



Simon GAUTIER



Doctrine

Relations entre entités

Concepts généraux

Relations unidirectionnelles One-To-One / Many-To-One / Many-To-Many

Les fixtures

Relations bidirectionnelles

Utiliser les jointures pour moins de requêtes

Concepts généraux

Intérêt des relations entre entités

- Créer des entités ne suffit pas
 - Souvent, nécessité de les lier entre elles
 - Exemple : Entité « Véhicule » \Leftrightarrow Entité « Modèle »
- Il existe plusieurs types de relations :
 - OneToOne (1:1)
 - Exemple : une session est liée à un et un seul client, un client peut avoir une et une seule session
 - ManyToOne (1:N)
 - Exemple : un client peut passer plusieurs commandes, une commande est liée à un et un seul client
 - ManyToMany (N:M)
 - Exemple : un produit peut être lié à plusieurs catégories, une catégorie peut contenir plusieurs produits

Concepts généraux

Notion de propriétaire et d'inverse

- Dans une relation entre 2 entités, une est dite « propriétaire », l'autre est dite « inverse »
- Exemple : lien entre client et commande
 - Un client peut avoir plusieurs commandes
 - Une commande n'est liée qu'à un client ➔ elle a au niveau de la base de données une clé étrangère (ex : customer_id) pour définir le client lié
 - Dans ce cas : la commande est l'entité propriétaire, le client est l'entité inverse
- Avec Doctrine : on ne crée pas « à la main » les clés étrangères
 - C'est Doctrine qui gère les relations qu'on lui décrit, on reste au niveau « objet »

Concepts généraux

Unidirectionnalité / Bidirectionnalité

- Relation unidirectionnelle :
 - Il est possible pour l'entité propriétaire de récupérer l'entité inverse
 - Exemple (commande / client) : **`$order->getCustomer()`**
- Relation bidirectionnelle :
 - Il est en plus possible pour l'entité inverse de récupérer les entités inverses
 - Exemple (commande / client) : **`$customer->getOrders()`**
- Si on travaille avec des relations unidirectionnelles, il est toutefois possible d'avoir une méthode qui retourne les entités inverses
 - On le fait soi-même dans le Repository (via une jointure)

Concepts généraux

Fonctionnement de Doctrine

- Doctrine travaille en « lazy loading » (chargement fainéant) :
 - Lorsqu'il charge une entité, il ne charge pas ses entités associées
 - Permet d'éviter de (gros) problèmes de performance car de proche en proche on peut charger beaucoup de données pour une seule entité demandée au départ !
- Si l'on souhaite charger par exemple un client avec ses commandes :
 - Soit on charge l'entité Client puis on appelle
« **\$customer->getOrders()** »
 - Dans ce cas, 2 requêtes distinctes sont jouées
 - Soit on demande à Doctrine de charger les 2 « en même temps » (dans une méthode de chargement d'entité que le développeur doit écrire...)
 - Via une jointure ➔ mieux pour les performances et le nombre de requêtes SQL

Relation One-To-One unidirectionnelle

- Rappel : One-to-One = relation unique entre 2 objets
- Chapitre précédent, l'entité « Vehicle » a été créée. Supposons que les équipements de sécurité sont stockés dans une autre entité « VehicleSecurity »
 - Un Vehicle n'a qu'un VehicleSecurity associé et inversement ➔ Relation One-To-One
- Via la console, création de l'entité VehicleSecurity :
 - Pour l'instant, c'est une entité « normale »

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\VehicleSecurityRepository")
 */
class VehicleSecurity
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @var int
     * @ORM\Column(type="integer")
     */
    private $euroNcapStars;

    /**
     * @var
     * @ORM\Column(type="integer")
     */
    private $airbagNumber;

    /**
     * @var boolean
     * @ORM\Column(type="boolean")
     */
    private $abs;

    /**
     * @var boolean
     * @ORM\Column(type="boolean")
     */
    private $esp;
}
```

Relation One-To-One

Une annotation pour définir la relation

- L'annotation est à ajouter sur l'entité propriétaire
 - On choisit Vehicle dans notre cas (plus logique mais on aurait pu choisir VehicleSecurity)
 - Rien à ajouter dans l'entité inverse (VehicleSecurity) si relation unidirectionnelle
 - On pourra faire `$vehicle->getVehicleSecurity()` mais pas `$vehicleSecurity->getVehicle()`
 - VehicleSecurity ne « sait » même pas qu'elle est liée à une autre entité
- L'annotation se fait sur une nouvelle propriété de l'entité
 - Interdit de le faire sur une propriété ayant déjà l'annotation `@ORM\Column`
 - Ajouter la propriété en « private » et lui associer l'annotation `@ORM\OneToOne`
 - Point important : **targetEntity** correspond au namespace complet de l'entité inverse

```
/**
 * @var VehicleSecurity
 * @ORM\OneToOne(targetEntity="App\Entity\VehicleSecurity", cascade={"persist"})
 */
private $vehicleSecurity;
```

- Générer le getter / setter dans PHPStorm
- Depuis la console : migrer la base de données (ajout d'une table + une colonne en clé étrangère et avec la contrainte UNIQUE)

Relation One-To-One

Relation facultative / obligatoire

- Par défaut, une relation One-To-One est facultative
 - Une entité Vehicle peut ne pas avoir de VehicleSecurity lié
- Pour rendre la relation obligatoire :
 - Ajouter une annotation **JoinColumn** :

```
/**
 * @var VehicleSecurity
 * @ORM\OneToOne(targetEntity="App\Entity\VehicleSecurity", cascade={"persist"})
 * @ORM\JoinColumn(nullable=false)
 */
private $vehicleSecurity;
```

- Relancer une mise à jour de la base de données
 - Toujours avec la console → attention : nécessite que chaque Vehicle ait un VehicleSecurity sinon la mise à jour ne pourra pas se faire

Relation One-To-One

Propriété « **cascade** »

- **cascade** : propager une opération de l'entité propriétaire Vehicle vers l'entité inverse VehicleSecurity
- **persist** : permet de propager la demande de persistance à l'entité liée si un **persist()** est fait sur l'entité propriétaire
 - Evite de faire : `$v->persist()` puis `$v->getVehicleSecurity()->persist()`
 - A la place : `$v->persist()` suffit et propage en cascade l'ordre
- **remove** : permet de la même manière de propager la suppression à l'entité liée si un **remove()** est fait sur l'entité propriétaire

```
@ORM\OneToOne(targetEntity="App\Entity\VehicleSecurity", cascade={"persist", "remove"})
```

➔ Dans l'exemple donné, un VehicleSecurity n'a pas de sens sans le Vehicle associé ➔ **remove** est donc utile

- D'autres opérateurs de cascade existent (cf. doc)
 - docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/working-with-associations.html#transitive-persistence-cascade-operations
- Remarque : la cascade définie n'a aucun lien avec un **CASCADE** du SGBD

Relation One-To-One

Clé étrangère ?

- Au niveau de l'entité, on définit seulement une propriété qui correspond à l'entité liée (c'est donc un objet)
 - Doctrine, lors de la mise à jour de la base de données, crée une clé étrangère dans la table de l'entité principale qui pointe vers la table de l'entité inverse
 - La clé étrangère N'EST PAS une propriété de l'entité
 - Pour accéder à cette « clé étrangère » depuis l'entité principale, il faut passer par l'entité inverse qui est la seule accessible depuis l'entité principale
- OUI : `$v->getVehicleSecurity()->getId()`
- NON : `$v->getVehicleSecurityId()`

```

public function testAddRelationOneToOneAction(EntityManagerInterface $em)
{
    $v = new Vehicle();
    $v->setPlate('AB-123-CD')->setMileage(92500)->setPrice(17000)
        ->setManufactureDate(new \DateTime('2015-01-02'))->setDescription('En bon état');

    $vs = new VehicleSecurity();
    $vs->setAbs(true)->setEsp(false)->setAirbagNumber(4)->setEuroNcapStars(4);

    $v->setVehicleSecurity($vs); // Affectation de l'objet VehicleSecurity à l'objet propriétaire de la relation

    $em->persist($v); // Il n'est pas nécessaire de persister l'entité liée grâce à la propriété cascade
    $em->flush();
    return new Response('<body></body>');
}

```

Exemple d'ajout

| | | | |
|---|----------|--|---|
| 1 | 0.05 ms | "START TRANSACTION" | Parameters: [] View formatted query View runnable query Explain query |
| 2 | 8.56 ms | INSERT INTO vehicle_security (euro_ncap_stars, airbag_number, abs, esp) VALUES (?, ?, ?, ?) | Parameters: [1 => 4 2 => 4 3 => 1 4 => 0] View formatted query View runnable query Explain query |
| 3 | 0.17 ms | INSERT INTO vehicle (plate, mileage, price, description, manu_date, vehicle_security_id) VALUES (?, ?, ?, ?, ?, ?) | Parameters: [1 => "AB-123-CD" 2 => 92500 3 => 17000.0 4 => "En bon état" 5 => "2015-01-02 00:00:00" 6 => 2] View formatted query View runnable query Explain query |
| 4 | 30.89 ms | "COMMIT" | Parameters: [] |

```

public function testUpdateRelationOneToOneAction(EntityManagerInterface $em)
{
    // Chargement de l'entité
    /** @var Vehicle $v */
    $v = $em->find('App:Vehicle', 2);
    $v->setMileage(93000);
    $v->getVehicleSecurity()->setAirbagNumber(6);

    // Pas besoin de persist car Doctrine connaît les entités
    $em->flush();
    return new Response('<body></body>');
}

```

Exemple de modification

| | | |
|---|----------|---|
| 1 | 2.41 ms | SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7 FROM vehicle t0 WHERE t0.id = ? Parameters: [▼ 2] View formatted query View runnable query Explain query |
| 2 | 0.23 ms | SELECT t0.id AS id_1, t0.euro_ncap_stars AS euro_ncap_stars_2, t0.airbag_number AS airbag_number_3, t0.abs AS abs_4, t0.esp AS esp_5 FROM vehicle_security t0 WHERE t0.id = ? Parameters: [▼ 2] View formatted query View runnable query Explain query |
| 3 | 0.03 ms | "START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query |
| 4 | 15.29 ms | UPDATE vehicle_security SET airbag_number = ? WHERE id = ? Parameters: [▼ 6 2] View formatted query View runnable query Explain query |
| 5 | 0.12 ms | UPDATE vehicle SET mileage = ? WHERE id = ? Parameters: [▼ 93000 2] View formatted query View runnable query Explain query |
| 6 | 17.30 ms | "COMMIT" Parameters: [] |

Relation Many-To-One unidirectionnelle

- Rappel : Many-to-One = lie une entité A à plusieurs entités B
- On peut imaginer dans la continuité des exemples avoir une entité « VehicleModel » qui peut contenir plusieurs entités « Vehicle »
 - Exemple de modèle : Clio, 308, ...
 - Via la console, création de l'entité puis ajout des champs métiers (pour simplifier, on imagine que la marque est juste un champ texte mais elle pourrait très bien avoir sa propre entité !)

```
class VehicleModel
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @var string
     * @ORM\Column()
     */
    private $name;

    /**
     * @var string
     * @ORM\Column()
     */
    private $make;
}
```

Relation Many-To-One

Annotation

- Quelle est l'entité propriétaire ?
 - On serait tenté de dire : « plusieurs véhicules appartiennent au même modèle, modèle est donc propriétaire »
 - Mais en fait, chaque entité « Vehicle » est liée à un « VehicleModel » → c'est donc elle qui « connaît » son modèle unique et qui contiendra en base de données la colonne référence
→ Vehicle est propriétaire
 - Retenir que dans Many-To-One, Many est propriétaire → Vehicle Many-To-One VehicleModel
 - Cascade ?
 - Si je supprime un véhicule, est-ce que je veux supprimer le modèle lié ? non
 - Si je persiste un véhicule, est-ce que le modèle lié doit aussi être persisté ? Pas très intéressant, disons non
 - Relation facultative ? Oui, je veux autoriser le fait qu'un Vehicle puisse ne pas avoir de VehicleModel
- J'ai répondu à tout, je crée mon annotation dans Vehicle
- Puis génération du nouveau getter / setter
- Enfin, avec la console, mise à jour la structure de la base de données

```
/**
 * @var VehicleModel
 * @ORM\ManyToOne(targetEntity="App\Entity\VehicleModel")
 */
private $model;
```

```

public function testAddRelationManyToOneAction(EntityManagerInterface $em)
{
    $m = new VehicleModel();
    $m->setName('208')->setMake('Peugeot');

    $v = new Vehicle();
    $v->setModel($m)->setPlate('AA-123-BB')->setMileage(45000)->setPrice(9800)->setManufactureDate(new \DateTime('2014-03-02'));
    $v2 = new Vehicle();
    $v2->setModel($m)->setPlate('BB-123-CC')->setMileage(1000)->setPrice(15900)->setManufactureDate(new \DateTime('2017-08-02'));

    // Il faut persister les 3 entités car aucune
    // cascade n'a été définie
    $em->persist($v);
    $em->persist($v2);
    $em->persist($m);
    $em->flush();
    return new Response('<body></body>');
}

```

| | | | |
|---|----------|---|--|
| 1 | 0.04 ms | "START TRANSACTION" | Parameters: [] View formatted query View runnable query Explain query |
| 2 | 9.90 ms | INSERT INTO vehicle_model (name, make) VALUES (?, ?) | Parameters: [1 => "208" 2 => "Peugeot"] View formatted query View runnable query Explain query |
| 3 | 0.15 ms | INSERT INTO vehicle (plate, mileage, price, description, manu_date, vehicle_security_id, model_id) VALUES (?, ?, ?, ?, ?, ?, ?) | Parameters: [1 => "AA-123-BB" 2 => 45000 3 => 9800.0 4 => null 5 => "2014-03-02 00:00:00" 6 => null 7 => 4] View formatted query View runnable query Explain query |
| 4 | 0.02 ms | INSERT INTO vehicle (plate, mileage, price, description, manu_date, vehicle_security_id, model_id) VALUES (?, ?, ?, ?, ?, ?, ?) | Parameters: [1 => "BB-123-CC" 2 => 1000 3 => 15900.0 4 => null 5 => "2017-08-02 00:00:00" 6 => null 7 => 4] View formatted query View runnable query Explain query |
| 5 | 27.83 ms | "COMMIT" | Parameters: [] |

Exemple d'ajout

Remarque : pour l'exemple, changements effectués :

- Rendre facultative la relation avec VehicleSecurity (suppression du nullable false)
- Rendre facultative la saisie de la description (ajout d'un nullable true)

Relation Many-To-One

Ajout d'un nouveau véhicule pour un modèle existant

```
public function testAddRelationManyToOneBisAction(EntityManagerInterface $em)
{
    $m = $em->find('App:VehicleModel', 4);

    $v = new Vehicle();
    $v->setModel($m)->setPlate('AB-123-AB')->setMileage(32900)->setPrice(7890)->setManufactureDate(new \DateTime('2016-03-02'));

    $em->persist($v);
    $em->flush();
    return new Response('<body></body>');
}
```

| | | |
|---|----------|---|
| 1 | 1.83 ms | SELECT t0.id AS id_1, t0.name AS name_2, t0.make AS make_3 FROM vehicle_model t0 WHERE t0.id = ? Parameters: [4] View formatted query View runnable query Explain query |
| 2 | 0.04 ms | "START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query |
| 3 | 12.37 ms | INSERT INTO vehicle (plate, mileage, price, description, manu_date, vehicle_security_id, model_id) VALUES (?, ?, ?, ?, ?, ?, ?) Parameters: [1 => "AB-123-AB" 2 => 32900 3 => 7890.0 4 => null 5 => "2016-03-02 00:00:00" 6 => null 7 => 4] View formatted query View runnable query Explain query |
| 4 | 20.30 ms | "COMMIT" Parameters: [] |

Relation Many-To-One

Récupération de tous les véhicules d'un modèle

```
public function testLoadRelationManyToOneAction(EntityManagerInterface $em)
{
    $m = $em->find('App:VehicleModel', 4);
    // Remarque : findByModel() n'existe pas ==> rappel : méthode magique
    $vehicles = $em->getRepository('App:Vehicle')->findByModel($m);
    dump($vehicles);
    return new Response('<body></body>');
}
```

| | | |
|---|---------|--|
| 1 | 4.54 ms | <pre>SELECT t0.id AS id_1, t0.name AS name_2, t0.make AS make_3 FROM vehicle_model t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[4]</pre> <p>View formatted query View runnable query Explain query</p> |
| 2 | 0.50 ms | <pre>SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.model_id = ?</pre> <p>Parameters:</p> <pre>[4]</pre> |

Relation Many-To-Many unidirectionnelle

- Rappel : Many-to-Many = une entité A peut être liée à plusieurs entités B et inversement
- Créons pour l'exemple une entité « VehicleEquipment » qui sera liée à l'entité « Vehicle ». Un véhicule peut avoir plusieurs équipements (ex : climatisation), un équipement peut être présent sur plusieurs véhicules
- Via la console, création de l'entité VehicleEquipment puis ajout des champs métiers :

```
class VehicleEquipment
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @var string
     * @ORM\Column()
     */
    private $name;

    /**
     * @var string
     * @ORM\Column(type="text")
     */
    private $description;
}
```

Relation Many-To-Many unidirectionnelle

Annotation

- Quelle est l'entité propriétaire ?
 - Nous avons le choix ! On choisit la plus « pratique »
 - On cherchera plus souvent à savoir quelles sont les équipements d'un véhicule plutôt que les véhicules ayant un équipement → Vehicle est propriétaire
 - Cascade ?
 - Si je supprime un véhicule, est-ce que je veux supprimer les équipements liés ? non
 - Si je persiste un véhicule, est-ce que l'équipement lié doit aussi être persisté ? Pas très intéressant, disons non
 - Relation facultative ? Ça n'a pas de sens en Many-To-Many !
- ➔ J'ai répondu à tout, je crée mon annotation dans Vehicle
- Important** : la propriété que je crée renverra plusieurs équipements ➔ **\$equipments**
- Puis, avec la console, je mets à jour la structure de la base de données

```
/**
 * @var PersistentCollection
 * @ORM\ManyToMany(targetEntity="App\Entity\VehicleEquipment")
 */
private $equipments;
```

Relation Many-To-Many unidirectionnelle

Setter / Constructeur

- Etant donné qu'on manipule une collection, le setter classique est remplacé par 2 méthodes :
 - **addEquipment()**
 - **removeEquipment()**
- Il faut penser à initialiser dans le constructeur la collection
- Aucun changement sur le getter

```
public function __construct()  
{  
    $this->equipments = new ArrayCollection();  
    $this->manufactureDate = new \DateTime();  
}
```

```
/**  
 * @return PersistentCollection  
 */  
public function getEquipments(): PersistentCollection  
{  
    return $this->equipments;  
}  
  
/**  
 * @param VehicleEquipment $vehicleEquipment  
 * @return Vehicle  
 */  
public function addEquipment(VehicleEquipment $vehicleEquipment): Vehicle  
{  
    $this->equipments->add($vehicleEquipment);  
    return $this;  
}  
  
/**  
 * @param VehicleEquipment $vehicleEquipment  
 * @return Vehicle  
 */  
public function removeEquipment(VehicleEquipment $vehicleEquipment): Vehicle  
{  
    $this->equipments->removeElement($vehicleEquipment);  
    return $this;  
}
```

Relation Many-To-Many unidirectionnelle

Structure générée en base de données

- Une relation Many-To-Many génère par défaut une table d'association entre les 2 entités concernées :
 - Colonne 1 : **vehicle_id**, clé étrangère vers la table correspondant à l'entité Vehicle
 - Colonne 2 : **vehicle_equipment_id**, clé étrangère vers la table correspondant à l'entité VehicleEquipment
 - Clé primaire : les 2 colonnes
- Le nom par défaut de la table est la concaténation des noms des 2 entités, **vehicle_vehicle_equipment** dans notre cas
 - Possibilité de personnaliser le nom :

```
/**  
 * @var PersistentCollection  
 * @ORM\ManyToMany(targetEntity="App\Entity\VehicleEquipment")  
 * @ORM\JoinTable(name="asso_vehicle_equipment")  
 */  
private $equipments;
```

Petite parenthèse – les fixtures

Installer : **composer req orm-fixtures**

- Les fixtures permettent d'ajouter des données dans la base
 - Ex : initialiser des jeux de données pour les entités de notre choix (cf. slide suivante)
- Les fichiers de fixtures sont à placer dans le répertoire **src/DataFixtures**
 - Il est possible d'avoir plusieurs fichiers de Fixture
 - Nom des classes arbitraire, il faut juste étendre de la classe **\Doctrine\Bundle\FixturesBundle\Fixture**
- Exécuter le chargement des fixtures via la console :
 - **php bin/console doctrine:fixtures:load**
 - Attention : la base de données est vidée, puis les fixtures sont ajoutées
 - **php bin/console doctrine:fixtures:load --append**
 - L'option **--append** permet d'ajouter les fixtures sans vider la base de données

```

class MyAppData extends Fixture
{
    /**
     * Load data fixtures with the passed EntityManager
     *
     * @param ObjectManager $manager
     */
    public function load(ObjectManager $manager)
    {
        $model = new VehicleModel();
        $model->setMake('Renault')->setName('Clio');
        $manager->persist($model);

        $model2 = new VehicleModel();
        $model2->setMake('Peugeot')->setName('308');
        $manager->persist($model2);

        $vehicleEquipment = new VehicleEquipment();
        $vehicleEquipment->setName('Climatisation')->setDescription('');
        $manager->persist($vehicleEquipment);

        $vehicleEquipment2 = new VehicleEquipment();
        $vehicleEquipment2->setName('Régulateur')->setDescription('');
        $manager->persist($vehicleEquipment2);

        $vehicleEquipment3 = new VehicleEquipment();
        $vehicleEquipment3->setName('Stop and start')->setDescription('');
        $manager->persist($vehicleEquipment3);

        $vehicleSecurity = new VehicleSecurity();
        $vehicleSecurity->setAirbagNumber(4)->setEuroNcapStars(4)->setEsp(1)->setAbs(0);
        $manager->persist($vehicleSecurity);

        $vehicle = new Vehicle();
        $vehicle->setModel($model)->setMileage(55089)->setManufactureDate(new \DateTime('2016-05-06'))
            ->setPlate('AA-123-BB')->setPrice('10500')->setDescription('Bon véhicule !')
            ->setVehicleSecurity($vehicleSecurity)->addEquipment($vehicleEquipment)->addEquipment($vehicleEquipment3);
        $manager->persist($vehicle);

        $manager->flush();
    }
}

```

Relation Many-To-Many unidirectionnelle

Exemple d'ajout

```
public function testAddRelationManyToManyAction(EntityManagerInterface $em)
{
    // Création d'un véhicule ayant tous les équipements existants :
    $equipments = $em->getRepository('App:VehicleEquipment')->findAll();
    $vehicle = new Vehicle();
    $vehicle->setPrice(6700)->setPlate('AZ-123-BB')->setMileage(159000);
    foreach ($equipments as $equipment) {
        $vehicle->addEquipment($equipment);
    }
    $em->persist($vehicle);
    $em->flush();
    return new Response('<body></body>');
}
```

| | | |
|---|----------|---|
| 1 | 1.45 ms | SELECT t0.id AS id_1, t0.name AS name_2, t0.description AS description_3 FROM vehicle_equipment t0 Parameters: [] View formatted query View runnable query Explain query |
| 2 | 0.03 ms | "START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query |
| 3 | 15.64 ms | INSERT INTO vehicle (plate, mileage, price, description, manu_date, vehicle_security_id, model_id) VALUES (?, ?, ?, ?, ?, ?, ?) Parameters: [1 => "AZ-123-BB" 2 => 159000 3 => 6700.0 4 => null 5 => "2018-01-03 15:56:46" 6 => null 7 => null] View formatted query View runnable query Explain query |
| 4 | 1.17 ms | INSERT INTO asso_vehicle_equipment (vehicle_id, vehicle_equipment_id) VALUES (?, ?) Parameters: [2 1] View formatted query View runnable query Explain query |
| 5 | 0.07 ms | INSERT INTO asso_vehicle_equipment (vehicle_id, vehicle_equipment_id) VALUES (?, ?) Parameters: [2 2] View formatted query View runnable query Explain query |
| 6 | 0.05 ms | INSERT INTO asso_vehicle_equipment (vehicle_id, vehicle_equipment_id) VALUES (?, ?) Parameters: [2 3] View formatted query View runnable query Explain query |
| 7 | 29.94 ms | "COMMIT" Parameters: [] View formatted query View runnable query Explain query |

Relation Many-To-Many unidirectionnelle

Exemple de suppression

```
public function testRemoveRelationManyToManyAction(EntityManagerInterface $em)
{
    $vehicle = $em->find('App:Vehicle', 2);
    foreach ($vehicle->getEquipments() as $key => $equipment) {
        if($key % 2 == 0) {
            continue; // Ne supprimer qu'un équipement sur 2...
        }
        $vehicle->removeEquipment($equipment);
    }
    $em->flush();
    return new Response('<body></body>');
}
```

| | | |
|---|----------|--|
| 1 | 1.84 ms | SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.id = ? Parameters: [▼ 2] View formatted query View runnable query Explain query |
| 2 | 0.24 ms | SELECT t0.id AS id_1, t0.name AS name_2, t0.description AS description_3 FROM vehicle_equipment t0 INNER JOIN asso_vehicle_equipment ON t0.id = asso_vehicle_equipment.vehicle_equipment_id WHERE asso_vehicle_equipment.vehicle_id = ? Parameters: [▼ 2] View formatted query View runnable query Explain query |
| 3 | 0.04 ms | "START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query |
| 4 | 8.46 ms | DELETE FROM asso_vehicle_equipment WHERE vehicle_id = ? AND vehicle_equipment_id = ? Parameters: [▼ 2 2] View formatted query View runnable query Explain query |
| 5 | 18.16 ms | "COMMIT" Parameters: [] |

Relation Many-To-Many avec attributs

- Pourquoi une telle relation ?
 - Une relation Many-To-Many classique peut dans certains cas être insuffisante
 - Ex : si nous ajoutons une entité réparation qu'on souhaite lier à l'entité véhicule, où stocker la date de la réparation
 - La date de réparation n'est pas un attribut du véhicule
 - La date de réparation n'est pas un attribut de la réparation (on y stocke que la nature de l'intervention, le prix, ...)
 - ➔ La date de réparation est un attribut de l'association entre les 2 entités
- Doctrine ne sait pas gérer ce type de relation nativement
 - Il faut la gérer « manuellement »
 - Comment ? En créant une entité de liaison entre les 2 entités existantes
 - Cette entité de liaison aura comme attribut la date de réparation
 - L'entité de liaison sera liée en Many-To-One aux véhicules
 - L'entité de liaison sera liée en Many-To-One aux réparations
 - ➔ On aura donc : véhicule One-To-Many liaison Many-To-One réparation
- Retenir : Many-To-Many avec attributs = 2 relations Many-To-One

Relation Many-To-Many avec attributs

Création des entités

- Comme d'habitude :
 - Via la console
 - Ajout des propriétés
 - Ajout getters / setters
 - Mise à jour structure base de données

```
class VehicleToVehicleRepairs
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @var \DateTime
     * @ORM\Column(type="datetime")
     */
    private $date;

    // Remarque : possible d'ajouter autant d'attributs qu'on le souhaite
}
```

```
class VehicleRepairs
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @var string
     * @ORM\Column(type="text")
     */
    private $description;

    /**
     * @var float
     * @ORM\Column(type="float", precision=8, scale=2)
     */
    private $price;
}
```

Relation Many-To-Many avec attributs Annotations

- L'entité VehicleToVehicleRepairs est propriétaire des 2 relations Many-To-One → Ajout des 2 propriétés avec les annotations
 - Remarque : relations obligatoires (présence de `@ORM\JoinColumn(nullable=false)`)

```
/**
 * @var Vehicle
 * @ORM\ManyToOne(targetEntity="App\Entity\Vehicle")
 * @ORM\JoinColumn(nullable=false)
 */
private $vehicle;
```

```
/**
 * @var VehicleRepairs
 * @ORM\ManyToOne(targetEntity="App\Entity\VehicleRepairs")
 * @ORM\JoinColumn(nullable=false)
 */
private $vehicleRepairs;
```

- Et comme d'habitude :
 - Génération des getters / setters
 - Mise à jour de la structure de la base de données
- Enfin, possibilité d'utiliser les fixtures pour mettre des données

Relation Many-To-Many avec attributs

- L'entité de liaison VehicleToVehicleRepairs est propriétaire des 2 relations (logique car elle est Many-To-One des 2 côtés)
- Conséquences (rappel, relation unidirectionnelle) :
 - On pourra faire depuis cette entité `$link->getVehicle()`
 - On pourra faire depuis cette entité `$link->getVehicleRepairs()`
 - On ne pourra pas faire `$vehicle->getLinks()` depuis l'entité Vehicle
 - On ne pourra pas faire `$vehicleRepair->getLinks()` depuis l'entité VehicleRepairs

`getLinks()` retournant dans les 2 cas des collections d'entités VehicleToVehicleRepairs
- Solution de contournement avant de faire des relations bidirectionnelles : utiliser `findBy()` du Repository de l'entité de liaison :
 - Inconvénient : beaucoup de requêtes déclenchées si dans une boucle pour chaque Véhicule on souhaite récupérer les réparations associées → on verra comment faire mieux !

```
// Pour récupérer tous les liens d'un véhicule :
$em->getRepository('App:VehicleToVehicleRepairs')->findBy(array('vehicle' => $vehicle));

// Pour récupérer tous les liens d'une réparation :
$em->getRepository('App:VehicleToVehicleRepairs')->findBy(array('vehicleRepair' => $vehicleRepair));
```

Relation Many-To-Many avec attributs

Exemple d'ajout de relations

```
public function testAddRelationManyToManyAttributeAction(EntityManagerInterface $em)
{
    // Ajout de types de réparation
    $repairs = new VehicleRepairs();
    $repairs->setDescription('Forfait vidange')->setPrice(149);
    $em->persist($repairs);
    $repairs2 = new VehicleRepairs();
    $repairs2->setDescription('Forfait montage / équilibrage 4 pneus')->setPrice(49);
    $em->persist($repairs2);

    // Réparations sur un véhicule existant :
    $vehicle = $em->find('App:Vehicle', 1);

    // Ajout des associations
    $link = new VehicleToVehicleRepairs();
    $link->setDate(new \DateTime('2018-01-03'))->setVehicle($vehicle)->setVehicleRepairs($repairs);
    $em->persist($link);
    $link = new VehicleToVehicleRepairs();
    $link->setDate(new \DateTime('2018-01-04'))->setVehicle($vehicle)->setVehicleRepairs($repairs2);
    $em->persist($link);

    $em->flush();
    return new Response('<body></body>');
}
```

Relation Many-To-Many

- Requêtes exécutées

| | | |
|---|----------|--|
| 1 | 1.07 ms | <pre>SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[1]</pre> <p>View formatted query View runnable query Explain query</p> |
| 2 | 0.03 ms | <pre>"START TRANSACTION"</pre> <p>Parameters:</p> <pre>[]</pre> <p>View formatted query View runnable query Explain query</p> |
| 3 | 7.66 ms | <pre>INSERT INTO vehicle_repairs (description, price) VALUES (?, ?)</pre> <p>Parameters:</p> <pre>[1 => "Forfait vidange" 2 => 149.0]</pre> <p>View formatted query View runnable query Explain query</p> |
| 4 | 0.02 ms | <pre>INSERT INTO vehicle_repairs (description, price) VALUES (?, ?)</pre> <p>Parameters:</p> <pre>[1 => "Forfait montage / équilibrage 4 pneus" 2 => 49.0]</pre> <p>View formatted query View runnable query Explain query</p> |
| 5 | 0.12 ms | <pre>INSERT INTO vehicle_to_vehicle_repairs (date, vehicle_id, vehicle_repairs_id) VALUES (?, ?, ?)</pre> <p>Parameters:</p> <pre>[1 => "2018-01-03 00:00:00" 2 => 1 3 => 2]</pre> <p>View formatted query View runnable query Explain query</p> |
| 6 | 0.02 ms | <pre>INSERT INTO vehicle_to_vehicle_repairs (date, vehicle_id, vehicle_repairs_id) VALUES (?, ?, ?)</pre> <p>Parameters:</p> <pre>[1 => "2018-01-04 00:00:00" 2 => 1 3 => 3]</pre> <p>View formatted query View runnable query Explain query</p> |
| 7 | 20.94 ms | <pre>"COMMIT"</pre> <p>Parameters:</p> <pre>[]</pre> |

Relation Many-To-Many avec attributs

Exemple de chargement de relations

```
public function testLoadRelationManyToManyAttributeAction(EntityManagerInterface $em)
{
    $vehicle = $em->find('App:Vehicle', 1);

    // Chargement de tous les VehicleToVehicleRepairs associés à ce véhicule
    $links = $em->getRepository('App:VehicleToVehicleRepairs')->findBy(array('vehicle' => $vehicle));

    return $this->render('list_repairs.html.twig', ['vehicle' => $vehicle, 'links' => $links]);
}
```

```
<h1>{{ vehicle.plate }}</h1>
Le véhicule a {{ vehicle.mileage }} km.
{% if links|length %}
    <ul>
        {% for link in links %}
            {# Important : link.vehicleRepairs génère une requête à chaque itération car à ce stade, la réparation
              n'a pas été chargée ==> pas très efficace.
              A l'inverse, link.date ne génère pas de requête car la donnée est stockée dans l'association. #}
            <li>Réparation du {{ link.date|date('d/m/Y') }} : {{ link.vehicleRepairs.description }}</li>
        {% endfor %}
    </ul>
{% else %}
    Le véhicule n'a jamais été réparé
{% endif %}
```


Relation Many-To-Many avec attributs

Exemple de chargement de relations – requêtes

1 : chargement véhicule (côté contrôleur)

2 : chargement liens
VehicleToVehicleRepairs (côté contrôleur)

3 et 4 : requêtes déclenchées lors de l'accès aux données d'une réparation dans les itérations côté Twig

➔ N+2 requêtes si N réparations

| | | |
|---|---------|--|
| 1 | 0.90 ms | <pre>SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[1]</pre> <p>View formatted query View runnable query Explain query</p> |
| 2 | 0.15 ms | <pre>SELECT t0.id AS id_1, t0.date AS date_2, t0.vehicle_id AS vehicle_id_3, t0.vehicle_repairs_id AS vehicle_repairs_id_4 FROM vehicle_to_vehicle_repairs t0 WHERE t0.vehicle_id = ?</pre> <p>Parameters:</p> <pre>[1]</pre> <p>View formatted query View runnable query Explain query</p> |
| 3 | 0.22 ms | <pre>SELECT t0.id AS id_1, t0.description AS description_2, t0.price AS price_3 FROM vehicle_repairs t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[2]</pre> <p>View formatted query View runnable query Explain query</p> |
| 4 | 0.29 ms | <pre>SELECT t0.id AS id_1, t0.description AS description_2, t0.price AS price_3 FROM vehicle_repairs t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[3]</pre> |

- On va voir que pour améliorer les performances, on va écrire une méthode personnalisée dans le Repository de VehicleToVehicleRepairs qui chargera tout d'un coup (une requête au lieu de N+1) ➔ Dans le contrôleur, le **findBy()** sera remplacé par cette méthode à créer.
- On pourra de la même manière faire une méthode qui charge tout y compris le véhicule ➔ une requête au lieu de N+2

Relations bidirectionnelles

- Relations présentées : unidirectionnelles
 - Seule l'entité propriétaire de la relation était modifiée (annotation, ...)
- Relation bidirectionnelle
 - Permettre depuis l'entité inverse, d'accéder à l'entité propriétaire
 - Ex : lien Vehicle Many-To-One VehicleModel
 - On a vu comment faire `$vehicle->getModel()`
 - Il serait intéressant de pouvoir faire `$model->getVehicles()`
- Objectif : rendre la relation bidirectionnelle

Relations bidirectionnelles

- Etape 1 : ajouter une annotation dans l'entité inverse (**VehicleModel** dans notre cas) :

```
/**
 * @var PersistentCollection
 * @ORM\OneToMany(targetEntity="App\Entity\Vehicle", mappedBy="model")
 */
private $vehicles;
```

- Noter le « s » à **\$vehicles** → un modèle contient bien plusieurs véhicules
 - **@ORM\OneToMany** → L'inverse de Many-To-One !
 - **targetEntity="App\Entity\Vehicle"** → l'entité inverse
 - **mappedBy="model"** → la propriété en Many-To-One dans l'entité inverse
- Etape 2 : compléter l'annotation de l'entité propriétaire (**Vehicle** dans notre cas) :
 - Dire à la propriété de l'entité propriétaire qu'elle a un inverse (une propriété symétrique)

```
/**
 * @var VehicleModel
 * @ORM\ManyToOne(targetEntity="App\Entity\VehicleModel", inversedBy="vehicles")
 */
private $model;
```



Relations bidirectionnelles

- Etape 3 : générer le getter / setter de la nouvelle propriété de l'entité inverse (**VehicleModel** dans notre cas). Idem relation Many-To-Many, les méthodes à prévoir sont :
 - Un constructeur (initialise **\$vehicles** comme un **ArrayCollection** vide)
 - Un getter (classique)
 - Une méthode d'ajout et une méthode de suppression de véhicule

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

```
/**  
 * @return PersistentCollection  
 */  
public function getVehicles(): PersistentCollection  
{  
    return $this->vehicles;  
}
```

```
/**  
 * @param Vehicle $vehicle  
 * @return VehicleModel  
 */  
public function addVehicle(Vehicle $vehicle): VehicleModel  
{  
    $this->vehicles->add($vehicle);  
    return $this;  
}  
  
/**  
 * @param Vehicle $vehicle  
 * @return VehicleModel  
 */  
public function removeVehicle(Vehicle $vehicle): VehicleModel  
{  
    $this->vehicles->removeElement($vehicle);  
    return $this;  
}
```

Relations bidirectionnelles

Remarque : ce code nécessite l'ajout du « ? » pour éviter une erreur PHP

- Problématique à traiter :

```
$vehicle = new Vehicle();  
$model = new VehicleModel();  
$model->addVehicule($vehicle);  
dump($vehicle->getModel()); // null !!
```

```
/**  
 * @return VehicleModel  
 */  
public function getModel():? VehicleModel  
{  
    return $this->model;  
}
```

- Le **addVehicle()** n'a aucun effet sur la propriété symétrique **\$model**
 - Logique d'un point de vue PHP car le seul moyen d'initialiser la propriété privée **\$model** est d'utiliser le setter **setModel()**
- Solution (pas pratique car on risque d'oublier) :
 - Toujours appeler **setModel()** après un **addVehicle()**
 - Inversement, toujours utiliser **addVehicle()** après un **setModel()**

Relations bidirectionnelles

- Solution plus « sécurisée » : appeler l'une des méthodes depuis l'autre
 - Avec cette solution, l'exemple précédent fonctionne et affiche bien un objet VehicleModel

```
/**
 * @param Vehicle $vehicle
 * @return VehicleModel
 */
public function addVehicule(Vehicle $vehicle): VehicleModel
{
    $this->vehicles->add($vehicle);
    $vehicle->setModel($this); // Lier inversement le modèle au véhicule
    return $this;
}
```

```
/**
 * @param Vehicle $vehicle
 * @return VehicleModel
 */
public function removeVehicle(Vehicle $vehicle): VehicleModel
{
    $this->vehicles->removeElement($vehicle);
    $vehicle->setModel(null); // Instruction autorisée seulement si la propriété peut être null
    return $this;
}
```

```
/**
 * @param VehicleModel $model
 * @return Vehicle
 */
public function setModel(?VehicleModel $model): Vehicle
{
    $this->model = $model;
    return $this;
}
```

Remarque : nécessite l'ajout du « ? »
pour éviter une erreur PHP

Relations bidirectionnelles

- ATTENTION : on ne fait cet appel que dans l'entité propriétaire ou l'entité inverse mais pas les 2 ➔ Sinon, boucle infinie !
- Si on veut faire l'appel des 2 côtés, il faut ajouter un paramètre permettant de casser la boucle infinie ➔ Exemple : ajout d'un booléen en second paramètre facultatif faux par défaut mais vrai quand appelé par l'autre setter
- TODO : faire un exemple de code

Utiliser les jointures pour moins de requêtes

Cas One-To-One

- Relation entre Vehicle et VehicleSecurity :

```
public function testOneToOneNoJoinAction(EntityManagerInterface $em)
{
    $vehicle = $em->getRepository('App:Vehicle')->find(1);
    // Le fait d'accéder à une propriété de l'entité
    // VehicleSecurity déclenche la seconde requête
    dump($vehicle->getVehicleSecurity()->getAirbagNumber());
    return new Response('<body></body>');
}
```



1 0.99 ms SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.id = ?
Parameters:
[
1
]
[View formatted query](#) [View runnable query](#) [Explain query](#)

2 0.14 ms SELECT t0.id AS id_1, t0.euro_ncap_stars AS euro_ncap_stars_2, t0.airbag_number AS airbag_number_3, t0.abs AS abs_4, t0.esp AS esp_5 FROM vehicle_security t0 WHERE t0.id = ?
Parameters:
[
1
]

- Eviter 2 requêtes : créer une méthode personnalisée dans le Repository de Vehicle :

```
public function findWithSecurity($id)
{
    return $this->createQueryBuilder('v') // SELECT * FROM vehicle v
        ->where('v.id=:id')->setParameter('id', $id) // WHERE v.id = $id
        ->leftJoin('v.vehicleSecurity', 's') // LEFT JOIN vehicle_security s ON v.vehicle_security_id = s.id
        // 1er paramètre de leftJoin() : propriété de l'entité principale (celle qui est définie dans le FROM,
        // en l'occurrence Vehicle) sur laquelle la jointure se fait => Vehicle a bien la propriété vehicleSecurity
        // 2ème argument de leftJoin() : alias de l'entité jointe utilisé dans la requête
        // On fait un LEFT JOIN car la relation est facultative. Sinon, on ferait un INNER JOIN
        ->addSelect('s') // Ajoute s.* au SELECT
        // On utilise addSelect() car select() remplace tout ce qui est dans le SELECT
        ->getQuery()->getOneOrNullResult(); // Le résultat ou null si pas de résultat
}
```


Utiliser les jointures pour moins de requêtes

Cas One-To-One – Utilisation de la nouvelle méthode

- Une seule requête exécutée au lieu de 2 :

```
public function testOneToOneJoinAction(EntityManagerInterface $em)
{
    $vehicle = $em->getRepository('App:Vehicle')->findWithSecurity(1);
    // Cette fois-ci, Doctrine n'a pas besoin d'exécuter une seconde requête car
    // il "connaît" les données de VehicleSecurity
    dump($vehicle->getVehicleSecurity()->getAirbagNumber());
    return new Response('<body></body>');
}
```

1 1.02 ms SELECT v0_.id AS id_0, v0_.plate AS plate_1, v0_.mileage AS mileage_2, v0_.price AS price_3, v0_.description AS description_4, v0_.manu_date AS manu_date_5, v1_.id AS id_6, v1_.euro_ncap_stars AS euro_ncap_stars_7, v1_.airbag_number AS airbag_number_8, v1_.abs AS abs_9, v1_.esp AS esp_10, v0_.vehicle_security_id AS vehicle_security_id_11, v0_.model_id AS model_id_12 FROM vehicle v0_ LEFT JOIN vehicle_security v1_ ON v0_.vehicle_security_id = v1_.id WHERE v0_.id = ?

Parameters:

```
[
  1
]
```

Utiliser les jointures pour moins de requêtes

Cas One-To-One

- Avec Doctrine, on ne peut faire une jointure que si l'entité du FROM possède un attribut vers l'entité à joindre
 - Dans l'exemple présenté, c'est le cas car Vehicle est propriétaire de la relation
 - Si on souhaitait faire l'équivalent depuis l'entité VehicleSecurity, il faudrait d'abord rendre la relation bidirectionnelle → VehicleSecurity aurait alors une propriété **\$vehicle** et on pourrait faire une jointure en partant de VehicleSecurity vers Vehicle
- Condition de jointure :
 - Implicite car Doctrine « connaît » la relation (cf. annotation) → on ne précise donc jamais « **ON v.vehicle_security_id = s.id** »
 - Pour ajouter une condition supplémentaire à la jointure : utiliser **WITH**

```
public function findWithSecurityBis($id)
{
    return $this->createQueryBuilder('v')
        ->where('v.id=:id')->setParameter('id', $id)
        ->leftJoin('v.vehicleSecurity', 's', 'WITH', 's.euroNcapStars >= 4')
        ->addSelect('s')
        ->getQuery()->getOneOrNullResult();
}
```

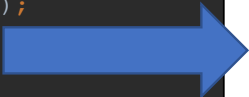
```
LEFT JOIN vehicle_security v1_ ON v0_.vehicle_security_id = v1_.id AND (v1_.euro_ncap_stars >= 4)
```

Utiliser les jointures pour moins de requêtes

Cas Many-To-One

- Relation Vehicle / VehicleModel :

```
public function testManyToOneNoJoinAction(EntityManagerInterface $em)
{
    $vehicle = $em->getRepository('App:Vehicle')->find(1);
    // Le fait d'accéder à une propriété de l'entité
    // VehicleModel déclenche la seconde requête
    dump($vehicle->getModel()->getName());
    return new Response('<body></body>');
}
```



```
1 1.29 ms SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5,
t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE
t0.id = ?
Parameters:
[
    1
]
View formatted query View runnable query Explain query

2 0.14 ms SELECT t0.id AS id_1, t0.name AS name_2, t0.make AS make_3 FROM vehicle_model t0 WHERE t0.id = ?
Parameters:
[
    1
]
```

- Créer une méthode dans le Repository de Vehicle :


```
public function findWithModel($id)
{
    return $this->createQueryBuilder('v')
        ->where('v.id=:id')->setParameter('id', $id)
        ->leftJoin('v.model', 'm')
        ->addSelect('m')
        ->getQuery()->getOneOrNullResult();
}
```

Utiliser les jointures pour moins de requêtes

Cas Many-To-One – Utilisation de la nouvelle méthode

- Une seule requête exécutée au lieu de 2 :

```
public function testManyToOneWithJoinAction(EntityManagerInterface $em)
{
    $vehicle = $em->getRepository('App:Vehicle')->findWithModel(1);
    // Cette fois-ci, Doctrine n'a pas besoin d'exécuter une seconde requête car
    // il "connaît" les données de VehicleModel
    dump($vehicle->getModel()->getName());
    return new Response('<body></body>');
}
```




```
1    1.04 ms  SELECT v0_.id AS id_0, v0_.plate AS plate_1, v0_.mileage AS mileage_2, v0_.price AS price_3, v0_.description AS description_4,
v0_.manu_date AS manu_date_5, v1_.id AS id_6, v1_.name AS name_7, v1_.make AS make_8, v0_.vehicle_security_id AS
vehicle_security_id_9, v0_.model_id AS model_id_10 FROM vehicle v0_ LEFT JOIN vehicle_model v1_ ON v0_.model_id = v1_.id WHERE
v0_.id = ?
Parameters:
[▼
    1
]
```

Utiliser les jointures pour moins de requêtes

Cas One-To-Many (si la relation est bidirectionnelle)


- Relation Vehicle / VehicleModel :

```
$model = $em->getRepository('App:VehicleModel')->find(1);  
foreach ($model->getVehicles() as $vehicle) {  
    dump($vehicle);  
}
```



```
1 1.01 ms SELECT t0.id AS id_1, t0.name AS name_2, t0.make AS make_3 FROM vehicle_model t0 WHERE t0.id = ?  
Parameters:  
[  
    1  
]  
View formatted query View runnable query Explain query  
  
2 0.16 ms SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5,  
t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE  
t0.model_id = ?  
Parameters:  
[  
    1  
]
```

```
$model2 = $em->getRepository('App:VehicleModel')->findWithVehicles(1);  
foreach ($model2->getVehicles() as $vehicle) {  
    dump($vehicle);  
}
```



```
3 0.19 ms SELECT v0_.id AS id_0, v0_.name AS name_1, v0_.make AS make_2, v1_.id AS id_3, v1_.plate AS plate_4, v1_.mileage AS mileage_5,  
v1_.price AS price_6, v1_.description AS description_7, v1_.manu_date AS manu_date_8, v1_.vehicle_security_id AS  
vehicle_security_id_9, v1_.model_id AS model_id_10 FROM vehicle_model v0_ LEFT JOIN vehicle v1_ ON v0_.id = v1_.model_id WHERE  
v0_.id = ?  
Parameters:  
[  
    1  
]
```

```
public function findWithVehicles($id)  
{  
    return $this->createQueryBuilder('m')  
        ->where('m.id = :id')->setParameter('id', $id)  
        ->leftJoin('m.vehicles', 'v')  
        ->addSelect('v')  
        ->getQuery()->getOneOrNullResult();  
}
```

Utiliser les jointures pour moins de requêtes

Cas Many-To-Many

- Relation Vehicle / VehicleEquipment

```
$vehicle = $em->getRepository('App:Vehicle')->find(1);  
foreach ($vehicle->getEquipments() as $equipment) {  
    dump($equipment->getName());  
}
```



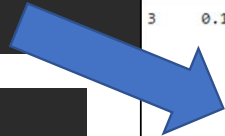
1 0.97 ms `SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.id = ?`
Parameters:
[
1
]
[View formatted query](#) [View runnable query](#) [Explain query](#)

2 0.16 ms `SELECT t0.id AS id_1, t0.name AS name_2, t0.description AS description_3 FROM vehicle_equipment t0 INNER JOIN asso_vehicle_equipment ON t0.id = asso_vehicle_equipment.vehicle_equipment_id WHERE asso_vehicle_equipment.vehicle_id = ?`
Parameters:
[
1
]

Remarque : Doctrine effectue une jointure automatique entre la table `vehicle_equipment` et la table d'association



```
$vehicle2 = $em->getRepository('App:Vehicle')->findWithEquipments(1);  
foreach ($vehicle2->getEquipments() as $equipment) {  
    dump($equipment->getName());  
}
```



3 0.17 ms `SELECT v0.id AS id_0, v0.plate AS plate_1, v0.mileage AS mileage_2, v0.price AS price_3, v0.description AS description_4, v0.manu_date AS manu_date_5, v1.id AS id_6, v1.name AS name_7, v1.description AS description_8, v0.vehicle_security_id AS vehicle_security_id_9, v0.model_id AS model_id_10 FROM vehicle v0 LEFT JOIN asso_vehicle_equipment a2 ON v0.id = a2.vehicle_id LEFT JOIN vehicle_equipment v1 ON v1.id = a2.vehicle_equipment_id WHERE v0.id = ?`
Parameters:
[
1
]

```
public function findWithEquipments($id)  
{  
    return $this->createQueryBuilder('v')  
        ->where('v.id=:id')->setParameter('id', $id)  
        // Noter que ce leftJoin() génère en fait  
        // 2 JOIN LEFT au niveau SQL  
        ->leftJoin('v.equipments', 'e')  
        ->addSelect('e')  
        ->getQuery()->getOneOrNullResult();  
}
```

Utiliser les jointures pour moins de requêtes

Cas Many-To-Many avec attributs

- Relation Vehicle / VehicleRepairs
- Etape préliminaire : rendre bidirectionnelle la relation côté Vehicle ➔ permettra de récupérer toutes les réparations d'un véhicule

```
class VehicleToVehicleRepairs
{
    // ...

    /**
     * @var Vehicle
     * @ORM\ManyToOne(targetEntity="App\Entity\Vehicle", inversedBy="repairsLinks")
     * @ORM\JoinColumn(nullable=false)
     */
    private $vehicle;
```

```
class Vehicle
{
    // ...

    /**
     * @var PersistentCollection
     * @ORM\OneToMany(targetEntity="App\Entity\VehicleToVehicleRepairs", mappedBy="vehicle")
     */
    private $repairsLinks;
```

Autres actions dans **Vehicle** :

- Initialisation **\$repairsLinks** dans le constructeur
- Ajout **getRepairsLinks()**
- Ajout **addRepairsLinks()**
- Ajout **removeRepairsLinks()**

Utiliser les jointures pour moins de requêtes

Cas Many-To-Many avec attributs – Sans jointure

```
$vehicle = $em->getRepository('App:Vehicle')->find(1);  
foreach ($vehicle->getRepairsLinks() as $link) {  
    dump($link->getVehicleRepairs()->getDescription());  
}
```

- Une requête pour le véhicule
- Une requête pour l'association
- Une requête par réparation

→ N+2 requêtes en tout
si N réparations

| | | |
|---|---------|--|
| 1 | 1.04 ms | <pre>SELECT t0.id AS id_1, t0.plate AS plate_2, t0.mileage AS mileage_3, t0.price AS price_4, t0.description AS description_5, t0.manu_date AS manu_date_6, t0.vehicle_security_id AS vehicle_security_id_7, t0.model_id AS model_id_8 FROM vehicle t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[1]</pre> <p>View formatted query View runnable query Explain query</p> |
| 2 | 0.16 ms | <pre>SELECT t0.id AS id_1, t0.date AS date_2, t0.vehicle_id AS vehicle_id_3, t0.vehicle_repairs_id AS vehicle_repairs_id_4 FROM vehicle_to_vehicle_repairs t0 WHERE t0.vehicle_id = ?</pre> <p>Parameters:</p> <pre>[1]</pre> <p>View formatted query View runnable query Explain query</p> |
| 3 | 0.16 ms | <pre>SELECT t0.id AS id_1, t0.description AS description_2, t0.price AS price_3 FROM vehicle_repairs t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[2]</pre> <p>View formatted query View runnable query Explain query</p> |
| 4 | 0.22 ms | <pre>SELECT t0.id AS id_1, t0.description AS description_2, t0.price AS price_3 FROM vehicle_repairs t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[3]</pre> |

Utiliser les jointures pour moins de requêtes

Cas Many-To-Many avec attributs – Jointure (solution 1)

- On économise une seule requête ➔ il en reste N+1

```
$vehicle = $em->getRepository('App:Vehicle')->findWithRepairsLinks(1);  
foreach ($vehicle->getRepairsLinks() as $link) {  
    dump($link->getVehicleRepairs()->getDescription());  
}
```

| | | |
|---|---------|--|
| 1 | 1.07 ms | <pre>SELECT v0_.id AS id_0, v0_.plate AS plate_1, v0_.mileage AS mileage_2, v0_.price AS price_3, v0_.description AS description_4, v0_.manu_date AS manu_date_5, v1_.id AS id_6, v1_.date AS date_7, v0_.vehicle_security_id AS vehicle_security_id_8, v0_.model_id AS model_id_9, v1_.vehicle_id AS vehicle_id_10, v1_.vehicle_repairs_id AS vehicle_repairs_id_11 FROM vehicle v0_ LEFT JOIN vehicle_to_vehicle_repairs v1_ ON v0_.id = v1_.vehicle_id WHERE v0_.id = ?</pre> <p>Parameters:</p> <pre>[1]</pre> <p>View formatted query View runnable query Explain query</p> |
| 2 | 0.17 ms | <pre>SELECT t0.id AS id_1, t0.description AS description_2, t0.price AS price_3 FROM vehicle_repairs t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[2]</pre> <p>View formatted query View runnable query Explain query</p> |
| 3 | 0.14 ms | <pre>SELECT t0.id AS id_1, t0.description AS description_2, t0.price AS price_3 FROM vehicle_repairs t0 WHERE t0.id = ?</pre> <p>Parameters:</p> <pre>[3]</pre> |

```
public function findWithRepairsLinks($id)  
{  
    return $this->createQueryBuilder('v')  
        ->where('v.id=:id')->setParameter('id', $id)  
        ->leftJoin('v.repairsLinks', 'rl')  
        ->addSelect('rl')  
        ->getQuery()->getOneOrNullResult();  
}
```

Utiliser les jointures pour moins de requêtes

Cas Many-To-Many avec attributs – Jointure (solution 2)

- Le même résultat avec une seule requête !

```
$vehicle = $em->getRepository('App:Vehicle')->findWithRepairsLinksAndRepairs(1);  
foreach ($vehicle->getRepairsLinks() as $link) {  
    dump($link->getVehicleRepairs()->getDescription());  
}
```

```
1    1.33 ms  SELECT v0_.id AS id_0, v0_.plate AS plate_1, v0_.mileage AS mileage_2, v0_.price AS price_3, v0_.description AS description_4,  
v0_.manu_date AS manu_date_5, v0_.name AS name_6, v1_.id AS id_7, v1_.date AS date_8, v2_.id AS id_9, v2_.description AS  
description_10, v2_.price AS price_11, v0_.user_id AS user_id_12, v0_.vehicle_security_id AS vehicle_security_id_13, v0_.model_id  
AS model_id_14, v1_.vehicle_id AS vehicle_id_15, v1_.vehicle_repairs_id AS vehicle_repairs_id_16 FROM vehicle v0_ LEFT JOIN  
vehicle_to_vehicle_repairs v1_ ON v0_.id = v1_.vehicle_id INNER JOIN vehicle_repairs v2_ ON v1_.vehicle_repairs_id = v2_.id WHERE  
v0_.id = ?  
Parameters:  
[▼  
  1  
]
```

```
public function findWithRepairsLinksAndRepairs($id)  
{  
    return $this->createQueryBuilder('v')  
        ->where('v.id=:id')->setParameter('id', $id)  
        ->leftJoin('v.repairsLinks', 'r1')  
        ->addSelect('r1')  
        // Seconde jointure en INNER car on est certain d'avoir une réparation dès lors qu'on a une association  
        ->innerJoin('r1.vehicleRepairs', 'r')  
        ->addSelect('r')  
        ->getQuery()->getOneOrNullResult();  
}
```

TP Doctrine – Relations entre entités



- Créer une entité Comment
 - title (string), author (string), created_at (datetime), message (text)
 - Faire une relation bidirectionnelle obligatoire Comment Many-To-One Article
 - Modifier la méthode de ArticleRepository créée au TP9 (partie 3) : la liste d'articles doit être complétée des commentaires associés triés par ordre décroissant (méthode **addOrderBy ()**) de date de création (faire évidemment une jointure !) + modifier le template Twig pour afficher les informations de chaque commentaire
- Créer une entité Category (un attribut de type string : name)
 - Faire une relation bidirectionnelle Article Many-To-Many Category en choisissant Article comme propriétaire + nom de table personnalisé : « asso_article_category »
 - Modifier le contrôleur / la vue de consultation d'un article pour afficher la liste des catégories à laquelle il appartient (ne pas faire de requête en plus ➔ jointure)
 - Menu de gauche : transmettre au template (en inclusion de contrôleur) une nouvelle variable : liste des catégories du blog. Les afficher au-dessus des derniers articles sortis. Au clic sur une catégorie, afficher la liste des articles de cette catégorie (nouvelle route « /blog/category/xxx » avec xxx ID de la catégorie / nouvelle action)
- Utilisation des fixtures :
 - Ajouter plusieurs commentaires associés à chaque fois à un article du blog pris au hasard
 - Ajouter plusieurs catégories et les associer aux articles de votre blog (de manière aléatoire, pour éviter d'avoir chaque article associé à toutes les catégories créées)