Architecture(s) et application(s) Web

Télécom SudParis Olivier Berger (TSP)

21/08/2025

CSC4101 - Cours Accès aux données avec l'ORM Doctrine

Table des matières

1 L'ORM Doctrine 3

2 Coder les classes 9

3 Utiliser les classes du modèle de données 12

4 Gérer des données de tests 22

Objectifs de cette séquence

Cette séquence de cours magistral abordera l'un des outils qui est utilisé dès les premiers TP, et qu'on utilisera ensuite tout au long du module, l'ORM *Doctrine*, qui gèrera la couche d'accès aux données dans nos développements en PHP avec Symfony.

1 L'ORM Doctrine

Cette section présente succinctement les fonctionnalités de l'ORM Doctrine, pour détailler le fonctionnement des mécanismes qu'on utilise dans le cours, et approfondir des éléments plus avancés.

1.1 Pourquoi une base de données ?

Exécution typique d'une application Web

1. Arrivée requête HTTP
   — L'application démarre
   — *Charge des données en mémoire*
2. Elle traite la requête
   — *Impact sur données* en mémoire
3. Fin
   — *Enregistre les données mises à jour*
   — Envoi réponse HTTP
   — L'application s'arrête

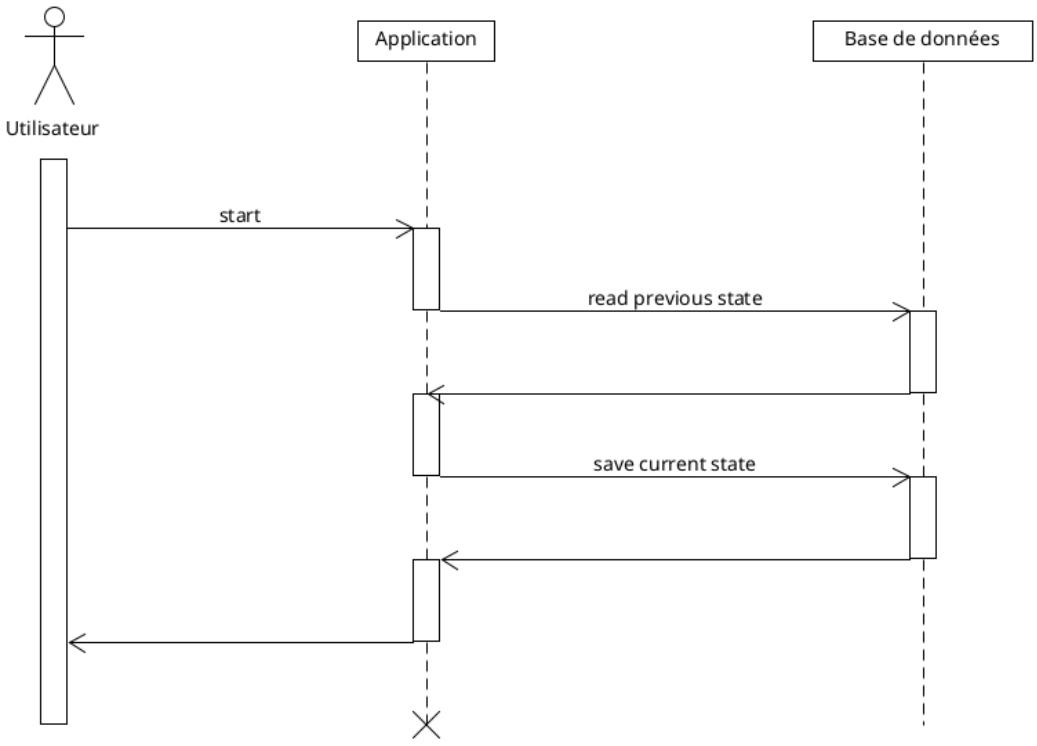


Figure 1 – Vie d'une appli Web (over-simplifiée)

Mais aussi interaction entre utilisateurs sur données partagées
Programmer :
— Modèle de données **en mémoire** (à chaud) :
  orienté objet :
  — UML
  — PHP (objet)
— Modèle de données en base de données (à froid, persistant) :
  relationnel :

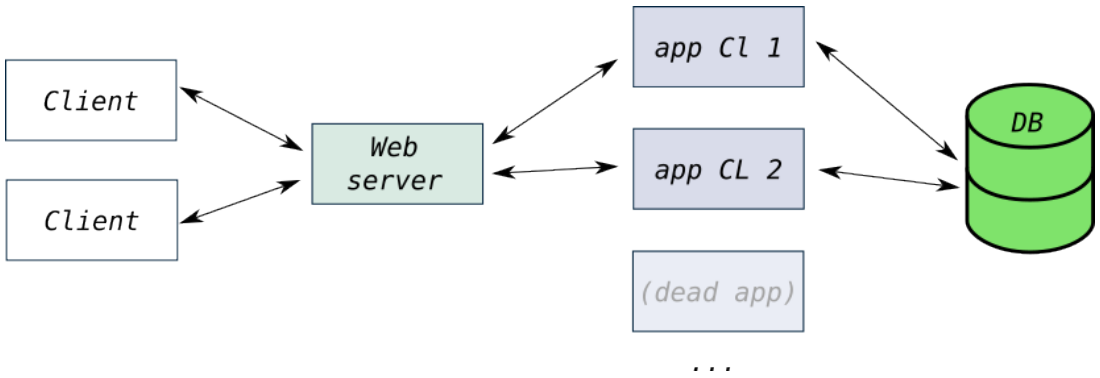


Figure 2 – Accès concurrent au RGBD

— entités + associations
— Système de Gestion de Base de données (SGBD) + SQL

1.2 Rôle d'un *ORM*

ORM *Object Relational Mapper*
— Principe ORM
  — Manipuler les données de l'application via des **classes / objets**
  — Implémentation du **Modèle de données** applicatif (en mémoire)
  — Persistence dans un SGBD
  — Conversion d'un modèle **objet** en un modèle **relationnel**

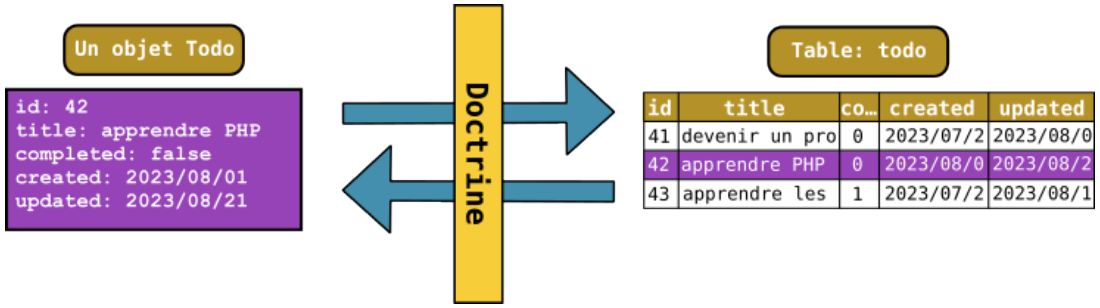


Figure 3 – Conversion objet - relationnel

Object Relational Mapping peut se traduire par *Mapping* (en bon franglais - sic) ou conversion objet-relationnel.
En principe, vous maîtrisez le modèle relationnel, qui est un pré-requis de ce module. De même pour le modèle objet.
Par contre, la conversion objet-relationnel a été étudiée, mais probablement pas approfondie, d'où la nécessité de l'approfondir ici.

1.2.1 Doctrine, l'*ORM* standard en PHP

`https://www.doctrine-project.org/projects/orm.html`

- composant standard applis PHP
- bien intégré avec Symfony :
  - gestion modèle de données
  - intégration avec formulaires saisie données
  - assistant génération de code dans Symfony
  - …

Doctrine est le composant d'ORM standard qui est intégré dans Symfony. Il gère l'accès et la persistance des données de notre application en base de données.
Il apporte de nombreuses fonctionnalités, en particulier :

- la génération d'une base de données relationnelle, à partir du Modèle objet défini par le programmeur dans les classes PHP. En général, on n'a pas besoin d'écrire quoi que ce soit en langage SQL, ou de s'occuper des détails de tel ou tel SGBD, quand on développe une application simple avec Symfony. Doctrine s'occupe « de tout » pour nous.
- l'articulation automatique avec les formulaires de saisie de données de Symfony, pour le support du typage des données (vérification de contraintes de saisie, sécurité, etc.)

Doctrine est un des composants de Symfony (`http://symfony.com/doc/current/doctrine.html`), mais il est aussi utilisable dans des applications PHP, en dehors de Symfony (`http://www.doctrine-project.org/projects/orm.html`).
La plupart des programmes PHP écrits avec Symfony qui seront étudiés dans ce cours utiliseront Doctrine. Il est donc utile de mieux savoir comment fonctionne Doctrine.

1.3 Concevoir les classes du modèle

Conception **orientée objet**

- Classes (PHP)
- Propriétés *mono-valuées* des classes :
  types de bases + *références* à des instances d'autres classes
- Propriétés *multi-valués* :
  Collections d'objets (ou de références d'objets)

La conception orientée objets n'est pas un domaine trivial. On peut mettre à profit les connaissances en modélisation UML, par exemple.
Dans le module on essaiera de traiter uniquement des structures de données relativement simples.
On étudiera de façon plus avancée la Conception Orientée Objet dans le module CSC4102 « *Introduction au Génie Logiciel Orienté Objet* ».
Examinons le modèle orienté objet d'une application simple, pour rappeler des éléments de vocabulaire.

1.3.1 Exemple modèle de données objet Todo

Modéliser des **tâches** : **classe** `Todo`
Propriétés mono-valuées :

- `title` : chaîne
- `completed` : booléen
- `created`, `updated` : dates

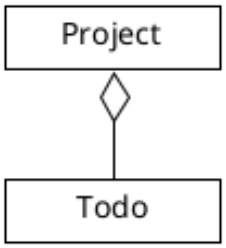
```php
class Todo
{
        private string $title;
        private bool $completed;
        private DateTime $created;
        private DateTime $updated;
```

Modéliser des **projets** : **classe** `Project`
Propriétés mono-valuées :

— `title` : chaîne
— `description` : chaîne

```
class Project
{
        private string $title;
        private string $description;
```

Association 1-n entre `Project` et `Todo`
En UML :



— tâches d'un projet (multi-valué) : Collection
— projet d'une tâche : référence
En PHP :
— les tâches d'un projet (`Project::todos`)

```
class Project
{
        private string $title;
        private description $completed;

        private array $todos = array();
```

— le projet d'une tâche (`Todo::project`)

```
class Todo
{
        private string $title;
        private bool $completed;
        private DateTime $created;
        private DateTime $updated;

        private Project $project;
```

En PHP, les types de base des propriétés mono-valuées sont disponible « de base », comme les chaînes, les entiers, ou les booléens. Laissons de côté les dates qui sont plus complexes (vu les soucis d'internationalisation, notamment).
Par contre, comme pour la programmation en Java, il est nécessaire de comprendre comment sont gérées les **associations** entre instances de différentes classes, pour les implémenter via des propriétés.
En PHP objet, la notion de **référence** existe naturellement, de façon assez similaire à ce qu'on trouve dans d'autres langages comme Java.
Une instance de `Todo` aura donc une propriété `project` (par convention `Todo::project`) qui pourra référencer une instance de `Project`, ou valoir `null` si elle n'est pas définie.
Pour ce qui concerne les **collections** (qui permettent de gérer des propriétés multi-valuées), la standardisation en PHP a été progressive, et on va s'appuyer, dans ce cours, sur les structures de données objet fournies par Doctrine, comme des « tableaux » / listes comme ArrayCollection.

1.3.2 Raffiner

Quelle est la « force » de l'association ?

- association : tâches sans projet possibles ?
- composition : pas de tâche sans projet ?

Pourra être approfondi dans CSC4102 « *Introduction au Génie Logiciel Orienté Objet* »

1.4 Vous êtes en terrain connu

Vous maîtrisez déjà :

- Conception du **modèle des données en Objet** (*comme découvert en CSC3101*)
- Programmation **objet en PHP** (*assez proche de Java, en fait, même si pas compilé*) :
  - références
  - collections

À l'exécution, sous le capot : génère des requêtes SQL dans SGBD relationnel (*appris en CSC3601*)… mais pas besoin de les programmer

Le principe d'un **ORM** *Object Relational Mapper* est de permettre de concevoir une application dans le paradigme objet, indépendamment de la technologie de bases de données sous-jacente.
La plupart des concepts que vous connaissez, comme les **références** vers des objets et les **associations** se retrouvent dans le modèle objet de PHP.
Les structures de données disponibles en PHP pour gérer des collections d'objets sont légèrement différentes de celles d'autres langages (p. ex. Java), mais reposent en général sur des tableaux (équivalents à des listes) ou des tableaux associatifs (dictionnaires). On verra que Doctrine apporte des classes particulières pour les structures de données, mais leur manipulation repose en grande partie sur la syntaxe de manipulation des tableaux déjà vue dans l'apprentissage de la syntaxe PHP en autonomie.
L'ORM réalise les opérations nécessaires pour assurer la conversion du modèle **objet** (références, associations, collections) en un modèle **relationnel**, pour permettre un stockage persistant, et donc l'intégration avec d'autres applications (via le langage SQL).

1.4.1 Utiliser l'ORM

- Ne pas écrire les requêtes SQL de chargement / modification
- **Programmer en objet** avec Doctrine
- L'ORM (*Object Relational Mapper*) détecte les données nouvelles ou modifiées et génère le SQL sous le capot

Le composant d'ORM permet de manipuler les données de l'application via des **objets** (instances de classes PHP).
Le programmeur réalise donc l'implémentation du **Modèle** applicatif grâce à des structures de données objet instanciées en mémoire, quand l'application s'exécute.
Sous le capot, l'ORM attribue des identifiants qui sont nécessaires au fonctionnement du modèle physique des données (dans le SGBD relationnel), alors que les objets PHP sont manipulés uniquement par leur référence (adresse mémoire).

1.4.2 Oublier SQL ?

- Pas toujours si simple
- Réviser un peu CSC3601 « *Modélisation, bases de données et systèmes d'information* » ?

Comprendre pour *debugger* si tout ne marche pas comme prévu automagiquement.

Le développeur Symfony n'est pas obligé d'apprendre SQL, s'il utilise Doctrine. Mais **Doctrine n'est pas magique pour autant**. Il faut notamment bien comprendre la syntaxe des attributs, et mieux vaut bien connaître **les bases du modèle relationnel** pour ne pas faire n'importe quoi.

1.4.3 Exemple : références traduites en clés étrangères

Examinons comment se traduisent les références entre entités associées

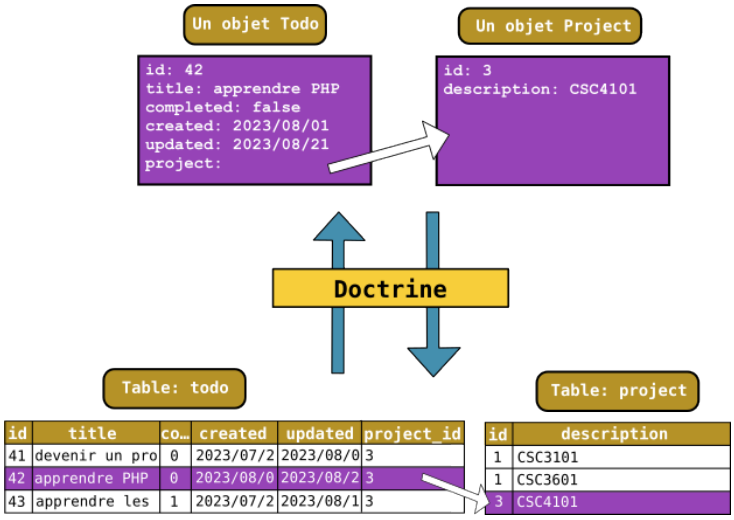


Figure 4 – Traduction des références en clés étrangères

Les propriétés `Project::todos` et `Todo::project` qui réalisent l'association `OneToMany/ManyToOne` entre un projet et ses tâches sont gérées, en mémoire, avec une collection, et respectivement une référence.
Mais si on s'intéresse au stockage en base de données relationnelle, on ne peut stocker de propriétés multi-valuées dans le modèle relationnel. Impossible donc de stocker la collection des instances de la propriété `Project::todo` dans une colonne de la table `project`. On pourra seulement stocker la référence d'une tâche à son projet via la clé étrangère `project_id` de `todo` qui sera une réfénce à l'identifiant `id` de `project`. Le chargement par jointure permettra de reconstituer la collection, à la demande.
Si vous ne maîtrisez pas cet aspect du modèle relationnel, vous risquez d'avoir du mal à comprendre certains soucis de mise au point, notamment sur la modification des données, et la persistence de ce genre d'associations.

2 Coder les classes

Cette section présente succinctement la façon dont on peut coder en PHP les classes du modèle de données de l'application, avec l'aide de Doctrine

2.1 Codage des entités du modèle de données

Observons comment a été mis en œuvre le modèle des données de l'application « fil-rouge » *Todo*.

2.1.1 Emplacement du code PHP

Classe de l'entité `Todo` :
- classe PHP `Todo`
- fichier source : `src/Entity/Todo.php`
- espace de nommage : module `App\Entity\Todo`

Les classes du modèle de données sont à coder dans un répertoire particulier du projet, et avec un espace de nommage particulier, par convention.
Notons qu'il n'est pas obligatoire de coder « à la main », mais qu'on utilisera aussi beaucoup des générateurs de code, pour aller plus vite et éviter les erreurs.

2.1.2 Classe PHP « naïve »

```php
class Todo
{
    private string $title;
    private bool $completed;

    public function getTitle() {
      // ...
    public function setTitle($title) {
      // ...
// ...
```

Pas de typage des propriétés, en « PHP basique », mais possible en PHP moderne (et objet).

Il n'y a pas de typage strict des propiétés, en PHP, comme dans d'autres langages interprétés. Mais c'est fortement recommandé dans un style de programmation moderne, y compris en objet.
Une fois cette classe disponible dans l'application, on peut gérer des instances de la classe en mémoire, par exemple en créant des objets, en les ajoutant dans un tableau, en parcourant ce tableau, etc.
Chaque objet est identifié par sa référence en mémoire.
Dans ce qui précède, le modèle objet de PHP ressemble fort à ce que vous connaissez déjà en Java.

2.1.3 Classe PHP documentée

Docblocks PHP « standard » (sans ORM) :

```php
class Todo {
          /**
           * @var string task title
           */
```

```
        private string $title;

        /**
         * @var bool Is the task completed/finished.
         */
        private bool $completed;

}
```

On voit que le programmeur a ajouté des annotations sous forme de « doc blocks » dans les « commentaires », qui ne sont pas utilisés par PHP en première instance, mais sont destinés à des outils comme PhpDocumentor.
Cela permet d'assister d'autres programmeurs qui utilise cette classe, s'ils utilisent des outils comme un éditeur de texte moderne ou un IDE comprenant ces annotations *PHPDoc*.
Le format de ces annotations *DocBlocks* de PHPDoc n'est pas un simple commentaire PHP : il doit commencer par « `/**` », au lieu d'un simple « `/*` » (cf. What does a DocBlock look like ?)
On retrouve la même syntaxe générale que pour Java avec JavaDoc.

2.2 Introduction de la persistance avec Doctrine

Sur la base de ce code objet standard PHP, on ajoute des méta-données qui introduisent des contraintes sur les classes et leurs propriétés, que Doctrine sait exploiter pour réaliser ses mécanismes d'ORM.

2.2.1 Attributs PHP

Ajout d'annotations, méta-données pour Doctrine

```
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
class Todo
{
        #[ORM\Id, ORM\GeneratedValue, ORM\Column]
        private int $id;

        #[ORM\Column(length: 255, nullable: true)]
        private string $title;

        #[ORM\ManyToOne(targetEntity: Project::class,
                        inversedBy: 'todos')]
        private Project $project;
        //...
}
```

```
#[ORM\Entity]
class Project
{
    #[ORM\Id, ORM\GeneratedValue]
    private int $id;

    #[ORM\Column(type: "text", nullable: true)]
    private string description;

    #[ORM\OneToMany(targetEntity: Todo::Class, mappedBy: 'project')]
    private $todos;
    //...
}
```

Les attributs définissent des règles de typage des valeurs des propriétés, ainsi que la définition de propriétés multi-valuées pour gérer les associations entre instances des classes du modèle comme étant des « tableaux » (listes d'éléments itérables).
Elles permettent aussi de définir des contraintes : propriété obligatoire, cardinalités des associations, compositions, etc.
Cf. Documentation Doctrine pour l'ensemble des fonctionnalités sur le *mapping* des propriétés des objets.
Notons en particulier, ici, la description d'une relation `OneToMany` permettant de gérer une association.
Une fois ces différentes méta-données ajoutées au code, Doctrine peut fonctionner.
Tant qu'on travaille en mémoire, sur des structures de données PHP, ces annotations entrent marginalement en ligne de compte.
Mais quand on souhaite charger ou sauvegarder dans la base de données les objets présents en mémoire, Doctrine fait le lien entre les objets PHP en mémoire et une base relationnelle, via la génération de requêtes SQL.

2.3 Générateur de code `make:entity`

Asssistant générateur de code : `make:entity`
Abus fortement recommandé !

```
symfony console make:entity

Class name of the entity to create or update (e.g. DeliciousPopsicle):
> Project

created: src/Entity/Project.php
created: src/Repository/ProjectRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> title

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Project.php

[...]

 Success!


Next: When you're ready, create a migration with symfony console make:migration
```

3 Utiliser les classes du modèle de données

Voyons maintenant la façon dont on peut utiliser ces classes du modèle de données, dans un projet Symfony.
On va examiner comment fonctionnent les mécanismes de Doctrine pour déclencher le chargement des données depuis la base de données.
On verra plus tard, bien plus en détails, les fonctions permettant de sauvegarder les données de l'application.

3.1 Programmer le chargement en mémoire

Intéressons-nous d'abord au chargement des **Collections** d'instances, qui correspond au `SELECT ... FROM ...`, où on doit allouer une nouvelle instance d'une classe, pour chaque occurrence du résultat de la requête.

3.1.1 Le *repository* d'instances

On gère le chargement des données grâce à un « générateur d'objets » (le *Repository*), associé à une classe Doctrine.

```php
use Doctrine\ORM\Mapping as ORM;

use App\Repository\TodoRepository;

#[ORM\Entity(repositoryClass: TodoRepository::class)]
class Todo
{
        #[ORM\Id, ORM\GeneratedValue, ORM\Column]
        private int $id;
```

Listing 1 : Extrait de `src/Entity/Todo.php`

Le *repository* est codé dans une classe, ici `TodoRepository` (dont le code est présent dans `src/Repository/TodoRepository.php`).
Cette classe utilitaire est associée à notre entité du modèle de données `Todo` via l'attribut `ORM\Entity`.
C'est un générateur d'instances d'une classe. C'est lui, dans Doctrine, qui sait générer des instances, ou des collections d'instances de nos classes PHP.
C'est par son intermédiaire qu'on va charger les données depuis la base relationnelle.

3.1.2 Chargement d'une collection d'instances (`findAll()`)

Utilisation pour charger toutes les instances de `Todo` :

La méthode `findAll()` des *repositories* Doctrine permet ainsi de **charger, dans une collection en mémoire**, toutes les instances d'une classe, qui correspondent à toutes les données correspondantes en base de données.
Avec un ORM comme Doctrine, on raisonne en objet. Plus besoin de faire un `SELECT * from todo;` en SQL et d'allouer manuellement des instances de `Todo` en mémoire, et d'y recopier le résultat de chaque ligne du résultat du SELECT. C'est le job d'un *repository*, et on s'appuie sur ses méthodes pour faire cela.

```
use App\Entity\Todo;
//...
        protected function execute()
        {
          // récupère une liste toutes les instances de Todo

          $todos = $this->todoRepository->findAll();

          foreach($todos as $todo)
          {
                  //...
          }

        }
```

Listing 2 : Extrait de `ListTodosCommand.php`

3.1.3 Accès au *repository* de Doctrine

```
use App\Entity\Todo;
use App\Repository\TodoRepository;
//...
class ListTodosCommand extends Command  {
    /**
     *  @var TodoRepository data access repository
     */
    private $todoRepository;

    public function __construct(ManagerRegistry $doctrineManager) {
        $this->todoRepository = $doctrineManager
                                ->getRepository(Todo::class);
        parent::__construct();
    }

    protected function execute(): int {
        // fetches all instances of class Todo from the DB
        $todos = $this->todoRepository->findAll();
        // ...
```

Ce code est un peu complexe, car il met en œuvre des patrons de conception avancés (Symfony est un *framework* moderne, complexe, si on essaye d'en comprendre tous les détails dès le début). Rassurez-vous : on essaye de ne pas réinventer la roue, et on utilise des générateurs de code, ou bien on s'inspire de la documentation.
On pourrait écrire le début de ce code de façon moins condensée, pour mettre en évidence les différentes composantes de Symfony et Doctrine mises en œuvre.
Essayons quand même de comprendre. Accrochez-vous :

1. `ListTodosCommand` est la classe d'une commande que nous avons codé, dans `src/Command/ListTodosCommand.php`, qui hérite d'une classe Symfony, `Command` ;
2. nous surchargeons le constructeur de `Command` pour recevoir en argument, une instance du `ManagerRegistry` de Doctrine (inutile d'approfondir ce mécanisme de la tuyauterie Symfony pour l'instant) :
   — cet utilitaire de Doctrine nous permet de récupérer le *repository* d'une classe de notre modèle de données : `->getRepository(Todo::class)`.
   — on garde la référence de ce repository dans une propriété de notre classe : `ListTodosCommand::todoRepository` (de type `TodoRepository`), pour pouvoir le récupérer lorsque la méthode `execute()` sera appelée
3. une fois que la commande est appelée, cette propriété nous permet d'appeler la méthode `findAll()` du repository de `Todo`.

Ce mécanisme d'injection de dépendances, permet à une classe d'accéder à des services de Symfony depuis son constructeur, et on le retrouvera à différents endroits.
C'est un peu complexe, mais ça fonctionne. Même si on ne comprend pas tous les détails de l'usine à gaz qu'est un cadriciel moderne comme Symfony, on peut faire confiance à la documentation pour nous dire comment faire. Et surtout, le code devient relativement compact à écrire.

3.2 Chargement depuis la base de données (`find...()`)

Chargement d'**un seul** objet depuis la base de données.
3 variantes :
— par identifiant
— par critères de sélection
— par un valeur d'une propriété (raccourci)
Encore grâce à la même classe *repository*.

3.2.1 Chargement via l'identifiant : `find($id)`

```php
$todo = $this->todoRepository->find($id);
```

génère, via Doctrine :

```sql
SELECT * FROM ... WHERE ID=[$id]
```

Attention : les identifiants sont uniques, mais un détail d'implémentation (valeur générée par le SGBD).

Cette fois on accède directement à une instance par son identifiant `find($id)`.
Mais le plus souvent, cet identifiant interne à la base de données reste caché dans les mécanismes internes de l'application.

3.2.2 Chargement par sélection sur des propriétés (`findOneBy()`)

Chargement d'une instance de `Todo`, étant donnés un *titre* et un *état terminé* (extrait de `ShowTodoCommand.php`) :

```
$todo = $this->todoRepository->findOneBy(
    ['title' => $title,
     'completed' => $completed]);
```

génère une requête SQL style :

```
SELECT ... FROM todo
  WHERE title = [$title]
  AND completed = [$completed]
  LIMIT 1
```

On utilise la méthode `findOneBy` du *repository*, en lui passant en argument un tableau contenant les différents critères de sélection/restriction de la requête à faire dans la base de données.
Vous verrez alors, dans le log, des requêtes du type :

```
[2018-08-07 10:14:47] doctrine.DEBUG: SELECT t0.tid AS tid_1, t0.title AS title_2, t0.completed AS c
```

Doc Doctrine : Documentation Doctrine ORM / Working with Objects / By Simple Conditions.
La documentation indiquée dans le lien ci-dessus permet de retrouver les différents critères qu'on peut passer aux méthodes de chargement d'instances, ou de collections d'instances.
Globalement, tous les critères de sélections qu'on utilise classiquement dans le modèle relationnel sont utilisables, typiquement pour filtrer sur des valeurs de propriétés particulières.

3.2.3 Sélection via *findBy...* suivi du nom de propriété

Méthodes `findBy*` nommées d'après les propriétés de la classe (*magic finders*) :
Exemple :

```
$todos = $this->todoRepository->findByCompleted(false);
```

donne :

```
..
  WHERE completed = 0;
```

Avec l'appel d'une méthode « magique » du type `findByPropriété(valeur)`, cette méthode charge « automagiquement » les instances de la classe ayant la propriété « `Propriété` » (issue du nom de la méthode) à la valeur « `valeur` »
Doctrine a généré la clause `WHERE` SQL correspondant à la valeur de la propriété `completed`, pour le `findByCompleted()`.
Notons que ces « *magic finders"* sont assez peu documentés. Mais on trouve une indication dans la documentation de l'API Doctrine sous l'intitulé sibyllin »*Adds support for magic method calls."*.
Une méthode `_call()` des *repository* permet d'intercepter l'appel aux méthodes et de détecter que la méthode `findByCompleted(false)` correspond en fait à un appel à `findBy(array('completed' => false))`.

3.2.4 Méthodes spécifiques de votre modèle applicatif

Les classes *repository* sont générées avec des fonctionnalités basiques.
Si besoin, on complète le *repository* pour ajouter des requêtes spécifiques à notre contexte applicatif :

```
class TodoRepository extends ServiceEntityRepository
{
        //...
        public function findLatest($page)
        {
                //...
        }
```

Si besoin, le programmeur dispose aussi d'un langage de requêtage proche de SQL, dans Doctrine. Cf. `https://symfony.com/doc/current/doctrine.html#querying-with-the-query-builder`
Mais la plupart du temps, pour des applications simples, on n'en aura pas besoin, et on exprimera les algorithmes dans une syntaxe objet, avec ces méthodes des *repositories* Doctrine.

3.3 Création de la base de tests

Voyons comment fonctionnent les outils de Doctrine qui nous permettent de créer une base de données de tests.

3.3.1 Attention : en *dév* ou en *prod*

Outils à utiliser dans env. de développement !

On n'étudie pas ici ce qui serait utilisé lors de la mise en production d'une application, car pour l'instant, notre besoin est de tester l'application en environnement de développement.

3.3.2 Base existante, ou base générée

Qui gère la base de données ?
- brancher Symfony sur une base existante
- ou générer une base de données à partir du modèle de données de notre application Symfony

Dans les deux cas Doctrine fait le *job*
Dans notre contexte, on est dans le second cas.

À chaque évolution de notre modèle de données, suite à une modification du code PHP, on re-teste en re-générant une nouvelle base de données à partir du nouveau code.

3.3.3 Génération d'une BD à partir du code

- Recherche des attributs Doctrine `ORM` dans code des classes PHP
- Génération requêtes SQL de création du schéma (*SQL /modeling language*)
  - Spécificités SGBD cible (SQLite, PostgreSQL, MariaDB, …)

```
CREATE TABLE todo (
       id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
       title VARCHAR(255) NOT NULL,
       completed BOOLEAN NOT NULL);
```

Listing 3 : Schéma généré pour SQLite (symfony console doctrine:schema:create -vvv) :

Doctrine sait examiner le code des classes PHP, pour y découvrir les attributs qui le concernent (mécanisme d'instrospection PHP).
L'ORM en déduit les opérations à mettre en œuvre pour gérer la persistance des données, quand le programmeur en aura besoin.
Doctrine peut ainsi générer, en fonction du type de SGBD utilisé dans le projet, un schéma de données relationnel.
Différents types de bases de données (y compris NoSQL, clé-valeurs) sont supportés avec Doctrine, même si nous utiliserons uniquement le modèle relationnel et SQLite, dans le cours.

3.3.4 Commandes de re-génération de la base de données

1. Configurer quelle base de données cible (**SQLite**, MySQL, PostGreSQL, …) : variable dans le fichier `.env`
2. Exécuter :

```
symfony console doctrine:database:create
symfony console doctrine:schema:create
```

La base de données (fichier SQLite) est créée dans le répertoire courant, avec des tables vides.

C'est la configuration de la base de données cible (variable `DATABASE_URL` dans le fichier `.env`) qui indique comment générer le modèle concret, avec une syntaxe SQL peut varier d'un SGBDR à l'autre.

3.3.5 Structure du schéma

Voir le schéma de données généré dans la base :

```
$ bin/console doctrine:schema:create --dump-sql
```

Exemple :

```
CREATE TABLE project (
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  title VARCHAR(255) NOT NULL,
  description CLOB DEFAULT NULL);

CREATE TABLE todo (
  tid INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  project_id INTEGER DEFAULT NULL,
  title CLOB DEFAULT NULL,
  completed BOOLEAN NOT NULL,
  -- ...
  CONSTRAINT FK_CD826255166D1F9C FOREIGN KEY (project_id)
    REFERENCES project (id) NOT DEFERRABLE INITIALLY IMMEDIATE);

...
```

Listing 4 : Contrainte d'intégrité référentielle pour clé étrangère `project_id`, pour matérialiser l'association 1-N, avec SQLite

C'est la configuration de la base de données cible qui permet de générer le modèle concret pour la sérialisation des données dans un SGBDR.
La syntaxe SQL des contraintes d'intégrité référentielle change en effet, selon le SGBD.

3.3.6 Relations m-n `ManyToMany`

Exemple : étiquettes (`Tag`) sur des tâches (`Todo`)

```
CREATE TABLE todo (
       id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
...);
CREATE TABLE tag (
       id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
...);

-- Génération d'une table de relation
CREATE TABLE todo_tag (
       todo_id INTEGER NOT NULL,
       tag_id INTEGER NOT NULL,
       PRIMARY KEY(todo_id, tag_id),
       CONSTRAINT FK_D767A0BAEA1EBC33
                  FOREIGN KEY (todo_id)
                  REFERENCES todo (id),
       CONSTRAINT FK_D767A0BABAD26311
                  FOREIGN KEY (tag_id)
                  REFERENCES tag (id) );
```

Pour les relations m-n (configurées par l'annotation `ManyToMany`), Doctrine génère une table de relation dédiée, qui groupe les clés étrangères des différents relations concernées.
Par exemple, ici, pour une association entre tâches et étiquettes (classe `Tag` du modèle de données), on obtient une table dont les seules colonnes sont les clés étrangères des deux tables `todo` et `tag`, qui constituent une clé primaire composite.

3.4 Programmer la modifications des données

Une application Symfony effectue en général des modifications des données, qu'il faut donc sauvegarder, avant l'arrêt de l'exécution.
Voyons comment on programme cela dans notre code PHP.

3.4.1 Objet + ORM

Sauvegarder les ajouts / suppressions / modifications
Deux étapes :

1. la création, modification ou suppression d'instances, **en mémoire**
2. **la synchronisation** entre le nouvel état obtenu et le contenu de la BD

C'est le rôle de l'*entity manager* (em) de l'ORM Doctrine d'effectuer cette synchronisation.

L'ORM sait charger les données, et gérer les modifications également.
On supporte alors l'ensemble des opérations CRUD sur les données du modèle de l'application :

— *Create* (création)
— *Request* / *Retrieve* (chargement)
— *Update* (modification)
— *Delete* (suppression)

On effectue des manipulations en codant en PHP objet, encore une fois.

3.4.2 Création et sauvegarde d'une nouvelle instance

```
public function createAction(ManagerRegistry $doctrine)
{
        $project = new Project();
        $project->setTitle('CSC4101');
        $project->setDescription("Architectures et applications Web");

        // entity manager
        $entityManager= $doctrine->getManager();
```

```
        // indique à Doctrine que vous aimeriez éventuellement
        // sauvegarder ce Projet (mais pas encore de requête)
        $entityManager->persist($project);

        // exécute effectivement la sauvegarde (ex. la requête INSERT)
        $entityManager->flush();

        // L'identifiant est enfin renseigné (généré par le SGBD)
        return new Response("Sauvegardé le projet d'id ".
                                                $project->getId());
}
```

`persist()` permet de « tagger » des instances qui ont été modifiées : nouvelles, modifiées ou à supprimer.
Les instances ont été modifiées en mémoire seulement, pour l'instant. Elles sont en attente de sauvegarde.
Ce n'est qu'à l'exécution du `flush()` que Doctrine effectuera réellement les requêtes de suppression effectives (en SQL), en parcourant toutes les données ainsi « taggées » et en vérifiant ce qui doit être fait : INSERT, UPDATE, ou DELETE.

3.4.3 *setter* pour collection *...ToMany*

```
class Project
{
        #[ORM\OneToMany(targetEntity: Todo::Class, mappedBy: 'project')]
        private $todos;

        public function addTodo(Todo $todo) {
                if (!$this->todos->contains($todo)) {
                        $this->todos->add($todo);
                        $todo->setProject($this);
                }
                //...
        }

        public function removeTodo(Todo $todo) {
                if ($this->todos->contains($todo)) {
                        $this->todos->removeElement($todo);
                        // set the owning side to null
                        // (unless already changed)
                        if ($todo->getProject() === $this) {
                                $todo->setProject(null);
                        }
                }
```

Le code ci-dessus est celui généré par `make:entity`.
Pour une collection d'instances liées comme `Project::todo`, il y a deux opérations de modifications qu'on effectue classiquement : ajout ou suppression d'un élément dans la collection.

3.4.4 Qui sauvegarder à l'ajout, pour les entités liées ?

`OneToMany` entre `Project` et `Todo`

```
class Project
{
          public function addTodo(Todo $todo): self
          {
                  if (!$this->todos->contains($todo)) {
                          $this->todos->add($todo);
                          $todo->setProject($this);
                  }
                  return $this;
```

```
        }
```

Attention : qui est modifié (pour le `persist()` ultérieur) ?

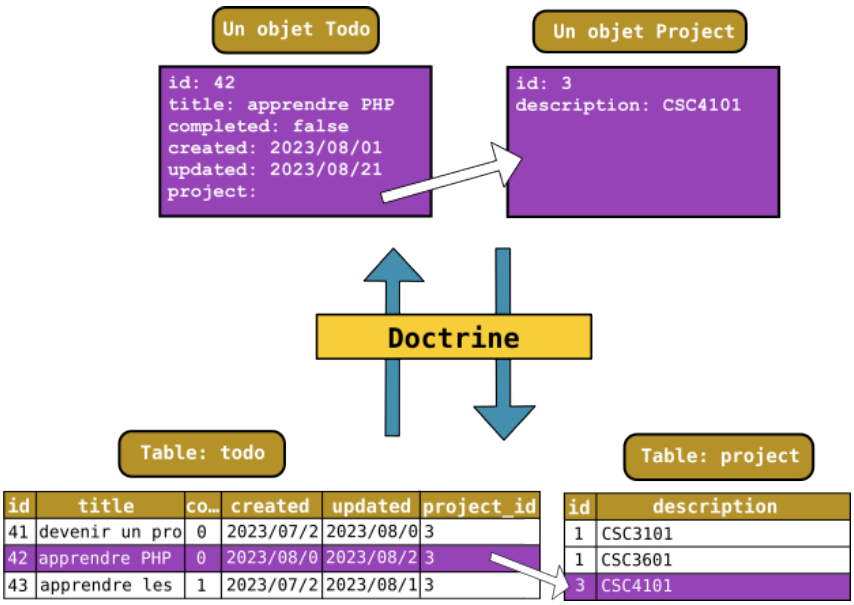


Figure 5 – Rappel du mapping d'une référence 1-N

Supposons qu'on a ajouté une nouvelle `Todo` à un `Project`.
Le projet déjà présent a-t-il besoin d'être modifié, dans la base de données ?
La collection des *Todos* du *Project, une relation /OneToMany,* est gérée en mémoire via `Project::todos`.
À l'ajout en mémoire, cette collection est bien modifiée.
Mais par rapport à la base de données relationnelle, il s'agit d'une propriété multi-valuée calculée, qui n'est pas stockée en base de données (cf. diagramme ci-dessus).
L'instance de projet présente en base n'est effectivement pas modifié : dans le schéma relationnel, ce sont les Todos contiennent une référence à leur projet.
Le `persist()` devra donc opérer sur la Todo.
Il sera donc inutile de faire un `persist()` sur l'instance du `Project`.

3.4.5 Suppression pour les entités liées

Code généré par l'assistant `make:entity` :

```php
class Project
{
        public function removeTodo(Todo $todo): self
        {
                if ($this->todos->contains($todo)) {
                        $this->todos->removeElement($todo);
                        // set the owning side to null
                        // (unless already changed)
                        if ($todo->getProject() === $this) {
                                $todo->setProject(null);
                        }
                }
                return $this;
        }
```

Persist ?
Possibilité gestion automatique des associations

Comme dans le cas de l'ajout, pour la suppression, on peut se demander également quelles instances modifiées sont effectivement concernées par le marquage dy `persist()`.
Heureusement, il existe un moyen d'automatiser les sauvegardes.

3.4.6 Propagation du `persist()` en cascade

Propagation en cascade :

```
#[OneToMany(... cascade: ['persist', 'remove'] ...)]
```

```
$todo = new Todo();
$project->addTodo($todo)

$entityManager->persist($project);
```

Listing 5 : Exemple de code

Sauvegarde de ses `todos` modifiés

Si on définit une option `cascade` pour l'attribut `OneToMany` de Doctrine, on peut alors gérer les sauvegardes en cascade, par transitivité, en parcourant les collections automatiquement. De même en cas de suppression.

3.4.7 Suppression des instances orphelines

Propriété orphanRemoval :

```
#[OneToMany(... cascade: ['persist'], orphanRemoval: true)]
```

```
$todo = ...
$todo->setProject(null);

$entityManager->persist($todo);
```

Listing 6 : Exemple de code

Suppression en base
En cas de doute : vérifier les requêtes générées (dans l'outil Doctrine de la barre d'outils Symfony, ou dans les *logs*)

De plus, si on définit l'option `orphanRemoval` pour l'attribut `OneToMany` on obtient une suppression automatique des instances d'entités faibles n'ayant plus d'association vers l'entité forte.
De façon générale, on testera le code du modèle de données avec précaution pour éviter d'oublier des attributs Doctrine, ce qui risquerait d'entrainer des bugs.

4 Gérer des données de tests

Examinons l'utilitaire des *DataFixtures* Doctrine qui permet de tester le code du modèle de données et de tester les fonctions de l'application sur des jeux de données de tests.

4.1 Initialiser la base avec données de tests

Coder une classe utilitaire *DataFixtures* pour Doctrine
Exemple :

— Chargement dans la base de données :

```php
private function loadProjects(ObjectManager $manager)
{
        foreach ($this->getProjectsData() as [$title, $description]) {

          $project = new Project();

          $project->setTitle($title);
          $project->setDescription($description);

          $manager->persist($project);
        }
        $manager->flush();
}
```

Listing 7 : `src/DataFixtures/ProjectFixtures.php`

— Définition des données dans un générateur :

```php
private function getProjectsData()
{
          // project = [title, description];
          yield ['CSC4101', "Architectures et applications Web"];
          yield ['CSC4102', "Introduction au Génie Logiciel Orienté Objet"];
}
```

Rôle de `yield` :

```php
$generator = $this->getProjectsData();

foreach($generator as $a) {
    print_r($a);
}
```

```
Array
(
    [0] => CSC4101
    [1] => Architectures et applications Web
)
Array
(
    [0] => CSC4102
    [1] => Introduction au Génie Logiciel Orienté Objet
)
```

L'instruction PHP `yield` permet de définir un tableau de n-uplets, qui seront renvoyés successivement en valeur de retour lors de l'appel à une méthode génératrice.

4.2 Lancer le chargement depuis la ligne de commande

À refaire à chaque recréation de la base de données dans l'environnement de développement :

```
$ symfony console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue y/N ?y
  > purging database
  > loading App\DataFixtures\ProjectFixtures
```

Les fixtures ne servent que pour initialiser des données de test, pour un administrateur de l'application ou un développeur.

Take Away

— ORM : Entités : classes, objets -> schéma relationnel
— ORM : Chargement des objets en mémoire
— Gestion des données liées dans les associations
— Programmer les modifications synchronisées dans la BD
— Outils de génération de base de données relationnelle
— Outil de données de tests *Fixtures*

Postface

Crédits illustrations et vidéos

— illustrations mapping Doctrine empruntées à la documentation Symfony

Copyright