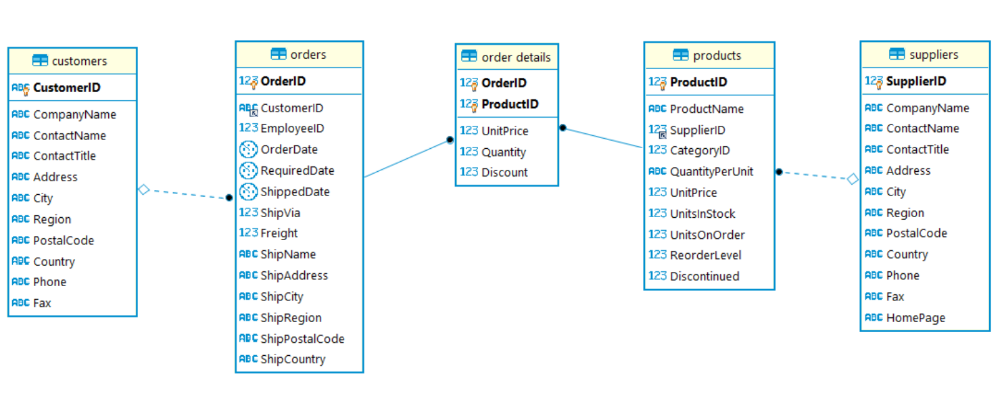
Découverte de Doctrine

Pour l'exemple, nous allons utiliser la base northwind que vous connaissez déjà. Vous pouvez également la base Record, ou appliquer ce tuto directement sur votre fil rouge.

Le modèle relationnel ressemble à ceci:



Découverte et mise en place de Doctrine

Si vous avez choisi le 'website-skeleton' lors de l'initialisation de votre projet symfony, Doctrine devrait être déja installé sur le projet.

Si ce n'est pas le cas, ouvrez un terminal et entrez les commandes suivantes :

```
composer require symfony/orm-pack
```

Il nous faut ensuite configurer l'accès à notre base.

Pour cela, nous allons nous rendre dans le fichier `.env` du projet :

```
1  # In all environments, the following files are loaded if they exist,
2  # the latter taking precedence over the former:
3  #
4  #  * .env                contains default values for the environment variables needed by the app
5  #  * .env.local          uncommitted file with local overrides
6  #  * .env.$APP_ENV       committed environment-specific defaults
7  #  * .env.$APP_ENV.local uncommitted environment-specific overrides
8  #
9  # Real environment variables win over .env files.
10 #
11 # DO NOT DEFINE PRODUCTION SECRETS IN THIS FILE NOR IN ANY OTHER COMMITTED FILES.
12 #
13 # Run "composer dump-env prod" to compile .env files for production use (requires symfony/flex >=1.2).
14 # https://symfony.com/doc/current/best_practices.html#use-environment-variables-for-infrastructure-configuration
15
16 ###> symfony/framework-bundle ###
17 APP_ENV=dev
18 APP_SECRET=179b22031468b9a7d0f13f278c01beb6
19 #TRUSTED_PROXIES=127.0.0.0/8,10.0.0.0/8,172.16.0.0/12,192.168.0.0/16
20 #TRUSTED_HOSTS='^(localhost|example\.com)$'
21 ###< symfony/framework-bundle ###
22
23 ###> symfony/mailer ###
24 # MAILER_DSN=smtp://localhost
25 ###< symfony/mailer ###
26
27 ###> doctrine/doctrine-bundle ###
28 # Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url
29 # For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
30 # For a PostgreSQL database, use: "postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=11&charset=utf8"
31 # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
32 DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
33 ###< doctrine/doctrine-bundle ###
34
```

La ligne 32 correspond à la connexion à la base de données. Nous devons donc la modifier pour que la connexion se fasse. Il suffit de remplacer `db_user`, `db_password`, `db_name` et éventuellement `127.0.0.1:3306` si ça en correspond à votre serveur local :

```
DATABASE_URL=mysql://root:@127.0.0.1:3306/northwind?serverVer
```

Création d'une première entité

Si la base Northwind n'est pas sur votre serveur, importez-la.

Nous allons nous intéresser à la table `products`. Nous allons créer une première entité nommée `Products`. Pour cela, plaçons-nous dans le dossier `Entity` de notre projet et créons un fichier `Products.php` voici le squelette de notre classe `Product`.

```
// Entity/Product.php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 */
class Products
{
    /**
     * @ORM\Column(name="ProductId", type="integer", nullable
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

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

}
```

- `@ORM\Table(name="products")` : permet de spécifier que la classe `Product` est associée à la table `products`
- `@ORM\Column(name="ProductId", type="integer", nullable=false)` : permet de mapper l'attribut `$id` avec la colonne `ProductId`
- `@ORM\Id` : spécifie que cet attribut représente l'`Id` (la clé primaire de la table).

Pour l'instant elle ne possède qu'un attribut `Id`, nous allons la tester avant d'aller plus loin.

Mise en place d'un test simple

Nous allons d'abord créer un contrôleur :

```
// Controller/TestController
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractControl
use Symfony\Component\Routing\Annotation\Route;

use App\Entity\Product;

class TestController extends AbstractController
{
    /**
     * @Route("/test", name="test")
     */
    public function index()
    {
        $repo = $this->getDoctrine()->getRepository(Product::
        $obj = $repo->findAll();

        return $this->render('test/index.html.twig', [
            'obj' =>  $obj
        ]);
    }
}
```

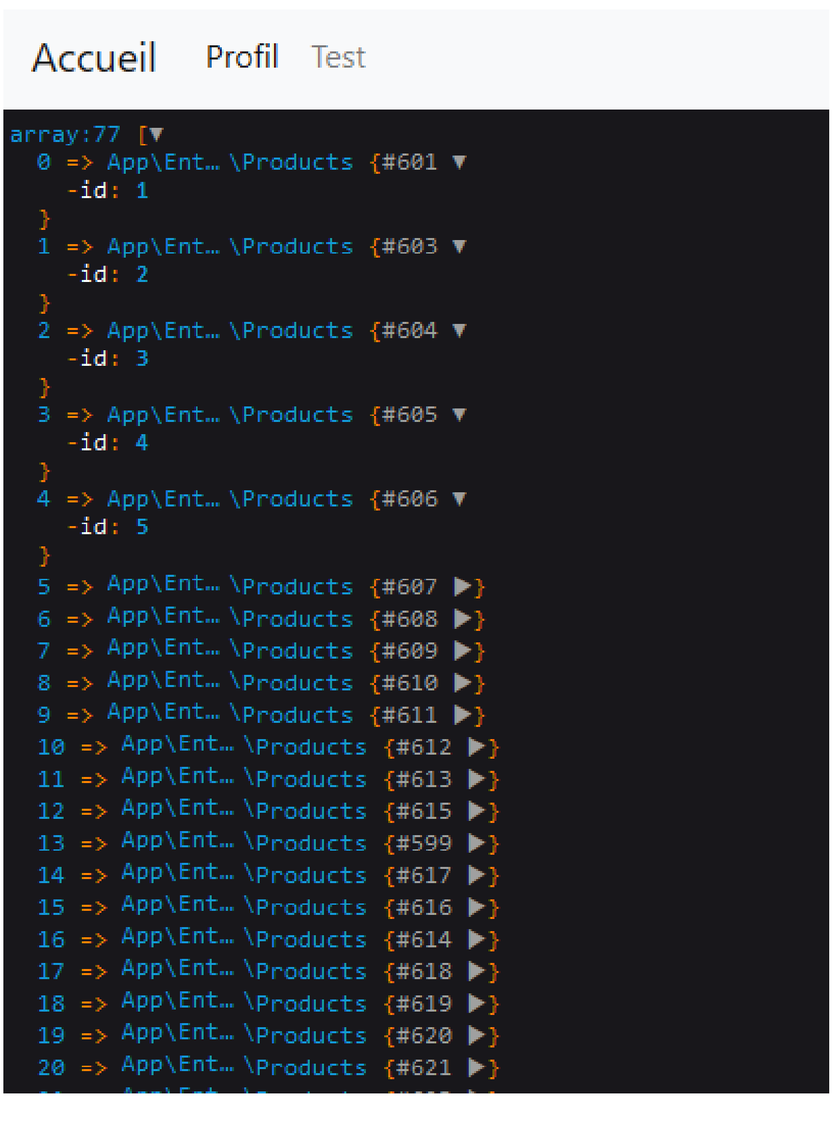
Dans l'exemple ci-dessus, nous utilisons le `Repository` par défaut avec la fonction `getRepository(Product::class)`. Il va nous permettre de récupérer toutes les lignes de la table `products` mappées dans des objets `Products` à l'aide de la méthode `findAll()`

Créons maintenant la vue nécessaire pour notre affichage et ajoutons un bouton (lien) sur notre navbar pour y avoir accès facilement :

```
<!-- templates/test/index.html.twig -->
{% extends 'base.html.twig' %}

{% block body %}
    {{ dump(obj) }}
{% endblock %}
```

Cette vue est volontairement très simple, elle permet d'afficher (à des fins de débuggage) la structure de la variable `$obj` initialisée dans le contrôleur :

Testez le résultat en ouvrant `http://127.0.0.1:8000/test` (ou `localhost/symfony/monprojet/public/index.php` selon votre structure) dans votre navigateur.



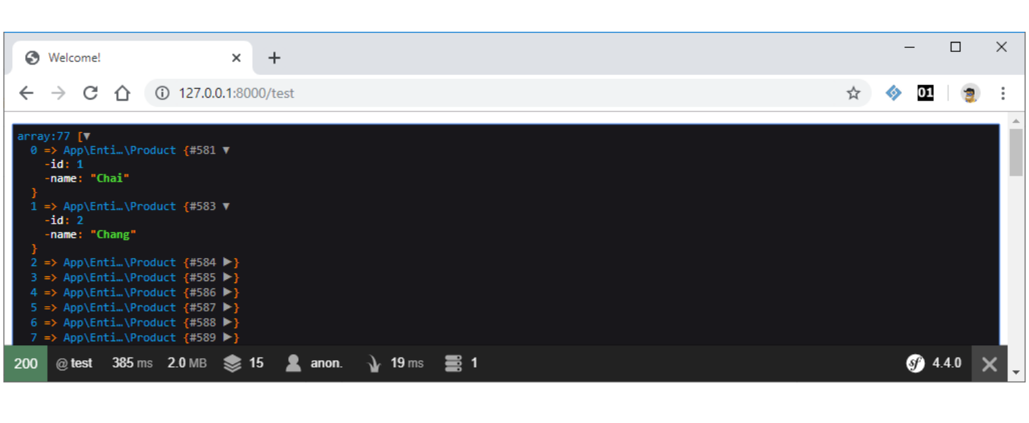
```
array:77 [▼
  0 => App\Ent… \Products {#601 ▼
    -id: 1
  }
  1 => App\Ent… \Products {#603 ▼
    -id: 2
  }
  2 => App\Ent… \Products {#604 ▼
    -id: 3
  }
  3 => App\Ent… \Products {#605 ▼
    -id: 4
  }
  4 => App\Ent… \Products {#606 ▼
    -id: 5
  }
  5 => App\Ent… \Products {#607 ▶}
  6 => App\Ent… \Products {#608 ▶}
  7 => App\Ent… \Products {#609 ▶}
  8 => App\Ent… \Products {#610 ▶}
  9 => App\Ent… \Products {#611 ▶}
  10 => App\Ent… \Products {#612 ▶}
  11 => App\Ent… \Products {#613 ▶}
  12 => App\Ent… \Products {#615 ▶}
  13 => App\Ent… \Products {#599 ▶}
  14 => App\Ent… \Products {#617 ▶}
  15 => App\Ent… \Products {#616 ▶}
  16 => App\Ent… \Products {#614 ▶}
  17 => App\Ent… \Products {#618 ▶}
  18 => App\Ent… \Products {#619 ▶}
  19 => App\Ent… \Products {#620 ▶}
  20 => App\Ent… \Products {#621 ▶}
```

Ajout d'une propriété `Name`

Dans l'entité `Products`, ajoutez une propriété `Name` pour mapper la colonne `ProductName`

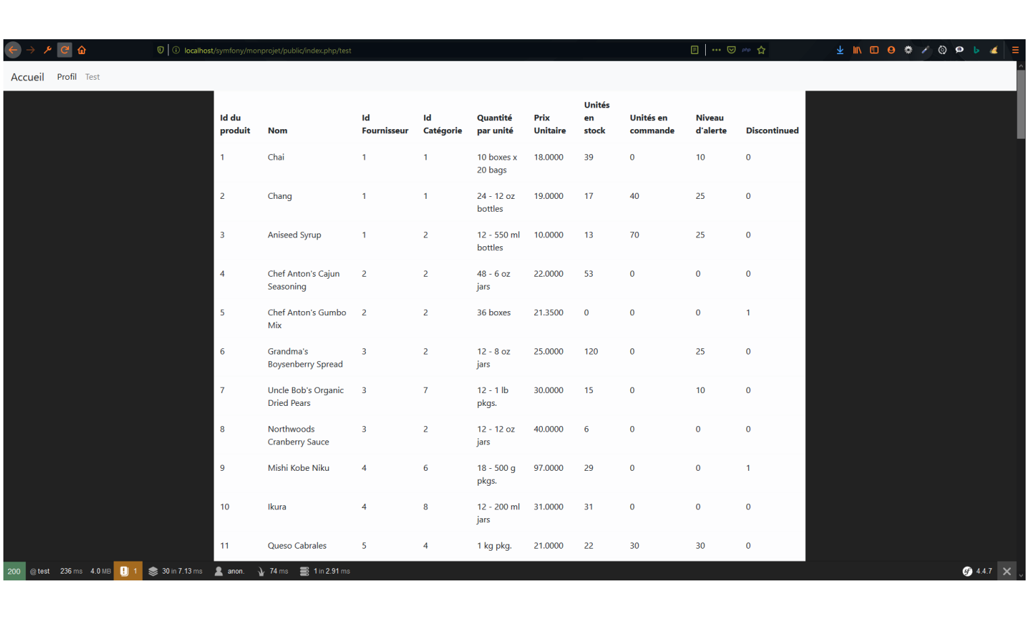
```
    /**
     * @ORM\Column(name="ProductName", type="string", length=
     */
    private $name;

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

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

        return $this;
    }
```

Testez le résultat, vérifiez l'existence de la propriété `name`.



```
array:77 [▼
  0 => App\Enti…\Product {#581 ▼
    -id: 1
    -name: "Chai"
  }
  1 => App\Enti…\Product {#583 ▼
    -id: 2
    -name: "Chang"
  }
  2 => App\Enti…\Product {#584 ▶}
  3 => App\Enti…\Product {#585 ▶}
  4 => App\Enti…\Product {#586 ▶}
  5 => App\Enti…\Product {#587 ▶}
  6 => App\Enti…\Product {#588 ▶}
  7 => App\Enti…\Product {#589 ▶}
```

Faites de même pour ajouter les autres propriétés de la table et formatez la vue pour obtenir l'affichage suivant :

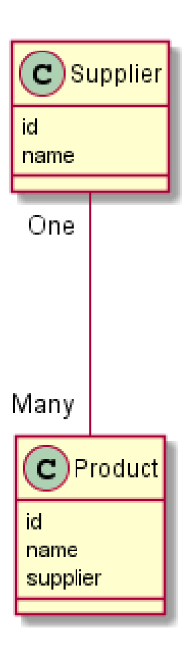


Id du produit	Nom	Id Fournisseur	Id Catégorie	Quantité par unité	Prix Unitaire	Unités en stock	Unités en commande	Niveau d'alerte	Discontinued
1	Chai	1	1	10 boxes x 20 bags	18.0000	39	0	10	0
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25	0
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25	0
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0	1
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25	0
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10	0
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0	0	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0	0	1
10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0	0	0
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30	0

Mise en place d'une relation entre deux tables

La table `products` est reliée avec la table `suppliers` par l'attribut `SupplierID`.

- Plusieurs produits sont associés à un seul fournisseur.
- Un fournisseur fournit plusieurs de produits.



> C'est une relation `ManyToOne` (point de vue produit). Et nous verrons plus loin, une relation `OneToMany` (point de vue fournisseur).

Relation `ManyToOne`

Commençons par créer une entité pour les fournisseurs en reprenant le même schéma que pour les produits.

```
// Entity/Suppliers
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="suppliers")
 */
class Suppliers
{
    /**
     * @ORM\Column(name="SupplierId", type="integer", nullabl
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

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

    /**
     * @ORM\Column(name="CompanyName", type="string", length=
     */
    private $name;

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

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

        return $this;
    }

}
```

Ensuite nous pouvons relier les deux entités. Dans la classe `Products` nous déclarons une propriété `suppliers` qui sera une instance de la classe `Suppliers`.

```
    /**
     * @var \Suppliers
     *
     * @ORM\ManyToOne(targetEntity="Suppliers")
     * @ORM\JoinColumn(name="SupplierId", referencedColumnNam
     *
     */
    private $suppliers;

    public function getSuppliers(): ?Suppliers
    {
        return $this->suppliers;
    }

    public function setSuppliers(?Suppliers $supplier): self
    {
        $this->suppliers = $suppliers;

        return $this;
    }
```

Dans les annotations:

`ManyToOne(targetEntity="Suppliers")` permet de spécifier le type de l'entité à mapper, dans ce cas l'attribut `$suppliers` contiendra un objet de type `Suppliers`

`JoinColumn(name="SupplierID", referencedColumnName="SupplierID")` représente la traduction en modèle relationnel : `name` pour la clé étrangère, `referencedColumnName` pour la clé primaire de la table jointe.

Maintenant chaque objet de type `Products` possède une propriété `Suppliers` qui contient une instance de la classe `Suppliers`.

Testez le résultat en utilisant soit `dump(obj)` dans la vue ou `dd($obj)` dans le contrôleur.

Vous constaterez que le `dump` dans la vue n'affiche pas le détail du fournisseur. En effet doctrine utilise un mode de chargement paresseux (`Lazy-loading`)

Ajoutez la ligne ci-dessous dans votre contrôleur

```
$obj[0]->getSuppliers()->getName();
```

Quelle différence cela fait-il ?

Comment faire pour charger tous les fournisseurs ?

Modifiez votre tableau pour afficher le nom du fournisseur à la place de l'id pour chaque produit.

Préciser la stratégie de récupération

`@ManyToOne(targetEntity="Supplier", fetch="LAZY")`

Le mode LAZY est celui par défaut. Les données sont chargées uniquement si nécessaires et l'appel aux données de relations provoque une requête supplémentaire.

`@ManyToOne(targetEntity="Supplier", fetch="EAGER")`

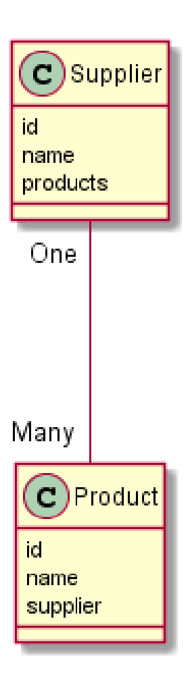
Le mode EAGER précharge les données de la relation automatiquement en réalisant la jointure par défaut. Ainsi votre entité sera nettement plus grande mais vous économiserez des requêtes.

Relation `OneToMany`

Avec Doctrine, une relation `OneToMany` est basée sur la relation `ManyToOne` associée. Il s'agit de la relation inverse.

Doctrine appelle cela une relation bidirectionnelle.

La relation bidirectionnelle requiert une attribut `mappedBy` du coté `One` et un attribut `inversedBy` du coté `Many`.



Ajout de la propriété `products` dans la classe `Suppliers`

Dans l'entité `Suppliers`, commençons par créer une propriété `products` qui portera la relation `OneToMany`

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

/**
* @ORM\OneToMany(targetEntity="Product", mappedBy="supplier",
*/
private $products;

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

Puis un accesseur

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

public function addProducts(Products $products): self
{
    if (!$this->products->contains($products)) {
        $this->products[] = $products;
        $products->setSuppliers($this);
    }

    return $this;
}
```

Maintenant dans l'entité `Products`, modifions la relation `ManyToOne` pour lui spécifier l'attribut `inversedBy` :

```
/**
* @ORM\ManyToOne(targetEntity="Supplier", inversedBy="product
*/
```

Mettez en place toutes les relations entre les différentes tables de votre base.