Actividad

ORM ,Doctrine y bases de datos

1. Trabajar con la Doctrine usando Asociaciones / Relaciones entre objetos

Hay **dos tipos principales de relaciones/asociaciones** entre entidades:

ManyToOne / OneToMany

> **La relación más común**, asigna en la base de datos a una columna la clave externa (por ejemplo, una columna proceso_id en la tabla tarea es una clave ajena).

`ManyToMany`

> Esta asociación **utiliza una tabla de combinación** y es necesaria cuando ambos lados de la relación pueden tener muchos del otro lado (por ejemplo, "estudiantes" y "clases": cada estudiante está en muchas clases y cada clase tiene muchos estudiantes).

En primer lugar, debes determinar qué relación utilizar. Si ambos lados de la relación contienen muchos de los otros lados (por ejemplo, "estudiantes" y "clases"), necesita una relación MuchosTodosMuchos. De lo contrario, es probable que necesite un ManyToOne.

Consejo

También existe una relación OneToOne (por ejemplo, un usuario tiene un perfil y viceversa). En la práctica, el uso de esto es similar a ManyToOne.

2. La Asociación ManyToOne / OneToMany

Supongamos que cada Tarea de nuestra aplicación pertenece exactamente a un proceso. En este caso, necesitaras una clase Proceso y una forma de relacionar un objeto Tarea con un objeto Proceso.

Comienza por **crear una entidad Proceso** con un campo de nombre como lo hiciste en la actividad anterior

```
daw@daw-VirtualBox:~/my_project_directory$ php bin/console make:entity Proceso
 created: src/Entity/Proceso.php
 created: src/Repository/ProcesoRepository.php

 Entity generated! Now let's add some fields!
 You can always add more fields later manually or by re-running this command.

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

 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]:
 >

 updated: src/Entity/Proceso.php

 Add another property? Enter the property name (or press <return> to stop adding fields):
 >


  Success!

 Next: When you're ready, create a migration with php bin/console make:migration
```

Esto genera la **entidad Proceso.php** como se puede ver a continuación

```php
<?php

namespace App\Entity;

use App\Repository\ProcesoRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: ProcesoRepository::class)]
class Proceso
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $nombre = null;

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

    public function getNombre(): ?string
    {
        return $this->nombre;
    }

    public function setNombre(string $nombre): static
    {
        $this->nombre = $nombre;

        return $this;
```

3. Asignación de la relación ManyToOne

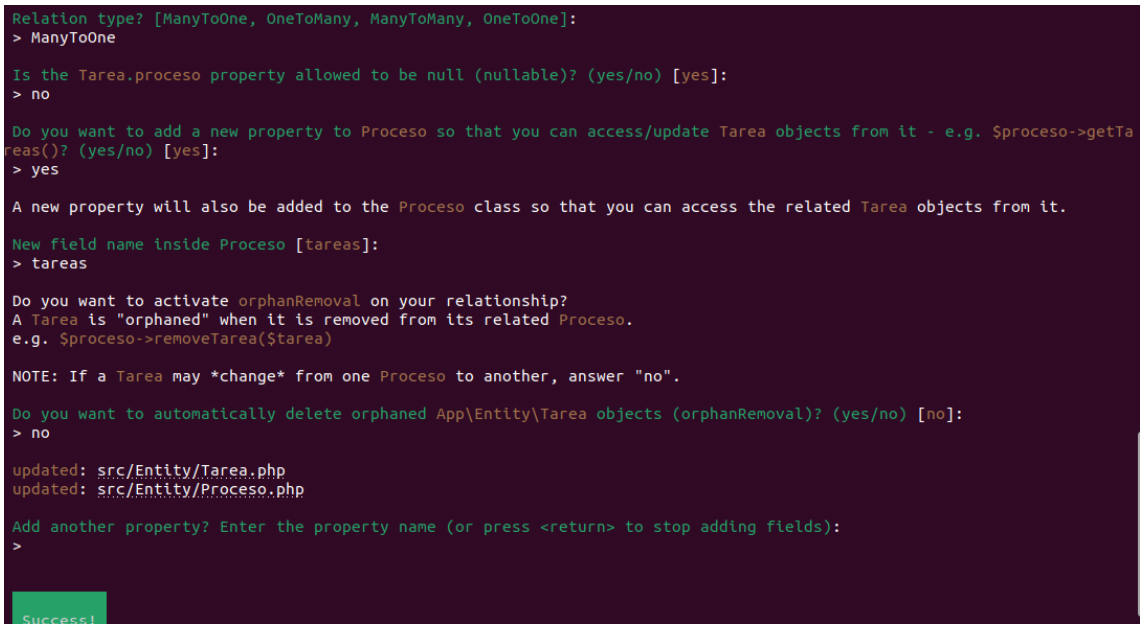
En este ejemplo, **cada Proceso se puede asociar a *muchas* Tareas. Sin embargo, cada Tarea solo se puede asociar a *un* solo Proceso**.

Desde la perspectiva de la entidad Tarea, se trata de una relación de varios a uno. Desde la perspectiva de la entidad Proceso, se trata de una relación de uno a varios.

Para implementarla, primero crea una propiedad de Proceso en la clase tarea con el atributo ManyToOne. **Puedes hacerlo a mano o usando el comando make:entity,** que te hará varias preguntas sobre tu relación. Si no estás seguro de la respuesta, ¡no te preocupes! Siempre puedes cambiar la configuración más adelante:

Vamos a configurar la relación mediante el **comando make:entity**

```
daw@daw-VirtualBox:~/my_project_directory$ php bin/console make:entity

 Class name of the entity to create or update (e.g. OrangePizza):
 > Tarea

 Your entity already exists! So let's add some new fields!

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

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

 What class should this entity be related to?:
 > Proceso

What type of relationship is this?
 ------------ ---------------------------------------------------------------------
  Type         Description
 ------------ ---------------------------------------------------------------------
  ManyToOne    Each Tarea relates to (has) one Proceso.
               Each Proceso can relate to (can have) many Tarea objects.

  OneToMany    Each Tarea can relate to (can have) many Proceso objects.
               Each Proceso relates to (has) one Tarea.

  ManyToMany   Each Tarea can relate to (can have) many Proceso objects.
               Each Proceso can also relate to (can also have) many Tarea objects.

  OneToOne     Each Tarea relates to (has) exactly one Proceso.
               Each Proceso also relates to (has) exactly one Tarea.
 ------------ ---------------------------------------------------------------------
```



Esto ha realizado cambios en las *dos* entidades. En primer lugar, se agregó una nueva propiedad proceso a la entidad Tarea (y los métodos getter y setter):

```php
#[ORM\ManyToOne(inversedBy: 'tareas')]
#[ORM\JoinColumn(nullable: false)]
private ?Proceso $proceso = null;
```

Esta asignación de ManyToOne es necesaria. Le dice a Doctrine que use la columna tarea (id) de la tabla de tareas para relacionar cada registro de esa tabla con un registro de la tabla de procesos.

A continuación, dado *que un* objeto Proceso se relacionará con muchos objetos *Tarea* el comando make:entity también *agregó una* propiedad tareas a la clase Proceso que contendrá estos objetos:

```php
class Proceso
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $nombre = null;

    #[ORM\OneToMany(mappedBy: 'proceso', targetEntity: Tarea::class)]
    private Collection $tareas;
```

El mapeo de ManyToOne mostrado anteriormente es *obligatorio*, pero, este OneToMany es opcional: solo agrégalo *si* deseas poder acceder a las Tarea que están relacionados con una Proceso (esta es una de las preguntas que le hace make:entity). En este

ejemplo, *será* útil poder llamar a $proceso->getTareas(). Si no lo deseas, tampoco necesita la configuración inversedBy o mappedBy.

ArrayCollection

El código dentro de __construct() es importante: La propiedad $procesos **debe ser un objeto de colección que implemente la interfaz Collection de Doctrine** . En este caso, se utiliza un objeto ArrayCollection

¡Tu base de datos está configurada! Ahora, ejecuta las migraciones pertinentes:

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

4. Guardar entidades relacionadas

Vamos a ver todo esto en acción. Para ello vamos a crear dentro del controlador de tarea **un nueva tarea relacionada con un proceso**, guardando estos nuevos objetos en la base de datos. El código que debes añadir es el mostrado en la función index:

```php
class TareaController extends AbstractController
{

    #[Route('/tarea/crear', name: 'tarea_crear')]
    public function index(EntityManagerInterface $entityManager): Response
    {
        $proceso = new Proceso();
        $proceso->setNombre('Analisis');

        $tarea= new Tarea();
        $tarea->setNombre('Diagrama ER');
        $tarea->setPrioridad(5);
        $tarea->setDescripcion('diagrama de datos');

        // relaciona esta tarea al proceso
        $tarea->setProceso($proceso);

        $entityManager->persist($proceso);
        $entityManager->persist($tarea);
        $entityManager->flush();

        return new Response(
            'Guardada tarea con id: '.$tarea->getId()
            .' en un nuevo proceso con id: '.$proceso->getId()
        );
    }
```

Al ir a /tarea desde el navegador, se agrega una sola fila a las tablas de productos y proceso. La columna tarea.producto_id de la nueva tarea se establece en el identificador de la nuevo Proceso creado. Doctrine gestiona toda la persistencia por ti:

Si no has trabajado nunca con un ORM, este es el concepto *más difícil*: **debes dejar de pensar en tu base de datos** y, en su lugar, ***solo* pensar en tus objetos**. **En lugar de establecer el identificador de Proceso en tarea, se establece todo el objeto *Proceso*. Doctrine se encarga del resto cuando guardas algo en la base de datos.**

5. Obtención de objetos relacionados

Cuando necesite recuperar objetos asociados, el flujo de trabajo tendrá el mismo aspecto que antes. En primer lugar, obtén un objeto $tarea y, a continuación, accede a su objeto Proceso relacionado (recuerda que una Tarea pertenece a un Proceso):

Añade el siguiente método a la clase **TareaController.php**

```php
#[Route('/tarea/proceso/{id}', name: 'tarea_proceso_show')]
public function showProcesoTarea(TareaRepository $tareaRepository, int $id): Response
{
    $tarea = $tareaRepository->find($id);
    // ...

    $procesoNombre = $tarea->getProceso()->getNombre();

    return new Response('la tarea seleccionada esta dentro del proceso : '.$procesoNombre);
}
```

IMPORTANTE. Observa como a través del objeto $tarea se accede a l objeto proceso.

Prueba este nuevo método con la siguiente llamada por ejemplo:

127.0.0.1/tarea/proceso/8

En este ejemplo, primero se consulta un objeto tarea en función del identificador del id. Más adelante, cuando se llama a $tarea->getPorceso()->getNombre(), Doctrine realiza una segunda consulta para encontrar el Proceso relacionado con esta tarea. Prepara el objeto $proceso y se lo devuelve.

I**MPORTANTE**. Lo importante es el hecho de que tienes acceso a Proceso relacionado con la Tarea, pero los datos de Proceso no se recuperan realmente hasta que solicitas .El Proceso , el objeto Proceso se "carga de forma perezosa o lazy".

Dado que asignamos el lado opcional de OneToMany, también puede consultar en la relación en la otra dirección:

Vamos ahora a **realizar el acceso desde el otro lado de la relación**. Es decir desde el proceso hacia las tareas. Para ello vamos a codificar el método showTareas() dentro del controlador de tareas TareaController.php, que dado un id de proceso permite obtener las tareas.

```php
#[Route('/tareas/proceso/{id}', name: 'tareas_proceso')]
public function showTareas(ProcesoRepository $procesoRepository, int $id): Response
{
    $proceso = $procesoRepository->find($id);

    $tareas = $proceso->getTareas();

    return new Response('El proceso '.$id.'tiene asociadas'.count($tareas));
}
```

6. Relaciones y clases de proxy

Esta "carga diferida" es posible porque, cuando es necesario, Doctrine devuelve **un objeto "proxy" en lugar del objeto verdadero**. Mira de nuevo el ejemplo anterior:

```
$tarea = $tareaRepository->find($id);

$proceso = $tarea->getCategory();
```

```
imprime "Proxies\AppEntityProcesoProxy"
dump(get_class($proceso));
```

Este objeto proxy extiende el verdadero objeto Proceso y tiene el mismo aspecto y actúa exactamente como él. **La diferencia es que, mediante el uso de un objeto proxy, Doctrine puede retrasar la consulta de los datos reales** de la Proceso hasta que realmente necesite esos datos (por ejemplo, hasta que llame a $proceso->getNombre()).

Las clases proxy son generadas por Doctrine y almacenadas en el directorio de caché. Probablemente ni siquiera te darás cuenta de que tu objeto $rpoceso es en realidad un objeto proxy.

En la siguiente sección, cuando recuperes los datos de tarea y el proceso de una sola vez (a través de una *combinación*), Doctrine devolverá el objeto *Proceso* verdadero , ya que no es necesario cargar nada de forma diferida.

7. Consultas (EntityManager createQuery)

Puedes hacer consultas de dos maneras diferentes. Para probar esto añade los siguientes dos métodos a la clase TareaRepository.php

```php
/***@return Tarea[]
 */
public function findTareasMayorPrioridad(int $prioridad): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery(
        'SELECT p
        FROM App\Entity\Tarea p
        WHERE p.prioridad > :prioridad
        ORDER BY p.prioridad ASC'
    )->setParameter('prioridad', $prioridad);

    // returns Tarea[]
    return $query->getResult();
}
```

La segunda forma es usar SQL tradicional

```php
public function findTareas(int $pri): array
{
    $conn = $this->getEntityManager()->getConnection();

    $sql = '
        SELECT * FROM tarea t
        WHERE t.prioridad > :pri
        ORDER BY t.prioridad ASC
        ';

    $resultSet = $conn->executeQuery($sql, ['pri' => $pri]);

    // returns an array of arrays (i.e. a raw data set)
    return $resultSet->fetchAllAssociative();
}
```

Para probar en la clase TareaController.php añade el siguiente método

```php
#[Route('/tareas/prioridad/{pri}', name: 'tareas_show')]
public function showTareasPrioridad(TareaRepository $tareaRepository, int $pri): Response
{
    //Devuelve objetos
    $tareas = $tareaRepository->findTareasMayorPrioridad($pri);
    //Devuelve registros raw es decir $tareas[campo]
    //$tareas = $tareaRepository->findTareas($pri);

        if (!$tareas) {
            throw $this->createNotFoundException('No tarea found ');
        }

        //Procesar objetos
    $tareas_string="";
    foreach ($tareas as $tarea ){
       $tareas_string=$tareas_string." :".$tarea->getNombre();
    }

    //Procesar datos raw es decir $tareas[campo]
    foreach ($tareas as $tarea ){
     $tareas_string=$tareas_string." :".$tarea['nombre'];
    }
    return new Response('la tareas seleccionadas son : '.$tareas_string);
}
```

IMPORTANTE. Este método usa ambos métodos uno de ellos devuelve datos en bruto(raw) desde la base de datos. El otro devuelve objetos.

Como podrás comprobar ambos método obtienen el mismo conjunto de Tareas

8. Operación de Join

En los ejemplos anteriores, se realizaron dos consultas: una para el objeto original (por ejemplo, una para Proceso) y otra para los objetos relacionados (por ejemplo, los objetos Tarea).

En este caso vamos a hacer un join directamente en el Repositorio, para que cuando se obtenga una tarea ya esté cargado el Proceso.

Para ello en TareaRepositoty.php añade el siguiente método

```php
    public function findOneTareaByIdJoinedProceso(int $tareaId): ?Tarea
    {
        $entityManager = $this->getEntityManager();

        $query = $entityManager->createQuery(
            'SELECT t, p
            FROM App\Entity\Tarea t
            INNER JOIN t.proceso p
            WHERE t.id = :id'
        )->setParameter('id', $tareaId);

        return $query->getOneOrNullResult();
    }

}
```

Importante. Fíjate como se realiza la operación de INNER JOIN

Ahora vamos a usar esta consulta de reunión en el acceso a una tarea y su proceso. Para ello codifica el siguiente método dentro de **TareController.php**

```php
#[Route('/tarea_join/proceso/{id}', name: 'tarea_proceso_show_join')]
public function show_join(TareaRepository $tareaRepository, int $id): Response
{
    $tarea= $tareaRepository->findOneTareaByIdJoinedProceso($id);
    $procesoNombre ="";

    if ($tarea){

        $proceso=$tarea->getProceso();

        if ($proceso){
            $procesoNombre = $proceso->getNombre();
        }
    }


    return new Response('JOIN la tarea seleccionada esta dentro del proceso : '.$procesoNombre);
}
```

Ahora se hace lo mismo que en el ejercicio anterior pero cuando se hace $tarea->getProceso() no se hace una segunda consulta a la base de datos.