



Simon GAUTIER



Doctrine

Créer des entités

Persister des entités

Charger des données : QueryBuilder / DQL

Récupération des résultats

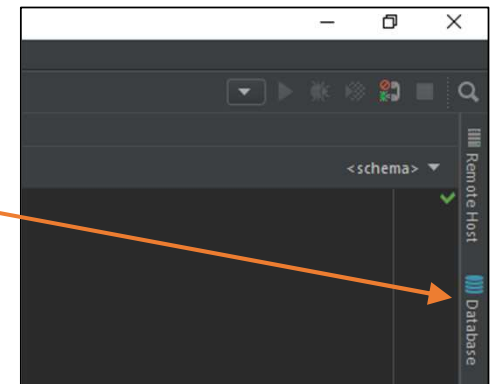
Gestion de la pagination

Qu'est-ce qu'un ORM ?

- ORM = Object-Relational Mapper (mapping objet-relationnel)
 - Définition wikipedia : *technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.*
- Exemple : site e-commerce (produits dans des catégories)
 - Pour ajouter un produit :
 - Sans ORM : requête SQL du type **INSERT into product...**
 - Avec ORM : on reste au niveau objet : « **\$product->save()** »
 - Pour accéder à des données :
 - Sans ORM : une requête en base de données qui retourne un tableau des valeurs du produit : **SELECT * FROM product WHERE id=5**
 - Puis on accède aux valeurs. Ex : **\$product['name']**
 - Avec ORM : **\$product->load(5)**
 - Puis on accède aux valeurs. Ex : **\$product->getName()**
 - Mais on peut aller beaucoup plus loin que sans ORM :
 - Catégorie du produit (dans une autre table) : **\$product->getCategory()**
 - Produits d'une catégorie

Qu'est-ce qu'un ORM ?

- Termes à retenir :
 - Objet géré par l'ORM = **entité**
 - **Persister** une entité = enregistrer l'entité dans la base de données
- Symfony : utilise Doctrine2 comme ORM
 - \$ **composer req orm**
- Initialiser l'accès à la base de données : fichier **.env** , variable **DATABASE_URL** :
 - Mysql : `DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?charset=utf8mb4&serverVersion=5.7"`
 - SQLite : `DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"`
- Créer la base de données :
 - \$ **php bin/console doctrine:database:create**
- Conseil : utiliser le composant « Database » de PhpStorm !



Aparté sur le composant **make**

```
$ composer req make
```


```
$ php bin/console | grep make
```

```
make
```

<code>make:auth</code>	Creates an empty Guard authenticator
<code>make:command</code>	Creates a new console command class
<code>make:controller</code>	Creates a new controller class
<code>make:entity</code>	Creates a new Doctrine entity class
<code>make:form</code>	Creates a new form class
<code>make:functional-test</code>	Creates a new functional test class
<code>make:serializer:encoder</code>	Creates a new serializer encoder class
<code>make:subscriber</code>	Creates a new event subscriber class
<code>make:twig-extension</code>	Creates a new Twig extension class
<code>make:unit-test</code>	Creates a new unit test class
<code>make:validator</code>	Creates a new validator and constraint class
<code>make:voter</code>	Creates a new security voter class

Créer une entité

- Entité = classe (répertoire Entity) + mapping avec la base de données
 - Mapping via annotations (recommandé), YAML, XML, PHP
- Étape 1 : créer la classe :
 - Solution simple : utiliser le maker (très pratique pour générer du code) :
\$ php bin/console make:entity
 - Puis choisir le nom de l'entité →
created: src/Entity/Vehicle.php
created: src/Repository/VehicleRepository.php
→ La console a généré notre entité et son Repository associé
 - Puis suivre le guide pour saisir les propriétés



```
<?php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

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

    // add your own fields
}
```

Créer une entité

Étape 2 : ajout des propriétés de l'entité (si la génération auto n'a pas été faite à l'étape précédente)

Classe : possibilité de forcer le nom de la table associée

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\VehicleRepository")
 * @ORM\Table(name="voiture")
 */
class Vehicle
```

Ajouter les propriétés souhaitées

Puis, générer les getters / setters :

- Uniquement un getter pour l'id
- Un getter / un setter pour les autres champs

Avec PhpStorm ➔ Menu « Code » ➔ « Generate »
➔ « Getters and Setters »

Remarque : choisir le mode « fluent » pour les setters
➔ ajoute un **return \$this;** aux setters pour permettre un chaînage.

```
/**
 * @var string
 * @ORM\Column(type="string", length=10)
 */
private $plate;

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

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

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

/**
 * @var \DateTime
 * @ORM\Column(type="datetime", name="manu_date")
 */
private $manufactureDate;
```

Par défaut, les noms des tables / des champs sont le snake case des noms de la classe / des attributs mais possibilité de spécifier des noms différents

Créer une entité

Étape 2 : liste des types possibles

Type Doctrine	Type SQL	Type PHP	Utilisation
string	VARCHAR	string	Toutes les chaînes de caractères jusqu'à 255 caractères.
integer	INT	integer	Tous les nombres jusqu'à 2 147 483 647.
smallint	SMALLINT	integer	Tous les nombres jusqu'à 32 767.
bigint	BIGINT	string	Tous les nombres jusqu'à 9 223 372 036 854 775 807. Attention, PHP reçoit une chaîne de caractères, car il ne supporte pas un si grand nombre (suivant que vous êtes en 32 ou en 64 bits).
boolean	BOOLEAN	boolean	Les valeurs booléennes true / false.
decimal	DECIMAL	double	Les nombres à virgule.
date ou datetime	DATETIME	objet DateTime	Toutes les dates et heures.
time	TIME	objet DateTime	Toutes les heures.
text	CLOB	string	Les chaînes de caractères de plus de 255 caractères.
object	CLOB	Type de l'objet stocké	Stocke un objet PHP en utilisant serialize / unserialize.
array	CLOB	array	Stocke un tableau PHP en utilisant serialize / unserialize.
float	FLOAT	double	Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur.

Cf. doc : docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html

Créer une entité

Étape 3 : générer la structure

- Générer la structure correspondant aux entités définies :
 \$ php bin/console doctrine:migrations:diff
 ➔ Génère un fichier dans **src/Migrations : VersionDATETIME**
- Contient une méthode **up ()** et une méthode **down ()** pour respectivement appliquer / supprimer la mise à jour

```
<?php declare(strict_types = 1);

namespace Doctrine\Migrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20171124151708 extends AbstractMigration
{
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'sqlite', 'Migration can only be executed safely on \'sqlite\'');

        $this->addSql('CREATE TABLE vehicle (id INTEGER NOT NULL, plate VARCHAR(10) NOT NULL, mileage INTEGER NOT NULL, price NUMERIC(8, 2) DEFAULT NULL, description CLOB NOT NULL, manu_date DATETIME NOT NULL, PRIMARY KEY(id))');
    }

    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'sqlite', 'Migration can only be executed safely on \'sqlite\'');

        $this->addSql('DROP TABLE vehicle');
    }
}
```


Créer une entité

Étape 3 : générer la structure

- Appliquer la migration :

```
$ php bin/console doctrine:migrations:migrate
```

- Si on ajoute d'autres champs et qu'on relance cette procédure, Doctrine appliquera un différentiel (ALTER TABLE...)
- La version courante est stockée dans une table auto-gérée :
migration_versions
- Bonne pratique : versionner les fichiers **src/Migrations/Version*** pour s'assurer que chaque environnement exécute les mêmes requêtes
 - Remarque : il existe une alternative non conseillée justement car elle ne permet pas de versionner les actions :

```
$ php bin/console doctrine:schema:update
```

Persister des objets en base de données

```
class VehicleController extends Controller
{
  /**
   * @Route("/vehicle", name="vehicle")
   */
  public function indexAction()
  {
    // you can fetch the EntityManager via $this->getDoctrine() or $this->get('doctrine')
    // or you can add an argument to your action: index(EntityManagerInterface $em)
    $em = $this->getDoctrine()->getManager();

    $vehicle = new Vehicle();
    $vehicle->setPlate('AB-123-CD')->setMileage(58000)->setPrice(19999.99)
      ->setManufactureDate(new \DateTime('2017-11-24'))->setDescription('Très belle voiture');

    // tell Doctrine you want to (eventually) save the Product (no queries yet)
    $em->persist($vehicle);

    // actually executes the queries (i.e. the INSERT query)
    $em->flush();

    return new Response('<body>Saved new product with id '.$vehicle->getId().'</body>');
  }
}
```

getManager() peut prendre un paramètre
\$name : permet de gérer la connexion à
plusieurs bases de données → facultatif :
pas de paramètre si une seule base de
données

Persister des objets en base de données



Le flush traite de manière transactionnelle tous les objets à persister :

- Si l'objet existe déjà → UPDATE
- Sinon → INSERT

Un objet déjà « connu » de Doctrine n'a pas besoin d'être explicitement persisté (on le verra plus loin).

3	3	38.43 ms	0
Database Queries	Different statements	Query time	Invalid entities
Queries			
Group similar statements			
# ▲	Time	Info	
1	0.07 ms	"START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query	
2	14.18 ms	INSERT INTO vehicle (plate, mileage, price, description, manu_date) VALUES (?, ?, ?, ?, ?) Parameters: [1 => "AB-123-CD" 2 => 58000 3 => 19999.99 4 => "Très belle voiture" 5 => "2017-11-24 00:00:00"] View formatted query View runnable query Explain query	
3	24.17 ms	"COMMIT" Parameters: [] View formatted query View runnable query Explain query	

Quelques méthodes de l'Entity Manager

persist()
detach()
flush()
contains()
clear()
refresh()
remove()

```
public function indexAction()
{
    $em = $this->getDoctrine()->getManager();
    $vehicle = new Vehicle();
    $vehicle->setPlate('AB-123-CD')->setMileage(58000)->setPrice(19999.99)
        ->setManufactureDate(new \DateTime('2017-11-24'))->setDescription('Très belle voiture');
    $vehicle2 = $vehicle3 = clone($vehicle);
    $em->persist($vehicle);
    $em->persist($vehicle2);
    $em->persist($vehicle3);

    $em->detach($vehicle); // Seuls $vehicle2 et $vehicle3 seront persistés
    if($em->contains($vehicle)) { /* false */ }
    if($em->contains($vehicle2)) { /* true */ }
    $em->clear('App\Entity\Vehicle'); // Toutes les entités Vehicle ne seront plus persistées
    $em->clear(); // Plus aucune entité ne sera persistée (tout type confondu)

    $em->persist($vehicle2);
    $em->flush();

    $vehicle2->setPlate('BA-321-DC');
    dump($vehicle2);
    $em->refresh($vehicle2); // $vehicle2 est rechargé avec ses données en base
    dump($vehicle2);
    $em->remove($vehicle2); // Supprimera $vehicle2 au prochain flush()
    $em->flush();

    return new Response('<body>Saved new product with id '.$vehicle2->getId().'</body>');
}
```

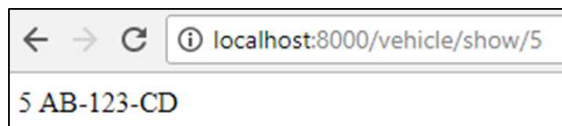
Repository – Récupérer les entités

Premier exemple d'utilisation de **find()**

```
/**
 * @Route("/vehicle/show/{id}", name="vehicle_show")
 * @param $id
 * @param EntityManagerInterface $em
 * @return Response
 */
public function showAction($id, EntityManagerInterface $em)
{
    // Préciser l'entité souhaitée. Remarque : la classe Repository n'a pas besoin d'exister
    // find() attend l'ID permettant de charger l'entité
    $vehicle = $em->getRepository('App:Vehicle')->find($id); // Ou $em->getRepository('App\Entity\Vehicle')

    if(!$vehicle) {
        throw $this->createNotFoundException("Véhicule non trouvé");
    }

    return new Response("<body>{$vehicle->getId()} {$vehicle->getPlate()}</body>");
}
```



Repository – Récupérer les entités

find(), **findAll()**, **findBy()**, **findOneBy()**

```
// Charger un seul objet
$vehicle1 = $em->getRepository('App:Vehicle')->find(5);
$vehicle2 = $em->find('App:Vehicle', 5);

// Charger tous les objets
$vehicles1 = $em->getRepository('App:Vehicle')->findAll();

// Charger tous les objets correspondant aux critères donnés
$vehicles2 = $em->getRepository('App:Vehicle')->findBy(array('plate' => 'AB-123-CD', 'mileage' => 58000));

// Utilisation plus complète de findBy
$vehicles2bis = $em->getRepository('App:Vehicle')->findBy(
    array('plate' => 'AB-123-CD', 'mileage' => 58000), // Les critères de sélection
    array('manufactureDate' => 'desc'), // Les critères de tri
    5, // Le nombre de résultats
    0 // Offset => 0 : commencer au premier résultat
);

// Charger un seul objet correspondant aux critères donnés. Si plusieurs résultats, le premier est pris
$vehicle3 = $em->getRepository('App:Vehicle')->findOneBy(array('plate' => 'AB-123-CD', 'mileage' => 58000));
```

Repository – Récupérer les entités

Méthodes magiques

```
// findByX($value) : méthode magique pour charger des entités à partir d'un de ses attribut
// Fonctionne si x est un attribut de l'entité
// Retourne comme findBy() un tableau d'entités
$vehicles = $em->getRepository('App:Vehicle')->findByPlate('AB-123-CD');

// findOneByX($value) : idem findByX() mais ne retourne que la première entité
// Fonctionne si x est un attribut de l'entité
// Retourne null si pas de résultat trouvé, une entité sinon
$vehicle = $em->getRepository('App:Vehicle')->findOneByPlate('AB-123-CD');
```

- Toutes ces méthodes sont disponibles même si la classe Repository (en l'occurrence App\Repository\VehicleRepository) n'a pas été créée. C'est la classe Doctrine\ORM\EntityRepository qui fournit toute cette logique.
 - ➔ En créant notre propre Repository, il est possible de définir nos méthodes spécifiques

Créer son propre Repository

- Un repository centralise tout ce qui touche à la récupération des entités
 - Utilise en interne un EntityManager pour fonctionner
- Pourquoi créer son propre Repository ?
 - Car les méthodes standard de **Doctrine\ORM\EntityRepository** ne suffisent pas
 - Exemple : récupérer tous les produits du site actifs et créés sur l'année X ➔ il est exclus d'utiliser un **findAll()** et de faire le travail côté PHP ! (performances, mémoire, ...)
- Ce qu'on y trouve (s'apparente à des requêtes SQL sans en être) :
 - Méthodes pour récupérer un objet par son ID ou par d'autres critères
 - Méthodes pour récupérer une liste d'objets en fonction d'un ou plusieurs critères
- 2 manières de fonctionner au sein du Repository :
 - Via le langage DQL (Doctrine Query Language = pseudo langage SQL)
 - Via le QueryBuilder (objet qui permet de construire des requêtes)

Le QueryBuilder

- Principe de construction de requêtes :
 - Le **QueryBuilder** permet comme son nom l'indique de construire les requêtes
 - Une fois la requête construite, le QueryBuilder fournit la « requête » sous forme d'objet **Query**
 - Enfin, l'objet Query permet d'obtenir **les résultats** (valeur simple, objet, tableau d'objets, ...)
- Accéder au QueryBuilder :

```
class VehicleRepository extends ServiceEntityRepository
{
    // ...
    public function myFindAll()
    {
        // Solution 1 :
        $queryBuilder = $this->_em->createQueryBuilder() // Créer un query builder vide
        ->select('v') // Préciser quelles colonnes sélectionner (toute ici)
        ->from($this->_entityName, 'v'); // Préciser sur quelle entité

        // Solution 2 équivalente (aller voir le code de createQueryBuilder()) :
        $queryBuilder = $this->createQueryBuilder('v');

        // ... suite de notre méthode
    }
    // ...
}
```

\Doctrine\ORM\EntityRepository

```
public function createQueryBuilder($alias, $indexBy = null)
{
    return $this->_em->createQueryBuilder()
        ->select($alias)
        ->from($this->_entityName, $alias, $indexBy);
}
```

Le QueryBuilder

- **from(\$entity, \$alias)**
 - **\$entity** : nom de l'entité sur laquelle faire la requête
 - Si utilisation de **\$this->_entityName** → nom de l'entité connu par le Repository
 - Sinon, spécifier à la main le nom de l'entité raccourci ou non :
 - Ex : **App\Entity\Vehicle** ou **App:Vehicle**
 - **\$alias** : Alias utilisé pour la table. Souvent, première lettre de l'entité (« **v** » pour « **Vehicle** »)
- **select(\$col1, \$col2, ...)** : colonnes à prendre dans la requête (nombre de paramètres libre)
 - Si l'alias utilisé pour l'entité → « ***** »
 - Sinon, utiliser un nom d'attribut de l'entité (attention pas le nom de la colonne dans la base de données mais le nom de l'attribut de l'objet)
 - Ex : « **mon_alias.manufactureDate** » et pas « **mon_alias.manu_date** »

Le QueryBuilder

Exemple : réécriture de **findAll()**

```
public function myFindAll()
{
    // Récupération du QueryBuilder ==> select * from vehicle
    $queryBuilder = $this->createQueryBuilder('v');

    // Pas besoin d'autres critères car on veut tout récupérer

    // Récupération de la Query à partir du QueryBuilder
    $query = $queryBuilder->getQuery();

    // Récupération des résultats à partir de la Query
    $results = $query->getResult();

    return $results;
}

// La même chose sans commentaires ni variables inutiles
public function myFindAllBis()
{
    return $this->createQueryBuilder('v')
        ->getQuery()
        ->getResult();
}
```

```
/**
 * @Route("/vehicle/my_find_all", name="vehicle_my_find_all")
 * @param EntityManagerInterface $em
 * @return Response
 */
public function testMyFindAllAction(EntityManagerInterface $em)
{
    $vehicles = $em->getRepository('App:Vehicle')->myFindAll();
    $vehicles2 = $em->getRepository('App:Vehicle')->myFindAllBis();
    dump($vehicles, $vehicles2);
    return new Response('<body></body>');
}
```

Le QueryBuilder

Exemple : utilisation de **where ()** / **andWhere ()** / **orWhere ()**

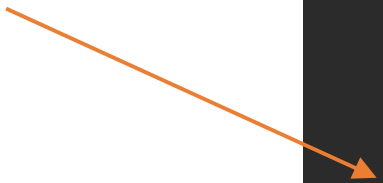
```
public function myFind($id)
{
    $qb = $this->createQueryBuilder('v')
        // Ajout d'une contrainte sur la propriété id de l'entité v
        ->where('v.id = :my_id')
        // Attribution de la valeur au paramètre (id reçu en paramètre)
        ->setParameter('my_id', $id)
    ;
    return $qb->getQuery()->getResult();
}

public function findByMileageAndYear($mileageMin, $mileageMax, $year)
{
    $qb = $this->createQueryBuilder('v')
        ->where('v.mileage >= :mileage_min')
        ->setParameter('mileage_min', $mileageMin)
        ->andWhere('v.mileage <= :mileage_max')
        ->setParameter('mileage_max', $mileageMax)
        ->andWhere('v.manufactureDate BETWEEN :begin AND :end')
        ->setParameter('begin', new \DateTime($year . '-01-01'))
        ->setParameter('end', new \DateTime($year . '-12-31'))
    ;
    return $qb->getQuery()->getResult();
}
```

Le QueryBuilder

Factorisation du code

- Si une sous-requête est récurrente ➔ la mutualiser dans une méthode séparée :



```
public function findByMileageAndYear($mileageMin, $mileageMax, $year)
{
    $qb = $this->createQueryBuilder('v')
        ->where('v.mileage >= :mileage_min')
        ->setParameter('mileage_min', $mileageMin)
        ->andWhere('v.mileage <= :mileage_max')
        ->setParameter('mileage_max', $mileageMax)
    ;
    $this->whereYear($qb, $year);
    return $qb->getQuery()->getResult();
}

public function whereYear(QueryBuilder $qb, $year)
{
    $qb
        ->andWhere('v.manufactureDate BETWEEN :begin AND :end')
        ->setParameter('begin', new \DateTime($year . '-01-01'))
        ->setParameter('end', new \DateTime($year . '-12-31'))
    ;
}
```

Le QueryBuilder

- Quelques autres méthodes du QueryBuilder :
 - Jointures : cf. chapitre suivant
 - **orderBy () / addOrderBy ()**
 - **having () / andHaving () / orHaving ()**
 - **groupBy () / addGroupBy ()**
 - ...
- Utiliser la documentation
<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html>

DQL

- DQL = Doctrine Query Language
 - Pseudo langage SQL adapté à Doctrine
 - Documentation : docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html
- Construire une Query : QueryBuilder VS DQL
 - Créer une Query depuis le QueryBuilder : méthode **getQuery()** du QueryBuilder
 - Créer une Query en DQL : méthode **createQuery()** de l'EntityManager
- Exemple de requête DQL :
`SELECT v FROM App:Vehicle v WHERE v.id=5`
- La console permet de tester les requêtes DQL :
`php bin/console doctrine:query:dql "MA_REQUETE"`

DQL

- Dans une requête DQL, on identifie une entité et pas une table :
 - « **App:Vehicle** » ou « **App\Entity\Vehicle** »
 - KO : « **vehicle** » si « **vehicle** » est le nom de la table
- Sélection des attributs de l'objet et pas des noms des colonnes :
 - **v.manufactureDate** et pas **v.manu_date**
- Toujours donner un alias à l'entité sélectionnée :
 - Généralement, la première lettre de l'entité mais aucune obligation
 - SELECT v.manufactureDate FROM App:Vehicle v → OK**
 - SELECT manufactureDate FROM App:Vehicle → ne fonctionne pas**

DQL – Examples

```
public function myFindAllDql()
{
    return $this->_em->createQuery("SELECT v FROM App:Vehicle v")->getResult();
}

public function myFindDql($id)
{
    $query = $this->_em->createQuery("SELECT v FROM App:Vehicle v WHERE v.id = :id");
    $query->setParameter('id', $id);
    return $query->getResult();
}
```

Query – Récupération des résultats

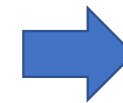
- Les exemples présentés n'utilisent que la méthode **getResult()** sur la Query. Il y en a d'autres !
- **getResult()** :
 - Retourne un tableau contenant les résultats sous forme d'objets
 - Attention : même si un seul résultat, il est dans un tableau

```
dump($this->_em->createQuery("SELECT v FROM App:Vehicle v")->getResult());
```



```
array:10 [  
  0 => Vehicle {#544 ▶}  
  1 => Vehicle {#546 ▶}  
  2 => Vehicle {#548 ▶}  
  ...  
]
```

```
$query = $this->_em->createQuery("SELECT v FROM App:Vehicle v WHERE v.id = :id");  
$query->setParameter('id', $id);  
dump($query->getResult());
```



```
array:1 [  
  0 => Vehicle {#567 ▶}  
]
```

Query – Récupération des résultats

- **getArrayResult () :**

- Retourne un tableau contenant les résultats sous forme de tableaux
- Attention : même si un seul résultat, il est dans un tableau (idem **getResult ()**)
- Plus rapide que **getResult ()** mais pas de modification possible des entités récupérées → lecture seule
- Dans Twig : pas de changement ! (rappel : `{{ my_var.attribute }}`)

```
dump($this->_em->createQuery("SELECT v FROM App:Vehicle v")->getArrayResult());
```



```
array:10 [  
  0 => array:6 [  
    1 => array:6 [  
      2 => array:6 [  
        ...  
      ]  
    ]  
  ]  
]
```

```
$query = $this->_em->createQuery("SELECT v FROM App:Vehicle v WHERE v.id = :id");  
$query->setParameter('id', $id);  
dump($query->getArrayResult());
```



```
array:1 [  
  0 => array:6 [  
    "id" => 5  
    "plate" => "AB-123-CD"  
    ...  
  ]  
]
```

Query – Récupération des résultats


- **getScalarResult () :**

- Retourne un tableau contenant les résultats sous forme de tableaux simples même si la requête mixe différentes données (cf. exemples)
- Attention : même si un seul résultat, il est dans un tableau (idem **getResult ()**)

```
$query = $this->_em->createQuery("SELECT v FROM App:Vehicle v  
    WHERE v.id = :id");  
$query->setParameter('id', $id);
```

```
dump($query->getArrayResult());
```

```
dump($query->getScalarResult());
```




```
array:1 [  
  0 => array:6 [  
    "id" => 5  
    "plate" => "AB-123-CD"  
    ...  
  ]  
]  
  
array:1 [  
  0 => array:6 [  
    "v_id" => 5  
    "v_plate" => "AB-123-CD"  
    ...  
  ]  
]
```

Query – Récupération des résultats

- **getScalarResult()** – Suite

```
$query = $this->_em->createQuery("SELECT v, v.manufactureDate  
    FROM App:Vehicle v WHERE v.id = :id");  
$query->setParameter('id', $id);
```



```
dump($query->getArrayResult());
```

```
dump($query->getScalarResult());
```

```
array:1 [  
    0 => array:2 [  
        0 => array:6 [  
            "id" => 5  
            "plate" => "AB-123-CD"  
            ...  
        ]  
        "manufactureDate" => DateTime @1511481600 {#545 ▶}  
    ]  
]  
  
array:1 [  
    0 => array:7 [  
        "v_id" => 5  
        "v_plate" => "AB-123-CD"  
        ...  
        "manufactureDate" => "2017-11-24 00:00:00"  
    ]  
]
```

Query – Récupération des résultats

- **getOneOrNullResult () :**

- Retourne une entité si un et un seul résultat est trouvé, null sinon
- Une exception est déclenchée s'il y a plus d'un résultat
- ➔ à utiliser si on sait que la requête n'est pas censée retourner plus d'un résultat

- **getSingleResult () :**

- Idem **getOneOrNullResult ()** mais une exception est déclenchée s'il n'y a aucun résultat
- ➔ A utiliser si on sait que la requête doit retourner un et un seul résultat

```
$query = $this->_em->createQuery("SELECT v FROM App:Vehicle v  
WHERE v.id = :id");  
$query->setParameter('id', $id);  
dump($query->getOneOrNullResult());
```

```
Vehicle {#567 ▼  
  -id: 5  
  -plate: "AB-123-CD"  
  -mileage: 58000  
  -price: "19999.99"  
  -description: "Très belle voiture"  
  -manufactureDate: DateTime @1511481600 {#550 ▶}  
}
```

Query – Récupération des résultats

- **getSingleScalarResult ()** :
 - Retourne une seule valeur : la requête doit retourner exactement un résultat
 - Une exception est déclenchée :
 - Si la requête ne retourne aucun résultat
 - Si la requête retourne plus d'un résultat
 - Si la requête a plus d'une colonne
 - Typiquement utilisé dans le cas d'un **SELECT COUNT (*) FROM ...**
- **execute ()** :
 - Exécute une requête non censée retourner un résultat
 - Généralement utilisé pour **UPDATE / INSERT / DELETE**
- Les méthodes présentées utilisent en réalité **execute ()** → cf. code

Focus sur la pagination

- Pour gérer une pagination, il faut :
 - Le nombre d'objets à afficher par page
 - Le nombre d'objets au total
 - Le numéro de la page courante
- Doctrine propose un composant qui gère les paginations : l'objet **Paginator**
- Pour comprendre son fonctionnement, enrichissons la méthode **myFindAll()** déjà présentée :

```
public function myFindAllWithPaging($currentPage, $nbPerPage)
{
    $query = $this->createQueryBuilder('v')
        ->getQuery()
        ->setFirstResult(($currentPage - 1) * $nbPerPage) // Premier élément de la page
        ->setMaxResults($nbPerPage); // Nombre d'éléments par page

    // Equivalent de getResult() mais un count() sur cet objet retourne le nombre de résultats hors pagination
    return new Paginator($query);
}
```


Focus sur la pagination

- Contrôleur :

```
public function testPageAction(EntityManagerInterface $em)
{
    // Afficher la page 2, 4 éléments par page
    $currentPage = 2;
    $nbPerPage = 4;
    $vehicles = $em->getRepository('App:Vehicle')->myFindAllWithPaging($currentPage, $nbPerPage);

    // Nombre total de pages construit à partir du nombre total d'objets
    $nbTotalPages = intval(ceil(count($vehicles) / $nbPerPage));
    dump($nbTotalPages);

    // Il est possible d'itérer sur l'objet Paginator comme s'il s'agissait d'un résultat classique
    foreach ($vehicles as $vehicle) {
        dump($vehicle);
    }
    return new Response('<body></body>');
}
```

- Requêtes générées :

- 1 pour le comptage
- 1 pour récupérer les IDs
- 1 pour les résultats

#	Time	Info
1	0.53 ms	<pre>SELECT COUNT(*) AS dctrn_count FROM (SELECT DISTINCT id_0 FROM (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 FROM vehicle v0_) dctrn_result) dctrn_table</pre> <p>Parameters: []</p> <p>View formatted query View runnable query Explain query</p>
2	0.19 ms	<pre>SELECT DISTINCT id_0 FROM (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 FROM vehicle v0_) dctrn_result LIMIT 4 OFFSET 4</pre> <p>Parameters: []</p> <p>View formatted query View runnable query Explain query</p>
3	1.16 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 FROM vehicle v0_ WHERE v0_.id IN (?)</pre> <p>Parameters: [5, 6, 7, 8]</p> <p>View formatted query View runnable query Explain query</p>

TP Doctrine – Partie 1



- Paramétrer l'accès à la base de données dans le fichier `.env` ou `.env.local`
- Créer une entité « **Article** » (nom de table « **article** ») avec les propriétés suivantes :
 - title (varchar 255)
 - content (text)
 - created_at (datetime) → doit être initialisé par défaut à la date courante
 - updated_at (datetime)
 - author (varchar 128)
 - nb_views (int)
 - published (boolean)
- Générer via l'assistant de migration la table en base de données

TP Doctrine – Partie 2



- Dynamiser l'action de listing des articles / Dynamiser l'action de push des X articles dans le menu
 - Afficher seulement les articles qui sont publiés. Trier les articles par ordre décroissant de date de création.
 - Pas de gestion de la pagination pour l'instant sur le listing des articles
 - Appliquer l'utilisation du **findBy()** / définir une limite seulement pour le push
- Dynamiser l'action de consultation d'un article
 - Appliquer l'utilisation du **find()** / vérifier que l'article existe bien et qu'il est publié. Sinon, déclencher une erreur 404 → **throw new NotFoundException()**
- Dynamiser l'action de suppression d'un article
 - Commencer par un **find()** pour charger l'article / vérifier qu'il existe bien (sinon, 404)
 - Puis faire le **remove()** pour le supprimer
- On ne dynamise pas encore les actions d'ajout / de modification d'un article car c'est lié à la gestion des formulaires que nous verrons plus tard...

TP Doctrine – Partie 3



- Créer un paramètre général à l'application (**services.yaml**) pour définir le nombre d'articles par page (section **parameters**)

```
# Put parameters here that don't need to change on each machine where the app is deployed
# https://symfony.com/doc/current/best_practices/configuration.html#application-related-configuration
parameters:
    %param_name%: %value%

services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
```

- Ajouter le mécanisme de pagination pour la liste des articles du blog
 - Dans le contrôleur :
 - Prendre en compte le paramètre précédemment défini
 - Déclencher une erreur 404 si la page demandée n'existe pas
 - Créer sa propre méthode dans le Repository des articles (avec les paramètres nécessaires : numéro de page courante + nombre de résultats par page)
 - Conserver l'ordre de tri établi (les articles les plus récents d'abord)
 - Utiliser sur le QueryBuilder la méthode **orderBy()** :
Exemple : **orderBy('a.createdAt', 'DESC')**
 - Enrichir le template qui affiche les articles : afficher les liens vers toutes les pages disponibles (le contrôleur doit passer les variables nécessaires). Le template doit tester s'il faut ou non afficher la pagination, ne pas mettre de lien sur la page courante