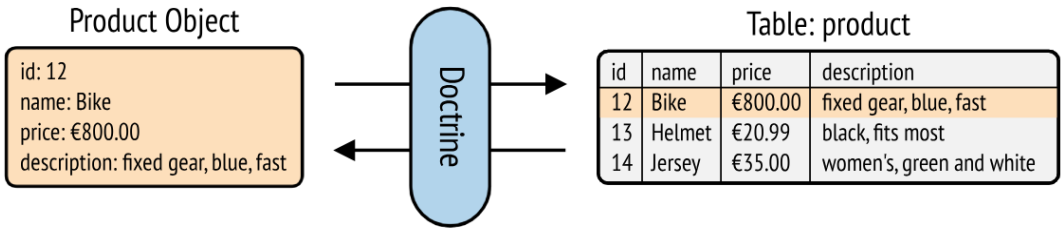
Actividad

ORM ,Doctrine y bases de datos

1. Introducción

Una de las tareas más comunes es el tratamiento de información con una base de datos. Symfony integra el ORM Doctrine por defecto, librería cuyo objetivo es proporcionar una herramienta potente para el tratamiento de la información, mediante el mapeo de objetos a relaciones y viceversa.

En el siguiente esquema se puede observar como Doctrine(ORM) realiza de puente entre los objetos del modelo de programación y las relaciones del modelo de persistencia en base de datos.



Doctrine **permite mapear objetos a una base de datos relacional**, como MySQL, PostgreSQL o SQLServer, **aunque también se puede hacer con MongoDB** con la librería Doctrine ODM y el bundle DoctrineMongoDBBundle.

2. Actividad práctica

En esta actividad práctica vamos a usar el Doctrinbe (ORM) para hacer persistencia de objetos en una base de datos relacional. Empezaremos con una sencilla clase, para luego en actividades posteriores trabajar con más de una clase

Antes de seguir arranca la base de datos mysql de tú instalación

```
sudo /opt/lammp/xammp start
```

```
aw@daw-VirtualBox:~$ sudo /opt/lampp/xampp start
sudo] contraseña para daw:
tarting XAMPP for Linux 8.2.4-0...
AMPP: Starting Apache...fail.
AMPP:  Starting diagnose...
AMPP:  Sorry, I've no idea what's going wrong.
AMPP:  Please contact our forum http://www.apachefriends.org/f/
ast 10 lines of "/opt/lampp/logs/error_log":
ail: no se puede abrir '/opt/lampp/logs/error.log' para lectura: No existe el
chivo o el directorio
AMPP: Starting MySQL...ok.
AMPP: Starting ProFTPD...ok.
```

Nota: Si al arrancar apache obtienes un error, para el servicio de apache2

```
Sudo service apache2 stop
```

Y vuelve a ejecutar la orden anterior

```
daw@daw-VirtualBox:~$ sudo /opt/lampp/xampp start
Starting XAMPP for Linux 8.2.4-0...
XAMPP: Starting Apache...ok.
XAMPP: Starting MySQL...already running.
XAMPP: Starting ProFTPD...already running.
```

3. Instalación de Doctrine

En primer lugar, instala Doctrine a través del Symfony, así como el MakerBundle **con los siguientes dos comandos:**

```
$composer require symfony/orm-pack
$ composer require --dev symfony/maker-bundle
```

Podrás observar en el archivo .env de tú proyecto que hay algunas lineas de configuración realacionadas con la base de datos

4. Configuración de la base de datos

La información de conexión a la base de datos **se almacena como una variable de entorno denominada** DATABASE_URL dentro del **archivo .env** en la raíz del proyecto. Abre el archivo y verás una entrada como la siguiente:

```
#
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=10.5.8-MariaDB"
```

Sustituye la parte **db_user:db_password** por **root** y el nombre de la base de datos por **dawdb**

```
DATABASE_URL="mysql://root@127.0.0.1:3306/dawdb?serverVersion=10.5.8-MariaDB"
```

Con esto le estas configurando la cadena de conexión a la base de datos del proyecto

5. Crear la base de datos con Doctrine

Doctrine puede crear la base de datos que aparece en la variable db_name con el siguiente comando, ejecútalo y veras como se crea la base de datos:

```
php bin/console doctrine:database:create
```

Este comando creará la base de datos que has puesto en el paso anterior (en el archivo .env) que **en nuestro caso es dawdb**.

Comprueba que se ha creado la base de datos correctamente. Para ello ejecuta el servidor apache de lammp y accede a la consola de administración de mysql

https://localhost/phpmyadmin

6. Creación de Entidades

Supongamos que está creando una aplicación que trabaja con tareas (por ejemplo una aplicación de gestión de proyectos). Necesitaras **crear una clase Tarea** para trabajar con este concepto. En terminología del ORM se denominan Entidades y son clases de objetos que tendrán persistencia.

A continuación sigue los siguientes pasos para crear la Entidad **Tarea**

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

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

created: src/Entity/Tarea.php
created: src/Repository/TareaRepository.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]:
> no

updated: src/Entity/Tarea.php

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

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

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

```
updated: src/Entity/Tarea.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
```

NOTA: Fíjate que si vas a VSCode se habrá creado un **nuevo archivo** denominado **Tarea.php** en la **carpeta Entity** del proyecto. Si abres el archivo verás una clase entidad denominada Tarea anotada con anotaciones de Doctrine para mapearla a la base de datos.

```php
<?php

namespace App\Entity;

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

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

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

    #[ORM\Column]
    private ?int $prioridad = null;

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

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

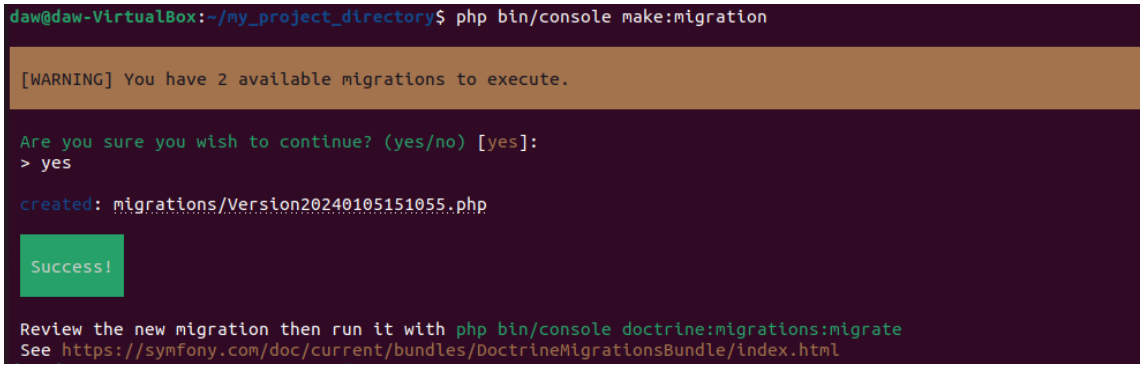
7. Migraciones: Creación de las tablas/esquemas de la base de datos

La clase Tarea está completamente configurada y lista para guardarse en una tabla de tarea de la base de datos. Si acabas de definir esta clase, la base de datos

aún no tiene la tabla de tareas. Para agregarla, se usa DoctrineMigrationsBundle, que ya está instalado:

```
php bin/console make:migration
```

Si todo funcionó, deberías ver algo como esto:



Si abre este archivo contiene el SQL necesario para actualizar su base de datos. Para ejecutar ese SQL, ejecute las migraciones:

```php
<?php

declare(strict_types=1);

namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20240105151055 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE TABLE tarea (id INT AUTO_INCREMENT NOT NULL, nombre VARCHAR(255)
NOT NULL, prioridad INT NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE
`utf8mb4_unicode_ci` ENGINE = InnoDB');
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('DROP TABLE tarea');
    }
}
```

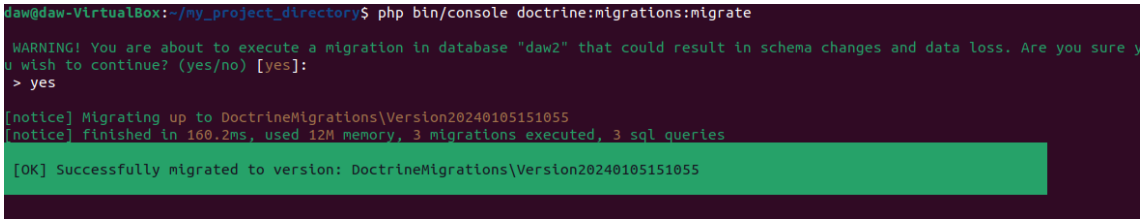
Nota: Abre el archivo terminado en php, para ver el conenido de los comandos SQL.

Este comando ejecuta todos los archivos de migración que aún no se han ejecutado en la base de datos. Debe ejecutar este comando en producción cuando realice la implementación para mantener actualizada la base de datos de producción.

Ejecuta ahora el comando para crear la tabla relación tarea

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

La salida del comando será así



Nota: Si ya has creado la tabla y vuelves a intentar crearla dará error

8. Migraciones y adición de más campos a las clases Entidad existentes

¿Qué sucede si necesita agregar una nueva propiedad de campo a una Tarea, como una descripción? Puedes editar la clase para agregar la nueva propiedad. Pero, también puedes usar make:entity de nuevo:

```
$ php bin/console make:entity

Class name of the entity to create or update
> Product

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

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

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)
```

Si abres el archivo Tarea.php verás que se ha añadido un nuevo campo de descripción con los métodos set y get

```php
#[ORM\Column(length: 255, nullable: true)]
private ?string $descripcion = null;
```

La nueva propiedad está asignada, pero aún no existe en la tabla tarea. Para ello ejecuta

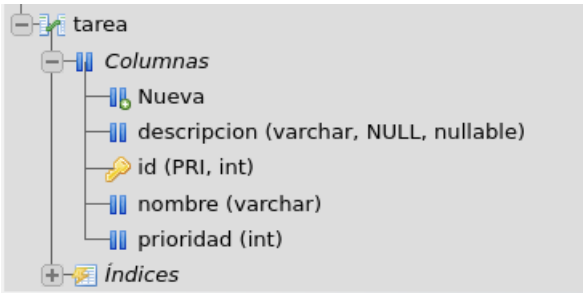
```
php bin/console make:migration
```

```php
<?php

declare(strict_types=1);

namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20240105152748 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE product DROP description');
        $this->addSql('ALTER TABLE tarea ADD descripcion VARCHAR(255) DEFAULT NULL');
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE product ADD description LONGTEXT NOT NULL');
        $this->addSql('ALTER TABLE tarea DROP descripcion');
    }
}
```

Y por último ejecuta:

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

Comprueba en la base de datos que la nueva columna se ha añadido a la tabla tarea:



9. Persistir objetos en la base de datos

¡Es hora de guardar un objeto Tarea en la base de datos! Vamos **a crear un nuevo controlador** para experimentar con esta función. Crea un controlador de entidades Tareas denominado **TareaController**:

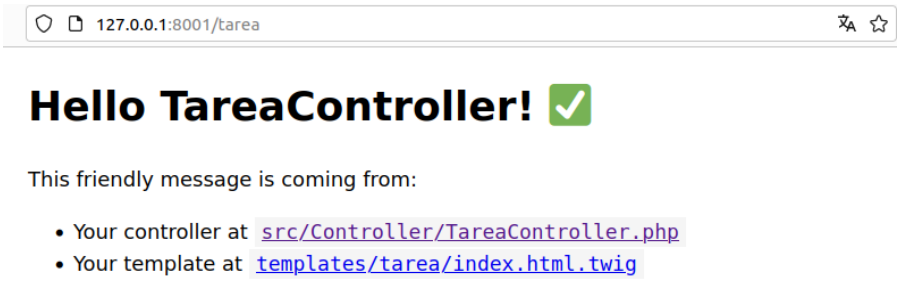
```
php bin/console make:controller TareaController
```

```
daw@daw-VirtualBox:~/my_project_directory$ php bin/console make:controller TareaController
 created: src/Controller/TareaController.php
 created: templates/tarea/index.html.twig

  Success!

 Next: Open your new controller class and add some pages!
```

El código creado para Controlador será el siguiente:

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class TareaController extends AbstractController
{
    #[Route('/tarea', name: 'app_tarea')]
    public function index(): Response
    {
        return $this->render('tarea/index.html.twig', [
            'controller_name' => 'TareaController',
        ]);
    }
}
```

Si ejecutas el controlador obtendrás un resultado como el siguiente (debes modificar algunas cosas antes):



Vamos a crear una nueva tarea en el controlador y la vamos a guardar en la base de datos. Para ello añade el siguiente código al controlador:

```php
<?php

namespace App\Controller;

use App\Entity\Tarea;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```php
class TareaController extends AbstractController
{
    #[Route('/tarea', name: 'app_tarea')]

    public function createTarea(EntityManagerInterface $entityManager): Response
    {
        $tarea= new Tarea();
        $tarea->setNombre("Diagrama ER");
        $tarea->setPrioridad(1);
        $tarea->setDescripcion("Diagrama ER inicial del proyecto");
        // tell Doctrine you want to (eventually) save the $tarea  (no queries yet)
        $entityManager->persist($tarea );
        // actually executes the queries (i.e. the INSERT query)
        $entityManager->flush();
        return new Response('Tareas guardadas ');
    }
}
```

Prueba ahora como el controlador crea tres tareas en la base de datos cuando le llamas:

http://127.0.0.1/tarea

Echa un vistazo al ejemplo anterior con más detalle:

- El argumento $entityManager EntityManagerInterface le dice a Symfony que inyecte el servicio Entity Manager en el método del controlador. Este objeto **es responsable de guardar objetos en la base de datos** y de obtener objetos de ella.

- Se crea una instancia y se trabaja con el objeto $tarea como cualquier otro objeto PHP normal.
-
- La llamada persist($tarea) le dice a Doctrine que "administre" el objeto $tarea. Esto **no** hace que se realice una consulta a la base de datos.
-
- Cuando se llama al método flush(), Doctrine examina todos los objetos que administra para ver si es necesario conservarlos en la base de datos. En este ejemplo, los datos del objeto $ no existen en la base de datos, por lo que el administrador de entidades ejecuta una consulta INSERT y crea una nueva fila en la tabla del tarea.

IMPORTANTE: Ya sea que estés creando o actualizando objetos, **el flujo de trabajo es siempre el mismo:** Doctrine es lo suficientemente inteligente como para saber si debe INSERTAR o ACTUALIZAR su entidad.

Crea por ti mismo dos tareas más

tarea	nombre	prioridad	descripción
	Diagrama de objetos	2	Diagrama dinámico
	Digrama de interfaces	1	Digrama de las interfaces del sistema

10. Alcanzando objetos de la base de datos

Recuperar un objeto de la base de datos es muy fácil. Supongamos que quieres poder ir a /tarea/5 para ver la tarea 5. Añade el siguiente método al TareaController.php

```php
#[Route('/tarea/{id}', name: 'tarea_show')]
public function show(EntityManagerInterface $entityManager, int $id): Response
{
    $tarea= $entityManager->getRepository(Tarea::class)->find($id);

    if (!$tarea) {
        throw $this->createNotFoundException(
            'No product found for id '.$id
        );
    }

    return new Response('la tarea seleccionada es : '.$tarea->getNombre());
ory/templates/product/index.html.twig • Untracked
}
```

Prébalo.

Otra posibilidad es utilizar el TareaRepository utilizando e inyectándolo por el contenedor de inyección de dependencias:

Añade para ello la siguiente importación en el controlador de tareas

```php
use App\Repository\TareaRepository;
```

Y cambia el método show como sigue

```php
#[Route('/tarea/{id}', name: 'tarea_show')]

public function show(TareaRepository $tareaRepository, int $id): Response
{
    $tarea = $tareaRepository->find($id);

        if (!$tarea) {
            throw $this->createNotFoundException('No product found for id '.$id);
        }

        return new Response('la tarea seleccionada es : '.$tarea->getNombre());
}
```

Pruebalo deberas obtener el mismo resultado

Cuando se consulta un tipo particular de objeto, siempre se utiliza lo que se conoce como su "repositorio". **Puedes pensar en un repositorio como una clase PHP cuyo único trabajo es ayudarte a obtener entidades de una determinada clase.**

Normalmente un objeto de repositorio tendrá muchos métodos auxiliares. Por ejemplo para nuestro caso la clase TareaRepository tiene la siguiente estructura (accede al archivo desde VSCode):

```php
<?php

namespace App\Repository;

use App\Entity\Tarea;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

/**
 * @extends ServiceEntityRepository<Tarea>
 *
 * @method Tarea|null find($id, $lockMode = null, $lockVersion = null)
 * @method Tarea|null findOneBy(array $criteria, array $orderBy = null)
 * @method Tarea[]    findAll()
 * @method Tarea[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class TareaRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Tarea::class);
    }

//    /**
//     * @return Tarea[] Returns an array of Tarea objects
//     */
//    public function findByExampleField($value): array
//    {
//        return $this->createQueryBuilder('t')
//            ->andWhere('t.exampleField = :val')
//            ->setParameter('val', $value)
//            ->orderBy('t.id', 'ASC')
//            ->setMaxResults(10)
```

11. Actualización de un objeto

Una vez que hayas obtenido un objeto de Doctrine, puedes interactuar con él de la misma manera que con cualquier modelo de PHP. En el siguiente ejemplo se actualiza el nombre de la Tarea seleccionada

```php
#[Route('/tarea/edit/{id}', name: 'tarea_edit')]
public function update(EntityManagerInterface $entityManager, int $id): Response
{
    $tarea = $entityManager->getRepository(Tarea::class)->find($id);

    if (!$tarea) {
        throw $this->createNotFoundException(
            'No product found for id '.$id
        );
    }

    $tarea->setNombre('Nuevo nombre actualizado');
    $entityManager->flush();

    return $this->redirectToRoute('tarea_show', [
        'id' => $tarea->getId()
    ]);
}
```

Prueba por ejemplo cambiando el nombre de una tarea de la siguiente forma:

127.0.0.1/tarea/edit/3

En este caso cambiaremos el nombre de la Tarea con id=3

El uso de Doctrine para editar un producto existente consta de tres pasos:

1. buscar el objeto ;
2. modificar el objeto;
3. Llamar a flush() en el administrador de entidades.

Puedes llamar a $entityManager->persist($tarea), pero no es necesario: Doctrine ya está "observando" tu objeto en busca de cambios.

12. Eliminación de un objeto

La eliminación de un objeto es muy similar, pero requiere una **llamada al método remove() del administrador de entidades**:

```
$entityManager->remove($tarea);
$entityManager->flush();
```

Prueba a eliminar una tarea, añade el siguiente método a la clase TareaController

```php
#[Route('/tarea/delete/{id}', name: 'tarea_delete')]
public function delete(EntityManagerInterface $entityManager, int $id): Response
{
    $tarea = $entityManager->getRepository(Tarea::class)->find($id);

    if (!$tarea) {
        throw $this->createNotFoundException(
            'No product found for id '.$id
        );
    }

    $entityManager->remove($tarea);

    $entityManager->flush();

    return new Response('la tarea seleccionada se ha borrado : ');
}
```

Como es de esperar, el método remove() notifica a Doctrine que desea eliminar el objeto dado de la base de datos. La consulta DELETE no se ejecuta realmente hasta que se llama al método flush().

Comprueba que en la base de datos no existe la tarea que acabas de borrar

13. Consulta de objetos: el repositorio

Ya has visto cómo el objeto repository te permite ejecutar consultas básicas sin ningún trabajo:

```php
$repository = $entityManager->getRepository(Tarea::class);
$product = $repository->find($id);
```

Pero ¿qué pasa si necesitas una consulta más compleja? Al generar la entidad con make:entity, el comando *también* generó una clase TareaRepository:

Cuando obtienes tu repositorio (es decir, ->getRepository(Tarea::class)), ¡en *realidad es* una instancia de *este* objeto! Esto se debe a que la configuración repositoryClass que se generó en la parte superior de la clase de entidad Tarea.

Supongamos que deseas consultar las Tareas con prioridad 1.

Actividad Agrega un nuevo método en la clase TareaRepository para recupere las tareas mayores de una prioridad dada:

```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();
    }
```

Doctrine también proporciona un Generador de consultas, una forma orientada a objetos para escribir consultas. Se recomienda usar esto cuando las consultas se construyen dinámicamente (es decir, basadas en condiciones de PHP):

El método findTareasMayorPrioridad() encuentra las tareas de mayor prioridad que una dada por parámetro

Ahora cambiamos el controlador **TareaController.php** para incluir un método que use este método **showTareasProridad()**

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

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

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

        return new Response('la tareas seleccionadas son : '.$tareas_string);
}
```

14. Consultas con SQL

Además, puede consultar directamente con SQL . Para probarlo vamos a crear n nuevo método para obtener las tareas de nuestra base de datos pero ahora usando SQL

```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();
}
```

En el controlador hacemos una llamada a este nuevo método

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

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

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

    foreach ($tareas as $tarea ){
     $tareas_string=$tareas_string." :".$tarea['nombre'];
    }
    return new Response('la tareas seleccionadas son : '.$tareas_string);
}
```

NOTA IMPORTANTE: Observa como ahora la consulta no devuelve objetos, sino un registro en cada posición del array