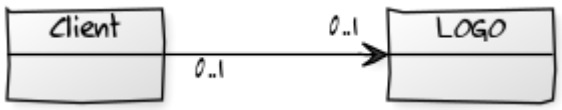
Symfony Partie 2
Relation entre entités

Relation uni directionnelle

Nous allons gérer les relations entre entités grâce aux annotations de Doctrine.
On doit définir une entité propriétaire. On choisira la classe Client.

Relation One to One




```
/**
 * @ORM\Entity
 */
class Client
{
  /**
   * @ORM\OneToOne(targetEntity="Ppe\SpaceBookingBundle\Entity\Logo")

  private $monLogo;
```

Il faut créer les deux entités, ajouter l'annotation précédente. On précisera le chemin complet depuis le fichier source de l'entité.

Il faut aussi ajouter les accesseurs et modificateurs de la nouvelle propriété. On ajoutera ceux-ci grâce à la console.

```
php app/console doctrine:generate:entities PpeSpaceBookingBundle:Client
```

Et on met à jour la BD.

```
php app/console doctrine:schema:update --force
```

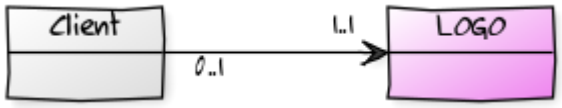
Le code généré :

```
   /**
   * @param Ppe\SpaceBookingBundle\Entity\Logo $logo
   */
  public function setLogo(\Ppe\SpaceBookingBundle\Entity\Logo $logo=null)
  {
    $this->monLogo = $logo;
  }

  /**
   * @return Ppe\SpaceBookingBundle\Entity\Logo
   */
  public function getLogo()
  {
    return $this->monLogo;
  }
}
```

Lors de la création des deux objets, on les lie grâce au modificateur de l'entité Client `$monClient->setLogo($monLogo)` . On enregistrera par la suite les entités dans la base.

One to One obligatoire

On peut rendre obligatoire la liaison :




```
/**
 * @ORM\Entity
 */
class Client
{
  /**
   * @ORM\OneToOne(targetEntity="Ppe\SpaceBookingBundle\Entity\Logo")
   * @ORM\JoinColumn(nullable=false)
 */
  private $monLogo;
```

```
   /**
    * Set logo
    *
    * @param Ppe\SpaceBookingBundle\Entity\Logo $logo
    */
   public function setLogo(\Ppe\SpaceBookingBundle\Entity\Logo $logo=null)
   {
     $this->monLogo = $logo;
   }

   /**
    * Get logo
    *
    * @return Ppe\SpaceBookingBundle\Entity\Logo
    */
   public function getLogo()
   {
     return $this->monLogo;
   }
}
```

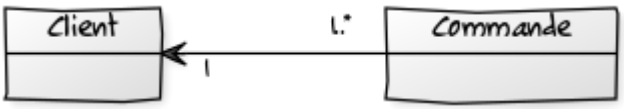
On ajoute l'annotation suivante pour rendre obligatoire la liaison et on supprime la possibilité de donner un paramètre null.

Les options

```
   * @ORM\OneToOne(targetEntity="Ppe\SpaceBookingBundle\Entity\Logo", cascade={"persist"})
```

Cette option va propager la persistance de l'entité client sur l'entité Logo.

Relation Many to One




```
/**
 * @ORM\Entity
 */
class Commande
{
  /**
   * @ORM\ManyToOne(targetEntity=" Ppe\SpaceBookingBundle\Entity\Client")
   * @ORM\JoinColumn(nullable=false)
   */
  private $monClient;
}
```

On ajoutera les accesseurs et modificateurs grâce à la console

```
php app/console doctrine:generate:entities PpeSpaceBookingBundle:Commande
```

Et on met à jour la base de données.

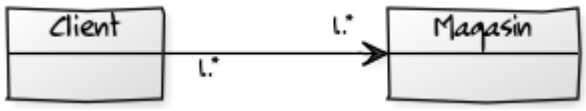
```
php app/console doctrine :schema :update –force
```

```
  /**
   * Set client
   *
   * @param Ppe\SpaceBookingBundle \Entity\Client $client
   */
  public function setClient(\Ppe\SpaceBookingBundle\Entity\Client $client)
  {
    $this->monClient = $client;
  }

  /**
   * Get client
   *
   * @return Ppe\SpaceBookingBundle\Entity\Client
   */
  public function getClient()
  {
    return $this->monClient;
  }
}
```

Le principe est le même, on change l'annotation OneToOne en ManyToOne.

Relation Many to Many

On représente ici la visite de clients dans des magasins. On doit définir une entité propriétaire. On choisira client. Seule l'entité propriétaire est à modifier :




```
/**
 * @ORM\Entity
 */
class Client
{
  /**
   * @ORM\ManyToMany(targetEntity=" Ppe\SpaceBookingBundle\Entity\Magasin",
cascade={"persist"})
   */
  private $magasins;
}
```

Doctrine fonctionne avec une table intermédiaire. On notera que l'entité Client contient une collection d'objets magasins (par convention, on applique le pluriel à l'attribut).

On générera le code des accesseurs et modificateurs via la console (on peut lancer cette commande sans crainte, s'ils sont déjà générés, ils ne le seront pas de nouveau).

```
php app/console doctrine:generate:entities PpeSpaceBookingBundle:Client
```

Et on met à jour la base de données.

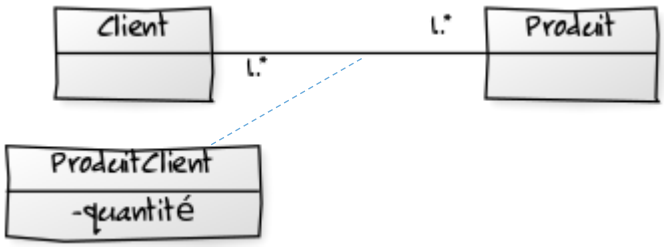
```
php app/console doctrine:schema:update --force
```

```
/**
     * Constructor
     */
    public function __construct()
    {
        $this->magasins = new \Doctrine\Common\Collections\ArrayCollection();
    }

    /**
     * Add magasin
     *
     * @param \Ppe\SpaceBookingBundle\Entity\Magasin $magasins
     * @return Client
     */
    public function addMagasin(\Ppe\SpaceBookingBundle\Entity\Magasin $magasin)
    {
        $this->magasins[] = $magasin;

        return $this;
    }

    /**
     * Remove magasin
     *
     * @param \Ppe\SpaceBookingBundle\Entity\Magasin $magasins
     */
    public function removeMagasin(\Ppe\SpaceBookingBundle\Entity\Magasin $magasin)
```

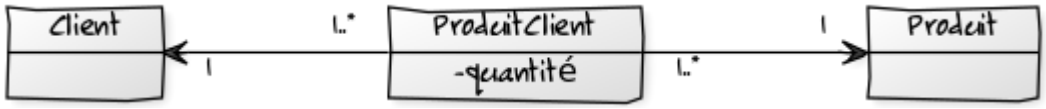
```
    {
        $this->magasins->removeElement($magasin);
    }

    /**
     * Get magasins
     *
     * @return \Doctrine\Common\Collections\Collection
     */
    public function getMagasins()
    {
        return $this->magasins;
    }
```

Comme Doctrine fonctionne avec une collection, il n'y a plus de set mais une fonction add et un constructeur pour initialiser la collection.

Relation Many to Many avec donnée(s) portée(s)



Comme en UML, il est nécessaire de réaliser une entité intermédiaire pour gérer la donnée portée. La relation Many to Many se transforme donc en deux relations Many to One. On mettra une propriété produit et client dans l'entité ProduitClient. Ce qui pourrait nous donner le schéma suivant :



```
/**
 * @ORM\Entity
 */
class ProduitClient
{
  /**
   * @ORM\Id
   * @ORM\ManyToOne(targetEntity=" Ppe\SpaceBookingBundle\Entity\Client")
   * @ORM\JoinColumn(nullable=false)
   */
  private $monClient;

 /**
    * @ORM\Id
   * @ORM\ManyToOne(targetEntity="Ppe\SpaceBookingBundle\Entity\Produit")
   * @ORM\JoinColumn(nullable=false)
   */
  private $monProduit;

    /**
     * @var quantite
     *
     * @ORM\Column(name="quantite", type="integer")
     */
  private $quantite;
}
```

On dispose d'une clef primaire concaténée dans la base de données, qui est représentée ici par les deux Id.

On générera le code des accesseurs et modificateurs via la console

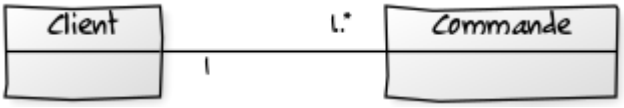
```
php app/console doctrine:generate:entities PpeSpaceBookingBundle:ProduitClient
```

Et on met à jour la base de données.

```
php app/console doctrine:schema:update --force
```

Relation bi directionnelle

On reprend l'exemple de la relation Many to One.
On veut pouvoir accéder aux commandes du client. Il faut pour cela rendre bi directionnelle la relation.



On change en premier la relation client

```
class Client
{
  /**
   * @ORM\OneToMany(targetEntity=" Ppe\SpaceBookingBundle\Entity\Commande",
mappedBy="monClient")
   */
  private $commandes;
}
```

On utilise l'inverse de la notation, soit One to Many (au lieu de Many to One).
L'attribut mappedBy permet de designer la propriété de la classe de destination qui correspond à notre entité (désigne l'attribut de Commande responsable de la liaison entre une instance de Commande et une instance de Client).

Puis Commande

```
 class Commande
{
  /**
   * @ORM\ManyToOne(targetEntity=" Ppe\SpaceBookingBundle\Entity\Client",
inversedBy="commandes")
   * @ORM\JoinColumn(nullable=false)
   */
  private $monClient;
}
```

Ici on précise la propriété de la classe Client.

Attention : la classe possédant inversedBy est considérée comme propriétaire par doctrine.
Tout changement du côté de la relation inverse est ignoré. Il faudra vérifier que le changement a lui aussi du côté propriétaire.

http://doctrine-orm.readthedocs.org/en/latest/reference/unitofwork-associations.html

On générera le code des accesseurs et modificateurs via la console uniquement pour le client.
La modification de la classe Commande n'a pas d'influence sur les accesseurs et modificateurs.

```
php app/console doctrine:generate:entities PpeSpaceBookingBundle:Client
```

Et on met à jour la BD.

```
php app/console doctrine:schema:update --force
```

Dans le code, lors de l'utilisation des classes, il sera nécessaire de lier les objets entre eux.

```
$leClient = new Client() ;
$laCommande = new Commande() ;
$client->addCommande($laCommande) ;
$commande->setClient($leClient) ;
```

Optimisation

Il est possible de réaliser une des deux liaisons à l'ajout de l'élément.

En éliminant cette ligne

```
$client ->addCommande($laCommande) ;
```

Et en ajoutant dans la fonction **setClient** de la classe Commande

```
public function setClient(\Ppe\SpaceBookingBundle\Entity\Client $client){
        $this->monClient = $client;
       $client->addCommande($this) ;

    }
```

La classe Client n'étant pas propriétaire, il n'est pas recommandé de réaliser l'inverse :

```
$commande ->setClient($leClient) ;
```

Dans la classe Client :

```
public function addCommande(\Ppe\SpaceBookingBundle\Entity\Commande $commande){
        $this->commandes[] = $commande;
       $commande->setClient($this) ;
        return $this;
    }

public function removeCommande(\Ppe\SpaceBookingBundle\Entity\Commande $commande)
    {
        $this->commandes->removeElement($commande);
       $commande->setClient(null) ;
    }
```