. Puisqu'il n'existe aucun setter, cela signifie que nous mettons à jour uniquement les parties de l'entité qui devraient déjà être valides.
- Au lieu d'avoir des getters et des setters simples, notre entité a maintenant un comportement réel : il est beaucoup plus facile de déterminer la logique dans le domaine.
- Les DTO peuvent être réutilisés dans d'autres composants, par exemple en désérialisant des contenus mélangés, en utilisant des formulaires...
- Les constructeurs classiques et statiques peuvent être utilisés pour gérer différentes manières de créer nos objets, et ils peuvent également utiliser des DTO.
- Les entités anémiques ont tendance à isoler l'entité de la logique, alors que les entités riches permettent de mettre la logique dans l'objet lui-même, y compris la validation des données.

## 7.3 Retour à la pratique

La prochaine étape pour la persistance avec Doctrine consiste à décrire la structure de l'entité **Product** à Doctrine à l'aide d'un langage de métadonnées. Le langage des métadonnées décrit comment les entités, leurs propriétés et leurs références doivent être conservées et quelles contraintes doivent leur être appliquées.

Les métadonnées d'une entité peuvent être configurées à l'aide d'attributs directement dans la classe Entity elle-même ou dans un fichier XML externe. Ce tutoriel présente les mappages de métadonnées à l'aide d'attributs.

### 7.3.1 Travail à faire



1. Dans le fichier `src/Product.php` ajoutez les métadonnées de l'entité **Product** :

```

<?php
// src/Product.php

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'products')]
class Product
{
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private int|null $id = null;
    #[ORM\Column(type: 'string')]
    private string $name;

    // .. (other code)
}

```

### Explications :

- La ligne précédant la définition de la classe `Product` spécifie le nom de la table (`products` ici) en base de données.
- La propriété `id` est définie avec la balise `Id` et la balise `GeneratedValue`, stipule que le mécanisme de génération de clé primaire doit utiliser la stratégie de génération automatique d'identifiant natif de la plateforme de base de données (par exemple, `AUTO INCREMENT` dans le cas de `MySQL`, ou `Sequences` dans le cas de `PostgreSql` et `Oracle`).

2. Maintenant que nous avons défini notre première entité et ses métadonnées, mettons à jour le schéma de la base de données :

```
php bin/doctrine orm:schema-tool:update --force --dump-sql
```

La spécification des deux options `--force` et `--dump-sql` entraîne l'exécution des instructions DDL (Data Definition Language) puis leurs affichages à l'écran :

```

Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php bin/doctrine orm:schema-tool:update --force --dump-sql
CREATE TABLE products (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name VARCHAR(255) NOT NULL);

Updating database schema...

1 query was executed

[OK] Database schema updated successfully!

PS C:\Users\alexa\doctrine2-tutorial>

```



Ajoutez tout, validez avec le message « Commençons par l'entité produit (partie 1) » et poussez.

### 7.3.2 Travail à faire



Maintenant, nous allons créer un nouveau script `create_product.php` dans le répertoire racine du projet (`doctrine2-tutorial`) pour insérer des produits dans la base de données :

```
<?php
// create_product.php <name>
require_once "bootstrap.php";

$newProductName = $argv[1];

$product = new Product();
$product->setName($newProductName);

$entityManager->persist($product);
$entityManager->flush();

echo "Created Product with ID " . $product->getId() . "\n";
```

Appelez ce script depuis la ligne de commande pour voir comment les nouveaux produits sont créés :

```
php create_product.php ORM
php create_product.php DBAL
```



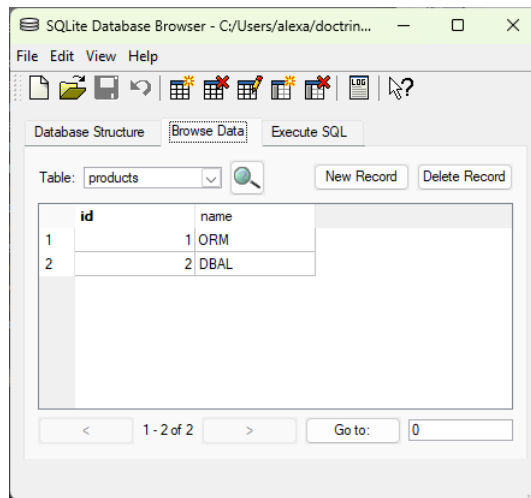
```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php create_product.php ORM
Created Product with ID 1
PS C:\Users\alexa\doctrine2-tutorial> php create_product.php DBAL
Created Product with ID 2
PS C:\Users\alexa\doctrine2-tutorial> |
```

#### Explications :

L'utilisation de la classe `Product` est un standard de POO (Programmation Orientée Objet). Voici comment le service `EntityManager` est utilisé :

- `persist()` avertit `EntityManager` qu'une nouvelle entité doit être insérée dans la base de données.
- `flush()` lance une transaction afin d'effectuer réellement l'insertion en base de données.

Vérifiez les deux insertions dans la table `products` de la base de données `db.sqlite` :



La distinction entre **persist** et **flush** permet l'agrégation de toutes les écritures de la base de données (INSERT, UPDATE, DELETE) en une seule transaction, qui est exécutée lorsque **flush()** est appelée. Grâce à cette approche, les performances d'écriture sont nettement meilleures que dans un scénario dans lequel les écritures sont effectuées sur chaque entité de manière isolée.



Ajoutez tout, validez avec le message « Commençons par l'entité produit (partie 2) » et poussez.

### 7.3.3 Travail à faire



1. Ensuite, nous allons récupérer une liste de tous les produits de la base de données. Créons un nouveau script pour cela :

```
<?php
// list_products.php
require_once "bootstrap.php";

$productRepository = $entityManager->getRepository('Product');
$products = $productRepository->findAll();

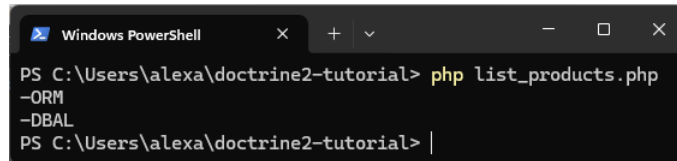
foreach ($products as $product) {
    echo sprintf("-%s\n", $product->getName());
}
```

#### Explications :

La méthode `EntityManager#getRepository()` peut créer un objet **Finder** (appelé ré-

férentiel ou repository) pour chaque type d'entité. Il est fourni par Doctrine et contient des méthodes de recherche comme `findAll()`.

Vérifiez que ce nouveau script fonctionne :



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php list_products.php
-ORM
-DBAL
PS C:\Users\alexa\doctrine2-tutorial> |
```

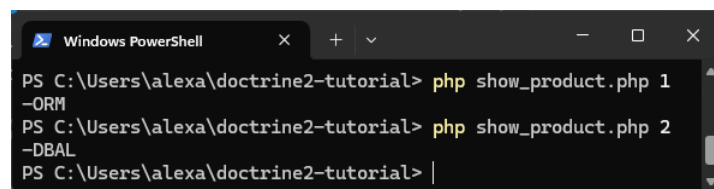
2. Continuons en créant un script pour afficher le nom d'un produit en fonction de son identifiant :

```
<?php
// show_product.php <id>
require_once "bootstrap.php";

$id = $argv[1];
$product = $entityManager->find('Product', $id);

if ($product === null) {
    echo "No product found.\n";
    exit(1);
}

echo sprintf("-%s\n", $product->getName());
```



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php show_product.php 1
-ORM
PS C:\Users\alexa\doctrine2-tutorial> php show_product.php 2
-DBAL
PS C:\Users\alexa\doctrine2-tutorial> |
```

3. Enfin, mettons à jour le nom d'un produit, en fonction de son identifiant :

```

<?php
// update_product.php <id> <new-name>
require_once "bootstrap.php";

$id = $argv[1];
$newName = $argv[2];

$product = $entityManager->find('Product', $id);

if ($product === null) {
    echo "Product $id does not exist.\n";
    exit(1);
}

$product->setName($newName);

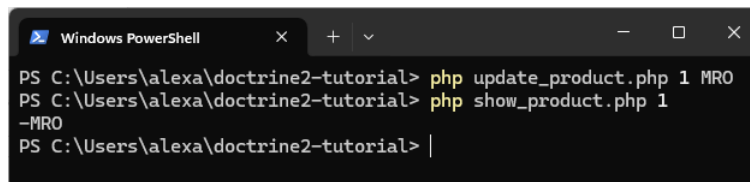
$entityManager->flush();

```

### Explications :

Ce dernier exemple illustre l'implémentation du [modèle UnitOfWork](#) : Doctrine garde une trace de toutes les entités qui ont été récupérées à partir du gestionnaire d'entités et peut détecter quand les propriétés de l'une de ces entités ont été modifiées. Par conséquent, plutôt que de devoir appeler `persist($entity)` pour chaque entité individuelle dont les propriétés ont été modifiées, un seul appel à `flush()` à la fin d'une requête suffit pour mettre à jour la base de données pour toutes les entités modifiées.

Après avoir appelé ce script sur l'un des produits existants, vous pouvez vérifier le nom du produit modifié en appelant le script `show_product.php` :



```

Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php update_product.php 1 MRO
PS C:\Users\alexa\doctrine2-tutorial> php show_product.php 1
-MRO
PS C:\Users\alexa\doctrine2-tutorial> |

```



Ajoutez tout, validez avec le message « Commençons par l'entité produit (partie 3) » et poussez.

## 8 Ajout des entités Bug et User

### 8.1 Travail à faire





Nous continuons avec l'exemple du traqueur de bug en créant les classes `Bug` et `User`. Nous les stockerons respectivement dans `src/Bug.php` et `src/User.php` :

```

<?php
// src/Bug.php

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'bugs')]
class Bug
{
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private int|null $id;

    #[ORM\Column(type: 'string')]
    private string $description;

    #[ORM\Column(type: 'datetime')]
    private DateTime $created;

    #[ORM\Column(type: 'string')]
    private string $status;

    public function getId(): int|null {
        return $this->id;
    }
    public function getDescription(): string {
        return $this->description;
    }
    public function setDescription(string $description): void {
        $this->description = $description;
    }
    public function setCreated(DateTime $created) {
        $this->created = $created;
    }
    public function getCreated(): DateTime {
        return $this->created;
    }
    public function setStatus($status): void {
        $this->status = $status;
    }
    public function getStatus(): string {
        return $this->status;
    }
}

```

```

<?php
// src/User.php

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'users')]
class User
{
    /** @var int */
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private int|null $id = null;

    /** @var string */
    #[ORM\Column(type: 'string')]
    private string $name;

    public function getId(): int|null
    {
        return $this->id;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }
}

```

Toutes les propriétés que nous avons vues jusqu'à présent sont de types simples (entier, chaîne et date/heure). Mais maintenant, nous allons ajouter des propriétés qui stockeront des objets de types d'entités spécifiques afin de modéliser les relations entre différentes entités.



Au niveau de la base de données, les relations entre entités sont représentées par des clés étrangères. Mais avec Doctrine, vous n'aurez jamais à (et ne devriez jamais) travailler directement avec les clés étrangères. Vous ne devez travailler qu'avec des objets qui représentent des clés étrangères à travers leur propre identité.

Pour chaque clé étrangère, vous avez soit une association Doctrine ManyToOne ou OneToOne. Sur les côtés inverses de ces clés étrangères, vous pouvez avoir des associations OneToMany. Évidemment, vous pouvez avoir des associations ManyToMany qui connectent deux tables entre elles via une table de jointure avec deux clés étrangères.



Ajoutez tout, validez avec le message « Ajout des entités Bug et User (partie 1) » et poussez.

## 8.2 Travail à faire



Maintenant que vous connaissez les bases des références dans Doctrine, nous pouvons étendre le modèle de domaine pour répondre aux exigences :

```
<?php
// src/Bug.php

use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;

class Bug
{
    // ... (previous code)

    private $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}
```

Avez-vous bien ajouté les deux premières lignes (`use ...`) ?

```

<?php
// src/User.php
use Doctrine\Common\Collections\ArrayCollection;

class User
{
    // ... (previous code)

    private $reportedBugs = null;
    private $assignedBugs = null;

    public function __construct()
    {
        $this->reportedBugs = new ArrayCollection();
        $this->assignedBugs = new ArrayCollection();
    }
}

```

Avez-vous pensé à la ligne `use Doctrine\Common\Collections\ArrayCollection;` ?



Ajoutez tout, validez avec le message « Ajout des entités Bug et User (partie 2) » et poussez.



Chaque fois qu'une entité est créée à partir de la base de données, une `Collection` du type `PersistentCollection` sera injectée dans votre entité au lieu d'un `ArrayCollection`. Cela aide Doctrine ORM à comprendre les changements survenus dans la collection en vue d'assurer leur persistance.

Parce que nous travaillons uniquement avec des collections pour les références, nous devons faire attention à implémenter une référence bidirectionnelle dans le modèle du domaine. Le concept de propriété ou de relation inverse est au cœur de cette notion et doit toujours être gardé à l'esprit. Les hypothèses suivantes sont faites sur les relations et doivent être suivies pour pouvoir travailler avec Doctrine ORM. Ces hypothèses ne sont pas propres à Doctrine ORM mais constituent les meilleures pratiques en matière de gestion des relations avec les bases de données et du mappage objet-relationnel :

- Dans une relation un-à-un, l'entité détenant la clé étrangère de l'entité associée sur sa propre table de base de données est **toujours** le côté propriétaire de la relation.
- Dans une relation plusieurs-à-un, le côté Plusieurs est le côté propriétaire par défaut car il détient la clé étrangère. En conséquence, le côté One est le côté inverse par défaut.

- Dans une relation plusieurs-à-un, le côté unique ne peut être le côté propriétaire que si la relation est implémentée en tant que ManyToMany avec une table de jointure, et le côté unique est limité pour autoriser uniquement les valeurs unique (contrainte UNIQUE) de base de données.
- Dans une relation plusieurs-à-plusieurs, les deux côtés peuvent être propriétaires de la relation. Cependant, dans une relation plusieurs-à-plusieurs bidirectionnelle, un seul côté est autorisé à être propriétaire.
- Les modifications apportées aux collections sont enregistrées ou mises à jour lorsque l'entité **propriétaire** de la collection est enregistrée ou mise à jour.
- L'enregistrement d'une entité du côté inverse d'une relation ne déclenche jamais d'opération de persistance pour les modifications apportées à la collection.



La cohérence des références bidirectionnelles du côté inverse d'une relation doit être gérée dans le code d'application de l'espace utilisateur. Doctrine ne peut pas mettre à jour comme par magie vos collections pour qu'elles soient cohérentes.

## 8.3 Travail à faire



1. Dans le cas des users et des bugs, nous avons des références aux bugs attribués et signalés par un utilisateur, ce qui rend cette relation bidirectionnelle. Nous devons changer le code pour assurer la cohérence de la référence bidirectionnelle :

```
<?php
// src/Bug.php
class Bug
{
    // ... (previous code)

    private User $engineer;
    private User $reporter;

    public function setEngineer(User $engineer): void
    {
        $engineer->assignedToBug($this);
        $this->engineer = $engineer;
    }

    public function setReporter(User $reporter): void
    {
        $reporter->addReportedBug($this);
        $this->reporter = $reporter;
    }

    public function getEngineer(): User
    {
        return $this->engineer;
    }

    public function getReporter(): User
    {
        return $this->reporter;
    }
}
```

```

<?php
// src/User.php
class User
{
    // ... (previous code)

    private $reportedBugs = null;
    private $assignedBugs = null;

    public function addReportedBug(Bug $bug): void
    {
        $this->reportedBugs[] = $bug;
    }

    public function assignedToBug(Bug $bug): void
    {
        $this->assignedBugs[] = $bug;
    }
}

```

### Explications :

- On a choisi de nommer les méthodes inverses au passé, ce qui devrait indiquer que l'affectation réelle a déjà eu lieu et que les méthodes ne sont utilisées que pour assurer la cohérence des références.
- Vous pouvez voir à partir de `User#addReportedBug()` et `User#assignedToBug()` que l'utilisation de cette méthode dans l'espace utilisateur seul n'ajouterait pas le Bug à la collection du côté propriétaire dans `Bug#reporter` ou `Bug#engineer`. L'utilisation de ces méthodes et l'appel de Doctrine pour la persistance ne mettraient pas à jour la représentation des collections dans la base de données.
- Seule l'utilisation de `Bug#setEngineer()` ou `Bug#setReporter()` enregistre correctement les informations de relation.
- Les propriétés `Bug#reporter` et `Bug#engineer` sont des relations Many-To-One, qui pointent vers un utilisateur. Dans un modèle relationnel normalisé, la clé étrangère est enregistrée sur la table du Bug, donc dans notre modèle de relation objet, le Bug est du côté propriétaire de la relation. Vous devez toujours vous assurer que les cas d'utilisation de votre modèle de domaine doivent déterminer quel côté est inverse ou propriétaire dans votre mappage de doctrine. Dans notre exemple, chaque fois qu'un nouveau bug est enregistré ou qu'un ingénieur est affecté au bug, nous ne voulons pas mettre à jour l'utilisateur pour conserver la référence, mais le bug.
- Les bugs font référence aux produits par une relation ManyToMany unidirectionnelle dans la base de données qui pointe des bugs vers les produits.



2. Les bugs font référence aux produits par une relation **ManyToMany** unidirectionnelle dans la base de données qui pointe des bugs vers les produits :

```
<?php
// src/Bug.php
class Bug
{
    // ... (previous code)

    private $products;

    public function assignToProduct(Product $product): void
    {
        $this->products[] = $product;
    }

    public function getProducts()
    {
        return $this->products;
    }
}
```



Ajoutez tout, validez avec le message « Ajout des entités Bug et User (partie 3) » et poussez.

## 8.4 Travail à faire

Nous avons maintenant terminé avec le modèle de domaine compte tenu des exigences.



1. Ajoutons des mappages de métadonnées pour l'entité **Bug**, comme nous l'avons fait pour l'entité **Product** auparavant :

```

<?php
// src/Bug.php

use DateTime;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'bugs')]
class Bug
{
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private int|null $id = null;

    #[ORM\Column(type: 'string')]
    private string $description;

    #[ORM\Column(type: 'datetime')]
    private DateTime $created;

    #[ORM\Column(type: 'string')]
    private string $status;

    #[ORM\ManyToOne(targetEntity: User::class, inversedBy: 'assignedBugs')]
    private User|null $engineer = null;

    #[ORM\ManyToOne(targetEntity: User::class, inversedBy: 'reportedBugs')]
    private User|null $reporter;

    #[ORM\ManyToMany(targetEntity: Product::class)]
    private $products;

    // ... (other code)
}

```

### Explications :

- Nous retrouvons ici les définitions de l'entité, de l'identifiant et des types primitifs. Pour le champ `created`, nous avons utilisé le type `DateTime`, qui traduit le format de base de données `AAAA-mm-jj HH:mm:ss` en une instance PHP `DateTime` et inversement.
- Après les définitions des champs, les deux références qualifiées à l'entité utilisateur sont définies. Elles sont créées par l'attribut `ManyToOne`. Le nom de classe de l'entité associée doit être spécifié avec le paramètre `targetEntity`, qui constitue suffisam-

ment d'informations pour que le mappeur de base de données puisse accéder à la table étrangère. Puisque **reporter** et **engineer** sont du côté propriétaire d'une relation bidirectionnelle, nous devons également spécifier le paramètre **inversedBy**. Il doit pointer vers le nom du champs du côté inverse de la relation. Nous verrons dans l'exemple suivant que le paramètre **inversedBy** a un homologue **mappedBy** qui fait la même chose du côté inverse.

- La dernière définition concerne la collection **Bug#products**. Elle contient tous les produits sur lesquels le bug spécifique se produit. Encore une fois, vous devez définir les paramètres **targetEntity** et **field** sur l'attribut **ManyToMany**.

2. Pour finir, nous ajoutons des mappages de métadonnées pour l'entité **User** :

```
<?php
// src/User.php

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'users')]
class User
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private int|null $id = null;

    #[ORM\Column(type: 'string')]
    private string $name;

    #[ORM\OneToMany(targetEntity: Bug::class, mappedBy: 'reporter')]
    private $reportedBugs;

    #[ORM\OneToMany(targetEntity: Bug::class, mappedBy: 'engineer')]
    private $assignedBugs;

    // .. (other code)
}
```

#### Remarque :

Concernant les balises **OneToMany**, n'oubliez pas que nous avons discuté des côtés inverse et propriétaire. Désormais, **reportedBugs** et **assignedBugs** sont des relations inverses, ce qui signifie que les détails de la jointure ont déjà été définis du côté propriétaire. Il suffit donc de spécifier la propriété de la classe **Bug** qui contient les côtés propriétaires.

3. Mettez à jour le schéma de votre base de données en exécutant :

```
php bin/doctrine orm:schema-tool:update --force --dump-sql
```



```
PS C:\Users\alexa\doctrine2-tutorial> php bin/doctrine orm:schema-tool:update --force --dump-sql
CREATE TABLE bugs (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, description VARCHAR(255) NOT NULL, created DATETIME NOT NULL, status VARCHAR(255) NOT NULL, engineer_id INTEGER DEFAULT NULL, reporter_id INTEGER DEFAULT NULL, CONSTRAINT FK_1E197C9F8D8CDF1 FOREIGN KEY (engineer_id) REFERENCES users (id) NOT DEFERRABLE INITIALLY IMMEDIATE, CONSTRAINT FK_1E197C9E1CFE6F5 FOREIGN KEY (reporter_id) REFERENCES users (id) NOT DEFERRABLE INITIALLY IMMEDIATE);
CREATE INDEX IDX_1E197C9F8D8CDF1 ON bugs (engineer_id);
CREATE INDEX IDX_1E197C9E1CFE6F5 ON bugs (reporter_id);
CREATE TABLE bug_product (bug_id INTEGER NOT NULL, product_id INTEGER NOT NULL, PRIMARY KEY(bug_id, product_id), CONSTRAINT FK_897D061DFA3DB3D5 FOREIGN KEY (bug_id) REFERENCES bugs (id) ON DELETE CASCADE NOT DEFERRABLE INITIALLY IMMEDIATE, CONSTRAINT FK_897D061D4584665A FOREIGN KEY (product_id) REFERENCES products (id) ON DELETE CASCADE NOT DEFERRABLE INITIALLY IMMEDIATE);
CREATE INDEX IDX_897D061DFA3DB3D5 ON bug_product (bug_id);
CREATE INDEX IDX_897D061D4584665A ON bug_product (product_id);
CREATE TABLE products (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name VARCHAR(255) NOT NULL);
CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name VARCHAR(255) NOT NULL);

Updating database schema...

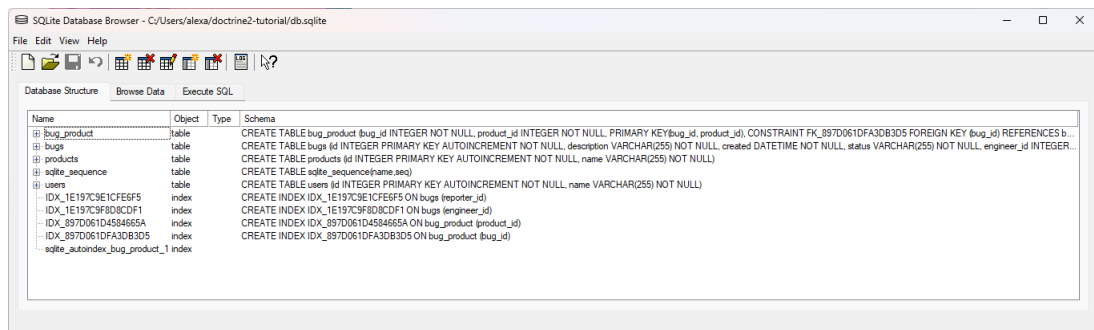
  8 queries were executed

[OK] Database schema updated successfully!

PS C:\Users\alexa\doctrine2-tutorial>
```

Vous aurez potentiellement dû corriger quelques éventuels bugs au préalable ;-)

Vérifiez que les requêtes ont bien été exécutées dans votre base de données :



Ajoutez tout, validez avec le message « Ajout des entités Bug et User (partie 4) » et poussez.

## 9 Implémenter davantage d'exigences

Jusqu'à présent, nous avons vu les fonctionnalités les plus élémentaires du langage de définition des métadonnées.

### 9.1 Travail à faire



1. Pour explorer des fonctionnalités supplémentaires, créons d'abord de nouvelles entités User :

```
<?php
// create_user.php
require_once "bootstrap.php";

$newUsername = $argv[1];

$user = new User();
$user->setName($newUsername);

$entityManager->persist($user);
$entityManager->flush();

echo "Created User with ID " . $user->getId() . "\n";
```

2. Maintenant appelez :

```
php create_user.php xavier
```



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php create_user.php xavier
Created User with ID 1
PS C:\Users\alexa\doctrine2-tutorial> |
```



Ajoutez tout, validez avec le message « Implémenter davantage d'exigences (partie 1) » et poussez.

## 9.2 Travail à faire



1. Nous disposons désormais des données nécessaires pour créer une nouvelle entité de Bug :

```

<?php
// create_bug.php <reporter-id> <engineer-id> <product-ids>
require_once "bootstrap.php";

$reporterId = $argv[1];
$engineerId = $argv[2];
$productIds = explode(",", $argv[3]);

$reporter = $entityManager->find("User", $reporterId);
$engineer = $entityManager->find("User", $engineerId);
if (!$reporter || !$engineer) {
    echo "No reporter and/or engineer found for the given id(s).\n";
    exit(1);
}

$bug = new Bug();
$bug->setDescription("Something does not work!");
$bug->setCreated(new DateTime("now"));
$bug->setStatus("OPEN");

foreach ($productIds as $productId) {
    $product = $entityManager->find("Product", $productId);
    $bug->assignToProduct($product);
}

$bug->setReporter($reporter);
$bug->setEngineer($engineer);

$entityManager->persist($bug);
$entityManager->flush();

echo "Your new Bug Id: ".$bug->getId()."\n";

```

2. Puisque nous n'avons qu'un seul utilisateur et produit, probablement avec l'identifiant 1, nous pouvons appeler ce script comme suit :

```
php create_bug.php 1 1 1
```



```

Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php create_bug.php 1 1 1
Your new Bug Id: 1
PS C:\Users\alexa\doctrine2-tutorial>

```



Ajoutez tout, validez avec le message « Implémenter davantage d'exigences (partie 2) » et poussez.



Rappelons que grâce au [pattern UnitOfWork](#), Doctrine mettra automatiquement à jour toutes les entités modifiées dans la base de données lors de l'appel à `flush()`.

## 10 Requêtes pour les Use-Cases de l'application

### 10.1 Liste des bugs

En utilisant les exemples précédents, nous pouvons remplir un peu la base de données. Cependant, nous devons maintenant discuter de la manière d'interroger le mappeur sous-jacent pour connaître les représentations de vue requises.

#### 10.1.1 Travail à faire



Lors de l'ouverture de l'application, les bugs peuvent être paginés via une vue de liste, qui est le premier cas d'utilisation en lecture seule :

```
<?php
// list_bugs.php
require_once "bootstrap.php";

$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r
        ORDER BY b.created DESC";

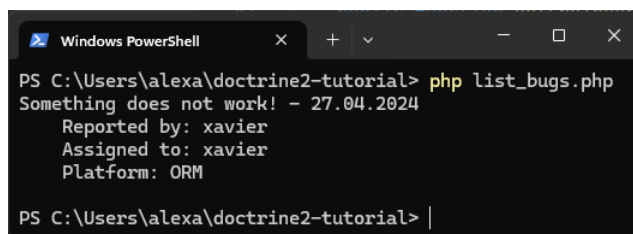
$query = $entityManager->createQuery($dql);
$query->setMaxResults(30);
$bugs = $query->getResult();

foreach ($bugs as $bug) {
    echo $bug->getDescription()." - ".$bug->getCreated()->format('d.m.Y')." \n";
    echo "    Reported by: ".$bug->getReporter()->getName()." \n";
    echo "    Assigned to: ".$bug->getEngineer()->getName()." \n";
    foreach ($bug->getProducts() as $product) {
        echo "        Platform: ".$product->getName()." \n";
    }
    echo " \n";
}
```

Explication :

La requête DQL (Doctrine Query Language) dans cet exemple récupère les 30 bogues les plus récents avec leur ingénieur et journaliste respectif dans une seule instruction SQL.

Vérifiez que la sortie console de ce script est :



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php list_bugs.php
Something does not work! - 27.04.2024
  Reported by: xavier
  Assigned to: xavier
  Platform: ORM
PS C:\Users\alexa\doctrine2-tutorial> |
```

## ! DQL is not SQL

Vous vous demandez peut-être pourquoi nous commençons à écrire du SQL au début de ce cas d'utilisation. N'utilisons-nous pas un ORM pour nous débarrasser de toute l'écriture manuscrite sans fin de SQL ? Doctrine introduit DQL qui est un ([langage de requête objet](#)) similaire à HQL ([Hibernate](#) Query Language) ou JPQL ([Java Persistence Query Language](#)). Il ne connaît pas la notion de colonnes et de tables, mais uniquement celles d'Entity-Class et de propriété. L'utilisation des métadonnées que nous avons définies auparavant permet d'effectuer des requêtes très courtes, distinctives et puissantes.

Une raison importante pour laquelle DQL est propice à l'API Query de la plupart des ORM est sa similitude avec SQL. Le langage DQL permet des constructions de requêtes que la plupart des ORM ne permettent pas : GROUP BY même avec HAVING, sous-sélections, récupération de classes imbriquées, résultats mixtes avec des entités et des données scalaires telles que les résultats COUNT() et bien plus encore. En utilisant DQL, vous devriez rarement arriver au point où vous souhaitez jeter votre ORM à la poubelle, car il ne prend pas en charge certains concepts SQL les plus puissants.

Si vous devez créer votre requête de manière dynamique, vous pouvez utiliser le `QueryBuilder` récupéré en appelant `$entityManager->createQueryBuilder()`. Vous trouverez plus de détails à ce sujet dans la partie correspondante de la documentation.

En dernier recours, vous pouvez toujours utiliser Native SQL et une description du jeu de résultats pour récupérer les entités de la base de données. DQL se résume à une instruction SQL native et à une instance `ResultSetMapping` elle-même. En utilisant Native SQL, vous pouvez même utiliser des procédures stockées pour la récupération de données ou utiliser des requêtes de base de données avancées non portables comme les requêtes récursives de PostgreSQL par exemple.



Ajoutez tout, validez avec le message « Requetes pour les Use-Cases de l'application : Liste des bugs » et poussez.



## 10.2 Tableau d'hydratation de la liste des bugs

Dans le cas d'utilisation précédent, nous avons récupéré les résultats en tant qu'instances d'objet respectives. Cependant, nous ne sommes pas limités à récupérer des objets uniquement à partir de Doctrine. Pour une vue de liste simple comme la précédente, nous n'avons besoin que d'un accès en lecture à nos entités et pouvons à la place basculer l'hydratation des objets vers de simples tableaux PHP.

L'hydratation peut être un processus coûteux, donc récupérer uniquement ce dont vous avez besoin peut générer des avantages considérables en termes de performances pour les requêtes en lecture seule.

### 10.2.1 Travail à faire



En implémentant la même vue de liste qu'avant en utilisant l'hydratation du tableau, nous pouvons réécrire notre code :

```
<?php
// list_bugs_array.php
require_once "bootstrap.php";

$dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
        "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
$query = $entityManager->createQuery($dql);
$bugs = $query->getArrayResult();

foreach ($bugs as $bug) {
    echo $bug['description'] . " - " . $bug['created']->format('d.m.Y') . "\n";
    echo "    Reported by: " . $bug['reporter']['name'] . "\n";
    echo "    Assigned to: " . $bug['engineer']['name'] . "\n";
    foreach ($bug['products'] as $product) {
        echo "        Platform: " . $product['name'] . "\n";
    }
    echo "\n";
}
```

#### Remarque :

Il y a cependant une différence significative dans la requête DQL : nous devons ajouter une jointure supplémentaire pour les produits liés à un bug. La requête SQL résultante pour cette instruction de sélection unique est assez volumineuse, mais toujours plus efficace à récupérer par rapport aux objets hydratants.

Vérifiez que la sortie console de ce script est :

```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php list_bugs_array.php
Something does not work! - 27.04.2024
Reported by: xavier
Assigned to: xavier
Platform: ORM
PS C:\Users\alexa\doctrine2-tutorial> |
```



Ajoutez tout, validez avec le message « Requêtes pour les Use-Cases de l'application : Tableau d'hydratation de la liste des bugs » et poussez.

## 10.3 Rechercher par clé primaire

Le cas d'utilisation suivant affiche un bug par clé primaire. Cela pourrait être fait en utilisant DQL comme dans l'exemple précédent avec une clause Where, cependant il existe une méthode pratique sur `EntityManager` qui gère le chargement par clé primaire, ce que nous avons déjà vu dans les scénarios d'écriture.

### 10.3.1 Travail à faire



Inspirez vous du script `show_product.php` pour écrire le script `show_bug.php` :

```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php show_bug.php 1
Bug: Something does not work!
Engineer: xavier
PS C:\Users\alexa\doctrine2-tutorial> |
```

#### Explication :

Puisque nous avons uniquement récupéré le bug par clé primaire, l'ingénieur et le rapporteur ne sont pas immédiatement chargés depuis la base de données mais sont remplacés par des « proxys LazyLoading ». Ces proxys se chargeront en arrière-plan, lorsque vous tenterez d'accéder à l'un de leurs états non initialisés.



Le LazyLoading (chargement paresseux) de données supplémentaires peut être très pratique, mais les requêtes supplémentaires créent une surcharge. Si vous savez que certains champs seront toujours (ou généralement) requis par la requête, vous obtiendrez de meilleures perfor-

- manances en les récupérant explicitement tous dans la première requête.



Ajoutez tout, validez avec le message « Requêtes pour les Use-Cases de l'application : Rechercher par clé primaire » et poussez.

## 11 Tableau de bord de l'utilisateur

Pour le prochain cas d'utilisation, nous souhaitons récupérer la vue du tableau de bord, une liste de tous les bugs ouverts signalés par l'utilisateur ou auxquels il a été affecté.

### 11.0.1 Travail à faire



Ceci sera réalisé en utilisant à nouveau DQL, cette fois avec quelques clauses WHERE et l'utilisation de paramètres liés :

```
<?php
// dashboard.php <user-id>
require_once "bootstrap.php";

$theUserId = $argv[1];

$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ".
      "WHERE b.status = 'OPEN' AND (e.id = ?1 OR r.id = ?1) ORDER BY b.created DESC";

$myBugs = $entityManager->createQuery($dql)
    ->setParameter(1, $theUserId)
    ->setMaxResults(15)
    ->getResult();

echo "You have created or assigned to " . count($myBugs) . " open bugs:\n\n";

foreach ($myBugs as $bug) {
    echo $bug->getId() . " - " . $bug->getDescription()."\n";
}
```



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php dashboard.php 1
You have created or assigned to 1 open bugs:

1 - Something does not work!
PS C:\Users\alexa\doctrine2-tutorial> |
```



Ajoutez tout, validez avec le message « Tableau de bord de l'utilisateur » et poussez.

## 12 Nombre de bugs

Jusqu'à présent, nous récupérons uniquement les entités ou leur représentation sous forme de tableau. Doctrine prend également en charge la récupération de valeurs scalaires via DQL. Ces résultats peuvent même être agrégés en utilisant les fonctions COUNT, SUM, MIN, MAX ou AVG.

### 12.1 Travail à faire



Nous utilisons une de ces fonctions pour récupérer le nombre de bugs ouverts regroupés par produit :

```
<?php
// products.php
require_once "bootstrap.php";

$dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
        "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
$productBugs = $entityManager->createQuery($dql)->getScalarResult();

foreach ($productBugs as $productBug) {
    echo $productBug['name'] . " has " . $productBug['openBugs'] . " open bugs!\n";
}
```



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php products.php
ORM has 1 open bugs!
PS C:\Users\alexa\doctrine2-tutorial> |
```



Ajoutez tout, validez avec le message « Nombre de bugs » et poussez.

## 13 Mise à jour des entités

Il manque un seul cas d'utilisation dans les exigences, les ingénieurs devraient être en mesure de fermer un bug.

### 13.1 Travail à faire



1. Commencez par ajouter une méthode `close()` à l'entité Bug :

```
<?php
// src/Bug.php

class Bug
{
    public function close()
    {
        $this->status = "CLOSE";
    }
}
```

2. Puis créez un script `close_bug.php` avec le contenu suivant :

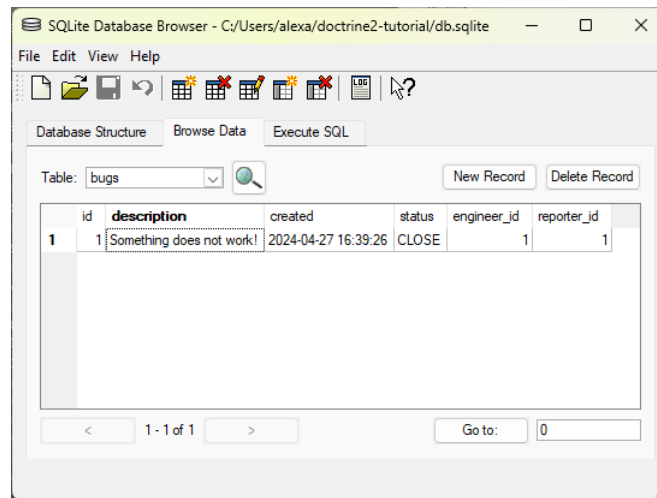
```
<?php
// close_bug.php <bug-id>
require_once "bootstrap.php";

$theBugId = $argv[1];

$bug = $entityManager->find("Bug", (int)$theBugId);
$bug->close();

$entityManager->flush();
```

3. Il ne vous reste plus qu'à tester la commande `php close_bug.php` et vérifier le résultat en base de données :



Lors de la récupération du bug de la base de données, il est inséré dans l'IdentityMap à l'intérieur de l'UnitOfWork de Doctrine. Cela signifie que votre bug avec exactement cet identifiant ne peut exister qu'une seule fois pendant toute la requête, quelle que soit la fréquence à laquelle vous appelez `EntityManager#find()`. Il détecte même les entités hydratées à l'aide de DQL et déjà présentes dans l'IdentityMap.

Lorsque flush est appelé, EntityManager parcourt toutes les entités de l'IdentityMap et effectue une comparaison entre les valeurs initialement extraites de la base de données et les valeurs que possède actuellement l'entité. Si au moins une de ces propriétés est différente, l'entité est planifiée pour un UPDATE sur la base de données. Seules les colonnes modifiées sont mises à jour, ce qui offre une assez bonne amélioration des performances par rapport à la mise à jour de toutes les propriétés.



Ajoutez tout, validez avec le message « Mise à jour des entités » et poussez.

## 14 Entity Repositories

Pour l'instant, nous n'avons pas expliqué comment séparer la logique de requête Doctrine de votre modèle. Dans Doctrine 1, il y avait le concept d'instances `Doctrine_Table` pour cette séparation. Le concept similaire dans Doctrine 2 est appelé *Entity Repositories*, intégrant le [pattern repository](#) au cœur de Doctrine.

Chaque entité utilise par défaut un référentiel par défaut et propose un certain nombre de méthodes pratiques que vous pouvez utiliser pour interroger les instances de cette entité. Prenons par exemple notre entité Produit. Si nous voulons effectuer une requête par nom, nous pouvons utiliser :

```
<?php
$product = $entityManager->getRepository('Product')
    ->findOneBy(array('name' => $productName));
```

La méthode `findOneBy()` prend un tableau de champs ou de clés d'association et les valeurs à comparer.

Si vous souhaitez trouver toutes les entités correspondant à une condition, vous pouvez utiliser `findBy()`, par exemple en interrogeant tous les bugs fermés :

```
<?php
$bugs = $entityManager->getRepository('Bug')
    ->findBy(array('status' => 'CLOSED'));

foreach ($bugs as $bug) {
    // do stuff
}
```

Par rapport à DQL, ces méthodes de requête manquent très rapidement de fonctionnalités.

## 14.1 Travail à faire

Doctrine vous offre un moyen pratique d'étendre les fonctionnalités du `EntityRepository` par défaut et d'y placer toute la logique de requête DQL spécialisée.



1. Pour cela, vous devez créer une sous-classe de `Doctrine\ORM\EntityRepository`, dans notre cas un `BugRepository` et y regrouper toutes les fonctionnalités de requête évoquées précédemment :

```

<?php
// src/BugRepository.php

use Doctrine\ORM\EntityRepository;

class BugRepository extends EntityRepository
{
    public function getRecentBugs($number = 30)
    {
        $dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r
        ORDER BY b.created DESC";

        $query = $this->getEntityManager()->createQuery($dql);
        $query->setMaxResults($number);
        return $query->getResult();
    }

    public function getRecentBugsArray($number = 30)
    {
        $dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
        "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
        $query = $this->getEntityManager()->createQuery($dql);
        $query->setMaxResults($number);
        return $query->getArrayResult();
    }

    public function getUsersBugs($userId, $number = 15)
    {
        $dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ".
        "WHERE b.status = 'OPEN' AND e.id = ?1 OR r.id = ?1
        ORDER BY b.created DESC";

        return $this->getEntityManager()->createQuery($dql)
            ->setParameter(1, $userId)
            ->setMaxResults($number)
            ->getResult();
    }

    public function getOpenBugsByProduct()
    {
        $dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
        "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
        return $this->getEntityManager()->createQuery($dql)->getScalarResult();
    }
}

```



2. Pour pouvoir utiliser cette logique de requête via une instruction du type :  
`$this->getEntityManager()->getRepository('Bug')`, nous devons ajuster légèrement les métadonnées :

```
<?php
// src/Bug.php

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: BugRepository::class)]
#[ORM\Table(name: 'bugs')]
class Bug
{
    // ...
}
```

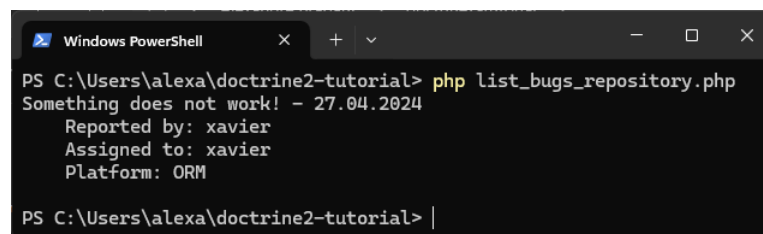
3. Nous pouvons désormais supprimer notre logique de requête à tous les endroits et les utiliser via `EntityRepository`. À titre d'exemple, voici le code du premier cas d'utilisation qui liste les bugs :

```
<?php
// list_bugs_repository.php
require_once "bootstrap.php";

$bugs = $entityManager->getRepository('Bug')->getRecentBugs();

foreach ($bugs as $bug) {
    echo $bug->getDescription()." - ".$bug->getCreated()->format('d.m.Y')." \n";
    echo "    Reported by: ".$bug->getReporter()->getName()." \n";
    echo "    Assigned to: ".$bug->getEngineer()->getName()." \n";
    foreach ($bug->getProducts() as $product) {
        echo "        Platform: ".$product->getName()." \n";
    }
    echo " \n";
}
```

4. Il ne vous reste plus qu'à tester :



```
Windows PowerShell
PS C:\Users\alexa\doctrine2-tutorial> php list_bugs_repository.php
Something does not work! - 27.04.2024
    Reported by: xavier
    Assigned to: xavier
    Platform: ORM
PS C:\Users\alexa\doctrine2-tutorial> |
```



- En utilisant `EntityRepositories`, vous pouvez éviter de coupler votre modèle avec une logique de requête spécifique. Vous pouvez également réutiliser facilement la logique de requête dans toute votre application.
- Enfin, notez que la méthode `count()` prend un tableau de champs ou de clés d'association et les valeurs à comparer. Cela vous offre un moyen pratique et léger de compter un ensemble de résultats lorsque vous n'avez pas besoin de le gérer.

```
<?php
$productCount = $entityManager->getRepository(Product::class)
    ->count(['name' => $productName]);
```



Ajoutez tout, validez avec le message « Entity Repositories » et poussez.