R4.01
Applications Web

MVC : Implémentation du Modèle Persistant
Mapping Objet/Relationnel (ORM)

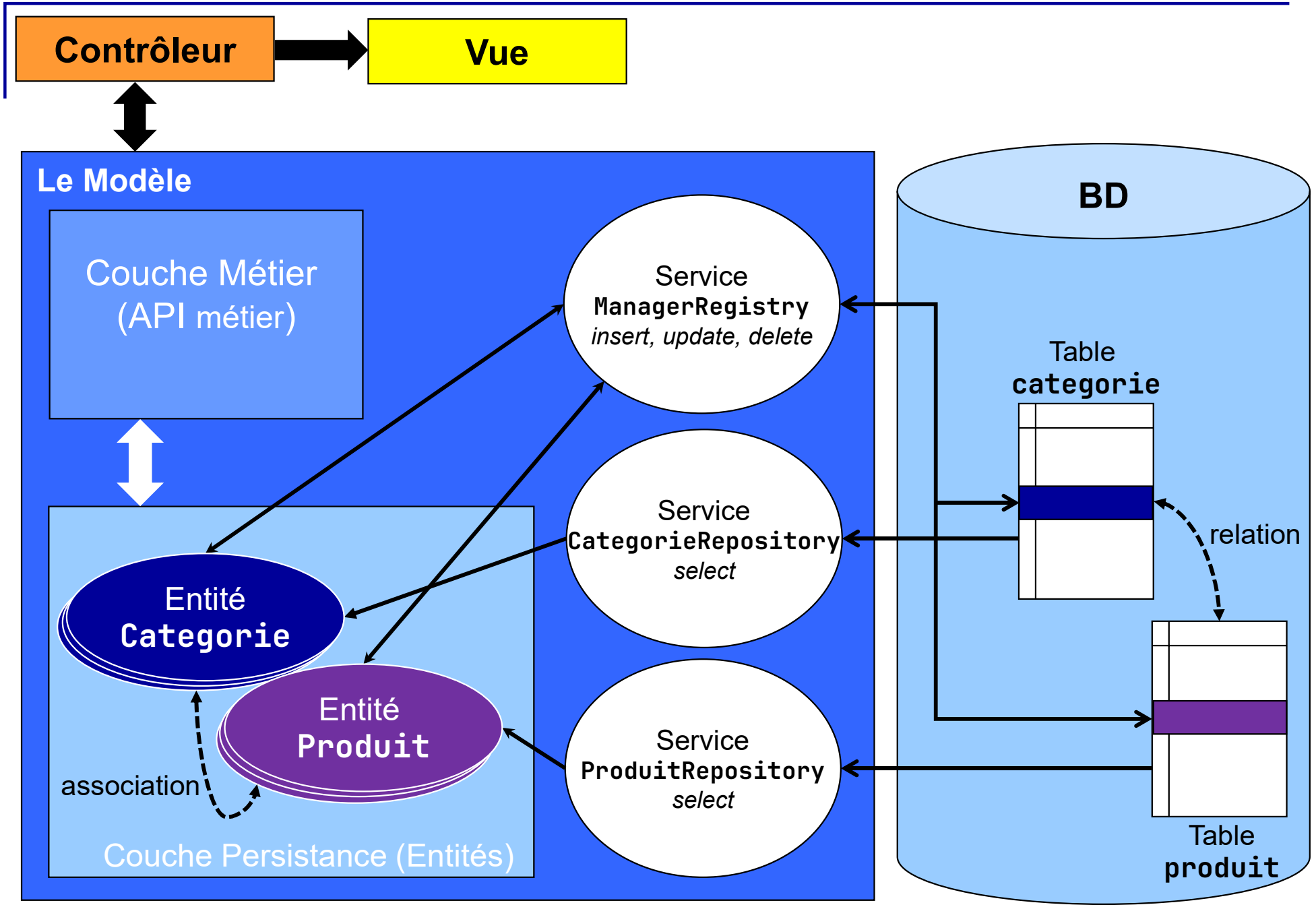
https://symfony.com/doc/current/doctrine.html

IUT2 A
Université Grenoble Alpes
Département INFO

Symfony

Motivation d'un ORM

- Lorsque l'on développe une application quia besoin de persistance, on est en général amené à manipuler une BD relationnelle et on est toujours confronté aux mêmes problèmes récurrents :
  - Trouver des enregistrements (n-uplets)
  - Mettre à jour des enregistrements
  - Ajouter de nouveaux enregistrements
  - Supprimer des enregistrements
  - Traiter une liste d'enregistrements
  - …*etc*
- En PHP, des outils permettent de construire un niveau d'abstraction de la base de données qui rend le code indépendant du type de SGBD utilisé mais ceci ne permet pas de s'abstraire du modèle relationnel (organisation des données en tables)
- Le concept de **DataObject**, issu des design patterns J2EE DataAccessObject et Transfer Object, permet quant à lui d'avoir un niveau **objet** d'accès aux données qui masque donc la couche SGBD relationnel qui réalise la persistance
  => permet de développer une couche métier sans SQL
- Les **Data Objects,** ou **Entités** en Symfony/Doctrine, permettent de réaliser un Mapping Objet/Relationnel (**Object-Relational Mapping**)
- Les **Data Access Objects (DAO)**, ou **Repository** en Symfony/Doctrine, permettent de réaliser des requêtes sur des collections d'objets persistants


Contrôleur
Vue
Le Modèle
Couche Métier
(API métier)
Service
ManagerRegistry
insert, update, delete
Service
CategorieRepository
select
Service
ProduitRepository
select
Entité
Categorie
Entité
Produit
association
Couche Persistance (Entités)
BD
Table
categorie
relation
Table
produit

Entité et Repository

- Une classe **Entité** est une classe PHP qui permet de modéliser un objet métier de notre application qui a besoin de persister en BD
  - La classe aura des attributs qui correspondront aux champs (colonnes) de la table qui assurera sa persistance
  - L'ORM Doctrine permet de produire automatiquement le code PHP des entités ainsi que le code SQL pour créer la table associée en BD
  - Les entités peuvent être spécifiées dans différents formats (Annotations, XML)
  - Les **relations** entre entités peuvent être modélisées facilement
- Un **Repository** est un **service symfony** (classe PHP) qui permet de réaliser des requêtes sur un **dépôt d'entités** (une table) et de récupérer les résultats sous forme d'**entités** (équivalent à un DAO)
  - Chaque type d'entité dispose de son propre **Repository**
  - L'ORM Doctrine propose automatiquement un certain nombre de requêtes standards
  - Il est possible d'ajouter à un **Repository** vos propres requêtes exprimées en DQL (Doctrine Query Language) ou codées à l'aide d'un « Query Builder »

Créer une classe Entité (1)

- En Ligne de Commande : **php bin/console make:entity**

```
Class name of the entity to create or update:
> Product

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

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

Field length [255]:
> 255

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

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

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

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

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

Créer une classe Entité (2)

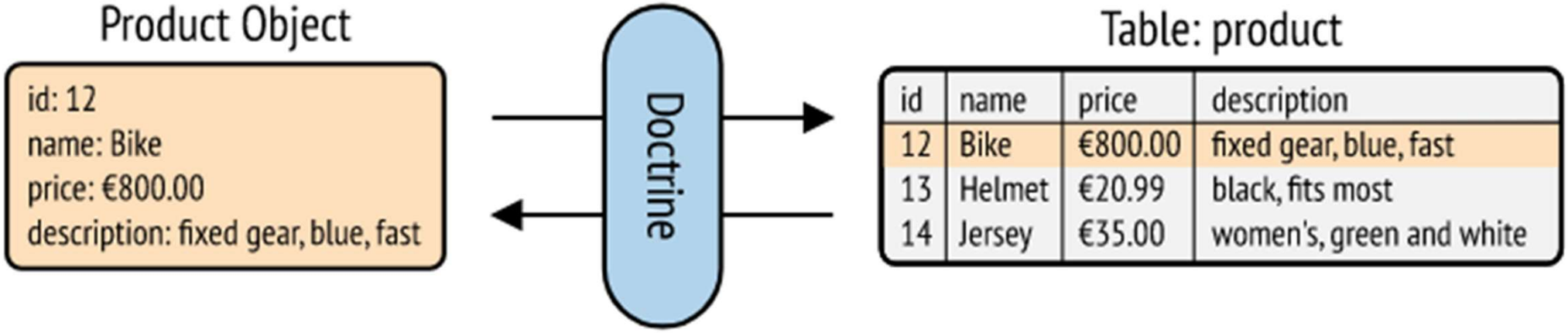
- L'exécution de la commande précédente crée une classe PHP :

```php
// src/Entity/Product.php
namespace App\Entity;
use App\Repository\ProductRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: ProductRepository::class)]
class Product {

    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type='integer')]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    private $name;

    #[ORM\Column(type: 'decimal', scale: 2, precision: 8)]
    private $price;

    public function getId() {
        return $this->id;
    }
    // ... Les getter and setter sont tous générés automatiquement !
}
```

Les attributs scale et precision permettent de préciser que les prix comporteront au plus 8 chiffres dont 2 après la virgule

Créer une classe Entité (3)

- Les objets de la classe PHP **Product** vont permettre de représenter/manipuler des n-uplets d'une table **product** créée dans la BD.
- Chaque propriété (attribut) de la classe PHP correspond à une colonne de la table SQL
- Le code PHP qui a été produit peut bien sûr être modifié/complété librement
- Il est également possible de ne pas utiliser la commande **make:entity** en écrivant directement le code PHP et ses annotations.



Les attributs de l'annotation **Column**

- **id** : will map to the column id using the type integer;
- **text** : will map to the column text with the default mapping type string;
- **postedAt** : will map to the posted_at column with the datetime type.
- **type**: (optional, defaults to 'string') The mapping type to use for the column.
- **name** : (optional, defaults to field name) The name of the column in the database.
- **length** : (optional, default 255) The length of the column in the database. (Applies only if a string-valued column is used).
- **unique** : (optional, default FALSE) Whether the column is a unique key.
- **nullable** : (optional, default FALSE) Whether the database column is nullable.
- **precision** : (optional, default 0) The precision for a decimal (exact numeric) column (applies only for decimal column), which is the maximum number of digits that are stored for the values.
- **scale** : (optional, default 0) The scale for a decimal (exact numeric) column (applies only for decimal column), which represents the number of digits to the right of the decimal point and must not be greater than precision

Les **types** Doctrine et leur traduction SQL/PHP (1)

- **string**: SQL VARCHAR / PHP string.
- **integer**: SQL INT / PHP integer.
- **smallint**: SQL SMALLINT / PHP integer.
- **bigint**: SQL BIGINT / PHP string.
- **boolean**: SQL boolean or equivalent (TINYINT) / PHP boolean.
- **decimal**: SQL DECIMAL / PHP string.
- **date**: SQL DATETIME / PHP DateTime object.
- **date_immutable**: SQL DATETIME / PHP DateTimeImmutable object.
- **time**: SQL TIME / PHP DateTime object.
- **time_immutable**: SQL TIME / PHP DateTimeImmutable object.
- **datetime**: SQL DATETIME/TIMESTAMP / PHP DateTime object with the current timezone.
- **datetimetz**: SQL DATETIME/TIMESTAMP / PHP DateTime object with the timezone specified in the value from the database.
- **datetime_immutable**: SQL DATETIME/TIMESTAMP / PHP DateTimeImmutable object with the current timezone.
- **datetimetz_immutable**: SQL DATETIME/TIMESTAMP / PHP DateTimeImmutable object with the timezone specified in the value from the database.

Les types Doctrine et leur traduction SQL/PHP (2)

- **dateinterval**: SQL interval / PHP DateInterval object
- **text**: SQL CLOB / PHP string.
- **object**: SQL CLOB / PHP object using serialize() and unserialize()
- **array**: SQL CLOB / PHP array using serialize() and unserialize()
- **simple_array**: SQL CLOB / one-dimensional PHP array using implode() and explode(), with a comma as delimiter. IMPORTANT Only use this type if you are sure that your values cannot contain a ",".
- **json**: SQL CLOB / PHP array using json_encode() and json_decode(). An empty value is correctly represented as null
- **float**: SQL Float (Double Precision) / PHP double. IMPORTANT: Works only with locale settings that use decimal points as separator.
- **guid**: a database GUID/UUID / PHP string. Defaults to varchar but uses a specific type if the platform supports it.
- **blob**: SQL BLOB / PHP resource stream
- **binary**: SQL binary / PHP resource stream

Migrations : synchronisation PHP/SQL

- Les migrations permettent de créer et/ou mettre à jour la **structure** des tables de la Base de Données en fonction du code PHP des entités
- Deux commandes importantes :
  - **php bin/console make:migration**
    *permet de **créer** le code SQL pour mettre en place une nouvelle version de vos entités*
    (à exécuter à chaque modification de la structure des entités)
  - **php bin/console doctrine:migrations:migrate**
    *permet **d'exécuter** le code SQL pour mettre à jour la BD*
    (à exécuter à chaque modification de la structure des entités)
- Le code SQL généré est consultable dans le répertoire **src/Migrations** de votre projet
- La commande **make:migration** est « intelligente ». Le code qu'elle génère prend en compte les modifications que vous avez faites côté PHP et ce qui existe déjà côté SQL => lorsque vous la relancez, elle n'écrase pas l'existant mais l'adapte en fonction des modifications
- Vous pouvez par exemple rajouter des attributs à une entité déjà existante :
  - En exécutant à nouveau la commande : **php bin/console make:entity**
  - Ou en modifiant le PHP puis en lançant : **php bin/console make:entity --regenerate**

  L'exécution de **make:migration** et de **doctrine:migrations:migrate** générera et exécutera alors le code SQL nécessaire pour modifier la table SQL correspondante. Elle ne supprimera pas la table existante ni les éventuelles données qui s'y trouvaient !

Persister en base

- Instancier une entité dans un contrôleur et la faire persister en BD se fait de la façon suivante :

```php
// src/Controller/DefaultController.php
use app\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
use Doctrine\Persistence\ManagerRegistry;
// ...

public function createProduct(ManagerRegistry $doctrine): Response {

    $product = new Product();
    $product->setName('Un nouveau produit');
    $product->setPrice(19.99);
    $product->setDescription('Lorem ipsum dolor');

    $em = $doctrine->getManager();
    $em->persist($product);
    $em->flush();

    return new Response('Produit créé avec id '.$product->getId());
}
```

***Gestionnaire d'Entités* ($em) :** service responsable de la persistance et de la récupération des objets vers et depuis la base de données

persist *indique au gestionnaire d'entités de « gérer » l'objet product*

flush *provoque la mise a jour de la BD pour tous les objets suivis par le gestionnaire d'entités*

*La classe **Product** créée par l'ORM contient des getters et des setters pour manipuler tous les attributs spécifiés dans le mapping objet-relationnel*

Mise à Jour / Suppression d'une entité

- Mettre à jour une entité nécessite trois étapes :
  - Récupérer l'objet depuis Doctrine puis le modifier
  - Appeler la méthode **flush()** du gestionnaire d'entités (Entity Manager) de Doctrine

```php
use Doctrine\Persistence\ManagerRegistry;

public function updateProduct(ManagerRegistry $doctrine, int $id) : Response {

    $product = $doctrine->getRepository(Product::class)->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }
    $product->setName('New product name!');
    $em = $doctrine->getManager();
    $em->flush();

    return $this->redirectToRoute('homepage')); // par exemple
}
```

- Pour supprimer une entité de la BD, il faut indiquer sa suppression au gestionnaire d'entités
- La suppression sera effective lors du prochain **flush()**

```php
$em->remove($product);
$em->flush();
```

Récupérer des Entités depuis la BD (1)

- Dans un contrôleur, pour récupérer un objet depuis une table de la BD, il faut :
  - Récupérer l'objet **Repository** (dépôt) associé à cette table
    - En passant par le Gestionnaire d'Entités (**`ManagerRegistry`**), comme vu page précédente
    - Ou en l'injectant explicitement (**`ProductRepository`**), comme illustré ci-dessous
  - Utiliser une des méthodes standard qu'il propose pour faire une requête :

*On accède au **Repository** associé à la classe **Product***

*La méthode **find** de ce repository permet de récupérer un objet **Product** en fournissant sa clé ($id)*

```php
public function showProduct(ProductRepository $productRepository,
                            int $id) : Response {

  $product = $productRepository->find($id);

  if (!$product) {
    throw $this->createNotFoundException('Produit non trouvé avec id '.$id);
  }

  // Faire qq chose… Par exemple passer l'objet $product à un template…
}
```

Récupérer des Entités depuis la BD (2)

- Un **Repository** propose plusieurs méthodes standards pour rechercher des objets :

```php
// Trouver un produit par sa clé primaire (souvent un id)
$product  = $repository->find($id);

// Trouver tous les produits
$products = $repository->findAll();

// Trouver un produit en combinant plusieurs attributs
$product  = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));

// Trouver tous les produits en combinant plusieurs attributs, avec tri
$products = $repository->findBy( array('name'  => 'foo', 'price' => 19.99),
                                 array('price' => 'asc')
                               );
```

- Lorsqu'une requête est susceptible de renvoyer plusieurs résultats (**`findAll`**, **`findBy`**…), elle le fait sous la forme d'un tableau (array) d'entités

- Lorsqu'une requête est censée ne renvoyer qu'un seul objet (**`find`**, **`findOneBy`**…) mais qu'il y a plusieurs candidats possibles, la requête lève une exception de type `Doctrine\ORM\NonUniqueResultException`

Programmer ses propres requêtes (DQL)

- La commande **make:entity** crée non seulement la classe PHP d'une entité mais aussi la classe PHP de son **repository** (dépôt)
- Il est possible de venir ajouter de nouvelles méthodes à un **repository** afin d'y programmer vos propres requêtes
- **Exemple** : recherche des produits dont le prix est supérieur à une valeur donnée :

```php
// src/Repository/ProductRepository.php

class ProductRepository extends ServiceEntityRepository {

  /**
   * @return Product[]
   */
  public function findAllGreaterThanPrice(float $price): array {
    $entityManager = $this->getEntityManager();
    $query = $entityManager->createQuery(
            'SELECT p
             FROM App\Entity\Product p
             WHERE p.price > :price
             ORDER BY p.price ASC'
    )->setParameter('price', $price);

    // On renvoie un tableau de Produits
    return $query->getResult();
  }
}
```

La requête, exprimée en DQL, peut porter sur un (ou plusieurs) Repository. En DQL on manipule des entités et non des n-uplets

***:price** est un paramètre de la requête auquel il faudra donner une valeur avec setParameter*

Le résultat de la requête est un tableau d'entités extraites du Repository

Programmer ses propres requêtes (Query Builder)

- Doctrine fournit un également un « constructeur de requêtes » (Query Builder) qui permet d'écrire une requête avec un formalisme purement objet

```php
public function findAllGreaterThanPrice($price,
                                $includeUnavailableProducts = false): array {
    // automatically knows to select Products
    // the "p" is an alias you'll use in the rest of the query
    $qb = $this->createQueryBuilder('p')
        ->where('p.price > :price')
        ->setParameter('price', $price)
        ->orderBy('p.price', 'ASC');

    if (!$includeUnavailableProducts) {
        $qb->andWhere('p.available = TRUE')
    }

    $query = $qb->getQuery();

    return $query->execute();

    // Pour obtenir un seul résultat :
    // $product = $query->setMaxResults(1)->getOneOrNullResult();
}
```

Utiliser ses propres requêtes

- Lorsque l'on a développé une nouvelle requête dans un Repository, celle-ci s'utilise évidemment comme les requêtes standards fournies par ce même repository :

```php
// Depuis un controleur par exemple

$minPrice = 1000;

$products = $doctrine->getRepository(Product::class)
                     ->findAllGreaterThanPrice($minPrice);

// ...
```

Relations entre Entités (1)

- Doctrine permet de modéliser tous les types possibles de **relations** entre entités et de « fabriquer » le code SQL nécessaire pour mettre en oeuvre ces relatinos au niveau de la BD :
  - OneToOne
  - ManyToOne / OneToMany
  - ManyToMany
- On va voir l'association la plus courante : ManyToOne / OneToMany en prenant l'exemple des Produits et des Catégories :
  - Plusieurs produits sont associés à une même catégorie : ManyToOne du côté Produit
  - Une catégorie est associée à plusieurs produits : OneToMany du côté Catégorie

  Pour traduite cela en SQL, il faudrait définir dans la table **produit** une clé étrangère (**idCategorie**) qui permette d'associer un produit à la catégorie auquel il appartient.

  Doctrine permet de traiter très facilement cela : il suffit de préciser qu'une **Entité Produit** possède un attribut **categorie** qui est une **Entité Categorie**... Et c'est réglé ! Magique !

- *Pour plus de détails sur les autres types d'association : voir la documentation détaillée de Doctrine: https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html*

Relation ManyToOne / OneToMany (1)

- Supposons qu'une entité **Category** a déjà été créée (avec un attribut **id** et un attribut **name**)
- Pour mettre en place l'association **ManyToOne** entre **Product** et **Category**, il suffit, lors de la création de l'entité Product, de lui ajouter un attribut de type « **relation** » :

```
> php bin/console make:entity
Class name of the entity to create or update (e.g. BraveChef):
> Product
               to stop adding fields):
> category
Field type (enter ? to see all types) [string]:
> relation
What class should this entity be related to?:
> Category
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne
Is the Product.category property allowed to be null (nullable)? (yes/no) [yes]:
> no
Do you want to add a new property to Category so that you can access/update
getProducts()? (yes/no) [yes]:
> yes
New field name inside Category [products]:
> products
Do you want to automatically delete orphaned App\Entity\Product objects
(orphanRemoval)? (yes/no) [no]:
> yes
                to stop adding fields):
>
(press enter again to finish)
```

Un objet Product aura un attribut category de classe Category

Un objet Category aura un attribut products de type tableau de Product

Relation ManyToOne / OneToMany (2)

- Dans la classe PHP **Product**, un attribut **category** (de type **Category)** a été ajouté, avec son annotation décrivant la relation (**ManyToOne**) ainsi que les méthodes nécessaires pour la gérer
- Côté SQL, une clé étrangère sera mise en place dans la table **product,** mais nous n'avons pas à nous en préoccuper lorsque nous utilisons la classe Product !

```php
// src/Entity/Product.php
// ...
class Product {
    // ...

    #[ORM\ManyToOne(targetEntity: Category::class, inversedBy: 'products')]
    private $category;

    public function getCategory(): ?Category {
        return $this->category;
    }

    public function setCategory(?Category $category): self {
        $this->category = $category;
        return $this;
    }
}
```

Relation ManyToOne / OneToMany (3)

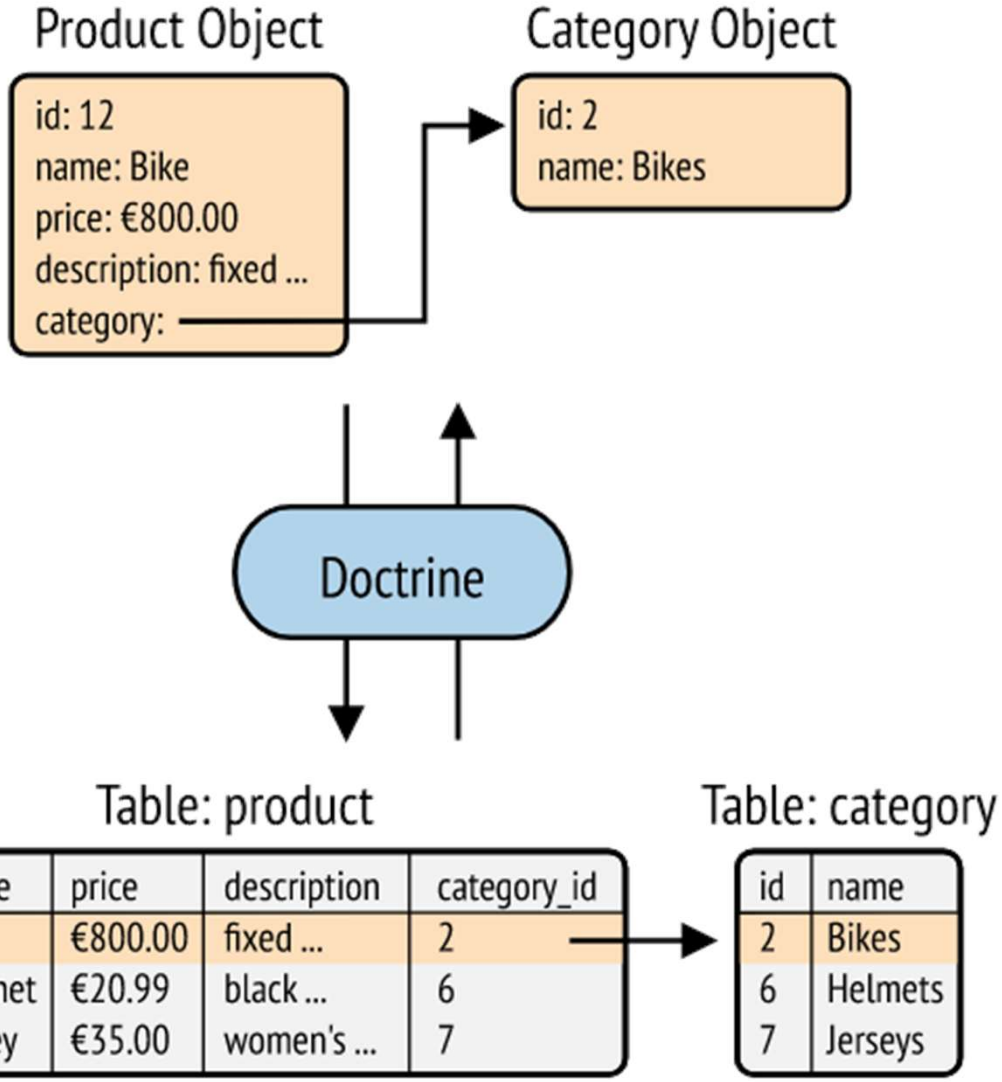
- Dans la classe **Category**, un attribut **products** a été rajouté avec son annotation décrivant la relation (**OneToMany**) ainsi que les méthodes nécessaires pour la gérer
- C'est un tableau (ArrayCollection) qui contiendra tous les produits de la catégorie

```php
// src/Entity/Category.php
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
class Category {
    // ...

    #[ORM\OneToMany(targetEntity: Product::class, mappedBy: 'category')]
    private $products;

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

    /**
     * @return Collection|Product[]
     */
    public function getProducts(): Collection {
        return $this->products;
    }
    // addProduct() and removeProduct() ont aussi été ajoutés
}
```

Relation ManyToOne / OneToMany (4)

- Pour mettre en place l'association côté SQL, il faut taper les deux commandes suivantes :
  - **php bin/console make:migration**
  - **php bin/console doctrine:migrations:migrate**

- On peut ensuite aller constater sur PHPMyAdmin qu'une clé étrangère a bien été créée dans la table **product**



Relation ManyToOne / OneToMany (5)

- La mise en place d'une **relation** entre deux entités permet de disposer, dans ces entités, de getters et de setters qui permettent de « **naviguer** » dans le modèle de données sans avoir besoin de se préoccuper de jointures !

```php
use App\Entity\Product;
use App\Entity\Category;
use App\Repository\CategoryRepository;
use App\Repository\ProductRepository;

public function showProduct(ProductRepository $productRepository,
                            int $idProduct): Response {

    $product = $productRepository->find($idProduct);

    // A partir d'un produit, récupérer SA catégorie
    $category = $product->getCategory();
}

public function showProducts(CategoryRepository $categoryRepository,
                             int $idCategory): Response {

    $category = $categoreyRepository->find($idCategory);

    // A partir d'une catégorie, récupérer SES produits
    $products = $category->getProducts();
}
```